Lecture Notes in Earth System Sciences

David A. Yuen · Long Wang
Xuebin Chi · Lennart Johnsson
Wei Ge · Yaolin Shi *Editors*

GPU Solutions to Multi-scale Problems in Science and Engineering

Springer

Lecture Notes in Earth System Sciences

For further volumes:
http://www.springer.com/series/10529

David A. Yuen · Long Wang
Xuebin Chi · Lennart Johnsson
Wei Ge · Yaolin Shi
Editors

GPU Solutions to Multi-scale Problems in Science and Engineering

Springer

Editors
David A. Yuen
Department of Earth Sciences and
Minnesota Supercomputing Institute
University of Minnesota
Minneapolis, MN
USA

and

School of Environmental Sciences
China University of Geosciences
Wuhan
People's Republic of China

Long Wang
Network Information Center
Beijing
People's Republic of China

Xuebin Chi
Supercomputing Center
Beijing
People's Republic of China

Lennart Johnsson
Computer Science
University of Houston
Houston, TX
USA

Wei Ge
Institute of Process Engineering
Chinese Academy of Sciences
Beijing
People's Republic of China

Yaolin Shi
Laboratory of Computational Geodynamics
Chinese Academy of Sciences
Beijing
People's Republic of China

ISSN 2193-8571 ISSN 2193-858X (electronic)
ISBN 978-3-642-16404-0 ISBN 978-3-642-16405-7 (eBook)
DOI 10.1007/978-3-642-16405-7
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012952572

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The past several years have witnessed a great transformative scenario occurring in the computational science by the extremely rapid incursion made by GPU and many-core architecture into the arena of high-performance computing (HPC). At the Supercomputing 2009 meeting in Portland, a group of us (X. Chi, D. A. Yuen, and H. Tufo from University of Colorado) gathered together and formulated plans for a GPU conference to be held in China. First, it was supposed to be held in Shanghai as proposed by Jifeng Yao from Shanghai Supercomputing Center, but because of the EXPO 2010, we had to move it to Harbin, which turns out to be a more delightful locale during the summer. Our timing for this conference was quite propitious, as the platforms armed with GPU and fast networking carried the day and captured three out of the top five slots of the TOP-500 list in Supercomputing 2010 in New Orleans. This auspicious event justified our thinking back in 2009 about the potential importance of GPU computing.

The first International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering (GMP-SMP2010) kicked off July 26, 2010 in Harbin, the capital of Heilongjiang province in Northeast (Dongbei) China. The workshop was organized by the Supercomputing Center of Computer Network Information Center, Chinese Academy of Sciences (CAS) in Beijing, the Graduate University of CAS, and the Chinese Society of Theoretical and Applied Mechanics. Nearly 100 computational experts and scholars from the world's well-known universities and institutes such as the University of Houston, the National Astronomical Observatory of Japan, Hong Kong Baptist University, the University of Chicago, the Tokyo Institute of Technology, Japan, Brown University, the University of Amsterdam, the University of Erlangen-Nuremberg, National Center of Atmospheric Center, University of Minnesota, Macalester College, University of Bonn, the Chinese Academy of Science, Tsinghua University, Peking University, Fudan University, China, University of Science and Technology of China (USTC), attended this international workshop.

The leadoff talk was given by Professor Xuebin Chi, the Director of the Supercomputing Center of CNIC, CAS. He stressed strongly that a revolutionary change is brewing in the field of technologies and applications of high-

performance computing due to the rapid development of GPU and many-core technology. He further expressed the hope that the discussion on GPU applications to multiscale problems in science and engineering be extremely fruitful. Following Professor Chi's speech, Professor David Yuen from University of Minnesota also gave an exciting opening talk, encouraging strongly vigorous participation by the students from China and the international contingent.

During the two-day workshop period, the attendees discussed topics on GPU solutions to multiscale problems in science and engineering. The workshop consisted of three sessions, the keynote lecture session, the invited lecture session, and the student and poster session. The keynote lectures are "*Development and application of a HPC system for multiscale discrete simulation Mole-8.5*" given by Professor Wei Ge from IPE, CAS, "*Acceleration for energy efficient, cost effective HPC*" given by Professor Lennart Johnsson from the University of Houston, USA, "*Practical Random Linear Network Coding on GPUs*" given by Professor Xiaowen Chu from the Hong Kong Baptist University, China and "*GRAPE and GRAPE-DR*" given by Professor Jun Makino from the National Astronomical Observatory of Japan, Japan. In the subsequent talks, issues on seeking GPU solutions to multiscale problems were addressed from different viewpoints, such as focusing on high-performance computing methods and algorithms, efficient software implementation techniques, the construction of scientific computing environment, the mainstream development trends, as well as other GPU-related issues in scientific computing and visualization technology. In the second afternoon students, both graduates and undergraduates from USA and China, gave a bulk of the presentations as well as interesting posters.

In the closing banquet, Professor David Yuen, on behalf of the organizing committee, announced that the conference proceedings will be published soon and the workshop will be held again in 2011, and Lanzhou was mentioned as a potential place. Plans for a book from the workshop were also laid out at the closing banquet on July 28.

As far as we are aware of, the few extant books on GPU and multi-core computing (e.g., Kirk and Hwu 2010; Kurzak et al. 2011) are written by computer scientists. Thus, there is now a dire need for a book with a strong applicational bent in order to encourage more people to join the GPU game.

Our papers are divided into two types: long expository papers with ample illustrations and examples and short contributions with a particular applicational focus in mind. The book is divided into eight sections. In section 1 we begin with this preface. Then we follow with an article by Matt Knepley and David Yuen - which addresses the reasons why scientists and engineers should be considering GPU. This is followed by a chapter by Lang, Wang, and Yuen offering photos of the workshop itself. This book is rather unique at this time has articles, spanning many different areas in science and engineering, which show the health of this burgeoning field. This book is to be contrasted with the recent book edited by Kurzak et al. (2011), which focused mainly on hardware and algorithms and is devoted to a computer science audience rather than people in computational science.

This book covers aspects of hardware and green computing in section 2. We discuss in section 3 software libraries from both China and the USA., namely PARRAY by Chen Yifeng and PETSc by the group at Argonne. In section 4 we shows the industrial applications on GPU. There Wei Ge's group performed an outstanding job in getting a super performance from a GPU–CPU system using thousands of GPU. In section 5, we shows the inroads made by GPU in density-functional theory and electronic structures. Section 6 deals with geophysical and fluid dynamical applications. We see the 145 Tflop performance on a weather code from the Japanese group at Tokyo Institute of Technology. There is also application of GPU on 3D elastic-wave propagation by Taro Okamoto and also by S. Song and his colleagues. In section 7, on algorithms and solvers have a thorough discussion of multigrid solver as applied to industrial problems by H. Koestler from Germany. In the final section on visualization, we present a variety of application on imaging and microtomography using GPU.

We plan to communicate broadly about the potential of GPU and many-core computing to the scientific and engineering community out there and not restricted to computer scientists. We hope this book will give the future perspectives of GPU to scientists and engineers and will stimulate further growth in this field. We thank both the Chinese Academy of Sciences and the OCI program of National Science Foundation for their generous support. We are very grateful for the help provided by Xianyu Lang, Qing Zhao, and Yichen Zhou in preparation of this volume.

Minnesota, USA — David A. Yuen
Beijing, China — Long Wang
Xuebin Chi

References

Kirk DB, Hwu WW (2010) Programming massively parallel processors: a hands-on-approach. Elsevier, Amsterdam, p 256

Kurzak J, Bader DA, Dongarra J (eds) (2011) Scientific computing with multicore and accelerators. CRC Press, Boca Raton, p 480

Contents

Part VI Geophysical and Fluid Dynamical Application

Part VIII Visualization

Part I
Introductory Material

Chapter 1
Why Do Scientists and Engineers Need GPU's Today?

Matthew G. Knepley and David A. Yuen

Abstract Recently, a paradigm shift in computer architecture has offered computational science the prospect of a vast increase in capability at relatively little cost. The tremendous computational power of graphics processors (GPU) provides a great opportunity for those willing to rethink algorithms and rewrite existing simulation codes. In this introduction, we give a brief survey of GPU computing, and its potential capabilities, intended for the general scientific and engineering audience. We will also review some challenges facing the users in adapting the large toolbox of scientific computing to these changes in computer architecture, and what the community can expect in the near future.

1.1 Introduction

In the past 4 years, the tremendous progress made by GPU-based computers has been revolutionary. In November 2010, the Tianhe 1A (see Fig. 1.1), which is a hybrid massively parallel computer consisting of thousands Intel Westmeres and Nvidia Fermi boards, assumed the No.1 position in the Top-500 list (Meuer et al. 2011), signaling a paradigm shift. That this machine, together with 3 more out of the first 5 places, were CPU-GPU machines based in Asia has come as a surprise to the computational community at large, particularly at Supercomputing 2010 in New Orleans. We would like to briefly review some successes of GPU computing

M. G. Knepley (✉)
Computation Institute, University of Chicago,
Chicago, IL 60637, USA
e-mail: knepley@ci.uchicago.edu

D. A. Yuen
Department of Earth Sciences and Minnesota Supercomputing Institute,
University of Minnesota, Minneapolis, MN 55455, USA
e-mail: daveyuen@gmail.com

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_1,

Fig. 1.1 The Tianhe-1A couples massively parallel GPUs with multi-core CPUs, employing 7,168 NVIDIA Tesla M2050 GPUs and 14,336 CPUs. It is a 2.507 Petaflop system consuming only 4.04 MW

solutions for scientists and engineers, and motivate them to explore further using the in-depth articles in this volume.

From the point of view of a computational scientist, a GPU is a massively parallel processor with large memory bandwidth. Its memory is organized hierarchically, into small local processor memories, slightly larger shared memories for vector processor groups, and global memories close to generic CPU sizes (see Fig. 1.2). Movement between the various memory levels is explicitly organized by the programmer. Volkov (2008) has given a lucid explanation of achievable performance with this memory organization based upon the classical Little's Law (Little 1961) from queuing theory, namely that the parallelism required to achieve peak performance is equal to the total throughput multiplied by the latency, as shown in Fig. 1.3. Thus, we can quantify the concurrency needed to take full advantage of this new devices. Moreover, early generations of GPUs had large gaps between single and double precision performance, but this is no longer true with the introduction of the Fermi (NVIDIA 2009), which puts the gap between single and double-precision at only a factor of two.

1.2 Why Move to the GPU?

The GPU platform can enable desktop computational science. For a modest investment, around $2000 at the time of writing, individual scientist can harness one teraflop that can be plugged into a common wall outlet. GPUs deliver exceptional performance both per watt and per dollar. Four of the top ten computing installations in the Green 500 use GPUs, including the third ranked system, the DEGIMA cluster in Nagasaki. The theme of green computing is addressed further in this volume in the chapter by Lennart Johnsson.

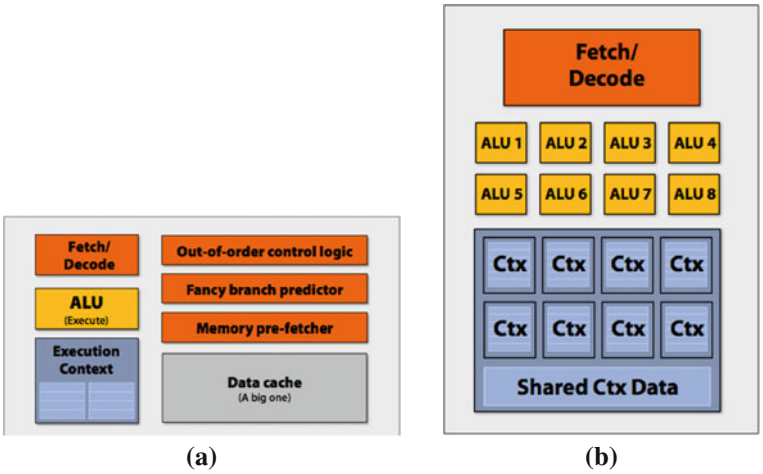


Fig. 1.2 Comparison between **a** a standard CPU core layout, and **b** the higher throughput GPU core. Image credit: Kayvon Fatahalian from Carnegie Mellon University

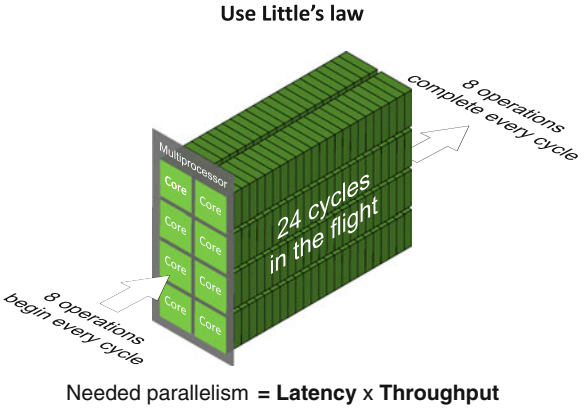


Fig. 1.3 Pictorial representation of the throughput on an example many-core machine. Since there is a 3 cycle latency and 24 operations can be in flight at once, 8-way parallelism is required to achieve peak performance. Image credit: Vasily Volkov from University of California Berkeley

Since the GPU can be easily combined with a common workstation, management of the system is also very easy. The bureaucracy and lengthy wait times of large computing centers are bypassed for all but the largest problems, and development costs are greatly reduced. Moreover, the GPU packages visualization along with the computation. This allows for immediate examination of the results and exploration through visual interactivity, which drives science and engineering innovation.

There are many good reasons for computational researchers to examine GPUs, but the most compelling is that increases in the number of cores is now inevitable.

Due to power limitations, the clock frequency of processors has stagnated. Future increases in performance will come almost entirely from increases in core count. In fact, Intel and AMD currently sell more multicore than single-core processors. GPUs are not as flexible as the Intel Sandy Bridge (2011) or AMD Magny Cours (2011) architectures, but represent a large increase in both flops per watt and flops per dollar.

1.3 Migration to the GPU

Migration of existing applications to a many-core platform can incur significant cost. In this section, we review pieces of software technology that can make this process as efficient as possible. Three broad categories of tools can be distinguished: support for GPU kernel development, high level libraries, and automated compiler solutions. Tools of this type have existed for quite some time on CPUs, however changing requirements dictate a shift in implementation strategies and goals. The memory latency penalty when changing levels in the memory hierarchy is even more severe on a GPU. The latency for global memory transactions on the new Fermi 2070 from NVIDIA is more than 100 times greater than a transaction to a register or shared memory. Moreover, the hardware provides very little help to the programmer since most "caches", such as shared memory or textures, are explicitly managed, and the recently introduced caches in the Fermi architecture are very small (16 K) compared to 12 MB on the Intel i7-970.

At the lowest level, application developers will want to produce kernels, using either CUDA or OpenCL, which will execute on the accelerator. The PyCUDA Klöckner (2011a) and PyOpenCL Klöckner (2011b) packages from Andreas Klöckner present an innovative way to develop and test kernels. CUDA or OpenCL source is handled as a Python string, and compiled modules become Python objects. They can be easily executed with different thread configurations using only a Python function call. Sophisticated Python string manipulation and templating packages can be used to easily generate a range of kernel variants. Another key component of development is debugging. NVIDIA has recently released the cuda-gdb program, which allows seamless debugging between CPU and GPU, switching between thread blocks, and examination of local state on the thread block.

There are several excellent library packages for linear algebra, both dense and sparse, on the GPU. NVIDIA alone supports CUBLAS (NVIDIA 2010), the Thrust libraries of STL-like iterators (Bell and Hoberock 2011), Cusp (Bell and Garland 2011) and CUDASPARSE (NVIDIA 2010) for sparse matrix multiplication, and OpenCurrent (Cohen 2011) which handles structured geometric multigrid. More advanced dense factorizations, such as QR and SVD, are also available in the FLAME (Quintana-Orti et al. 2003; Zee 2009) package from UT Austin and the Magma (Dongarra et al. 2011) package from UT Knoxville. FFT packages are available from NVIDIA (CUFFT (2007)), and Peking University (Parray) which is detailed in the chapter by Yifeng Chen. The CUDA Data Parallel Primitives library

(Owens et al. 2011) from UC Davis is similar to Thrust, but provides more graph-based algorithms. There are notable holes in library support, such as the lack of an efficient sparse triangular solve, needed both for direct factorization and multigrid preconditioners, but overall potential users have a rich set of linear algebra support to choose from.

At a higher level, the PETSc package (Balay et al. 1997, 2011) provides linear and nonlinear algebraic solvers, by building on the serial linear algebra cited above. PETSc handles parallel communication through MPI, and can hook together a host of accelerators. Moreover, algorithms in PETSc, such as Krylov solvers, the Newton iteration, or Implicit-Explicit (IMEX) time-stepping solvers (Kennedy and Carpenter 2003; Hairer and Wanner 1996), all handle logic on the CPU, but algebraic computation on GPU. Vectors and matrices reside on the GPU throughout the algorithm, so that costly communication is avoided, resulting in complete reuse of algorithmic structure of the CPU versions. This same strategy allows parallel sparse eigenvalue problems to be solved on the GPU through the SLEPc package. The PETSc implementation is detailed in the chapter by Minden, et al. in this volume.

In the past, accessing GPU resources required code development on the part of the user. However, now one can substantially accelerate some application codes using fully automated approaches. The Jacket system from Accelereyes (2011) is able to take sections of standard Matlab code and convert them to CUDA kernels. It executes these kernels on the GPU and returns the results, which are seamlessly integrated into the running Matlab program. In a similar vein, the PGI compiler (Portland Group Inc 2011) is able to take small code sections with custom annotations, similar to OpenMP, and compile them as CUDA kernels. The compiler handles all the necessary glue code to launch kernels and transfer input and output data. While not appropriate for the most complex procedures, if small kernels can be isolated, these approaches provide a very user-friendly avenue to use GPUs, and has had signal successes shown in the chapter by Zhang et al.

1.4 Example Applications

PyClaw (Mandli et al. 2011; Alghamdi et al. 2011) is a Python-based structured grid solver for general systems of hyperbolic PDEs. PyClaw provides a powerful and intuitive interface to the algorithms of the existing Fortran codes Clawpack and SharpClaw. Due to efficient interfacing tools and design, this incurs very little performance cost while dramatically simplifying code development and use. PyClaw also incorporates the PETSc library, enabling massive parallelism and scalable solvers. The package is further augmented by use of PyWENO Emmett (2011) for generation of efficient high-order WENO code. The combination of these tools interconnected through their Python bindings allows for the efficient solution of a wide range of problems through application of numerical tools that heretofore have never been brought together. It has both an extension for scalable distributed-memory parallelism, and for Riemann solvers and limiters on the GPU, which work together seamlessly.

The GeoClaw (George 2011; Berger et al. 2011) package specializes PyClaw for geophysical simulation, notably tsunami investigation. With a GTX580, PyClaw can achieve 10 Gflops in double precision for the entire run, which by comparison is almost by two orders of magnitude faster than the same run on an intel i9 processor.

Hammond and DePrince have implemented the quantum many-body method coupled-cluster theory on GPUs using CUDA and associated math libraries (DePrince and Hammond 2011). This code is fully hybrid in that it uses threaded kernels on the CPU that overlap with streaming GPU calls, and also uses both task and data parallelism. Moreover, this coupled-cluster iterative solver runs an order of magnitude faster than the best CPU-only codes, such as Molpro. This is the first step in developing a coupled-cluster code for GPU-based supercomputers, such as the Cray Titan to be delivered to ORNL in 2012.

In this volume, Sanchez et al. from the University of Minnesota have reported on their implementation of two- and three-dimensional Rayleigh-Bénard thermal convection with infinite Prandtl number on Nvidia GPUs using a 2nd-order finite difference method together with a discrete-Fourier transform (see Fig. 1.4). By exploiting the massive parallelism of GPU using both CUDA for C and optimized CUBLAS routines, they have run simulations of Rayleigh number up to $6 \cdot 10^{10}$ in two dimensions and up to 10^7 in three dimensions on a single Fermi GPU. On a single Nvidia Tesla C2070 GPU, the implementation have single-precision performance of 535 and 100 Gflop/s respectively. Andreas Klöckner of New York University has developed an unstructured, high-order, parallel Discontinuous Galerkin code, Hedge, for the simulation of electrodynamics and incompressible fluid dynamics. In the results shown in the included chapter, he is able to realize 250 GF on a single NVIDIA GTX 280 for the entire simulation, and nearly 4 TF on a network on 16 GPUs. The ASUCA weather forecasting program, a research code from the Japan Meteorological Agency, is examined in the chapter by Prof. Aoki of Tokyo Institute of Technol-

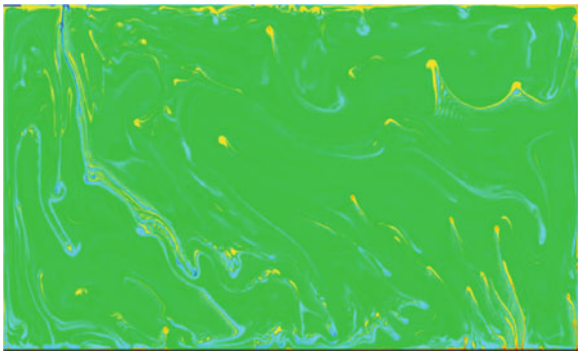


Fig. 1.4 A visualization of the global mantle circulation from Sanchez et al. for Rayleigh number $\mathrm{Ra} = 6 \cdot 10^{10}$ on 2D 2000^2 grid. They use a nonlinear color map of the temperature field, where *blue* represents colder material, *red* and *yellow* represent hot, and *green* represents a temperature with $\pm 2.5\,\%$ of the center of the normalized temperature scale. Refer to the chapter for details of the calculation

ogy. The dynamical core uses a flux form non-hydrostatic compressible form for the equations and is third order accurate in space and time. It achieves 50 GF on a single NVIDIA M2050 node of TSUBAME 2.0 which represents a speedup of twelve times over the Xeon X5670. The code also scales up to the entire machine, computing at a sustained 145 TF in single precision on almost 4,000 GPUs. In the chapter by Taro Okamoto from Titech, large-scale 3D seismic wave propagation using domains up to 1000^3 has reported performance exceeding several hundreds of Gflops.

1.5 Future Perspectives

With the advent of Tianhe-1A, with more than 7,000 GPUs, and TSUBAME2, with over 4,200, supercomputing has clearly begun to embrace manycore and hybrid architectures. As power efficiency requirements become more stringent, this trend toward GPUs should only increase. Although the issue of the stability of the entire computing system remain murky, as the likelihood of failure of some core increases with the number of cores and runtime, the articles in this volume demonstrate that practical science and engineering calculation is possible on the GPU, and that a solid foundation of software infrastructure is currently being constructed to support this. In the future, we hope to see a focus on aspects of the problem which have to date remained largely unexplored.

For many algorithms, particularly those of linear algebra and algebraic solvers, memory bandwidth is the critical resource. This provides another powerful reason to employ single precision floating computation, since it consumes half the bandwidth of double precision. However, single precision cannot be used naively in most solvers due to the sensitivity to roundoff error. Exploration of mixed precision algorithms, and conceptual frameworks for their construction and analysis are sorely needed for modern GPU computing.

As the number of independent functional units, or cores, increases, so does the penalty for synchronization across the computing ensemble. Synchronization is required for any global reduction, such as those required for norms used in convergence tests or dot products necessary for orthogonalization. There are also many source of process jitter, such as OS noise, overlap of multiple kernels, communication between CPU and GPU, scheduling problems, and even excessive heat. Formerly, on machines with fewer than 1,000 cores, these operations were rarely a major factor in runtime. However, at several thousand cores, a Krylov solver can spend a third of its time waiting to synchronize processes. Now with a thousand cores on a single GPU board, synchronization reduction is the next great hurdle for scientific software.

Lastly, GPU and many-core CPU manufacturers, such as Nvidia and Intel, do not have nearly as much experience in massive integration as CPU manufactures, such as IBM, Cray, and Fujitsu. It will most likely take a partnership in this area to bring about truly reliable large scale machines which incorporate GPUs and/or many-core CPUs. The interplay between chips such as the Intel Many Integrated Core (MIC) and NVidia Kepler will present both architectural and programming challenges.

Acknowledgments We thank Gordon Erlebacher, Paul R. Woodward, and Jonathan Cohen for useful discussions, and Kayvon Fatahalian, Vasily Volkv, and David Sanchez for illustrative graphics. We are grateful for the support given by the Minnesota Supercomputing Institute and the CMG program and CIG collaboration of the National Science Foundation. Dave Yuen expresses thanks to the Chinese Academy of Sciences for a senior Visiting Professorship during this period.

References

Accelereyes (2011) Jacket. http://www.accelereyes.com/products/jacket

Alghamdi A, Ahmadia A, Ketcheson D, Knepley M, Mandli K, Dalcin L (2011) PetClaw: a scalable parallel nonlinear wave propagation solver for python. ACM (2011). http://web.kaust.edu.sa/faculty/davidketcheson/petclaw.pdf

AMD (2011) AMD developer central: mangy cours zone. http://developer.amd.com/zones/magny-cours

Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing, Birkhäuser Press, Basel, pp 163–202

Balay S, Brown J, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2011) PETSc. http://www.mcs.anl.gov/petsc

Bell N, Garland M (2011) The cusp library. http://code.google.com/p/cusp-library/

Bell N, Hoberock J (2011) The thrust library. http://code.google.com/p/thrust/

Berger MJ, George DL, LeVeque RJ, Mandli K (2011) The geoclaw software for depth-averaged flows with adaptive refinement. Adv Water Resour (2011). doi:10.1016/j.advwatres.2011.02.016. (in press)

Cohen J (2011) The OpenCurrent library. http://code.google.com/p/opencurrent/

DePrince AE, Hammond JR (2011) Coupled cluster theory on graphics processing units i. the coupled cluster doubles method. J Chem Theory Comput 7(5):1287–1295. doi:10.1021/ct100584w. http://pubs.acs.org/doi/abs/10.1021/ct100584w

Dongarra J et al (2011) Magma. http://icl.cs.utk.edu/magma/

Emmett M (2011) PyWENO documentation. http://memmett.github.com/PyWENO/

George D (2011) GeoClaw documentation. http://depts.washington.edu/clawpack/users/geoclaw.html

Hairer E, Wanner G (1996) Solving ordinary differential equations II: stiff and differential-algebraic problems. Springer series in computational mathematics, vol 14. Springer, Berlin

Intel: Sandy Bridge (2011) The 2nd gen intel core processors. http://www.intel.com/SandyBridge

Kennedy CA, Carpenter MH (2003) Additive runge-kutta schemes for convection-diffusion-reaction equations. Appl Num Math 44:139–181. doi:10.1016/S0168-9274(02)00138-1. http://dl.acm.org/citation.cfm?id=639155.639164

Klöckner A (2011a) PyCUDA. http://mathema.tician.de/software/pycuda

Klöckner A (2011b) PyOpenCL. http://mathema.tician.de/software/pyopencl

Little JDC (1961) A proof for the queuing formula: $L = \lambda W$. Oper Res 9(3): 383–387. http://www.jstor.org/stable/167570

Mandli K, Ketcheson D et al (2011) PyClaw documentation. http://numerics.kaust.edu.sa/pyclaw/

Meuer H, Strohmaier E, Dongarra J, Simon H (2011) The top 500 Supercomputer Sites. http://www.top500.org/

NVIDIA: CUDA: CUFFT library (2007) Technical report, PG-00000-003 V1.1, NVIDIA

NVIDIA: NVIDIA's next generation CUDA compute architecture: fermi (2009) Technical report, NVIDIA

NVIDIA: CUDA: CUBLAS library (2010) Technical report, PG-00000-002 V3.1, NVIDIA

NVIDIA: CUDA: CUSPARSE library (2010) Technical report, PG-05329-032 V01, NVIDIA

Owens J et al (2011) CUDPP. http://code.google.com/p/cudpp/
Portland Group Inc (2011) PGI accelerator compilers. http://www.pgroup.com/resources/accel.htm
Quintana-Ortí G, Quintana-Ortí ES, van de Geijn RA, Van Zee FG, Chan E (2003) Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Trans Math Softw 36(3):14:1–14:26. http://doi.acm.org/10.1145/1527286.1527288
Volkov V, Demmel JW (2008) Benchmarking gpus to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing (SC08) (2008). http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf
Zee FGV (2009) libflame: the complete reference. www.lulu.com

Chapter 2
Happenings at the GPU Conference

Xian-yu Lang, Long Wang and David A. Yuen

The following pages should convey the lively convivial atmosphere at the GPU conference in nice surroundings of Harbin in China's Dongbei region.

1. July 26th: Registration Day

Harbin HANLIN HYATT Hotel Registration

The conference was held in Harbin HANLIN HYATT Hotel at XueFu Road No. 56, NanGang, Harbin, Heilongjiang, China. July 26th is the registration day, in which

X. Lang (✉)
Supercomputing Center of Chinese Academy of Science, Beijing,
People's Republic of China
e-mail: lxy@sccas.cn

X. Lang · L. Wang
Computer Network Information Center, Zhong Guan Cun 4, Beijing 100190,
People's Republic of China

D. A. Yuen
Department of Earth Sciences and Minnesota Supercomputing Institute,
University of Minnesota, Pillsbury Hall 23, Minneapolis, MN 55455, USA
e-mail: daveyuen@gmail.com

D. A. Yuen
School of Environmental Sciences, China University of Geosciences,
Wuhan, People's Republic of China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_2,

almost 100 computational experts and scholars from the world's well-known universities and institutes arrived.

2. July 27th: The First Day

The Group Photo of attendees from the conference

At the beginning, Prof. Xuebin Chi, the director of the Supercomputing Center of CNIC, Chinese Academy of Sciences and Prof. David Yuen from the University of Minnesota gave the exciting opening speeches. Prof. Chi pointed out that a revolutionary change is brewing in the field of technologies and applications of high performance computing due to the rapid development of GPU technology. He further expressed the hope that the discussion on GPU applications to multi-scale problems in science and engineering be fruitful.

Prof. Chi Xuebin, the director of the Supercomputing Center of CNIC, CAS gave the Opening Speech

Prof. Dave A. Yuen from University of Minnesota gave a Second Opening Speech

Mr. John Xie, NVIDIA, China PSG Sr. Sales Manager Spoke on ≪TESLA GPU Computing Update≫

Prof. Wei Ge, IPECAS, China spoke on ≪Development and application of a HPC system for multi-scale discrete simulation—Mole-8.5≫

Prof. Lennart Johnsson, University of Houston, USA ≪Acceleration for energy efficient, cost effective HPC≫

Dr. Matthew Knepley, University of Chicago, USA ≪Fast Multiple Solvers for Vortex Method on GPU≫

Prof. Wei Ge, IPECAS, China spoke on ≪Development and application of a HPC system for multi-scale discrete simulation—Mole-8.5≫

Dr. Peng Wang, NVIDIA, USA ≪GPU Best Practices for HPC Applications at Industry Scale≫

Dr. Taro Okamoto, Tokyo Institute of Technology, Japan ≪GPU-Accelerated Simulation of Seismic Wave Propagation≫

Prof. Hong Liu, IGGCAS, China ≪Accelerating RTM on CUDA platform of CPU/GPU≫

Prof. Yangdong Deng, Tsinghua University, China ≪Accelerating Simulation of Seismic Wave Propagation by Multi GPUs and 3D Decomposition≫

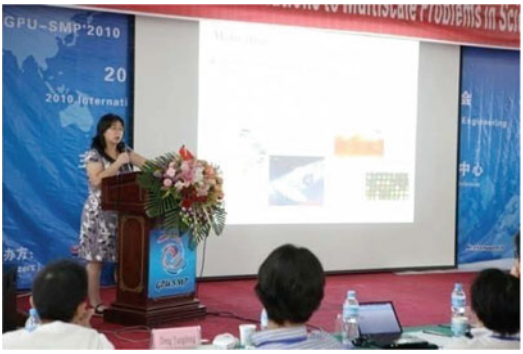
Prof. Takayuki Aoki, Tokyo Institute of Technology, Japan ≪Large-scale CFD Applications on GPU-based Supercomputer at Tokyo Institute of Technology≫

Prof. Yunquan Zhang, ISCAS, China ≪Accelerating Linpack Performance with Mixed Precision Algorithm on CPU+GPGPU Heterogeneous Cluster≫

Prof. Yifeng Chen, Peking University, China ≪Towards Developing Portable FFT Library for GPU Clusters≫

Dr. Junjie Peng, Shanghai University, China ≪Parallel Lattice Boltzmann Method on CUDA Architecture≫

Dr. Thomas Geenen, University of Amsterdam, the Netherlands ≪Iterative deblurring of large 3D datasets from Cryomicrotome imaging using an array of GPUs≫



Dr. Ying Liu, GUCAS, China ≪Some GPU Statistical Applications in SCCAS≫

Dr. Xianfeng He, IPECAS, China ≪Parallel Visualization of Multi-scale simulation: GPU implement and application in Mole-8.5≫

Dr. Long Wang, SCCAS, CNIC, China ≪Some GPU Applications in SCCAS≫

Eighteen talks on GPU solutions to multi-scale problems in science and engineering were given on the first day of the conference. The banquet was held in PORTMAN Russian type restaurant, which is located on Central Avenue, the evening was extremely beautiful.

Banquet PORTMAN Russian Type Restaurant

Holy Sophia Cathedral in the night

3. July 28th: The Second Day

Mr. Rob M. Weiss, Macalester College, USA ≪WebViz≫

Prof. Xiaowen Chu, Hong Kong Baptist University, China ≪Practical Random Linear Network Coding on GPUs≫. Prof. Jun Makino, National Astronomical Observatory of Japan, Japan ≪GRAPE and GRAPE-DR≫

Dr. Andreas Kloeckner, Brown University, USA ≪High-Order Discontinuous Galerkin Methods by GPU Metaprogramming≫

Dr. Harald Koestler, University of Erlangen-Nuremberg, Germany ≪Software and Performance Engineering for numerical codes on GPU clusters≫

Prof. Xisheng Luo, USTC, China ≪GPU accelerated CESE Method for 1D shock tube problems≫

Dr. Rory Kelly, NCAR, Boulder, USA ≪GPU application to Large Numerical simulations in Atmospheres≫

Prof. Wensheng Bian, ICCAS, China ≪Multi-scale Simulation of Hydrogen-oriented Chemical Reactions≫

Prof. Dongsheng Xu, IMRCAS, China ≪Nucleation and reaction of dislocations in some metals and intermetallic compound TiAl≫

Dr. Shuxia Zhang, University of Minnesota, USA ≪High Throughput Computing and Interactive Visualization on a Desktop Supercomputer≫

Dr. Gaojin Wen, SIAT, CAS, China ≪Parallelized Binomial Option Pricing on Heterogeneous Supercomputer≫

Dr. Ying Xu, SSC, China ≪Fully-resolved direct numerical simulation of particle-laden turbulent flow using GPUs≫

Dr. Yanbin Yang, BAO, CAS, China ≪Accelerating Multi-Scale Astrophysical Particle Simulations with Many-Core Hardware≫

Dr. Bojing Zhu, GUCAS, China ≪Correlation of Zipingpu reservoir and 2008 Wenchuan earthquake by fluid flow driven pore-network crack mode≫

Mr. Sen Li, SCCAS, CNIC, China ≪GPU Computing On Computational Fluid Dynamics≫

Mr. David Sanchez, University of Minnesota, USA ≪High Rayleigh Number 3D Mantle Convection on GPU≫

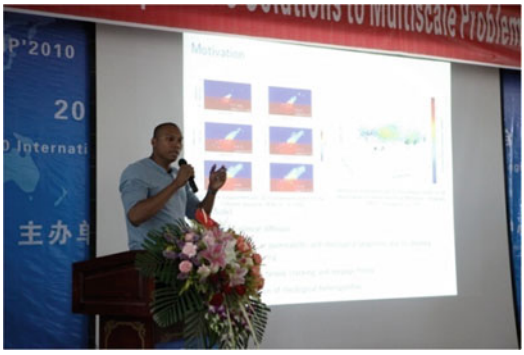
Mr. Liang Zheng, University of Chinese Academy of Sciences, China ≪GPU Applications to Multigrid Solver for Stokes Equations≫

Mr. Robin Weiss, Macalester College, USA ≪GPU-Accelerated Swarm Intelligence Algorithms for Data Mining≫

Ms. Zhoujun Liu, IACAS, China ≪Particle-based Simulation of sand-ripples and Interaction with Obstacle using GPU technique≫

Ms. Xiaohua Wan, ICTCAS, China ≪Three-dimensional reconstruction of electron tomography using graphic processing units (GPUs)≫

Mr. Weile Jia, SCCAS, CNIC, China ≪N-body simulation on GPU cluster≫



Mr. Boris Galvan, Bonn University, German ≪GPU simulation of evolving fracture networks in a poro-elastoplastic medium with pressure-dependent permeability≫

Posters

4. July 29th: Sightseeing

On the last day we went touring at three sites around greater Harbin, which included the tiger zoo, Russian Church and had Natural park on Sun Island by the river.

In all, the participants enjoyed a convivial atmosphere in which many ideas were exchanged and new friendships were forged.

Sun Island adjacent to the river

Siberian Tiger Artificial Propagation Center

Central Avenue in downtown Harbin

Holy Sophia Cathedral

Part II
Hardware and Installations

Chapter 3
Efficiency, Energy Efficiency and Programming of Accelerated HPC Servers: Highlights of PRACE Studies

Lennart Johnsson

Abstract During the last few years the convergence in architecture for High-Performance Computing systems that took place for over a decade has been replaced by a divergence. The divergence is driven by the quest for performance, cost-performance and in the last few years also energy consumption that during the life-time of a system have come to exceed the HPC system cost in many cases. Mass market, specialized processors, such as the Cell Broadband Engine (CBE) and Graphics Processors, have received particular attention, the latter especially after hardware support for double-precision floating-point arithmetic was introduced about three years ago. The recent support of Error Correcting Code (ECC) for memory and significantly enhanced performance for double-precision arithmetic in the current generation of Graphic Processing Units (GPUs) have further solidified the interest in GPUs for HPC. In order to assess the issues involved in potentially deploying clusters with nodes consisting of commodity microprocessors with some type of specialized processor for enhanced performance or enhanced energy efficiency or both for science and engineering workloads, PRACE, the Partnership for Advanced Computing in Europe, undertook a study that included three types of accelerators, the CBE, GPUs and ClearSpeed, and tools for their programming. The study focused on assessing performance, efficiency, power efficiency for double-precision arithmetic and programmer productivity. Four kernels, matrix multiplication, sparse matrix-vector multiplication, FFT, random number generation were used for the assessment together with High-Performance Linpack (HPL) and a few application codes. We report here on the results from the kernels and HPL for GPU and ClearSpeed accelerated systems. The GPU performed surprisingly significantly better than the CPU on the sparse matrix-vector multiplication on which the ClearSpeed performed surprisingly poorly. For matrix-multiplication, HPL and FFT the ClearSpeed accelerator was by far the most energy efficient device.

L. Johnsson (✉)
Department of Computer Science, University of Houston, Houston, TX, USA

L. Johnsson
School of Computer Science and Communications, KTH, Stockholm, Sweden

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_3,

3.1 Introduction

3.1.1 Architecture and Performance Evolution

High-Performance Computing (HPC) has traditionally driven high innovation in both computer architecture and algorithms. Like many other areas of computing it has also challenged established approaches to software design and development. Many innovations have been responses to opportunities offered by the exponential improvements of capabilities of silicon based technologies, as predicted by "Moore's Law" (Moore 1965), and constraints imposed by the technology as well as packaging constraints. Taking full advantage of computer system capabilities require architecture aware algorithm and software design, and, of course, problems for which algorithms can be found that can take advantage of the architecture at hand. Conversely, architectures have historically been targeted for certain workloads. In the early days of electronic computers, even at the time transistor technologies replaced vacuum tubes in computer systems, scientific and engineering applications were predominantly based on highly structured decomposition of physical domains and algorithms based on local approximations of continuous operators. Global solutions were achieved through a mixture of local or global steps depending on algorithm selected (e.g., explicit vs. implicit methods, integral vs. differential methods). In most cases methods allowed computations to be organized into similar operations on large parts of the domains and data accessed in a highly regular fashion. This fact was exploited by vector architectures, such as the very successful Cray-1 (Cray-1 Computer System 1976), and highly parallel designs such as the Illiac IV (1976) (ILLIAC IV 1972, 2011a,b; Thelen 2005), the Goodyear MPP (Massively Parallel Processor) (1983) (Goodyear MPP 2011) with 16,896 processors, the Connection Machine (Cray-1 Computer System 1976; Hills 1989; Thelen 2003) CM-1 (1986) with 65,536 processors and the CM-2 (1987) with 2048 floating-point accelerators, These machines all were of the SIMD (Single Instruction Multiple Data) (Cell 2011), data-parallel, or vector type, thus amortizing instruction fetch and decode over several, preferably large number of operands. The memory systems were designed for high bandwidth, which in the case of the Cray-1 (Cray-1 Computer System 1976) and the Control Data Corp. 6600 (Thornton 1963, 1970) was achieved by optimizing it for access of streams of data (long vectors), and in the case of MPPs through very wide memory systems. The parallel machines with large numbers of processors had very simple processors, indeed only 1-bit processors. (It is interesting to note that the data parallel programming model is the basis for Intel's recently developed Ct technology (Ghuloum et al. 2007a,b) and was also the basis for RapidMind (RapidMind 2011) acquired by Intel in 2009).

The emergence of the microprocessor with a complete CPU on a single chip (Single Chip 4-Bit P-Channel Microprocessor 1987; Intel 4004 2011a,b; Kanellos 2001) targeted for a broad market and produced in very high volumes offered large cost advantages over high-performance computers designed for the scientific and engineering market and led to a convergence in architectures also for scientific

Table 3.1 June 1993 Top 500 list by process architecture (Top500 2011)

Processor architecture	Count	Share (%)	R_{max} sum (GF)	R_{peak} sum (GF)	Processor sum
Vector	334	66.80	650	792	1,242
Scalar	131	26.20	408	859	15,606
SIMD	35	7.00	64	135	54,272
Totals	500	100	1,122.84	1,786.21	71,120

computation. According to the first Top500 (Top500 2011) list from June 1993, 369 out of 500 systems (73.8 %) were either "Vector" or "SIMD", while by November 2010 only one Vector system appears on the list, and no SIMD system. Since vector and SIMD architectures were specifically targeting scientific and engineering applications whereas microprocessors were, and still are, designed for a broad market, it is interesting to understand the efficiencies, measured as fraction of peak performance, achieved for scientific and engineering applications on the two types of platforms. The most readily available data on efficiencies, but not necessarily the most relevant, is the performance measures reported on the Top500 lists based on High-Performance Linpack (HPL) (Petitet et al. 2008) that solves dense linear systems of equations by Gaussian elimination. The computations are highly structured and good algorithms exhibit a high degree of locality of reference. For this benchmark, the average floating-point rate as a fraction of peak for all vector systems was 82 % in 1993, Table 3.1, with the single vector system on the 2010 list having an efficiency of over 93 %, Table 3.2. The average HPL efficiency in 1993 for "Scalar" systems was 47.5 %, but improved significantly to 67.5 % in 2010. The microprocessors, being targeted for a broad market with applications that do not exhibit much potential for "vectorization", focused on cache based architectures enabling applications with high locality in space and time to achieve good efficiency, despite weak memory systems compared to the traditional vector architectures. Thus, it is not all that surprising that microprocessor based systems compare relatively well in case of the HPL benchmark. The enhanced efficiency over time for microprocessor based systems is in part due to increased on-chip memory in the form of three levels of cache in current microprocessors, and many added features to improve performance, such as, e.g., pipelining, pre-fetching and out-of-order execution that add complexity and power consumption of the CPU, and improved processor interconnection technologies. Compiler technology has also evolved to make more efficient use of cache based architectures for many application codes.

Table 3.2 November 2010 Top500 list by processor architecture (Top500 2011)

Processor architecture	Count	Share (%)	R_{max} sum (GF)	R_{peak} sum (GF)	Processor sum
Vector	1	0.20	122,400	131,072	1,280
Scalar	497	99.40	43,477,293	64,375,959	6,459,463
N/A	2	0.40	73,400	148280	11,584
Totals	500	100	43,673,092.54	64,655,310.70	6,472,327

The scientific and engineering market also had a need for good visualization of simulated complex physical phenomena, or visualization of large complex data sets as occurring for instance in petroleum exploration. Specialized processor designs, like the LDS-1 from Evans & Sutherland (Evans and Sutherland 2011a,b) that initially targeted training simulators, evolved to also cover the emerging digital cinema market as well as engineering and scientific applications. As in the case of standard processors, semiconductor technology evolved to a point where much of the performance critical processing could be integrated on a single chip, such as the Geometry Engine (Clark 1980, 1982) by Jim Clark who founded Silicon Graphics Inc (Silicon Graphics 2011) that came to dominate the graphics market until complete Graphics Processing Units (GPUs) could be integrated onto a single chip (1999) (GeForce 256 2011; Graphics Processing Unit 2011) at which time the cost had become sufficiently low that the evolution became largely driven by graphics for gaming with 432 million such units shipped in 2010 (Jon Peddie Research 2011) (compared to about 350 million PCs (Pettey 2011) and 9 million servers (Pettey and Stevens 2011) according to the Gartner group). Thus, since in the server market two socket servers are most common, but four and even 8-socket servers are available as well, the volumes of discrete GPUs (as opposed to GPUs integrated with CPUs, e.g. for the mobile market) and CPUs for PCs and servers are almost identical. Today, GPUs are as much of a mass market product as microprocessors are, and prices are comparable (from about a hundred dollars to about two thousand dollars depending on features).

With their design target having been efficient processing for computer graphics GPUs lend themselves to vector/stream processing. As in the case of the vector machines for scientific and engineering applications GPUs are optimized for applying the same operation to large amounts of (structured) data and have memory systems that support high execution rates. Over time GPUs have enhanced their floating-point arithmetic performance significantly and since 2008 also incorporated hardware support for double-precision floating-point operations and moved towards support of the IEEE floating-point standard. Double-precision floating-point performance and compliance with the IEEE floating-point standard are critical for many scientific and engineering applications. The evolution of GPU floating-point performance since 2002 is shown in Fig. 3.1 (Introduction to Parallel GPU Computing 2010).

As seen in Fig. 3.1, in 2003 the GPU single-precision floating-point performance was only modestly higher than that of common IA-32 (IA-32 2011) microprocessors by, e.g., AMD and Intel, and there was no hardware support for double-precision floating-point arithmetic, so many application developers in science and engineering did not find the benefits of porting codes to GPUs sufficiently large to warrant the effort to do so. However, as is also apparent from the figure, the performance trajectories for GPUs have been quite different from those of CPUs, so that today a GPU may have 10–30 times higher single-precision performance than a CPU, with the AMD/ATI Radeon HD5870 (ATI Radeo HD 5870 Graphics 2011; Comparison of AMD Graphics Processing Units 2011) having a peak single-precision performance of 2.7 TF (10^{12} flops/s (floating-point operations per second)). Moreover, today GPUs not only support double-precision arithmetic, but the performance advantage compared to a CPU may be a factor of five or more.

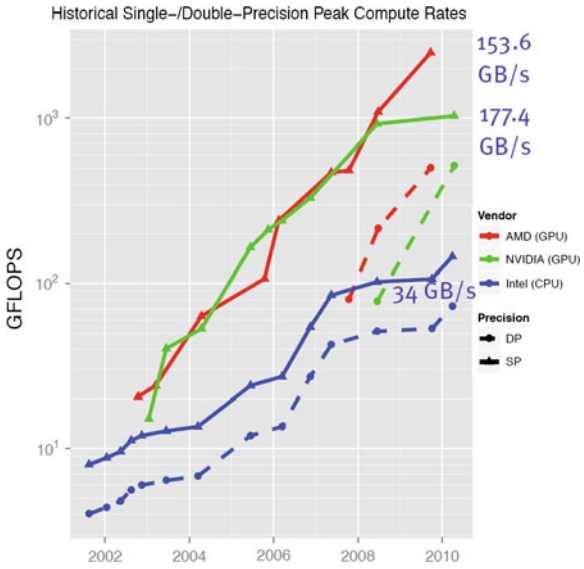


Fig. 3.1 Performance growth of GPUs and CPUs 2002–2010 (Introduction to Parallel GPU Computing 2010)

Good application performance also requires high memory bandwidth. Today, the memory bandwidth for high-end GPUs is about 150 GB/s (ATI Radeo HD 5870 Graphics 2011; Comparison of AMD Graphics Processing Units 2011; Comparison of Nvidia Graphics Processing Units 2011; Tesla C2050/C2070 GPU Computing Processor 2010), which compares very favorably with that of IA-32 microprocessors by AMD and Intel that today has a memory bandwidth of 25 to 30+ GB/s. (The Intel Westmere-EP 6-core CPU has three memory channels each with a peak data rate of 10.8 GB/s (32.4 GB/s total with DDR3 1.333 GHz DIMMs (Intel 56XX 2011), whereas the AMD Magny-Cours 8- and 12-core CPUs have a peak memory data rate of 28.8 GB/s across four channels for DDR3 1.333 GHz DIMMs due to limitations in the North Bridge (Gelas 2010). Observed Stream (McCalpin 2011) benchmark numbers are 20.5 GB/s (HP Challenge Benchmark Record 2011) and 17.9 GB/s (Gelas 2010) for the Intel Westmere-EP CPU and 27.5 GB/s (Memory Bandwidth (STREAM)—Two-Socket Servers 2011), 24.7 GB/s (Gelas 2010) and 19.4 GB/s (HPC Challenge Benchmark Record 2011) for the AMD Magny-Cours CPU (on a per CPU basis).

Thus, today GPUs offer about five times the memory bandwidth and about a factor of five higher peak double-precision floating-point performance than IA-32 microprocessors, and the cost is comparable. For instance, nVidia's Tesla C2050 lists for about $2,500, and the ATI FirePro 3D V9800 is priced similarly, compared to a list price of $1,663 for the top-of-the line Intel Westmere-EP CPU (3.46 GHz, 6-cores, 12 MB L3 cache) (Intel 56XX 2011) whereas the top-of-the line AMD Magny-Cours CPU has a list price of $1,514 (2.5 GHz, 12-cores, 12GB L3 cache)

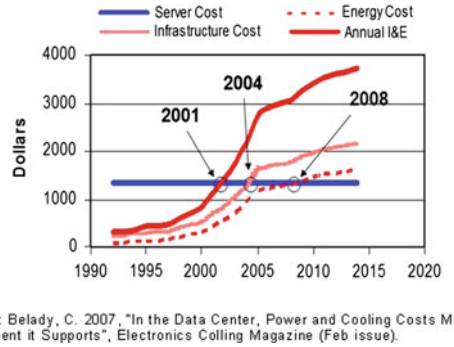


Fig. 3.2 Evolution of US power and cooling costs for a standard IA-32 server (Belady 2007)

(Ghuloum et al. 2007a). The lowest costs versions of CPUs may cost as little as 20 % of the top-of-the line CPUs, comparable to the GPUs targeted for the low end consumer market.

3.1.2 Energy Efficiency

Performance and cost-performance are the traditional measures affecting choice of technology and platforms for high-performance scientific and engineering applications. In recent years energy efficiency in computation has become another important and sometimes deciding factor in the choice of platform. Since a few years ago the life-time energy cost including cooling of servers has exceeded the cost of the server itself, Fig. 3.2 (Belady 2007).

For microprocessors a large contribution to the performance gain from one generation to the next was increased clock frequency, until about a decade ago. The first microprocessor, the Intel 4004 (Single Chip 4-Bit P-Channel Microprocessor 1987; Intel 4004 2011a,b; Kanellos 2001) introduced in 1971 had a clock frequency of 0.74 MHz. By the end of 2002, Intel introduced a Pentium 4 clocked at 3.06 GHz using its Northwood core (Pentium 4 2011). The clock frequency was further increased to 3.4 GHz in a version available in early 2004 and further to 3.8 GHz in the Prescott core introduced later that year. (The 3.8 GHz Prescott Pentium 4 is the highest clock frequency ever used in an Intel CPU.) Thus, over a period of about 30 years clock frequencies for Intel microprocessors increased by a factor of about 5,000, followed by a slight decline since its peak in 2004, Fig. 3.3. The evolution is similar for CPUs from AMD, though traditionally AMD CPUs have operated at somewhat lower clock rates, as shown in Fig. 3.4, and lower power consumption.

The reason for the apparent limit on clock frequency is that, for CMOS technology, the dominating technology for microprocessors, the dynamic switching power P depends on voltage and clock frequency as $P \propto cV^2f$. This relationship is due

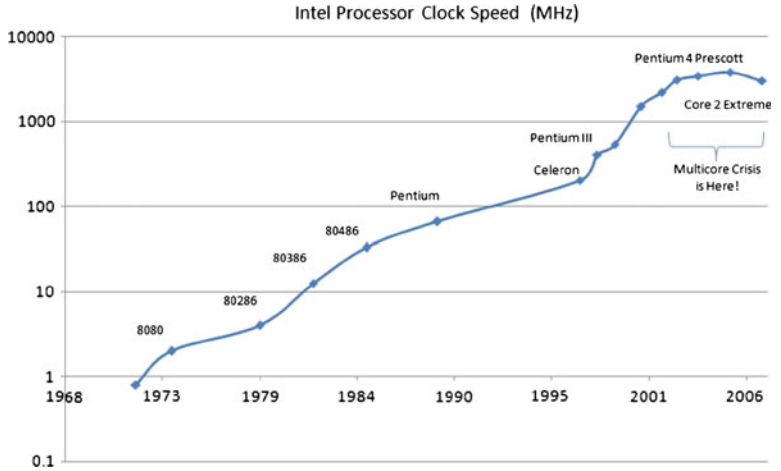


Fig. 3.3 Intel CPU clock rates 1971–2007 (Intel Processor 2011)

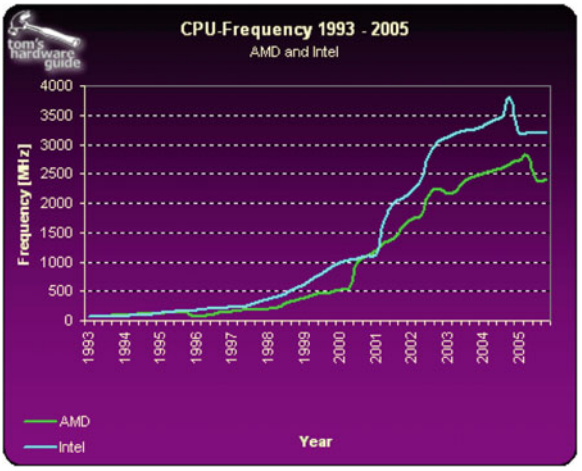


Fig. 3.4 AMD and Intel CPU clock rates, 1993–2005 (Team TsG 2005)

to the fact that CMOS is a charge transfer technology in which charges on gates of transistors effectively acting as capacitors are drained and restored in switching transistors on or off. The energy stored on a capacitor (gate) is $\propto cV^2$. Furthermore, for CMOS the clock frequency $f \propto V$. Hence, the power dissipation increases very rapidly with the clock frequency. In fact, even though V typically has been reduced form one chip generation to the next, the power density for Intel CPUs doubled for each generation as shown in Fig. 3.5. The evolution of the power consumption for AMD CPUs (Team TsG 2005) has been similar, Fig. 3.6. In 1999 Fred Pollack of Intel stated in his keynote at Micro 32 that "We are on the Wrong side of a Square

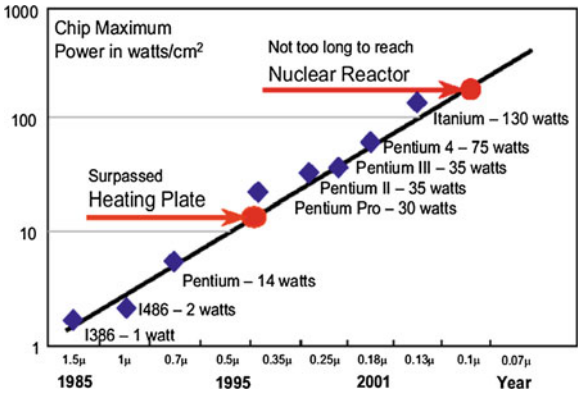


Fig. 3.5 Heat density of Intel CPUs, *source* Shekhar Borkar, Intel

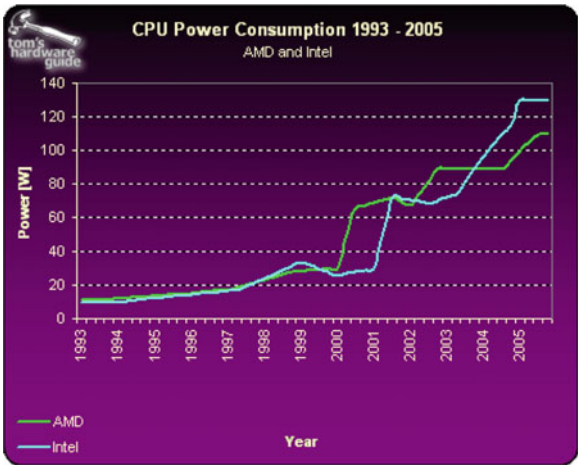


Fig. 3.6 Comparison of the power consumption of AMD and Intel IA-32 CPUs (Team TsG 2005)

Law" (Pollack 1999) and concluded with a new goal for CPU design: "Double *Valued Performance* every 18 months, at the same power level", something that the industry has largely adhered to since almost a decade ago.

The energy per instruction for a range of Intel CPUs (Grochowski and Annavaram 2006) is shown in Table 3.3. The approach taken to achieve "Double valued performance every 18 months, at the same power level" has been to introduce multi-core CPUs exploiting reduced feature sizes in CPU manufacturing, and slightly reducing the maximum clock frequencies. This approach has enabled "double valued performance" to continue for applications that can take advantage of parallelism, but at a cost in application porting and development, and a challenge for compiler developers. High parallelism is becoming main stream, not only by increased core count per chip, but also by increased number of operations a core can perform in a single

Table 3.3 Energy per instruction for Intel CPUs (Grochowski and Annavaram 2006)

Product	Normalized performance	Normalized power	EPI on 65 nm at 1.33 V (nJ)
i486	1.0	1.0	10
Pentium	2.0	2.7	14
Pentium Pro	3.6	9	24
Pentium 4 (Willamette)	6.0	23	38
Pentium 4 (Cedarmill)	7.9	38	48
Pentium M (Dothan)	5.4	7	15
Core Duo (Yonah)	7.7	8	11

clock cycle, from one floating-point operation per cycle about a decade ago for IA-32 designs to currently four and in the next generation eight, resulting in a capability to currently carry out 48 floating-point operations per cycle in the case of the AMD 12-core chip.

The power consumption of CMOS processors, as mentioned above, raises steeply with the clock frequency, and of course the number of transistors. The most recent IA-32 CPU by Intel, the 6-core Westmere-EP CPU, (3.46 GHz, 1.17 billion transistors, 240 mm^2 in 32 nm technology) (Shimpi 2010) and by AMD, the 8- and 12-core Magny-Cours CPU (2.5 GHz, 2 billion transistors, 692 mm^2 in 45 nm technology) (Gelas 2010) both dissipates up to 130–140 W in their highest clock rate versions, while the current generation GPUs from AMD/ATI (0.825 GHz, 2.15 billion transistors, 334 mm^2 in 40nm technology) (Comparison of AMD Graphics Processing Units 2011; Bell 2009) and nVidia (0.575 GHz, 3 billion transistors, 550 mm^2 also in 40 nm technology) (Comparison of Nvidia Graphics Processing Units 2011; Valich 2010) both have a maximum power rating of 225 W. But, since the GPUs have a peak double-precision performance about five times higher than that of the IA-32 CPUs, the GPUs still may deliver higher energy efficiency for applications. We summarize this information in Table 3.4.

Estimates of the peak double-precision floating-point rate per W at the chip level is shown in Table 3.5 (Johnsson 2011) for a few processors. The table shows an advantage by a factor of 2.5 to about 4 of GPUs over CPUs. Thus, GPUs in addition

Table 3.4 Some chip characteristics for CPU and GPU processors

	nm	Trans. (Billions)	Die	Cores (mm^2)	Memory BW (GB/s)	I/O BW (GB/s)	GF DP	W
Nehalem-EP	45	0.731	263	4	3×10.8	2×25.6	53.3	130
Westmere-EP	32	1.17	240	6	3×10.8	2×25.6	83.0	130
AMD Magny-Cour	45	2	692	12	4×10.8[a]	4×25.6	120.0	137
Tesla C1060	65	1.4	576	240	102.4	8	77.8	188
Tesla C2050	40	3	550	448	144	8	515.2	225
ATI HD5870	40	2.15	334	1600	153.6	8	544	225

[a]Limited to 28.8 GB/s by the Northbridge

Table 3.5 Estimates of theoretical performance per W for some processor alternatives (Johnsson 2011)

ARM Coretx-9			ATOM			AMD 12-core			Intel 6-core			ATI 9370		
Cores	W	GF/W	Cores	W	GF/W	Cores	W	GF/W	Cores	W	GF/W	Cores	W	GF/W
4	∼2	∼0.5	2	2+	∼0.5	12	115	∼0.9	6	130	∼0.6	1600	225	∼2.3
nVidia				TMS320C6678				IBM				ClearSpeed		
Fermi								BQC				CX700		
Cores	W	GF/W		Cores	W	GF/W		Cores	W	GF/W		Cores	W	GF/W
512	225	∼2.3		8	10	∼4		16	∼50	∼4		192	10	∼10

to offering potentially higher performance and lower cost-performance in regards to hardware cost, GPUs also have the potential to offer a further cost advantage by being more energy efficient and more environmentally friendly despite their higher power rating.

The potential for higher energy efficiency than that of IA-32 CPUs is indeed real as demonstrated by measurements for HPL. The Green500 list ranks systems on the Top500 list based on their HPL energy efficiency. On the November 2010 list eight of the ten most energy efficient systems use some form of accelerator, with five using GPUs and three using the Cell Broadband Engine (CBE) (Cell 2011; Cell Project at IBM Research 2001; Chen et al. 2005). Systems using GPUs ranked 2nd, 3rd, 8th, 9th, and 10th. The IBM Blue Gene/Q to be delivered late in 2011 or 2012 ranked 1st with an energy efficiency of 1,684 MF/W. Compared to the Blue Gene/P, its predecessor, the BG/Q has double the execution width of each core, and twice the number of cores per node. Few details are available at this time. The most energy efficient GPU accelerated system achieved an efficiency of 958 MF/W, while the most energy efficient system using the CBE for acceleration achieved 773 MF/W (Homberg 2009). This system used an experimental interconnection network connecting nodes via the CBE internal high-speed bus. Non-accelerated systems using the latest generation IA-32 CPUs achieved an energy efficiency of about 350–400 MF/W for HPL (Fig. 3.7).

3.1.3 GPU Integration and Programming

Programming and code generation for both CPUs and GPUs today requires effective exploitation of parallelism for high efficiency. IA-32 CPUs support common programming languages, such as C, C++, Fortran, etc with a choice of mature compilers

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	1684.20	IBM Thomas J. Watson Research Center	NNSA/SC Blue Gene/Q Prototype	38.80
2	958.35	GSIC Center, Tokyo Institute of Technology	HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows	1243.80
3	933.06	NCSA	Hybrid Cluster Core i3 2.93Ghz Dual Core, NVIDIA C2050, Infiniband	36.00
4	828.67	RIKEN Advanced Institute for Computational Science	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect	57.96
5	773.38	Forschungszentrum Juelich (FZJ)	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	57.54
5	773.38	Universitaet Regensburg	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	57.54
5	773.38	Universitaet Wuppertal	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	57.54
8	740.78	Universitaet Frankfurt	Supermicro Cluster, QC Opteron 2.1 GHz, ATI Radeon GPU, Infiniband	385.00
9	677.12	Georgia Institute of Technology	HP ProLiant SL390s G7 Xeon 6C X5660 2.8Ghz, nVidia Fermi, Infiniband QDR	94.40
10	636.36	National Institute for Environmental Studies	GOSAT Research Computation Facility, nvidia	117.15

Fig. 3.7 The 10 most energy efficient system on the November 2010 Top 500 list (Intel 4004 2011b)

that generate efficient code. GPUs on the other hand with a quite different memory architecture and different instruction set have traditionally required specialized and sometimes proprietary languages and compilers. This fact, and the lack of architectural support for many operations commonly used in science and engineering applications have been a limiting factor on their wide-spread adoption. However, the hardware support for general purpose use of GPUs is improving rapidly, thus lowering the barrier towards wide adoption. The good double-precision arithmetic performance and support for IEEE arithmetic are also important factors in today's strong interest in GPUs. However, GPUs are not stand-alone processors and requires a host, which typically for HPC applications is a common microprocessor. GPUs are "add-on" units typically integrated into the system using the I/O bus of the CPU. This bus can be a performance bottleneck in many cases since data needs to move between the CPU memory and the smaller but faster GPU memory for many applications. As GPUs become integrated onto CPU chips this bottleneck will disappear, but at least initially the GPUs integrated with CPUs on the same chip will not have their own high bandwidth memory system one of the key advantages of today's GPUs. In future generation CPUs the role of GPUs or stream processors may very well change for the scientific and engineering market and stream or vector architectures taking on the primary role, as in the case of Intel's Many Integrated Core (MIC) CPUs (Introducing Intel many Integrated Core Architecture 2011).

To alleviate some of the programming issues associated with having to produce code for both CPUs and GPUs in a heterogeneous node the Open Computing Language (OpenCL 2011), OpenCL, was conceived with version 1.0 published in December 2008 and version 1.1 in September 2010 (OpenCL 2010). OpenCL has been developed by the Khronos Group that also developed OpenGL. Because of the potential benefits of being an Open Standard, OpenCL was included in the assessment despite the fact that only prerelease compilers were available.

3.1.4 Concurrency Comparison Between CPUs and GPUs

On-chip parallelism is increasing rapidly for both CPUs and GPUs. The current generation CPUs can carry out up to about 50 double-precision floating-point operations concurrently (48 for the AMD 12 core Magny-Cours CPUs) whereas GPUs can carry out in the order of 500–600 double-precision floating-point operations concurrently (640 for the AMD/ATI HD5870 and FirePro 3D V9800GPUs). Though the concurrency for GPUs is about 10 times higher than for CPUs, the peak performance difference is smaller because the GPUs operate at lower clock frequency (e.g. max 2.5 GHz for the AMD 12-core CPU versus max 0.825 GHz for the AMD/ATI GPU). As silicon technologies evolve to allow for smaller feature sizes enabling more transistors to be put on the same die, chip designers so far has used the increased capability for additional cores, increased on-chip memory, and less often for execution units of increased width. However, for CPUs the next generation from both AMD and Intel will double the width of the execution units as well as increase the number of

cores, thus significantly increasing the peak capabilities, and bringing the parallelism required for peak performance of a IA-32 chip to a level of 100 operations or more. Over about a decade the number of floating-point operations per cycle per core will have increased from one to eight. Hence, though there will be a difference in the degree of parallelism to be expressed and managed, both CPUs and GPUs will have comparable challenges in regards to concurrency. In regards to the viability of GPUs for "general purpose" scientific and engineering computations Shalf et al. at LBNL (John et al. 2006) made the interesting observation that only 80 instructions out of the close to 300 instructions on IA-32 platforms were used across a broad range of codes.

3.2 Highlights of a PRACE Study of Accelerated IA-32 Servers

3.2.1 Background

The potential performance, cost/performance and energy efficiency advantages of GPUs are significant, but the programming, and in particular the code porting challenges, are also quite significant. In order to assess the benefits and the code porting challenges PRACE, the Partnership for Advanced Computing in Europe (PRACE 2009), undertook an evaluation of GPU accelerated servers during the second half of 2008 and 2009. The evaluation was made from a data center perspective, i.e., the perspective that codes to be run on a GPU accelerated system could largely only be ported with modest effort using tools targeting heterogeneous node architectures, and not be completely rewritten or hand optimized. Furthermore, the focus was on double-precision arithmetic performance since the intent was to evaluate the merits of GPU accelerated nodes across "all" codes used at partner centers. The tools evaluated were HMPP (Hybrid Multi-core Parallel Programming) (HMPP Open Standard 2011; Hybrid Multi-Core Parallel Programming Workbench 2011) from CAPS (Bell 2009), RapidMind (RapidMind 2011; Writing Applications for the GPU Using the RapidMind™ Development Platform 2006) and to a lesser degree the Portland Group Inc's (PGIs) Accelerator Compilers (PGI Accelerator Programming Model for Fortran and C 2010; Portland Group Inc 2011) because the PGI products were not available at the time this evaluation started, and OpenCL, as already mentioned. For the GPU test systems the results were compared with nVidia's CUDA (CUDA 2011; NVIDIA Corporation 2011) whenever possible. In addition to nVidia C1060 accelerated servers, ClearSpeed (ClearSpeed 2011) CSX700 (CSX700 Processor 2011) accelerated systems were also assessed, as were systems with CBEs. However, since IBM has decided not to continue with the CBE we do not include results related to it.

The reference platform for the evaluations was a dual socket server equipped with Intel Nehalem 2.53 GHz quad-core CPUs and 3GB DDR3 memory per core. The theoretical peak performance per core of this reference platform thus was 10.12

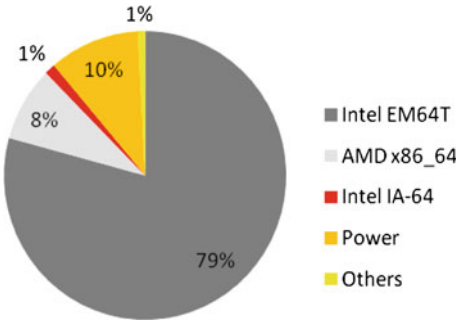


Fig. 3.8 November 2009 Top500 (Cray-1 Computer System 1976) processor family statistics

GF/s. The choice of the Nehalem CPU for the reference platform was motivated by the dominance of Intel EM64T on the November 2009 Top500 (Top500 2011) list on which this processor family accounted for 79 % of the CPUs, see Fig. 3.8, and the Nehalem CPU being the most recent EM64T CPU from Intel at the time of this evaluation.

GPU evaluations were made on dual socket, quad-core 2.8 GHz Intel Harpertown servers with two nVidia Tesla servers for each node and two C1060 cards for each Tesla server. The Tesla servers were connected to the hosts over PCI Express Gen2 16x (8 GB/s) for each node. The C1060 has 30 stream processors each with eight single-precision (SP) Floating-Point Units (FPUs) and one double-precision (DP) FPU. The peak SP performance is 624 GF and the peak DP performance is 78 GF.

ClearSpeed results were obtained from two platforms; (1) dual socket 2.53 GHz Intel Nehalem servers with 4 GB/core with a ClearSpeed-Petapath e710 unit for each server connected via PCI express Gen2 16x (PRACE 2011; Sagar et al. 2010)., (2) dual socket 2.67 GHz Nehalem servers with 3 GB/core and ClearSpeed-Petapath e740 and e780 units, one per CPU socket, connected via PCI express Gen 2 16× (PRACE 2011; Sagar et al. 2010). The ClearSpeed-Petapath units use 1, 4 or 8 ClearSpeed CSX700 units, each with a peak double-precision arithmetic performance of 96 GF. A ClearSpeed CSX700 is in turn made up of two Multi-Threaded Array Processors (MTAPs) (CSX700 Datasheet 2011), each with a peak performance of 48 GF, double-precision.

The benchmarks used for the evaluations were a few kernels common in scientific and engineering applications: dense matrix multiplication, solution of dense systems of linear equations (HPL), sparse matrix-vector multiplication, FFT and random number generation. This selection was based on a study of application codes used at PRACE partner sites (Alan et al. 2008). These kernels also represent a subset of Phil Colella's well known "Seven Dwarf's" (Colella 2004) described in Asanovic et al. (2006). The benchmark software used for these functions was EuroBen (EuroBen Benchmark 2011), except for the linear system solution for which High-Performance Linpack (HPL) (Petitet et al. 2008) was used. The EuroBen routines used were

- mod2am for dense matrix-matrix multiplication $C = A \times B$

- mod2as for sparse matrix-vector multiplication c = A × b with the matrix in Compressed Sparse Row (CSR) format
- mod2f for 1-D complex-to-complex Fast Fourier Transform using a radix-4 algorithm
- mod2h for random number generation.
- All benchmarks were based on C codes.

3.2.2 Results for the Reference Platform

For the reference platform we report both single core and eight core results. The memory system supports a single core well, but not fully all four cores on a CPU for memory intensive applications. Furthermore, a node has NUMA (Non-Uniform Memory Access) (Non-Uniform Memory Access 2011) characteristics in that in a node each CPU with four cores has its own memory not directly accessible by the cores on the other CPU in a two socket system.

3.2.2.1 Single Core Results

Matrix Multiplication

The single core dense matrix multiplication using mod2am calling Intel's Math Kernel Library (MKL) (Intel Math Kernel Library 2011) is shown in Fig. 3.9. The peak achieved performance is 9.387 GF, 92.8 % of peak (Sagar et al. 2010).

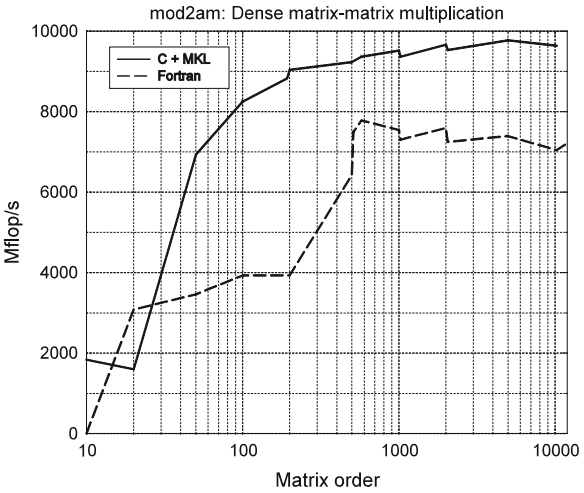


Fig. 3.9 Mod2am results on a single Nehalem 2.53 GHz core (Sagar et al. 2010)

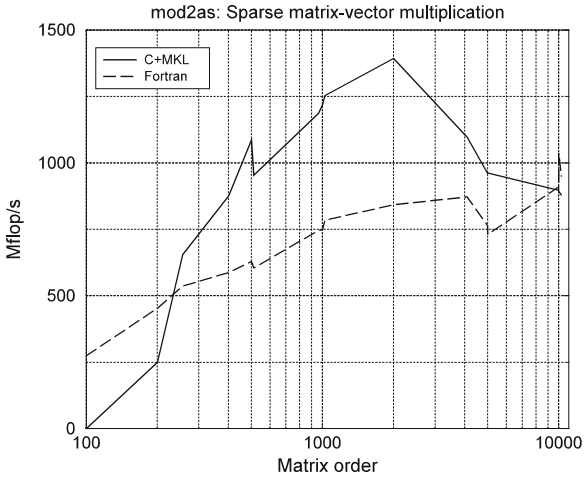


Fig. 3.10 Mod2as results on a single Nehalem 2.53 GHz core (Sagar et al. 2010)

Sparse Matrix-Vector Multiplication

The single core sparse matrix-vector results (Sagar et al. 2010) are shown on Fig. 3.10. As expected the performance is much lower. Sparse matrix-vector multiplication using compressed formats has a relatively low number of floating-point operations compared to integer operations for address calculations and, for randomly generated sparse matrices, a random memory access pattern that tend to result in poor cache behavior. The peak observed performance is about 13.6 % of theoretical peak (10.12 GF). Due to the randomness of the matrix sparsity the performance as a function of matrix size does not follow a smooth progression unlike the case for dense matrix multiplication. The sparse matrix was filled to 15 % in all cases.

FFT

The single core FFT results (Sagar et al. 2010) are shown in Fig. 3.11. The peak achieved performance was 2.778 GF, 27.5 % of peak. Unlike matrix multiplication and matrix-vector multiplication complex-to-complex FFT computations do not have a balanced number of additions and multiplications. Thus, for this type of FFT the peak core performance of 10.12 GF is never attainable. Complex multiplication requires 4 real multiplications and 2 real additions. A radix-4 computation requires 3 complex multiplications and 4 complex additions/subtractions. In a straightforward organization of the complex operations the complex multiplication results at best in 6 arithmetic operations out of 8 potential hardware arithmetic operations, i.e. 75 % utilization, and a complex addition results in 2 out of four potential operations, or 50 % utilization. FFTs also have a somewhat complex memory reference patterns using strided access with different strides for different phases of the

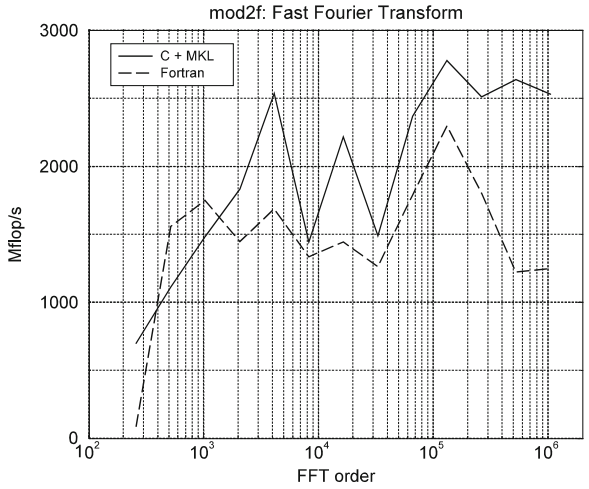


Fig. 3.11 Mod2f radix-4 complex-to-complex 1-D FFT on a single Nehalem 2.53 GHz core (Sagar et al. 2010)

algorithm. The strided access can result in poor cache behavior. In Ali et al. (2007) and Mirkovic et al. (2000), a performance difference by more than a factor of 10 was observed for different strides for a few different processors.

Random Number Generation

The single core random number results (Sagar et al. 2010) are shown in Fig. 3.12. Since the random number generator use very few floating-point operations the

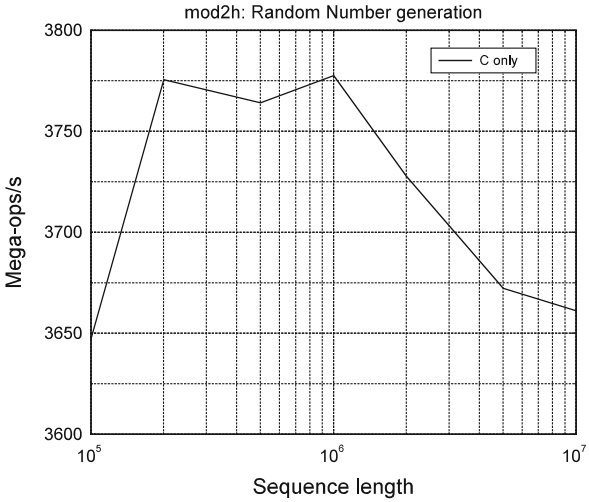


Fig. 3.12 Mod2h random number generation results on a single Nehalem 2.53 GHz core (Sagar et al. 2010)

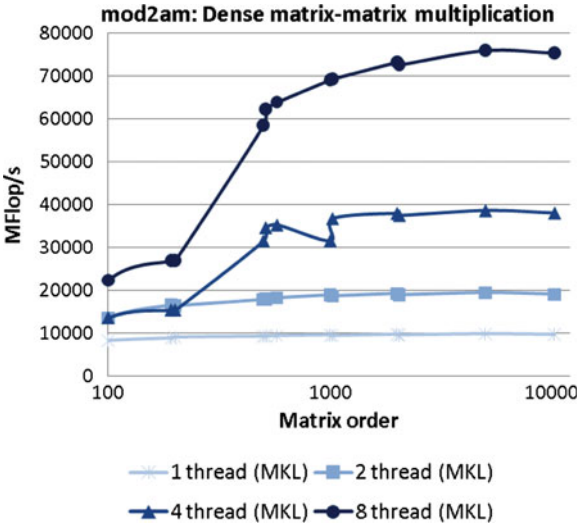


Fig. 3.13 Mod2am results on a dual socket, 8-core Intel Nehalem 2.53 GHz node with 24 GB memory (Sagar et al. 2010)

performance is measured in operations/s. The MKL library does not include a random number generator so results are reported for a C code.

3.2.2.2 Node Results

The reference node has two sockets each with a quad-core 2.53 GHz Intel Nehalem CPU. Thus, eight threads can be run concurrently on the reference platform, 16 with hyper-threading (Intel HT Technology 2011) with two threads per core. In our tests we did not enable hyper-threading since it is known to reduce performance in compute intensive cases. Results for 1, 2, 4 and 8 threads are shown in Figs. 3.13, 3.14, 3.15 and 3.16. The MKL version used for the benchmarks supported multi-threading for dense matrix-matrix and sparse matrix-vector multiplication, but not for the FFT. Thus, for the FFT MPI was used to in effect create multiple threads on a reference node. However, at this time MKL does have multi-threaded FFT support (Petrov and Fedorov 2010). For the random number generator multiple instances were run since neither an MPI nor an OpenMP version did exist, and was not developed.

Matrix Multiplication

The peak matrix multiplication performance achieved on eight cores using the MKL was 76 GF, which is 93.9 % of theoretical peak.

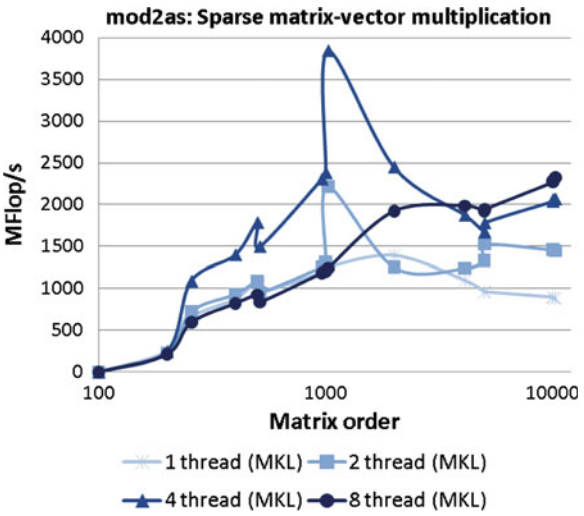


Fig. 3.14 Mod2as results on a dual socket, 8-core Intel Nehalem 2.53 GHz node with 24 GB memory (Sagar et al. 2010)

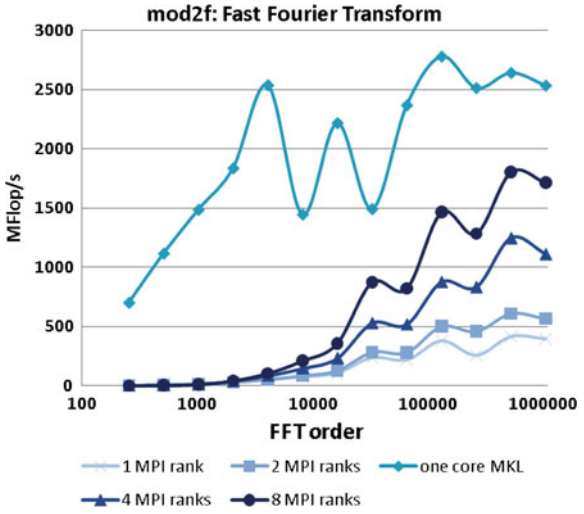


Fig. 3.15 Mod2f results for on a dual socket, 8-core Intel Nehalem 2.53 GHz node with 24 GB memory (Sagar et al. 2010)

Sparse Matrix-Vector Multiplication

For sparse matrix-vector multiplication the performance is highly variable as can be expected due to the randomness of the problem, with a performance peak for four threads of close to 5 % of theoretical peak performance. For eight threads the

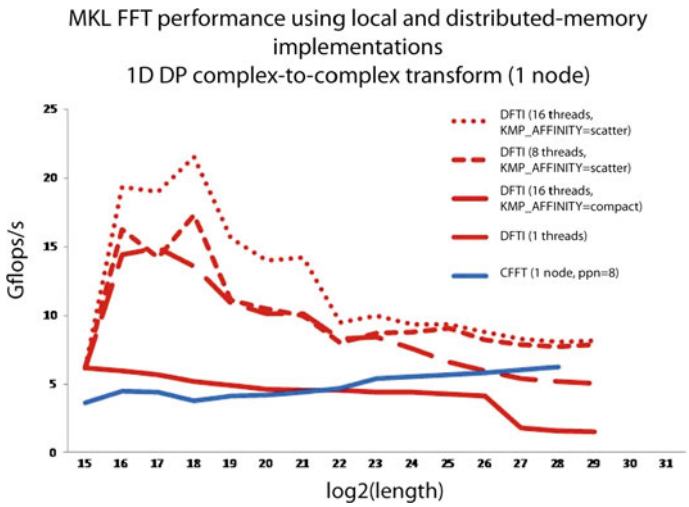


Fig. 3.16 Performance for Intel's recently released multi-threaded MKL FFT on a 2.8 GHz dual socket Nehalem platform (Petrov and Fedorov 2010)

performance is less variable and increases fairly monotonically with matrix size to a peak efficiency of about 3 %, Fig. 3.14.

FFT

From Fig. 3.15 it is apparent that the single node MPI code for the FFT is performing poorly. Indeed the performance is much worse than the single thread code regardless of the number of MPI processes on a node. Since these benchmarks were carried out Intel has released a multi-threaded MKL FFT code (Petrov and Fedorov 2010) with much improved performance also for a single thread. The results reported for a 2.8 GHz dual socket Nehalem are shown in Fig. 3.16. The single thread performance is about twice what we observed for the MKL version we used, and the multi-threaded version using one thread per core has a peak performance about six times higher than the single thread performance we measured. Using hyper-threading with two threads per core results in a performance boost that for some sizes may exceed 30 % and result in an efficiency of up to about 25 % for the node, similar to our observed single core performance without hyper-threading.

Random Number Generation

For the random number generator the aggregate performance increases almost in proportion to the number of instances run, as seen in Fig. 3.17.

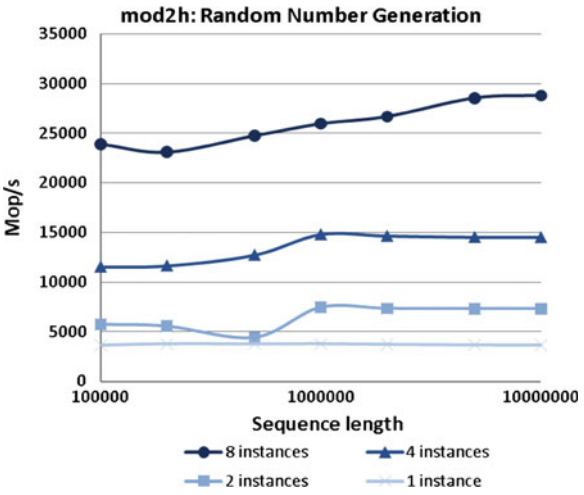


Fig. 3.17 Mod2h results on a dual socket, 8-core Intel Nehalem 2.53 GHz node with 24 GB memory (Sagar et al. 2010)

HPL

For HPL a best single node efficiency of close to 87 % has been reported for the Intel Nehalem, see e.g. (Feldman 2009; Gelas 2008). The measurements performed on the reference platform are in line with these results.

3.2.2.3 Energy Efficiency

In regards to energy efficiency matrix multiplication is known to exercise the CPU heavily and hence result in high power consumption. The HPL benchmark that is used for the Green500 (The Green500 2010) list depends heavily on matrix multiplication. For the reference platform we measured a maximum power consumption of 303 W for matrix multiplication (Sagar et al. 2010), resulting in 251MF/W at the achieved 76 GF. For HPL a power efficiency of 230 MF/W was observed (Sagar et al. 2010), which is in line with the expected power efficiency given the difference in efficiencies of matrix multiplication and HPL using the MKL. No power measurements were carried out for the sparse matrix-vector multiplication, the FFT and the random number generation. The FFT is fairly floating-point intensive, but not as intensive as matrix multiplication, but relatively more memory reference intensive. On this basis we estimate the maximum power consumption to about 250 W for the FFT resulting in an estimated power efficiency of 50–80 MF/W for the performance reported in Fig. 3.16.

3.2.3 nVidia C1060 GPUs

Matrix Multiplication

For matrix multiplication on the C1060 nVidia's CUBLAS was used in analogy with using MKL on the reference platform. Since in many applications the data set on which the computations are performed is allocated to the memory of the host processor, subsets of data on which computations are to be performed need to be transferred to the GPU memory and results transferred back. Thus, performance was measured both for the computations on the GPU itself with data fetched and stored in its local memory and for the situation when data needs to be fetched from the CPU memory and results stored in it. Figure 3.18 shows the results, with the lower performance curve including the pre and post computation data transfers between CPU memory and the GPU. Since matrix multiplication requires $2N^3$ operations but only $3N^2$ data elements need to be transferred, the data transfer time decreases in significance as N increases. The peak of the on GPU performance with CUBLAS is about 82 %, which drops to a peak of about 76 % if data transfers are included. These results are in agreement with the results reported in Phillips and Fatica (2010).

Sparse Matrix-Vector Multiplication

For sparse matrix-vector multiplication the results are shown in Fig. 3.19. It is interesting to note that with the data on the GPU the peak observed performance is about 9 GF, or about 11.5 % of peak, a higher fraction of peak than on the CPU. This result

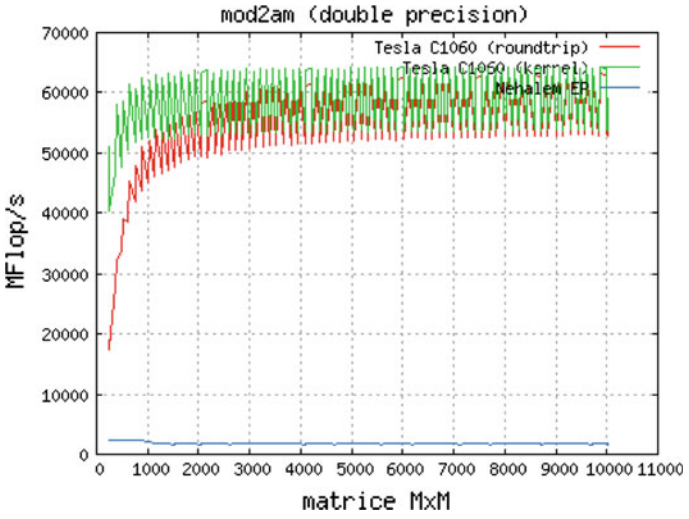


Fig. 3.18 Mod2am results on nVidia C1060 GPU with 78 GF peak performance (PRACE 2011)

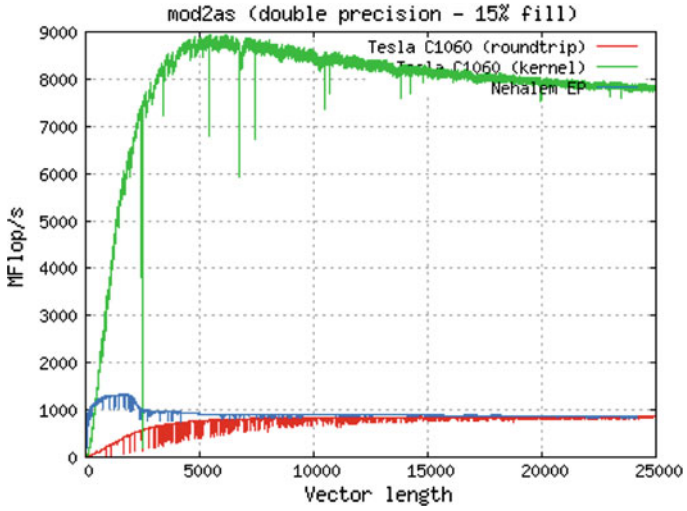


Fig. 3.19 Mod2as results on nVidia C1060 GPU with 78 GF peak performance (PRACE 2011)

is in line with the results in Matsuoka and Dongarra (TESLA GPU computint). However, if data needs to be fetched from CPU memory and results transferred back, then the data transfer time dominates and the efficiency drops to about 1 %. For sparse matrix-vector multiplication both operation count and data transfer is of order O(N).

FFT

The FFT performance on the C1060 is shown in Fig. 3.20. At the time of the benchmark there was no double-precision CUDA FFT available so a complete port of the mod2f FFT to CUDA was necessary resulting in a CUDA code with about 3,000 lines. The peak performance achieved including data transfers to the CPU memory was about 4 GF, about 5 % of peak. At this time the nVidia CUFFT is available and is reported to achieve close to 30 GF on a C1060 (CUDA Case Studies 2009) excluding data transfer. For FFT the operations count is O(NlogN), and thus the impact of the data transfer expected to be less significant than for sparse matrix-vector multiplication but more significant than for matrix multiplication. The peak efficiency of the single core Nehalem FFT is about 25 %. The recently released multi-threaded MKL FFT (Petrov and Fedorov 2010) has an improved single thread performance that is estimated to about 5.4 GF for a single core of the reference platform and about 20 GF for 16 threads on the reference platform, scaling the results in Petrov and Fedorov (2010) with the ratios of the clock frequencies of the reference platform and the platform in Petrov and Fedorov (2010) (the MKL hyper-threaded version performs better than the single thread per core version). Thus, the recent MKL release achieves about 54 % efficiency on a single core and a peak of about 25 % on the node, while CUFFT achieves a peak efficiency of about 38 % on the C1060.

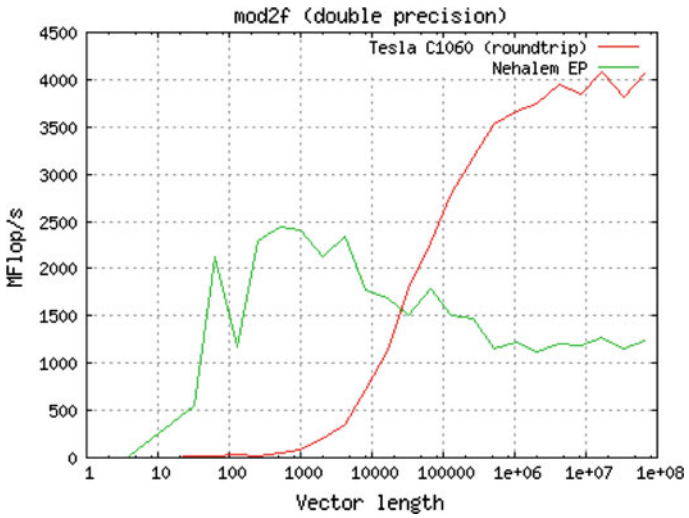


Fig. 3.20 Mod2f results in hand coded CUDA on nVidia C1060 GPU with 78 GF peak performance (PRACE 2011)

HPL

For HPL a peak efficiency for one Nehalem core and one C1060 GPU was measured to be 59.5 GF, 68 %, whereas the efficiency dropped to 52.5 % using all 8 cores of the host and four C1060 GPUs (Sagar et al. 2010). The peak power efficiency was 270 MF/W. The single C1060 results are in line with what is reported in Phillips and Fatica (2010).

Energy Efficiency

GPUs draw significant power with the C1060 having a specified max power of 188 W (Tesla C1060 Computing Processor Board Specification 2010) and an estimated typical power consumption of 160 W. The Intel Nehalem CPU used for the reference platform has a maximum power dissipation of 80 W (Intel Xeon Processor E5540 2011).

For the reference platform during maximum load for matrix multiplication the CPUs account for about 50 % of the power consumption of the reference platform. With the C1060 reaching close to 60 GF for matrix multiplication, Fig. 3.18, and assuming the maximum specified power consumption for this case, the GPU power efficiency is estimated at 300 MF/W. Similarly, for the CPUs alone, the achieved performance using MKL was 76 GF and assuming the maximum CPU power consumption the CPU power efficiency is estimated to be 475 MF/W. The fact that the GPU in case of HPL improves the combined energy efficiency is due to the fact that

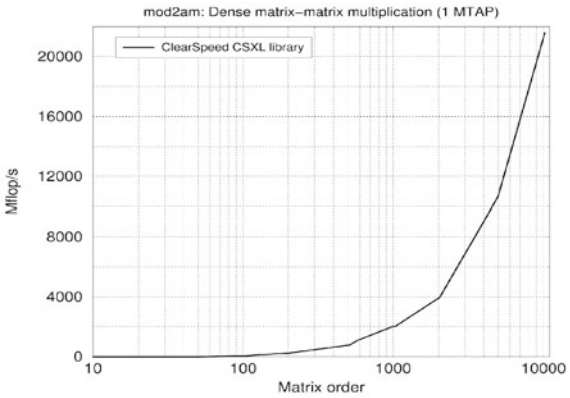


Fig. 3.21 Mod2am results on one MTAP with a peak performance of 48 GF (Sagar et al. 2010)

the power consumption by the memory, fans, power supplies, motherboard etc is already accounted for in the reference platform power efficiency (that is about half of the CPU power efficiency).

3.2.4 ClearSpeed CSX700

Matrix Multiplication

Matrix multiplication carried out on a single Multi-Threaded Array Processor (MTAP) (CSX700 Datasheet 2011) of which there are two on a CSX700 is shown in Fig. 3.21. For the CSX700 the peak observed performance was 85 GF (PRACE 2011), or 88.5 % of peak. For the e780 with 8 CSX700 units the peak observed performance was 520 GF (PRACE 2011), 68 % of peak.

As is clear from Fig. 3.21 the ClearSpeed performance is not significant in comparison with the host CPU until the matrix dimensions are in the order of a few thousands. The library (CXSL User Guide 2010) that comes with the ClearSpeed hardware recognize this and leaves the multiplication of the matrices to be performed on the host for small matrices. In fact, the software allows for load sharing between the host and the ClearSpeed board. Figure 3.22 shows the aggregate performance for matrix multiplication as a function of the host assist. The choice of matrix dimensions for the benchmark was compliant with the CSX700 unit working with tiles that for M and N are multiples of 192 and for K a multiple of 288, for multiplication of an $M \times K$ matrix by a $K \times M$ matrix. For other matrix shapes the CSX700 library partitions the matrices into tiles compliant with these restrictions and has the host execute the remaining matrix parts for a correct result. For the matrix shapes studied in this benchmark the maximum performance exceeds 130 GF at 42 % host assist for the largest $M = N$. The combined peak performance represents 71 % of the

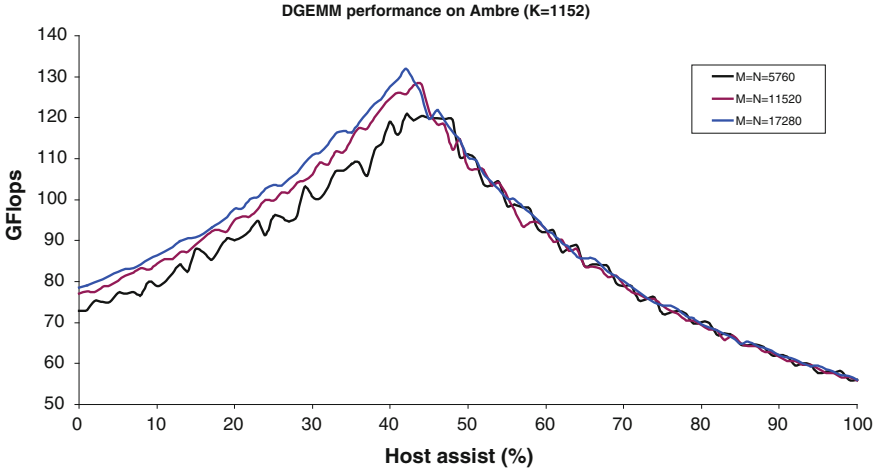


Fig. 3.22 Mod2am results on the reference platform equipped with a ClearSpeed CSX700 accelerator as a function of the host assist percentage. Peak host performance 80.96 GF, peak CSX700 performance 96 GF (Sagar et al. 2010)

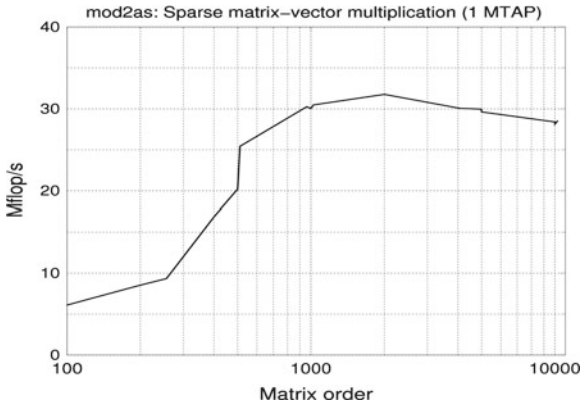


Fig. 3.23 Mod2as results on one MTAP with a peak performance of 48 GF (Sagar et al. 2010)

combined theoretical peak performance. This is lower than the peak efficiency for the CSX700 card (88.5 %) and the host (93.9 %), but the matrices chosen for this experiment did not maximize performance for either.

Sparse Matrix-Vector Multiplication

The sparse matrix-vector performance is shown in Fig. 3.23. The performance is exceedingly poor with a peak performance of only close to 30 MF, or less than 0.1 % of the peak performance. The MTAP has an architecture that favors streams, like GPUs, but clearly its performance for random memory accesses is very poor.

Table 3.6 Mordf results on the CSX700 with peak performance of 96 GF (48 GF per MTAP) (Sagar et al. 2010)

Size	1 MTAP	2 MTAP
256	2.8	5.7
512	3.4	6.7
1024	3.8	7.4
2048	4.2	9.4
4096	5.0	9.9
8192	3.7	7.9

FFT

For complex-to-complex 1-D FFTs the results are shown in Table 3.6. The best observed performance was 9.9 GF, 10.3 % of peak. Comparing to the MKL performance reported in Petrov and Fedorov (2010) the reference node performs better than the CSX700, but a CSX700 delivers a peak performance about twice that of a single core of the reference platform.

Random Number Generation

The performance for random number generation is shown in Fig. 3.24 for a single MTAP. The MTAP performance is about 10 % lower than the performance of a single core of the reference platform.

HPL

For HPL that depends heavily on matrix multiplication the CSX700 contributed 43.75 GF at 42 % host assist, yielding an overall efficiency of 63 % (Sagar et al.

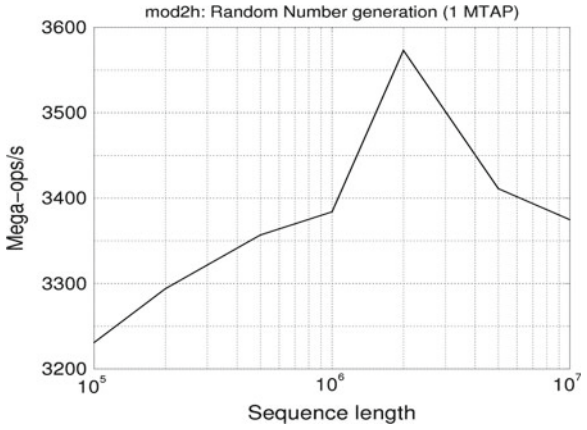


Fig. 3.24 Mod2h results on 1 MTAP (Sagar et al. 2010)

2010). The results on the manufacturer web site indicates a peak HPL performance of 56.1 GF (Linpack, ClearSpeed 2010) corresponding to an efficiency of 58.4 %.

Energy Efficiency

In regards to energy efficiency the CSX700 was observed to consume about 10 W in idle state (9.5–10.5 W observed) (Sagar et al. 2010) and about 16 W performing matrix multiplication (Sagar et al. 2010). Thus, with a peak matrix multiplication performance of 85 GF the power efficiency is about 5300 MF/W for the CSX700, while for HPL our results yield in excess of 2700 MF/W for the CSX700 alone at a delivered rate of 43.75 GF and a combined power efficiency of 350 MF/W for the reference platform with one CSX700.

For FFT the peak measured performance was 9.9 GF. The power consumption for the FFT was not measured, but it clearly must be in the 10–16 W range (Sagar et al. 2010) resulting in a power efficiency in the 600–1000 MF/W range. For the reference platform the idle power was measured to be about 140 W and the peak power 303 W (Sagar et al. 2010) resulting in a power efficiency range of 70–150 MF/W. Thus, though the absolute performance for the CSX700 is inferior to the MKL multi-threaded reference platform performance, the energy efficiency is a factor 6–8 times better.

For random number generation the aggregate performance for the reference platform is about 4 times higher than the CSX700 performance, but the power consumption is estimated to be 10–20 times higher and hence the CSX700 considerably more power efficient.

3.2.5 Performance Comparison

Figure 3.25 summarizes the performance results for matrix multiplication normalized to the reference platform. The C1060 has slightly lower theoretical peak double-precision performance (78 GF) and the CSX700 has slightly higher theoretical peak performance (96 GF) than the reference platform (81 GF). The combined peak performance of the reference platform and a CSX700 is close to 2.2 times that of the reference platform itself, while adding a C1060 results in a node with 1.96 times the performance of the reference platform.

For sparse matrix-vector multiplication both the C1060 and CSX70 do not offer any performance advantage, Fig. 3.26.

For the complex-to-complex 1-D radix-4 FFT the relative results we observed are shown in Fig. 3.27. However, since our measurements were made, a new version of the MKL library has been released that improved the reference platform performance with up to more than 7 times thus making the reference platform performance superior to the CSX700. nVidia has also released a CUFFT version that supports

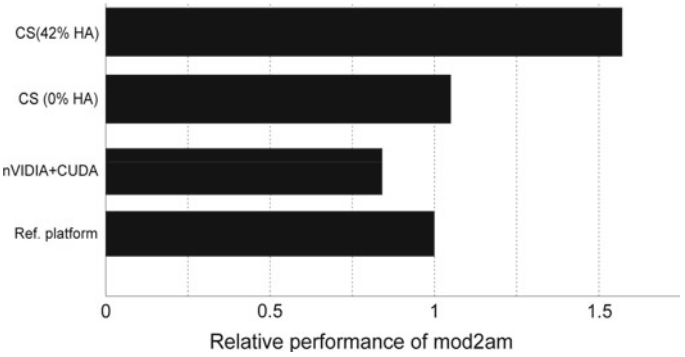


Fig. 3.25 Mod2am performance on the nVidia C1060 GPU and ClearSpeed CSX700 relative to the reference platform (Sagar et al. 2010)

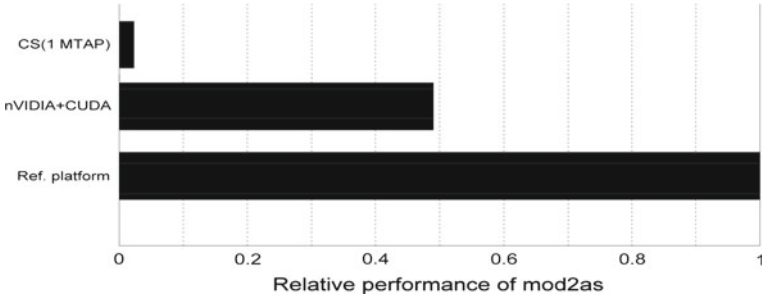


Fig. 3.26 Mod2as performance on the nVidia C1060 GPU and ClearSpeed CSX700 relative to the reference platform (Sagar et al. 2010)

double-precision arithmetic and that achieves about 50 % better performance than that of MKL on the reference platform.

For random number generation a single CSX700 MTAP has a performance comparable to a single core of the reference platform. No random number generator was available for the C1060 at the time of the benchmark.

For HPL, a single core of the reference platform in combination with one C1060 GPU was measured to yield 59.5 GF (Sagar et al. 2010) corresponding to 68 % efficiency while all eight cores together with four C1060 resulted in a peak node performance of 206 GF out of a possible 393 GF corresponding to 52.5 % efficiency.

We summarize our own measurements and some from the literature in Table 3.7 in order to compare efficiencies of the selected benchmarks on the different architectures, and the energy efficiencies of the devices in isolation and together as an integrated system.

For the CSX700 the HPL performance is derived from Linpack, ClearSpeed (2010). This estimate compares fairly well with estimating the performance from the CSX600 performance reported in Kozin (2008) by scaling the performance with the ratio of the peak performances of the CSX700 and CSX600 units, thus assuming

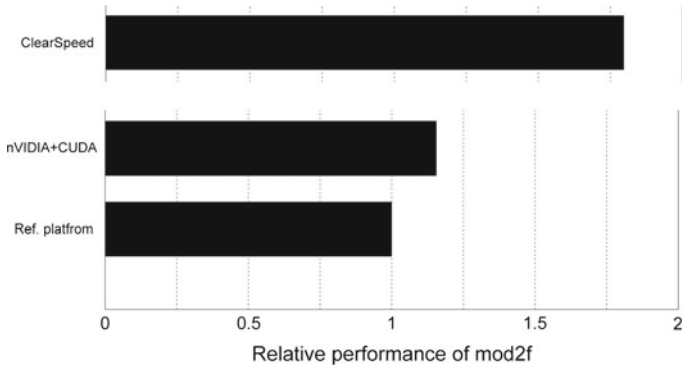


Fig. 3.27 Mod2f relative performance using MKL version 10.1 on the reference platform alone and with nVidia C1060 GPU or ClearSpeed CSX700 acceleration (Sagar et al. 2010). MKL release 10.2 having a multi-threaded version of the FFT and improved single core performance has resulted in the reference platform achieving about twice the performance of the CSX700, and a new release of the CUFFT has resulted in the C1060 achieving a peak performance about 50 % higher than the reference platform

the same efficiency for the units. For the host plus CSX700 HPL performance the number is estimated from the measured performance of an eight node system with four CSX700 per node (Sagar et al. 2010). The performance of one such node was measured at 206.25 GF with 43.75 GF contributed by each CSX700. Thus, in this in this configuration the four CSX units in a node contributed 175 GF to the node performance and the host 31.25 GF.

In regards to efficiency we notice that for matrix multiplication all three architectures do well, as expected, with the host having a slight advantage. For sparse matrix-vector multiplication none does well, with the CSX700 performing by far the worst. Surprisingly the C1060 perfromed better than the host, but in combination with the host the C1060 is not efficient due to the low computational intensity of sparse matrix-vector multiplication (computations and data transfer are both of order O(N)).

For the FFT the C1060 offers the best efficiency using the optimized CUFFT from nVidia which has about 50 % higher efficiency than the optimized MKL for the reference platform (38.5 vs. 24.7 %). The CSX700 efficiency is less than half of that of the reference platform and about 25 % of the efficiency of the C1060.

The HPL performance as expected is somewhat lower than that of matrix multiplication on which it depends heavily, and the relative merits of the host, the C1060 and the CSX700 are about the same with the CSX700 however ending up with an efficiency about the same as that of the C1060.

Table 3.7 Summary of peak performance and efficiency

	Host (81 GF)		C1060 (78 GF)		C1060 incl transf		CSX700 (96 GF)		Host + CSX700	
	GF	Eff (%)	GF	Eff (%)	GF	Eff (%)	GF	Eff (%)	GF	Eff (%)
Mod2am	76	93.9	64	82.1	61	78.2	85	88.5	130	73.4
Mod2as	3.8	4.7	9	11.9	1	1.3	0.03	0	–	–
Mod2f	20[a] (Pettey 2011)	24.7	30 (Productivity benefits of Intel Ct Technology 2010)	38.5	4	5.1	9.9	10.3	–	–
HPL		87 (PGI Accelerator Programming Model for Fortran and C 2010)	50 (Silicon Graphics 2011)	64.1		52.5	56 (Shimpi 2010)	58.3	75[a]	42.4[a]

[a]Estimated values

Table 3.8 Power efficiency of the configurations evaluated

	Host			Host + C1060			Host + CSX700		
	GF	W	GF/W	GF	W	GF/W	GF	W	GF/W
Mod2am	76	303	0.251	130[a]	490[a]	0.265[a]	130	315[a]	0.410[a]
Mod2f	20[a] (Pettey 2011)	250[a]	0.080[a]	40[a]	420[a]	0.095[a]	25[a]	260[a]	0.096[a]
HPL	69[a]	303[a]	0.230			0.270	75[a]	315[a]	0.238[a]

[a]Estimated values

3.2.6 Power Efficiency Comparison

As previously mentioned the peak performances of the reference platform, the C1060 and the CSX700 are fairly comparable, but the efficiencies achieved on the platforms are quite different and the maximum power consumption is also quite different. We did not have the opportunity to carry out power measurements for all benchmarks. The results are summarized in Table 3.8.

Adding a CSX700 to a node increases its maximum power consumption by about 5 %, while the C1060 increases it with more than 60 %. For matrix multiplication the CSX700 resulted in a total node performance of 130 GF in our tests and hence the power efficiency increased from about 250 MF/W to about 410 MF/W. The power efficiency for the CSX700 itself is about 5.3 GF/W (85 GF, 16 W) whereas for the Nehalem itself is about 0.475 GF/W (76 GF, 160 W).

The power estimates for the FFT assumes about 80 % (250 W) of maximum power for the reference platform. The C1060 in itself has a power efficiency of about 0.175 GF/W (30 GF, 170 W) whereas the Nehalem itself has a power efficiency of about 0.155 GF/W (10 GF, 65 W). The CSX700 itself has significantly higher power efficiency; about 0.700 GF/W (10 GF, 14 W)

For HPL the power efficiency improves for a host with accelerator compared to the host itself, as expected from the results for matrix multiplication. The marginal improvement for a host with CSX700 is surprising. Considering the CPU itself it has a power efficiency of about 0.440 GF/W (35 GF, 80 W), whereas the C1060 itself is estimated to 0.265 GF/W (50 GF, 190 W) and the CSX700 is estimated to 3 GF/W (45 GF, 15 W).

The power efficiency of the CSX700 is a factor of 4–10 higher than that of the CPU itself for matrix multiplication, FFT and HPL, but unfortunately for FFT and HPL the relatively low fraction of peak realized cause the total platform power efficiency to increase only marginally for a host combined with a single CSX700. The C1060 power efficiency for matrix multiplication, 0.34 GF/W (64 GF, 190 W) is less than that of the Nehalem, which is also the case for HPL, but the power efficiency is slightly higher for the FFT.

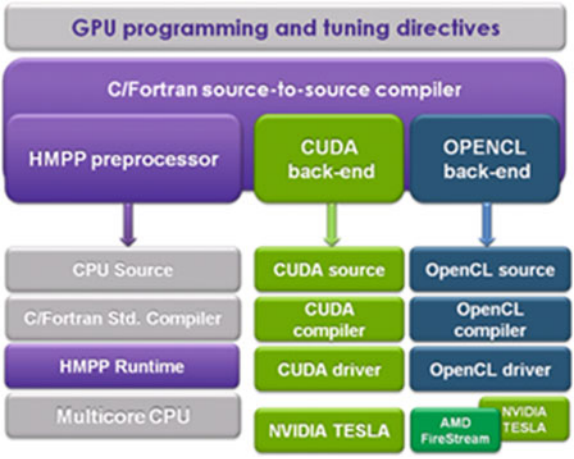


Fig. 3.28 The architecture of the HMPP preprocessor (Hybrid Multi-Core Parallel Programming Workbench 2011)

3.3 Programming Tools Assessment

3.3.1 HMPP (Hybrid Multi-Core Parallel Programming)

The Hybrid Multi-core Parallel Programming (HMPP) preprocessor by CAPS (Dolbeau et al. 2007; HMPP Open Standard 2011; Hybrid Multi-Core Parallel Programming Workbench 2011) use directives inserted into the source code to control code generation. The directives have the form of special comments in Fortran and pragmas in C. Using the directives the HMPP preprocessor directs the code generation to be made for the desired device by a compiler for that device. The HMPP preprocessor generates the code necessary to manage the data transfers between the host and accelerators and seeks to optimize it. By using directives an annotated code can be compiled by any compiler for any desired platform and hence the annotated code is as portable as the original code. The HMPP preprocessor has a fallback mechanism should an executable code fail to be generated for a particular target accelerator. Should that be the case code is generated for the host by the compiler used for it. The HMPP directives are designed to target functions (codelets) that can be executed on accelerators and for optimizing the data transfers between the host and accelerators.

The architecture of the preprocessor is shown in Fig. 3.28 in which two back-ends of current interest are shown. The HMPP memory model is illustrated in Fig. 3.29. Our focus was on the CUDA back-end because the OpenCL specification was just released at the time of this study. Our target was the nVidia C1060 GPU as accelerator for IA-32 servers. The test platform had dual socket quad-core 2.8 GHz Intel Harpertown CPUs. Initially the HMPP Workbench 1.5.3 was used, later release 2.1.0sp1

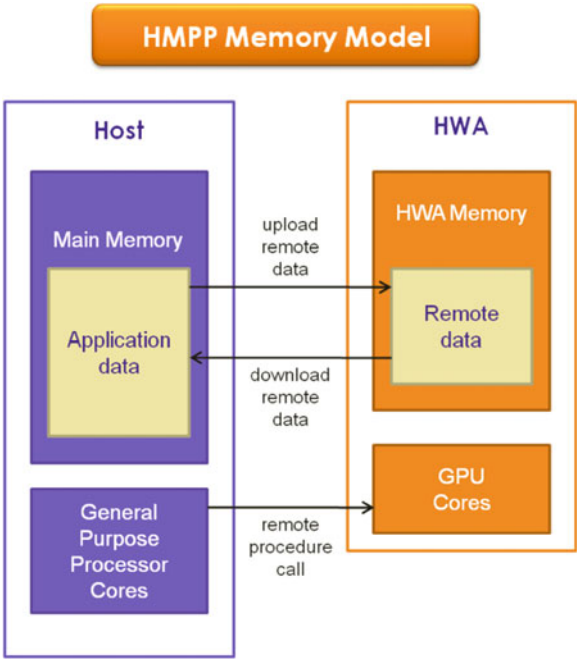


Fig. 3.29 The HMPP memory model (HWA = HardWare Accelerator) (HMPP Open Standard 2011)

when it became available. For the host the Intel compiler version 11.1 was used and for the C1060 the CUDA 2.3 environment.

An example of the use of the HMPP directives is shown in Fig. 3.30.

The result of using HMPP for matrix multiplication for the C1060 is shown in Fig. 3.31 and for sparse matrix-vector multiplication in Fig. 3.33. These two routines were the only two ported during the course of this study. For matrix multiplication the CUDA code generated by HMPP for a "simple" port has a performance of 60–75 % of the CUBLAS performance as seen by comparing Figs. 3.31 and 3.18, which is a very good result for a small effort. However, after code optimization using good knowledge of the target architecture and HMPP performance comparable to, or even better than, that of CUBLAS was obtained, as seen in Fig. 3.32.

The lessons learned from the limited use of HMPP are Sagar et al. (2010).

Modifying a code to use HMPP to generate a functional code for a GPU is simple. The resulting performance may be quite good for a modest effort, or fairly poor depending on the nature of the computations. For "optimal" performance on a GPU the original code is likely to require modification, unless designed to work well on a streaming architecture.

```
  // simple codelet declara-
tion
  #pragma hmpp Hmxm codelet,
args[a;b].io=in,
args[c].io=out,
args[a].size={m,l},
args[b].size={l,n},
args[c].size={m,n}, TARGET=CUDA
  void mxm(int m, int l, int
n, const double a[m][l], const
double                 b[l][n],
double c[m][n])
  {    int i, j, k;
      for (i = 0; i < m; i++)
{
   for (j = 0; j < n;
j++) {
              c[i][j] = 0.0;}}
      for (i = 0; i < m; i++)
{
          for (k = 0; k < n;
k++) {
              for (j = 0; j <
l; j++) {
                  c[i][k]     =
 c[i][k] + a[i][j] * b[j][k];}}}
```

```
  // usage of the codelet
  #pragma hmpp Hmxm advancedload,
args[a;b],       args[a].size={m,l},
args[b].size={l,n}
  for (i = 0; i < nrep; i++) {
  #pragma hmpp Hmxm callsite,
args[a;b].advancedload=true
  #pragma hmpp Hmxm callsite
            mxm(m, l, n, (double
(*)[m]) a, (double (*)[n]) b,
(double (*)[n]) c);
  }
  #pragma hmpp Hmxm delegatedstore,
args[c]
```

Fig. 3.30 Illustration of use of HMPP pragma's for definition and use of codelets (Sagar et al. 2010)

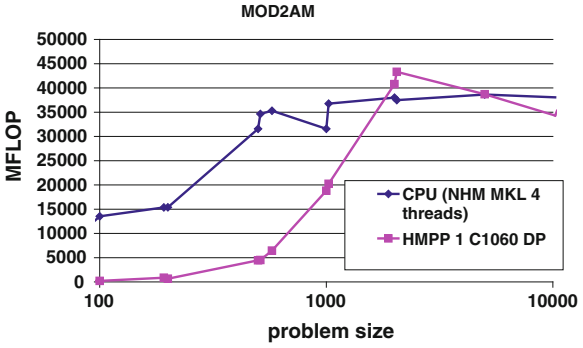


Fig. 3.31 Mod2am performance on the C1060 using HMPP (Sagar et al. 2010)

- Some constructions (such as reductions) are difficult to parallelize and do not perform well on GPUs (or many other highly parallel architectures, some of which have special hardware for reduction operations).
- Producing optimized code for heterogeneous node architectures requires in-depth knowledge of the hardware (not specific to HMPP or GPUs).
- Astute directives for code generation (such as loop reordering, loop fusion, etc.) are a great help to boost performance.
- The performance of codes generated by using HMPP can be equal to or better than that offered by vendor libraries, which is very encouraging.

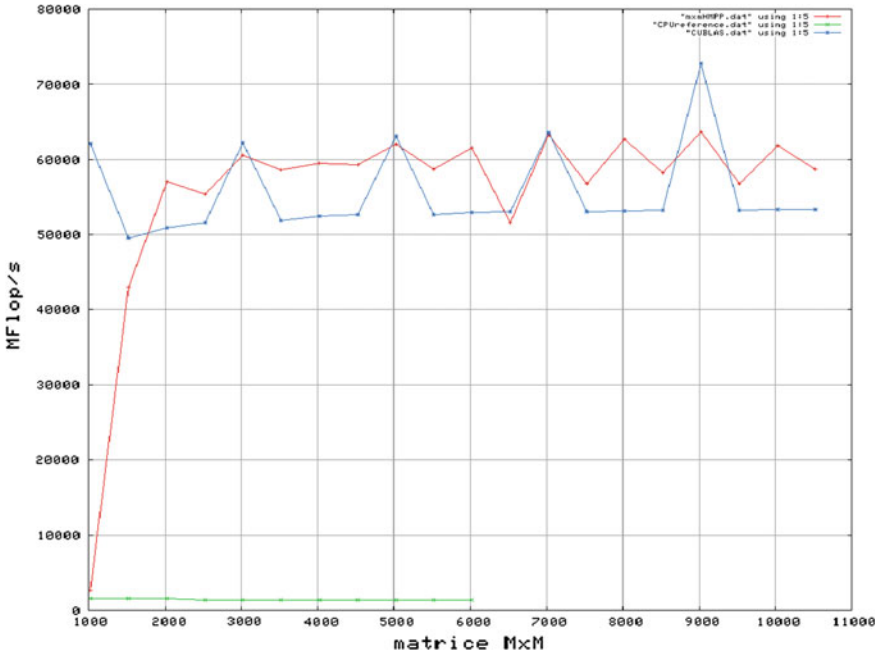


Fig. 3.32 Optimized performance of matrix multiplication using HMPP compared to CUBLAS for the C1060 (PRACE 2011)

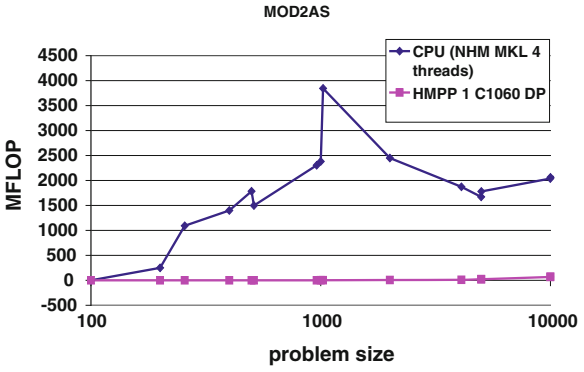


Fig. 3.33 Mod2as results on the C1060 using HMPP (Sagar et al. 2010)

3.3.2 RapidMind

The RapidMind Multi-Core Development Platform (McCool 2007, 2008) was designed for application code portability across platforms, including multi-core CPUs, GPUs and the CBE (Chen et al. 2005). About a year after this study was

initiated RapidMind was acquired by Intel and the RapidMind technology integrated with Intel's Ct technology (Ghuloum et al. 2007a,b; McCool 2007; Productivity benefits of Intel Ct Technology 2010) and some of it recently released as part of Intel's Array Building Blocks (ArBB) (Intel(R) Array Building Blocks for Linux OS, User's Guide 2011; Intel(R) Array Building Blocks Virtual Machine, Specification 2011; Sophisticated Library for Vector Parallelism 2011). RapidMind targeted a data parallel programming model (as did Ct) but did support task-parallel operations. RapidMind added special types and functions to C++ enabling a programmer to define operations (functions) on streams (special arrays). By the freedom to define array operations RapidMind supported more powerful array operations than, e.g., those available in Fortran. Data dependencies and data workflows could be easily described and information necessary for an efficient parallelization included. The compiler and the runtime environment had sufficient information to decide how to auto-parallelize code.

We report results using RapidMind to generate code for the C1060 for matrix multiplication, Fig. 3.34, sparse matrix-vector multiplication, Fig. 3.35, and the radix-4 complex-to-complex 1-D FFT, Fig. 3.36. As can be seen from Fig. 3.34 RapidMind only achieves about 25 % of the performance of CUBLAS. The "simple" version was created by adding 20 lines of RapidMind code to the mod2am code from EuroBen. The GPU-optimized code made use of code downloaded from the RapidMnd developer web site. For sparse matrix-vector multiplication RapidMind again achieved about a quarter of the performance of CUBLAS, and for the FFT it achieved about 20 % of the performance of our CUDA code. Using RapidMind a first executable was fairly easy to generate, but to achieve good performance significant work and insight into RapidMind and the target architectures was necessary. A more in-depth discussion of the RapidMind porting effort can be found in Christadler and Weinberg (2010).

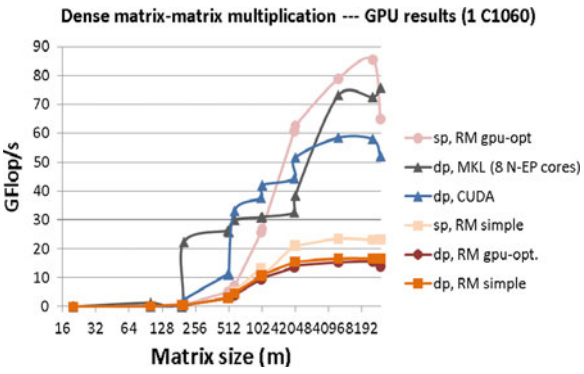


Fig. 3.34 Mod2am results using RapidMind compared to using CUDA on the C1060 and MKL on the reference platform (Sagar et al. 2010)

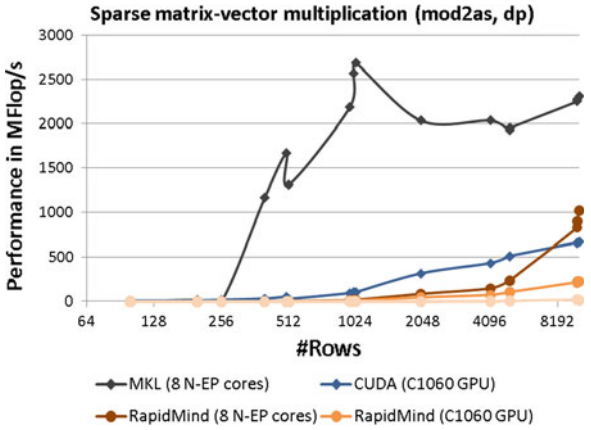


Fig. 3.35 Mod2as results using RapidMind compared to CUDA on the C1060 and MKL on the reference platform (Sagar et al. 2010)

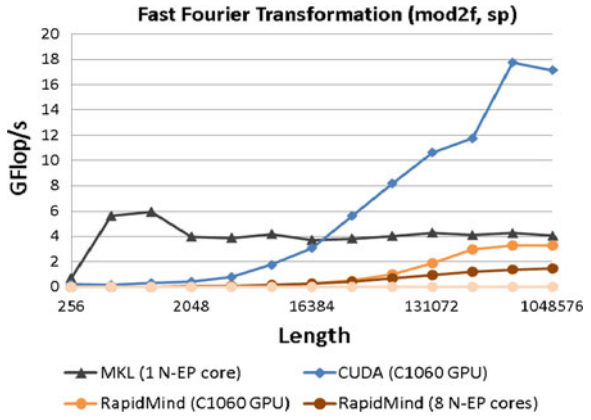


Fig. 3.36 Mod2f results using RapidMind compared to CUDA on the C1060 and MKL with one thread on the reference platform (Sagar et al. 2010)

3.3.3 PGI Accelerator Compilers

PGI in 2009 released enhanced versions of their C and Fortran compilers that use directives to control code generation for specialized hardware (Portland Group Inc 2011; PGI Accelerator Programming Model for Fortran and C 2010) like GPUs using a similar approach to the one used for the HMPP preprocessor. We investigated the PGI accelerator compiler capabilities on the matrix multiplication and sparse matrix-vector multiplication EuroBen codes. The results are shown in Figs. 3.37 and 3.38. For matrix multiplication the generated code achieved a peak performance slightly in excess of 8 GF, or 11 % of the peak C1060 performance. On the host platform the PGI compiler generated code achieved at best 17 % of theoretical peak. For the sparse

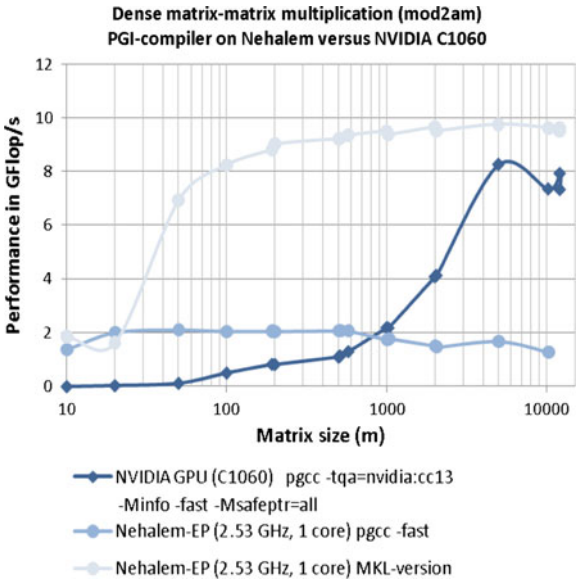


Fig. 3.37 Mod2am results using the PGI Accelerator C compiler for the C1060 (Sagar et al. 2010)

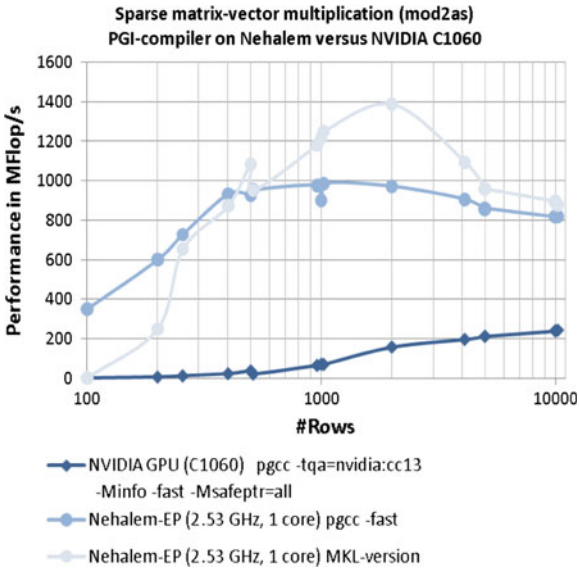


Fig. 3.38 Mod2as results using the PGI Accelerator C compiler for the C1060 (Sagar et al. 2010)

matrix-vector multiplication it is interesting to note that he PGI compiler generated code achieves a performance comparable to that of MKL. However, on the GPU the performance of the PGI generated code is significantly lower than that of the CUBLAS, see Fig. 3.19.

3.3.4 Programming Tools Comparison

The CAPS HMPP preprocessor as well as the PGI Accelerator Compiler and RapidMind were immature tools at the time of this study. OpenCL was also included but the beta compilers available during this effort were buggy and the porting efforts had significant problems. For matrix multiplication, which is ideal for many architectures, including GPUs, and a function for which many compilers perform well, the GPU codes generated by RapidMind and the PGI Accelerator Compiler were not good. The performance achieved was at best about 20 and 11 % of peak, respectively. The CAPS HMPP preprocessor did better and after an optimization effort generated code that performed comparable to the CUBLAS, a very good result. For sparse matrix-vector multiplication all tools generated poor code for the C1060 with a peak performance of much less than 1 % of peak and a factor 5–10 worse than CUBLAS. The PGI Accelerator Compiler generated good code though for the host (comparable in performance to MKL).

Clearly, producing code that performs well is a very important aspect of a programming tool for HPC. However, ease of use including debugging is also important for productivity (Kennedy et al. 2004) as determined in the DARPA High Productivity Computing Systems program (Dongarra et al. 2008). Measuring productivity is difficult since many hard to measure factors may influence the outcome, such as the programmers experience with similar programming tasks, familiarity with the tools, platform etc, and the extent to which code needs to be modified or entirely redesigned, rewritten and debugged. Though the number of lines of code is a debated measure it is generally agreed that error rates and debugging time are likely to increases with increased code size, and that fewer lines of code is an indication of the expressiveness and quality of a language. Thus, for the porting efforts undertaken the number of lines of code was measured, and so was the time to produce and debug the code, and in some cases optimize it. Though there is much uncertainty in the data it nevertheless appears true that languages and programming models targeting expressiveness do result in shorter codes (Erbacci et al. 2009).

Table 3.9 shows a sample of the measurements. The lines of codes reported are true but unfortunately misleading in that the HMPP code includes several versions that were produced in attempting to get good performance (Sagar et al. 2010). But the code size for one version is nevertheless larger than for RapidMind. The reported time for RapidMind, which include learning the tool, does show that the learning curve can be significant for new tools that address a complex programming situation.

Table 3.9 Programmer productivity measurements for mod2am and mod2as using CAPS HMPP and RapidMind

	Lines of code		Development time		Performance (% of peak)	
	Mod2am	Mod2as	Mod2am	Mod2as	Mod2am	Mod2as
CAPS HMPP	976	979	5	0.5	78.99	0.09
RapidMind	30	27	18.5	12	19.85	0.29

3.4 Conclusions

Though double-precision arithmetic performance was not a strong point for GPU at the time of the evaluation, the expected evolution of GPU performance and programmability and potential advantage in energy efficiency made it interesting for PRACE to evaluate GPUs as accelerators for future HPC systems, in particular from a programming and energy efficiency point of view. The programming issues associated with heterogeneous node architectures and the streaming architecture of GPUs are likely to remain as support for double-precision arithmetic and programming flexibility in future generations of GPUs improve. This expectation has already been realized to some degree since our study was undertaken.

In regards to the fraction of peak performance achieved for the Nehalem CPU, and the C1060 and the CSX700 by themselves, the Nehalem performed the best for matrix multiplication with an efficiency of 93.9 %, with the CSX700 being second at 88.5 % and the C1060 being third at 82.1 %. From an energy perspective the CSX700 outperformed the CPU by a factor of more than 11 and the C1060 by an estimated factor of more than 15. For HPL the differences were somewhat less dramatic with the CSX700 having a power efficiency about 7 times higher than the Nehalem CPU and a power efficiency more than 10 times that of the C1060. For FFT, the C1060 with a good library implementation achieved a peak efficiency of close to 40 %, about 50 % better than the CPU. The CSX700 did not perform well and only achieved 25 % efficiency. However, because it only consumes less than 10 % of the power of the C1060, it still had the best power efficiency, estimated at more than three times that of the C1060. The estimated power efficiency of the Nehalem for FFT was about half of that of the C1060, but the Nehalem CPU itself has comparable power efficiency. The C1060 performed surprisingly well on the sparse matrix-vector multiplication benchmark and indeed performed significantly better than the CPU, and should also, despite its high power consumption have a power efficiency much better than the CPU. The CSX700 performed very poorly on the sparse matrix-vector multiplication.

In regards to an integrated system as expected the data transfers between the CPU memory and the accelerators have a significant impact on the benefit for less compute intensive tasks, such as FFT and in particular sparse matrix-vector multiplication. For compute intensive tasks such as matrix multiplication and HPL the accelerators offers both a performance boost and improved power efficiency, while for computations such as FFT the performance improvement may be less but a power efficiency improvement may still be possible because “shared infrastructure”, such as memory, is included in the host measures.

For all the power efficiency measurements and estimate we caution that results can easily be misleading depending on the intended objective. Measuring total power consumption and performance is fairly straightforward, but in attempting to assess what to expect for future generation systems it is necessary to have observations for the various components themselves, since the components are likely to change in different ways. Component energy measurements are difficult and may require hardware modification, in particular for current measurements.

The programming tools that were evaluated were all immature, with OpenCL being so immature that reliable results were not obtained. For HMPP and RapidMind creating a usable code was quite simple and only required a modest learning effort, but creating an efficient code required a measurable effort requiring good knowledge of both the tool and the architecture of the target devices.

Acknowledgments The results reported here are due to efforts by many members of the PRACE Preparatory Phase Work Package 8 and documented in a deliverable to the European Commission under grant agreement RI-211528 within the EU Commission's infrastructure initiative INFRA-2007-2.2.2.1. Support for this effort has also been received from SNIC, the Swedish National Infrastructure for Computing a meta-center for HPC under the Swedish Research Council which is gratefully acknowledged.

References

Ali A, Johnsson L, Mirkovic D (2007) Empirical auto-tuning code generator for FFT and trigonometric transforms. Paper presented at the 5th workshop on optimizations for DSP and embedded systems. International symposium on code generation and optimization, San Jose

AMD™ Processor Pricing (2011) Advanced Micro Devices, Inc. Accessed 2 May 2011, from http://www.amd.com/us/products/pricing/Pages/server-opteron.aspx, Advanced Micro Devices, Inc

Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Yelick KA (2006) The landscape of parallel computing research: a view from Berkeley. (UCB/EECS-2006-183). http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

ATI Radeo HD 5870 Graphics (2011) Advanced Micro Devices, Inc. Accessed 2 May 2011, from http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx#2, Advanced Micro Devices, Inc

Belady CL (2007) In the data center, power and cooling costs more than the IT equipment it supports. Electronics cooling

Bell BS (2009) RV870 architecture, FS Media, Inc. Accessed 2 May 2011, from http://www.firingsquad.com/hardware/ati_radeon_hd_5870_performance_preview/page3.asp, FS Media, Inc

CAPS (2011) CAPS enterprise. Accessed 2 May 2011, from http://www.caps-entreprise.com/index.php, CAPS enterprise

Cell (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/w/index.php?title=Cell&oldid=426379510, Wikipedia

Cell Project at IBM Research (2011) IBM. Accessed 2 May 2011, from http://www.research.ibm.com/cell/, IBM

Chen T, Raghavan R, Dale J, Iwata E (2005) Cell broadband engine architecture and its first implementation. Accessed from https://www.ibm.com/developerworks/power/library/pa-cellperf/

Christadler I, Weinberg V (2010) RapidMind: portability across architectures and its limitations. Paper presented at the facing the multi-core challenge (conference proceedings), Heidelberg

Clark J (1980) A VLSI geometry processor for graphics. Comput Mag 13(7):59–68

Clark J (1982) The geometry engine: a VLSI geometry systems for graphics. Comput Graph 16(3):127–133

ClearSpeed (2011) ClearSpeed Technology. Accessed 2 May 2011, from http://www.clearspeed.com/, ClearSpeed Technology

Colella P (2004) Defining software requirements for scientific computing

Comparison of AMD Graphics Processing Units (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/w/index.php?title=Comparison_of_AMD_graphics_processing_units&oldid=427053994, Wikipedia

Comparison of Nvidia Graphics Processing Units (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units, Wikipedia

Connection Machine (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Connection_Machine, Wikipedia

Cray-1 Computer System (1976) Cray Research, Inc, Minnesota

CSX700 Datasheet (2011) (06-PD-1425 Rev 1E). http://www.clearspeed.com/products/documents/CSX700_Datasheet_Rev1E.pdf

CSX700 Processor (2011) http://www.clearspeed.com/products/csx700.php

CUDA (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/w/index.php?title=Special:Cite&page=CUDA&id=427059959, Wikipedia

CUDA Case Studies (2009) http://www.lunarc.lu.se/Documents/nvidia-workshop/files/presentation/50_Case_Studies.pdf

CXSL User Guide (2010) (06-RM-1305), p 54. http://support.clearspeed.com/resources/documentation/CSXL_User_Guide_3.1_Rev1.C.pdf

Dolbeau R, Bihan S, Bodin F (2007) HMPP: a hybrid multi-core parallel programming environment. Paper presented at the proceedings of the workshop on general purpose processing on graphics processing units (GPGPU 2007), Boston. http://www.caps-entreprise.com/upload/ckfinder/userfiles/files/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf

Dongarra J, Graybill R, Harrod W, Lucas R, Lusk E, Luszczek P, Tikir M (2008) DARPA's HPCS program: history, models, tools, languages. Adv Comput 72:1–100

Erbacci G, Cavazzoni C, Spiga F, Christadler I (2009) Report on petascale software libraries and programming models. Deliverable 6.6(RI-211528), 163. http://www.prace-project.eu/documents/public-deliverables-1/public-deliverables/d6-6.pdf

ESC Corporation (ed) LDS-1/PDP-10 display system. Evans and Sutherland Computer Corporation, Salt Lake City

EuroBen Benchmark (2011) EuroBen. Accessed 2 May 2011, from http://www.euroben.nl/index.php, EuroBen

Evans and Sutherland (2011a) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Evans_%26_Sutherland, Wikipedia

Evans and Sutherland (2011b) Evans and Sutherland. Accessed 2 May 2011, from http://www.es.com/, Evans and Sutherland

Feldman M (2009) Benchmark challenge: Nehalem versus Istanbul, HPC wire. HCP wire. Accessed from http://www.hpcwire.com/hpcwire/2009-06-18/benchmark_challenge_nehalem_versus_istanbul.html

Flynn's Taxonomy (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Flynn's_taxonomy, Wikipedia

GeForce 256 (2011) NVIDIA corporation. Accessed 2 May 2011, from http://www.nvidia.com/page/geforce256.html, NVIDIA corporation

Gelas JD (2008) Linpack: Intel's Nehalem versus AMD Shanghai. Anandtech. Accessed from http://www.anandtech.com/show/3470

Gelas JD (2010) AMD's 12-core "Magny-Cours" Opteron 6174 versus Intel's 6-core Xeon Anandtech. Accessed 2 May 2011, from http://www.anandtech.com/show/2978, Anandtech

Ghuloum A, Sprangle E, Fang J, Wu G, Zhou Z (2007a) Ct: a flexible parallel programming model for tera-scale architectures. http://software.intel.com/file/25739

Ghuloum A, Smith T, Wu G, Zhou X, Fang J, Guo P, So B, Rajagopalan M, Chen Y, Chen B (2007b) Future-proof data parallel algorithms and software on Intel® multi-core architecture. Intel Technol J 11(4):333–348

Goodyear MPP (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Goodyear_MPP, Wikipedia

GPU Shipments Report by Jon Peddie Research (2011) Jon Peddie Research. Accessed 2 May 2011, from http://jonpeddie.com/publications/market_watch/, Jon Peddie Research
Graphics Processing Unit (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=427152592, Wikipedia
Grochowski E, Annavaram M (2006) Energy per instruction trends in Intel® microprocessors
Hills WD (1989) The connection machine. MIT Press, Cambridge
HMPP Open Standard (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/w/index.php?title=HMPP_Open_Standard&oldid=415481893, Wikipedia
Homberg W (2009) Network specification and software data structures for the eQPACE architecture Jülich supercomputing center (JSC). Accessed 2 May 2011, from http://www2.fz-juelich.de/jsc/juice/eQPACE_Meeting/, Jülich supercomputing center (JSC)
HP Challenge Benchmark Record (2011) University of Tennessee. Accessed 2 May 2011, from http://icl.cs.utk.edu/hpcc/hpcc_record.cgi?id=403, University of Tennessee
HPC Challenge Benchmark Record (2011) University of Tennessee. Accessed 2 May 2011, from http://icl.cs.utk.edu/hpcc/hpcc_record.cgi?id=434, University of Tennessee
Hybrid Multi-Core Parallel Programming Workbench (2011) CAPS enterprise. Accessed 2 May 2011, from http://www.caps-entreprise.com/fr/page/index.php?id=49&p_p=36, CAPS enterprise
IA-32 (Intel Architecture 32-bit) (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/IA-32, Wikipedia
ILLIAC IV (1972) Corporation system characteristics and programming manual. Burroughs corporation
ILLIAC IV (2011a) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/ILLIAC_IV, Wikipedia
ILLIAC IV (2011b) Burroughs corporation. Accessed 2 May 2011, from http://archive.computerhistory.org/resources/text/Burroughs/Burroughs.ILLIAC%20IV.1974.102624911.pdf
Intel 4004 (2011a) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Intel_4004, Wikipedia
Intel 4004 (2011b) A big deal then, a big deal now. Intel corporation. Accessed 2 May 2011, from http://www.intel.com/about/companyinfo/museum/exhibits/4004/facts.htm, Intel corporation
Intel 56XX (2011) Series products (formerly Westemere-EP_). Intel corporation. Accessed 2 May 2011, from http://ark.intel.com/ProductCollection.aspx?codeName=33174, Intel corporation
Intel Hyper-Threading Technology (Intel HT Technology) (2011) Intel Corporation. Accessed 2 May 2011, from http://www.intel.com/technology/platform-technology/hyper-threading/index.htm, Intel corporation
Intel Math Kernel Library (2011) Intel corporation. Accessed 2 May 2011, from http://software.intel.com/en-us/articles/intel-mkl/, Intel corporation
Intel Processor (2011) Clock speed (MHz). Accessed 2 May 2011, from http://smoothspan.files.wordpress.com/2007/09/clockspeeds.jpg
Intel Xeon Processor E5540 (2011) Intel corporation. Accessed 2 May 2011, from http://ark.intel.com/Product.aspx?id=37104&processor=E5540&spec-codes=SLBF6, Intel corporation
Intel(R) Array Building Blocks for Linux OS, User's Guide (2011) (324171-006US), p 74. http://software.intel.com/sites/products/documentation/arbb/arbb_userguide_linux.pdf
Intel(R) Array Building Blocks Virtual Machine, Specification (2011) (324820-002US), p 118. http://software.intel.com/sites/products/documentation/arbb/vm/arbb_vm.pdf
Intel's Ct Technology Code Samples (2010) Intel. Accessed 2 May 2011, from http://software.intel.com/en-us/articles/intels-ct-technology-code-samples/, Intel
Introducing Intel many Integrated Core Architecture (2011) Intel corporation. Accessed 2 May 2011, from http://www.intel.com/technology/architecture-silicon/mic/index.htm, Intel corporation
Introduction to Parallel GPU Computing (2010) Center for scalable application development software
Johnsson L (2011) Overview of data centers energy efficiency evolution. In: Ranka S, Ahmad I (eds) Handbook of green computing. CRC Press, New York

Kanellos M (2001) Intel's accidental revolution. CNET news. Accessed from CNET News website
Kennedy K, Koelbel C, Schreiber R (2004) Defining and measuring the productivity of programming languages. Int J High Perform Comput Appl 18(4):441–448
Kozin IN (2008) Evaluation of ClearSpeed accelerators for HPC, p 15. http://www.cse.scitech.ac.uk/disco/publications/Clearspeed.pdf
Linpack, ClearSpeed (2010) CleerSpeed technology limited. Accessed 2 May 2011, from http://www.clearspeed.com/applications/highperformancecomputing/hpclinpack.php, CleerSpeed technology limited
Matsuoka S, Dongarra J TESLA GPU computint. http://www.microway.com/pdfs/TeslaC2050-Fermi-Performance.pdf
McCalpin JD (2011) STREAM: sustainable memory bandwidth in high-performance computers, University of Virginia. Accessed 2 May 2011, from http://www.cs.virginia.edu/stream/, University of Virginia
McCool MD (2007) RapidMind multi-core development platform. CASCON Cell Workshop
McCool MD (2008) Developing for GPUs, cell, and multi-core CPUs using a unified programming model. Linux J
Memory Bandwidth (STREAM)—Two-Socket Servers (including AMD™ 6100 Series Processors) (2011) Advanced Micro Devices, Inc. Accessed 2 May 2011, from http://www.amd.com/us/products/server/benchmarks/Pages/memory-bandwidth-stream-two-socket-servers.aspx, Advanced Micro Devices, Inc
Mirkovic D, Mahasoom R, Johnsson L (2000) An adaptive software library for fast fourier transforms. Paper presented at the 2000 international conference on supercomputing, Santa Fe
Moore GE (1965) Craming more components onto integrated circuits. Electronics 38(8):114–117
Non-Uniform Memory Access (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access, Wikipedia
NVIDIA Corporation (2011) What is CUDA? Accessed 2 May 2011, from http://www.nvidia.com/object/what_is_cuda_new.html, NVIDIA corporation
OpenCL (2010) Specification Version: 1(1), p 379. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
OpenCL (2011) The open standard for parallel programming of heterogeneous systems. Khronos Group. Accessed 2 May 2011, from http://www.khronos.org/opencl/, Khronos Group
Pentium 4 (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Pentium_4, Wikipedia
Petitet A, Whaley RC, Dongarra J, Cleary A (2008) HPL–a portable implementation of the high-performance Linpack benchmark for distributed-memory computers, University of Tennessee Computer Science Department. Accessed 2 May 2011, from http://www.netlib.org/benchmark/hpl/, University of Tennessee Computer Science Department
Petrov V, Fedorov G (2010) MKL FFT performance—comparison of local and distributed-memory implementations. Intel software network. Retrieved from http://software.intel.com/en-us/articles/mkl-fft-performance-using-local-and-distributed-implementation/
Pettey C (2011) Gartner says worldwide PC shipments in fourth quarter of 2010 grew 3.1 percent; year-end shipments increased 13.8 percent. Accessed from http://www.gartner.com/it/page.jsp?id=1519417, Gartner, Inc
Pettey C, Stevens H (2011) Gartner says 2010 worldwide server market returned to growth with shipments up 17 percent and revenue 13 percent. Gartner, Inc. Accessed 2 May 2011, from http://www.gartner.com/it/page.jsp?id=1561014, Gartner, Inc
PGI Accelerator Programming Model for Fortran and C (2010) p 36. http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf
Phillips E, Fatica M (2010) CUDA accelerated Linpack on clusters, E. Phillips. http://www.nvidia.com/content/GTC-2010/pdfs/2057_GTC2010.pdf
Pollack F (1999) New microarchitecture challenges in the coming generations of CMOS process technologies. Paper presented at the proceedings of the 32nd annual IEEE/ACM international symposium on microarchitecture, Haifa

Portland Group Inc (2011) Accelerated compilers. STMicroelectronics. Accessed 2 May 2011, from http://www.pgroup.com/resources/accel.htm, STMicroelectronics
PRACE (2009) Preparatory phase project, Deliverable 8.3.1, technical component assessment and development, report
PRACE (2011) PRACE. Accessed 2 May 2011, from http://www.prace-ri.eu/, PRACE
Productivity benefits of Intel Ct Technology (2010) Intel corporation. Accessed 2 May 2011, from http://software.intel.com/en-us/articles/productivity-benefits-of-intel-ct-technology/, Intel corporation
RapidMind (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/RapidMind, Wikipedia
Sagar RS, Labarta J, van der Steen A, Christadler I, Huber H (2010) PRACE preparatory phase project, Deliverable 8.3.2, final technical report and architecture proposal. http://www.prace-project.eu/documents/public-deliverables/d8-3-2-extended.pdf
Shalf J, Donofrio D, Oliker L, Wehner M (2006) Green flash: application driven system design for power efficient HPC. Paper presented at the Salishan conference on high-speed computing
Shimpi AL (2010) New westmere details emerge: power efficiency and 4/6 core plans. AnandTech, Inc. Accessed 2 May 2011, from http://www.anandtech.com/show/2930, AnandTech, Inc
Silicon Graphics (2011) Wikipedia. Accessed 2 May 2011, from http://en.wikipedia.org/wiki/Silicon_Graphics, Wikipedia
Simpson AD, Bull M, Hill J (2008) http://www.prace-project.eu/documents/Identification_and_Categorisatio_of_Applications_and_Initial_Benchmark_Suite_final.pdf
Single Chip 4-Bit P-Channel Microprocessor (1987) Intel corporation
Sophisticated Library for Vector Parallelism (2011) Intel array building blocks: a flexible parallel programming model for multicore and many-core architectures. Intel corporation. Accessed 2 May 2011, from http://software.intel.com/en-us/articles/intel-array-building-blocks/, Intel corporation
Team TsG (2005) The mother of All CPU charts 2005/2006. Bestofmedia network. Accessed 2 May 2011, from http://www.tomshardware.com/reviews/mother-cpu-charts-2005,1175.html, Bestofmedia network
Tesla C1060 Computing Processor Board Specification (2010) (BD-04111-001-v06). http://www.nvidia.com/docs/IO/43395/BD-04111-001v-06.pdf
Tesla C2050/C2070 GPU Computing Processor (2010) NVIDIA Corporation
The Green500 (2010) Green 500: ranking the worlds most energy-efficient supercomputers. Accessed 2 May 2011, from www.green500.org, The Green500
Thelen E (2003) The connection machine -1-2-5. Ed-Thelen.org. Accessed 2 May 2011, from http://ed-thelen.org/comp-hist/vs-cm-1-2-5.html, Ed-Thelen.org
Thelen E (2005) ILLIAC IV. Ed-Thelen.org. Accessed 2 May 2011, from http://ed-thelen.org/comp-hist/vs-illiac-iv.html, Ed-Thelen.org
Thornton JE (1963) Considerations in computer design–leading up to the control data 6600. http://www.bitsavers.org/pdf/cdc/cyber/cyber_70/thornton_6600_paper.pdf
Thornton JE (1970) The design of a computer: the control data 6600. Scott, Foresman and Company, Glenview
Top 500 (2011) Top500.org. Accessed 2 May 2011, from http://www.top500.org/, Top500.org
Valich T (2010) nVidia GF100 architecture: alea iacta est. Accessed from http://www.brightsideofnews.com/print/2010/1/18/nvidia-gf100-architecture-alea-iacta-est.aspx
Writing Applications for the GPU Using the RapidMind™ Development Platform (2006) p 7. Accessed from http://www.cs.ucla.edu/palsberg/course/cs239/papers/rapidmind.pdf

Chapter 4
GRAPE and GRAPE-DR

Junichiro Makino

Abstract We describe the architecture and performance of GRAPE-DR (Greatly Reduced Array of Processor Elements with Data Reduction). It operates as an accelerator attached to general-purpose PCs or x86-based servers. The processor chip of a GRAPE-DR board have 512 cores operating at the clock frequency of 400 MHz. The peak speed of a processor chip is 410 Gflops (single precision) or 205 Gflops (double precision). A GRAPE-DR board consists of four GRAPE-DR chips, each with its own local memory of 256 MB. Thus, a GRAPE-DR board has the theoretical peak speed of 1.64 SP and 0.82 DP Tflops. Its power consumption is around 200 W. The application area of GRAPE-DR covers particle-based simulations such as astrophysical many-body simulations and molecular-dynamics simulations, quantum chemistry calculations, various applications which requires dense matrix operations, and many other compute-intensive applications. The architecture of GRAPE-DR is in many ways similar to those of modern GPUs, since the evolutionary tracks are rather similar. GPUs have evolved from specialized hardwired logic for specific operations to a more general-purpose computing engine, in order to meet the perform complex shading and other operations. The predecessor of GRAPE-DR is GRAPE (GRAvity PipE), which is a specialized pipeline processor for gravitational N-body simulations. We have changed the architecture to extend the range of applications. There are two main differences between GRAPE-DR and GPGPU. One is the transistor and power efficiency. With 90 nm technology and 400M transistors, we have integrated 512 processor cores and achieved the speed of 400 Gflops at 400 MHz clock and 50 W. A Fermi processor from NVIDIA integrates 448 processors with 3B transistors and achieved the speed of 1.03 Tflops at 1.15 GHz and 247 W. Thus, Fermi achieved 2.5 times higher speed compared to GRAPE-DR, with 2.9 times higher clock, 8 times more transistors, and 5 times more power consumption. The other is the external memory bandwidth. GPUs typically have the memory bandwidth

J. Makino (✉)
Center for Computational Astrophysics, National Astronomical Observatory of Japan,
2-21-1 Osawa, Mitaka-shi, 181-8588 Tokyo, Japan
e-mail: makino@cfca.jp

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_4,

of around 100 GB/s, while our GRAPE-DR card, with 4 chips, have only 16 GB/s. Thus, the range of application is somewhat limited, but for suitable applications, the performance and performance per watt of GRAPE-DR is quite good. The single-card performance of HPL benchmark is 480 Gflops for matrix size of t 48 k, and for 81 cards 37 Tflops.

4.1 Introduction

In many simulations in astrophysics, it is necessary to solve gravitational N-body problems. To solve the gravitational N-body problem, we need to evaluate the gravitational force on each particle (body) in the system from all other particles in the system. If relatively low accuracy is sufficient, we can use fast solvers such as the Barnes-Hut tree algorithm (Barnes and Hut 1986) or FMM (Greengard and Rokhlin 1987). If high accuracy is required, the direct summation is still the most practical approach. In both cases, the calculation of the gravitational interactions between particles is the most time-consuming part of the calculation.

We can, therefore, greatly speed up the entire simulation, just by accelerating the speed of the calculation of particle-particle interactions. This is the basic idea behind GRAPE computers, as shown in Fig. 4.1. The system consists of a host computer and special-purpose hardware, and the special-purpose hardware handles the calculation of gravitational interactions between particles. The host computer performs other calculations such as the time integration of particles, I/O, and diagnostics.

4.2 History of GRAPE Systems

Figure 4.2 shows the evolution of GRAPE systems and general-purpose parallel computers. GRAPE Project was started in 1988. The first machine we completed, the GRAPE-1 (Ito et al. 1990), was a single-card unit on which around 100 IC and LSI chips were mounted and wire-wrapped. The pipeline unit of GRAPE-1 was implemented with commercially available IC and LSI chips. The design of the pipeline unit of GRAPE-1 is rather unusual. The position vectors are expressed in a 16-bit fixed-point format, and subtraction was done using 16-bit ALU chips. After the subtraction, the result is converted to an 8-bit (1 sign and 7 magnitude bits) logarithmic format, with effective mantissa of 3 bits. This conversion was done by a single EPROM chip.

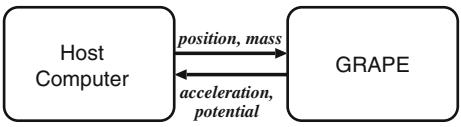


Fig. 4.1 Basic structure of a GRAPE system

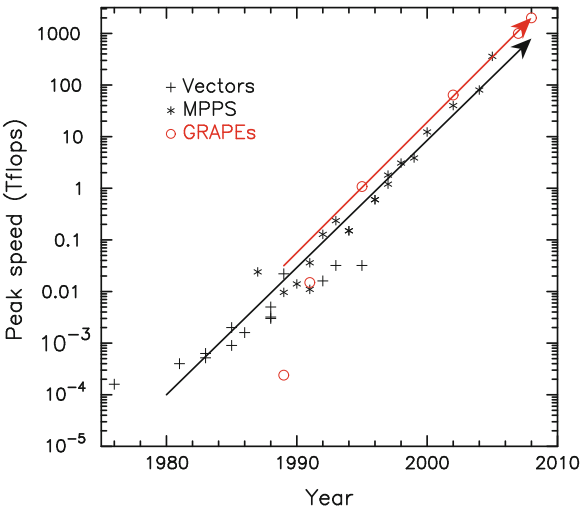


Fig. 4.2 The evolution of GRAPE and general-purpose parallel computers. The peak speed is plotted against the year of delivery. *Open circles*, *crosses* and *stars* denote GRAPEs, vector processors, and parallel processors, respectively

For all operations between two 8-bit words, EPROM chips were used. Thus, we could greatly simplify the hardware design. One EPROM chip can calculate $dx^2 + dy^2$ from dx and dy, and another EPROM chip can calculate dx/dr^3 from dx and dr^2. The accumulation of the force was done in a 48-bit fixed-point format, so that forces from distant particles do not cause underflow.

One might think the effective mantissa of 3 bits must be too short. We have performed theoretical and experimental analysis of error propagation (Makino et al. 1990) and confirmed that this accuracy is sufficient for at least a certain class of important problems.

GRAPE-2 was similar to GRAPE-1A, but with much higher numerical accuracy. In order to achieve higher accuracy, commercial LSI chips for floating-point arithmetic operations such as TI SN74ACT8847 and Analog Devices ADSP3201/3202 were used. The pipeline of GRAPE-2 processes the three components of the interaction sequentially. So it accumulates one interaction in every three clock cycles. This approach was adopted to reduce the circuit size. Its speed was around 40 Mflops, but it is still much faster than workstations or minicomputers at that time.

After GRAPE-2, we started to design custom LSI chips. GRAPE-3 (completed in 1992) and GRAPE-5 (Kawai et al. 2000) are low-accuracy machines similar to GRAPE-1, but with much faster speed. GRAPE-4 (Makino et al. 1997) and 6 (Makino et al. 2003) are high-accuracy machines.

Table 4.1 summarizes the history of GRAPE project. In 1992, we completed GRAPE-2A (Ito et al. 1993), an enhanced version of GRAPE-2 designed to be used for molecular dynamics simulations. MD-GRAPE system (Fukushige et al

Table 4.1 History of GRAPE project

GRAPE-1	(89/4–89/10)	310 Mflops, low accuracy
GRAPE-2	(89/8–90/5)	50 Mflops, high accuracy (32 bit/64 bit)
GRAPE-3	(90/9–91/9)	18 Gflops, high accuracy
GRAPE-4	(92/7–95/7)	1 Tflops, high accuracy
GRAPE-5	(96/4–99/8)	5 Gflops/chip, low accuracy
GRAPE-6	(97/8–02/3)	64 Tflops, high accuracy

1996) with custom LSI design followed GRAPE-2A, and the latest machine, Protein Explorer, Taiji et al. (2003) have reached the peak speed of 1 Pflops in 2006.

We can see that GRAPE and MD-GRAPE systems achieved very high speed. Since the total development cost of each GRAPE system was less than 5 M USD (except for Protein Explorer), they typically achieved the price performance two orders of magnitude better than that of typical general-purpose supercomputers of the same time. GRAPE systems were also extremely power efficient. For example, GRAPE-6, with the peak speed of 64 Tflops, consumed roughly 30 kW. The Earth Simulator, with the peak speed of 40 Tflops, consumed 6 MW. Thus, GRAPE-6 is 300 times more power efficient than the Earth Simulator. Even in 2010, latest x86 processors still require around 2 W per Gflops. So GRAPE-6 is more power-efficient than microprocessors of year 2010 by a factor of 4 or more.

4.3 "Problem" with GRAPE Approach

We have seen that GRAPE systems have achieved price-performance and power efficiency roughly two orders of magnitude better than general-purpose computers of the same time. This success comes from the simple fact that within one GRAPE processor chip, almost all transistors are used for the pipeline processors which perform the evaluation of particle-particle interactions. Thus, the number of arithmetic units in a GRAPE-6 chip exceeds 300, and the number of transistors per arithmetic unit is around 30,000. The Fermi processor from NVIDIA integrates 512 arithmetic unit (adder and multiplier) with 3 billion transistors. Thus, it uses around 3,000,000 transistors per arithmetic unit. The difference between GRAPE-6 and Fermi is about a factor of 100. If we compare GRAPE-6 with latest x86 microprocessors, this ratio is close to 1,000.

For the last two decades, microprocessors has been evolving to more and more complex design, and thus the number of transistors per arithmetic unit has been increasing. The move from CPU to GPU resulted in roughly a factor of 10 reduction in the number of transistors per arithmetic unit, or a factor of 10 increase in the number of arithmetic units in a chip, but they are still less efficient compared to microprocessors 20 years ago, which in turn was already less efficient compared to GRAPE-6 chip by a factor of 30.

If we were to make the follow-up of GRAPE-6 processor using today's 45 nm technology, we could make a chip with 200 pipelines (12,000 arithmetic units) operating at the clock speed of 700 MHz or so, resulting in the speed of around 10 Tflops and power consumption of around 50 W.

However, there is one economical factor. The initial design cost of a custom LSI chip has been increasing exponentially for the last two decades. In 1990, the initial cost for a custom chip was around 100 K USD. By the end of the 1990s, it has become higher than 1 M USD. By 2010, the initial cost of a custom chip is around 10 M USD. Thus, it has become difficult to get a budget large enough to make a custom chip, which has a rather limited range of applications.

One way to reduce the initial design cost is to use FPGA (Field-Programmable Gate Array) chips. An FPGA chip consists of a number of "programmable" logic blocks and also "programmable" interconnection network. A logic block is essentially a small lookup table implemented using static RAM cells, which can implement any combinatorial logic for the input data. Interconnection network is used to implement more complex logic. Recent FPGA chips have more complex design, with larger SRAM blocks and also many small (typically 9 bit) multipliers.

It is possible to implement pipeline processors to FPGA chips. However, because of the programmable design, FPGAs are much more inefficient in transistor usage than full-custom chips. Several successful approaches have been reported (Hamada et al. 1999; Kawai and Fukushige 2006). However, if higher accuracy is necessary, the advantage of FPGA chips becomes small.

One could also use so-called structured ASIC chips, which fall in between FPGAs and full-custom chips. Typically, a chip based on the structured ASIC technology can integrate three to five times more logics compared to fairly large FPGA chips, for much lower replication cost (but with much higher initial cost). Again, it is ideal for low-accuracy pipeline.

The use of GPUs is an obvious alternative by now, but when we started planning the followup project for GRAPE-6 in 2002, it was not clear to which direction GPUs were going, and as we've seen above, GPUs are still quite inefficient in the use of transistors.

4.4 GRAPE-DR

A very different solution for the problem of the high initial cost is to widen the application range by some way to justify the high initial cost. With the GRAPE-DR project (Makino et al. 2007), we followed this approach.

With GRAPE-DR, the hardwired pipeline processor of previous GRAPE systems were replaced by a collection of simple SIMD programmable processors. The internal network and external memory interface was designed so that it could emulate GRAPE processor efficiently and could be used for several other important applications, including the multiplication of dense matrices.

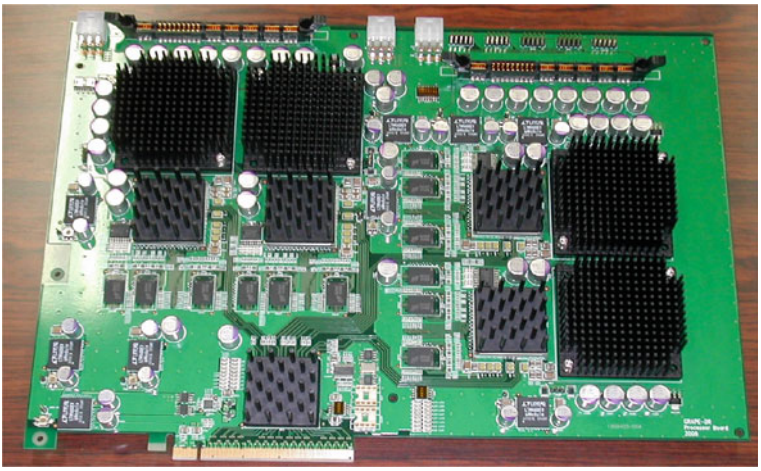


Fig. 4.3 The GRAPE-DR processor board

GRAPE-DR is an acronym of "Greatly Reduced Array of Processor Elements with Data Reduction". The last part, "Data Reduction", means that it has an on-chip tree network which can do various reduction operations such as summation, max/min and logical and/or.

The GRAPE-DR project was started in FY 2004, and finished in FY 2008. The GRAPE-DR processor chip consists of 512 simple processors, which can operate at the clock cycle of 500 MHz, for the 512 Gflops of single precision peak performance (256 Gflops double precision). It was fabricated with TSMC 90 nm process and the size is around 300 mm^2. The peak power consumption is around 60 W. The GRAPE-DR processor board (Fig. 4.3) houses 4 GRAPE-DR chips, each with its own local DRAM chips. The memory bandwidth to DRAM is 4 GB/s per chip, when the clock frequency is 400 MHz. It communicates with the host computer through Gen1 16-lane PCI-Express interface.

If we compare the GRAPE-DR chip with latest (as of 2010) GPU chips, the number of processors are rather similar (both GRAPE-DR and NVIDIA Fermi have 512 processors), and performance is also rather close (256 DP peak for GRAPE-DR and 515 DP peak for Fermi), though latest GPUs use semiconductor technology two and half generation advanced (90 nm for GRAPE-DR and 40 nm for Fermi) and more than 10 times more transistors. Compared to GRAPE-6, GRAPE-DR is roughly a factor of 10 less efficient in transistor usage, but even so it is a factor of 10 more efficient than GPUs. The reason for this difference with GPU is that GRAPE-DR is designed as a collection of simple processors with small register files and local memory, connected with simple networks, so that the arithmetic units occupy a significant fraction of silicone real estate. Thus, the design of GRAPE-DR is not very far from the theoretical limit for the number of transistors per double-precision arithmetic unit (within a factor of five).

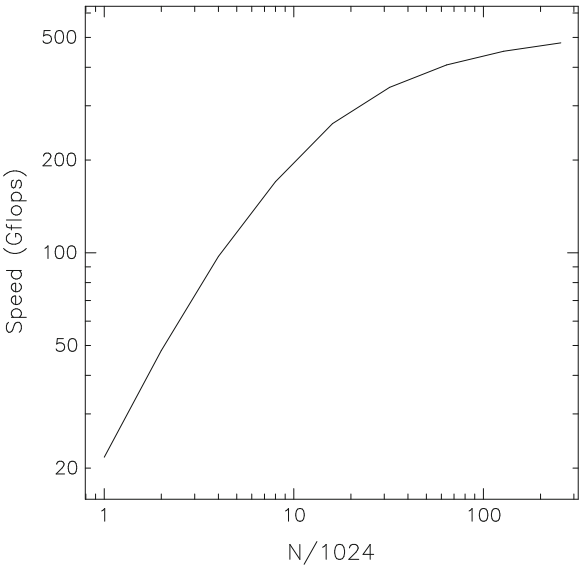


Fig. 4.4 The performance of individual-timestep scheme on single-card GRAPE-DR in Gflops, plotted as a function of the number of particles

This card gives the theoretical peak performance of 819 Gflops (in double precision) at the clock speed of 400 MHz. The actual performance numbers are 740 Gflops for matrix–matrix multiplication, 480 Gflops for LU-decomposition, and 500 Gflops for direct N-body simulation with individual timesteps (Fig. 4.4). These numbers are roughly a factor of two or more better than the best performance number so far reported with GPGPUs of year 2010.

In the case of parallel LU decomposition, the measured performance was 37.44 Tflops on 81-board, 81-node system (Fig. 4.5). The power consumption of this system during the calculation, measured following the rule of Green 500 list, was 25.4 KW, and thus performance per Watt is 1474 Mflops/W. This number is listed as No. 2 in the Green 500 list of Nov 2010. Also, a calculation on 64-node system was listed as No. 1 in the June 2010 "Little" list. The machine which outperformed GRAPE-DR in Nov 2010 list is the prototype of IBM BlueGene/Q. Its first commercial delivery is scheduled to the end of year 2011. Thus, in terms of the power efficiency, GRAPE-DR achieved the world's best performance for the machine in the production use in the year of 2010, with the 90 nm technology which is at least two generations older than what is used on other machines, including GPGPUs.

Thus, from a technical point of view, we believe GRAPE-DR project is highly successful, in making multi-purpose computers with the highest single-card performance and the highest performance per watt.

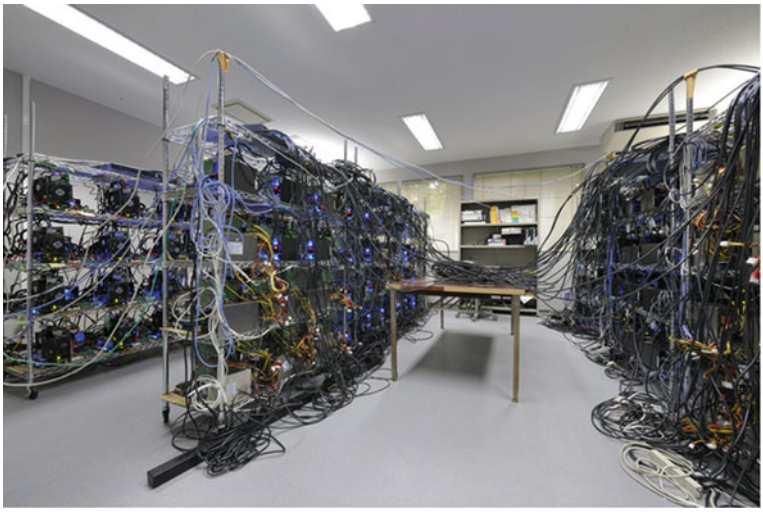


Fig. 4.5 The GRAPE-DR cluster

4.5 Future Directions

Whether or not GRAPE-DR will be competitive with other approaches, in particular GPGPUs, is another question. Though GRAPE-DR is significantly more efficient in transistor usage, the price of GPUs is much lower than that of GRAPE-DR, partly because of much larger production quantities and partly because the pricing strategy of GPU companies. Currently, GPU companies sell GPU cards with huge chips for incredibly low price, and that is the reason why they are attractive to the HPC community. If GPGPUs were designed specifically for HPC applications, its market would be much smaller and thus its per-chip price need to be much higher. Even now, NVIDIA Tesla cards are sold for the price much higher than that of equivalent graphic cards.

Thus, if we or someone else would continue the development of machines similar to GRAPE-DR, they will be able to compete with other HPC platforms such as x86 CPUs and GPGPUs, even if the production volume is much smaller and silicon technology is one or two generations behind.

Another important factor is that with current GPUs, the price of electricity is already higher than the hardware cost. Very roughly, in Japan, the continuous use of one Watt for one year costs one USD. If we use a GPU card with 250 W of power consumption for five years, it would cost 1250 USD, and if we include the cost of air conditioning, it might approach 2000 USD. Thus, whether the price of the card is 500 or 1000 USD does not matter. If something else can provide the performance same as that of GPU for, say, 50 W, even if it costs 2000 USD, the total cost would be less.

If we were to design a followup machine for GRAPE-DR, using 28 nm technology, we can integrate 10 times more arithmetic units than what we have in the current 90 nm chip. We can improve the throughput of double-precision floating-point operations by another factor of two by replacing 25×50 bit multiplier of GRAPE-DR by full 50×50 bit multipliers, without significantly increasing the chip size. With the clock speed kept low (for example 700 MHz), a single chip deliver the peak double-precision performance of 7 Tflops, for the power consumption of around 100 W.

References

Barnes J, Hut P (1986) A hiearchical o(nlogn) force calculation algorithm. Nature 324:446–449

Fukushige T, Taiji M, Makino J, Ebisuzaki T, Sugimoto D (1996) A highly-parallelized special-purpose computer for many-body simulations with an arbitrary central force: Md-grape. ApJ 468:51–61

Greengard L, Rokhlin V (1987) A fast algorithm for particle simulations. J Comput Phys 73:325–348

Hamada T, Fukushige T, Kawai A, Makino J (1999) Prograpе-1: a programmable, multi-purpose computer for many-body simulations. PASJ (submitted)

Ito T, Makino J, Ebisuzaki T, Sugimoto D (1990) A special-purpose n-body machine grape-1. Comput Phys Commun 60:187–194

Ito T, Makino J, Fukushige T, Ebisuzaki T, Okumura SK, Sugimoto D (1993) A special-purpose computer for n-body simulations: Grape-2a. PASJ 45:339–347

Kawai A, Fukushige T (2006) $158/gflop astrophysical n-body simulation with a reconfigurable add-in card and a hierarchical tree algorithm, 2006

Kawai A, Fukushige T, Makino J, Taiji M (2000) Grape-5: a special-purpose computer for n-body simulations. PASJ 52:659–676

Makino J, Ito T, Ebisuzaki T (1990) Error analysis of the grape-1 special-purpose n-body machine. PASJ 42:717–736

Makino J, Taiji M, Ebisuzaki T, Sugimoto D (1997) Grape-4: a massively parallel special-purpose computer for collisional n-body simulations. ApJ 480:432–446

Makino J, Fukushige T, Koga M, Namura K (2003) GRAPE-6: massively-parallel special-purpose computer for astrophysical particle simulations. PASJ 55:1163–1187

Makino J, Hiraki K, Inaba M (2007) Grape-dr: 2-pflops massively-parallel computer with 512-core, 512-gflops processor chips for scientific computing. In: Proceedings of SC07. ACM, 2007 (Online)

Taiji M, Narumi T, Ohno Y, Futatsugi N, Suenaga A, Takada N, Konagaya A (2003) Protein explorer: a petaflops special-purpose computer system for molecular dynamics simulations. In: The SC2003 proceedings, pages CD-ROM, 2003. IEEE, Los Alamitos

Part III
Software Libraries

Chapter 5
PARRAY: A Unifying Array Representation for Heterogeneous Parallelism

Yifeng Chen, Xiang Cui and Hong Mei

Abstract This paper introduces a programming interface called PARRAY (or Parallelizing ARRAYs) that supports system-level succinct programming for heterogeneous parallel systems like GPU clusters. The current practice of software development requires combining several low-level libraries like Pthread, OpenMP, CUDA and MPI. Achieving productivity and portability is hard with different numbers and models of GPUs. PARRAY extends mainstream C programming with novel array types of the following features: (1) the dimensions of an array type are nested in a tree structure, conceptually reflecting the memory hierarchy; (2) the definition of an array type may contain references to other array types, allowing sophisticated array types to be created for parallelization; (3) threads also form arrays that allow programming in a Single-Program-Multiple-Codeblock (SPMC) style to unify various sophisticated communication patterns. This leads to shorter, more portable and maintainable parallel codes, while the programmer still has control over performance-related features necessary for deep manual optimization. Although the source-to-source code generator only faithfully generates low-level library calls according to the type information, higher-level programming and automatic performance optimization are still possible through building libraries of sub-programs on top of PARRAY. The case

Y. Chen (✉) · X. Cui · H. Mei
HCST Key Lab School of EECS, Peking University, Beijing 100871,
People's Republic of China
e-mail: cyf@pku.edu.cn

X. Cui
e-mail: cuixiang08@sei.pku.edu.cn

H. Mei
e-mail: meih@pku.edu.cn

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_5,

study on cluster FFT illustrates a simple 30-line code that 2×-outperforms Intel Cluster MKL on the Tianhe-1A system with 7168 Fermi GPUs and 14336 CPUs.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming · D.3.3 [Programming Languages]: Language Constructs and Features—Concurrent programming structures · D.3.4 [Programming Languages]: Processors—Code generation

General Terms Languages · Performance · Theory

Keywords Parallel programming · Array representation · Heterogeneous paralelism · GPU clusters

5.1 Introduction

Driven by the demand for higher performance and lower hardware and energy costs, emerging supercomputers are becoming more and more heterogeneous and massively parallel. Several GPU-accelerated systems are now ranked among the top 20 fastest supercomputers. Despite the rapid progress in hardware, programming for optimized performance is hard.

The existing programming models are designed for specific forms of parallelism: Pthread and OpenMP for multicore parallelism, CUDA and OpenCL for manycore parallelism, and MPI for clustering. A simple combination of these low-level interfaces does not provide enough support for software productivity and portability across different GPU clusters with varied numbers/models of GPUs on each node. A current common practice is to combine MPI and CUDA. However on a cluster of GPU-accelerated multicore nodes, the number of MPI processes cannot be the number of CPU cores (to use the CPU cores) and the number of GPUs (to use GPUs) at the same time. A seemingly obvious solution is to use the number of GPUs for MPI processes and use OpenMP to control CPU cores, but the complexity of programming with all MPI, CUDA and OpenMP will discourage most application developers.

Such difficulties have led to a variety of new ideas on programming languages (more detailed comparisons in Sect. 5.5). Language design is a tradeoff between abstraction and performance. For high-performance applications, the concern of performance is paramount. We hence ask ourselves a question:

how abstractly can we program heterogeneous parallel systems without introducing noticeable compromises in performance?

The design of PARRAY follows the approach of bottom-up abstraction. That means if a basic operation's algorithm or implementation is not unique (with considerable performance differences), the inclination is to provide more basic operations at a lower level. Our purpose is not to solve all the programmability issues but to provide a bottom level of abstraction on which performance-minded programmers can

directly write succinct high-performance code and on which higher-level language features can be implemented without unnecessary performance overheads. This kind of "performance transparency" allows other software layers to be built on top of this layer, and the implementation will not be performance-wise penalized because of choosing PARRAY instead of using low-level libraries directly. A programmer can then choose the right programming level to work with.

A common means to achieve abstraction is to adopt a unifying communication mechanism, e.g. synchronous message passing in process algebra (Hoare 1985), distributed memory sharing or Partitioned Global Address Space (PGAS). However, heterogeneous parallel systems are often equipped with different kinds of hardware-based communication mechanisms such as sequentially-consistent shared memory for multicore parallelism, asynchronous message-passing or RDMA for clustering, inconsistent shared memory (with explicit but expensive consistency-enforcing synchronization) for manycore parallelism, as well as PCI data transfer between servers and their accelerators. These mechanisms exhibit significantly different bandwidth, latency and optimal communication granularity (i.e. the size of contiguous data segments).

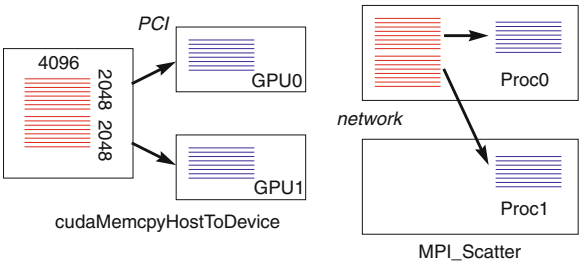
Encoding all data-transfer operations of a source program into a unified communication pattern does not always yield efficient code. For example, programs with assumption of a global-address space (either Distributed Shared Memory or PGAS) tend to issue data-transfer commands when data are needed for computation, but a better strategy may instead use prefetching to better overlap communication and computation. Another challenge is to maximize communication granularity. Shared-memory programs tend to issue individual data-access commands in the actual code for computation, but communication channels such as PCI (between CPU and GPU) and Infiniband (among nodes) require granularity to reach a certain level to achieve peak bandwidths. The source-program information about granularity is easily lost with global-address accesses. Compile-time and runtime optimization alleviates the issues but it remains to be seen to what extent such lost source information is recoverable automatically.

In this paper, we take a different perspective and do not attempt to devise any unifying memory model. Instead we invent novel types that unify the representation of different communication mechanisms. It allows programmers to specify what needs to be done in a communicating operation, and the compiler will then generate the corresponding library calls. This is not restricted to generating hardware-based communication calls. If the required data transfer is one-sided and there exists a well optimized PGAS protocol, the compiler simply generates PGAS calls (Nieplocha et al. 1996) instead.

How can we uniformly represent such a diverse variety of communication mechanisms?

The answer is to identify their common unifying mathematical structure. To get a glimpse of the intended representational unification, consider an example to partition a 4096×4096 row-major array in the main host memory (hmem) to two 2048×4096 blocks, each being copied to a GPU device memory via PCI. Experiments show that

the overall bandwidth of two parallel PCI transfers is 1.6x of single PCI transfers. The two CPU multicore threads that control GPUs are therefore programmed to invoke `cudaMemcpy` simultaneously, followed by inter-thread synchronization. Interestingly similar communication patterns arise elsewhere e.g. in collective communication `MPI_Scatter` that scatters the two-block source array to two MPI processes. This is illustrated in the following figures.



Although the two seemingly disparate communication patterns describe data transfers of different library calls (CUDA and MPI), via different media and to different memory devices, they do share the same logical pattern of data partition, distribution and correlation. Such similarities motivate us to unify them as one communication primitive and treat the data layouts of communication's origin and destination as typing parameters.

There have been a large body of research on arrays (e.g. HTA (Ganesh et al. 2006)), but the existing array notations are not expressive and flexible enough to reflect the complex memory hierarchy of heterogeneous parallel systems, control various system-level features, unify sophisticated communication patterns including dimensional transposition, noncontiguous transfer, uncommon internode connections etc., and convey enough source information to the compiler to generate performance-optimal code. The solution is to increase representational expressiveness.

PARRAY adopts a new array type system that allows the programmer to express additional information about parallelization that can guide the compiler to generate low-level library calls. The following features are claimed to be original contributions not seen in other array notations that we know about:

1. Dimension tree
   The dimensions of an array type are nested in a tree, conceptually reflecting the memory hierarchy. Unlike HTA's dimension layers that have default memory types, PARRAY's dimension tree is logical and independent of any specific system architecture. The memory-type information is specifiable for each sub-dimension. Such flexibility allows dimensions to form a hierarchy within individual memory devices.
2. Array-type references
   The definition of an array type may contain references to other array types' definitions. This allows sophisticated array types to be created for transposition, partition, distribution and distortion. Such types form an algebraic system with a complete set of algebraic properties (see Sect. ??).

3. Thread arrays
   Threads also form arrays. A thread array type indicates the kind of processor on which the threads are created, invoked and synchronized. The SPMD codes of different thread arrays are compacted in a nested-loop-like syntactical context (Single-Program Multiple-Codeblocks or SPMC) so that the order of commands in the nested context directly reflects that of the computing task. An array type may consist of a mixture of dimensions that refer to data and thread array types and represent the distribution of array elements over multiple memory devices. A communication pattern is represented as the pairing between such types.

PARRAY essentially organizes various array types in a unifying mathematical framework, but:

how do we know to what extent the basic framework's design is already expressive enough and unnecessary to be extended for future applications and architectural changes?

The answer derives from a theoretical analysis showing that any location-indexing expression (called *offset expression*) for array elements is representable with PARRAY types, as long as it just consists of integer (independent) expressions, multiplication, division and modulo operators, and additions and compositions between expressions. Such level of expressiveness is not seen in any existing array notations.

PARRAY is implemented as a preprocessing compiler that translates directives into C code and macros. As the array types already convey detailed information about the intended communication patterns, the code generation is straightforward and faithful in the sense that the compiler need not second-guess the programmer's intention, and hence little runtime optimization is required. The compiler does generate various conditional-compilation commands, which are usually optimized by the underlying C compiler without causing performance overheads. In a sense, the PARRAY's data-transfer command is like an extremely general MPI collective that invokes the actual communication libraries according to the type parameters. Little overhead-inducing code is generated before or after the invocations. Paradoxically, by not relying on runtime optimization, PARRAY can guarantee performance (for well-coded programs) and form a performance-transparent portable layer of abstraction.

Section 5.2 introduces PARRAY notation. Section ?? studies the mathematical foundation of PARRAY; Sect. 5.3 explains the rationale behind the implementation including the concept of SPMC; Sect. 5.4 investigates into a case study on large-scale FFT; Sect. 5.5 reviews previous works.

5.2 Parallelizing Array Types

This section introduces the PARRAY type notation on which the actual programming syntax is based.

5.2.1 Dimension Tree

Let us first look at a simple definition:

$$A \mathrel{\widehat{=}} \texttt{pinned float}[[3][2]][4].$$

The type describes $3 \times 2 \times 4 = 24$ floating-point elements in three dimensions, but it is also a two-dimensional array type with 6×4 floats and or a one-dimensional array type with 24 floats. The indicated memory type is "`pinned`" which denotes the main host memory allocated by CUDA for fast DMA with GPU.

Note that in the actual code generation, the size of a dimension can be any arbitrary integer expression of C.

As slightly simplified code generation, the compiler translates the *offset expression* (i.e. mapping from indices to element offsets) $A[[x][y]][z]$ into a C expression $x * 8 + y * 4 + z$, $A[x][y]$ into expression $x * 4 + y$, and $A[x]$ into the index x itself. Such offset expressions are used for row-major element accesses in program.

Let the *partial array type* A^0 denote the left subtree of A, i.e. a 2D type of size 3×2, and A^1 denote the right 1D subtree of size 4. The compiler will translate $A^0[x][y]$ into $x * 8 + y * 4$ (instead of $x * 2 + y$)!

5.2.2 Type Reference

A dimtree may contain references to other array types. The following device-memory (or dmem) array type

$$B \mathrel{\widehat{=}} \texttt{dmem float}[[\#A^{01}][\#A^{00}]][\#A^1]$$

contains references to A with the sub-dimensions of the left subtree transposed. Their offset expressions satisfy an equation:

$$B[[x][y]][z] = A[[y][x]][z].$$

Unlike the elements of A, those of type B are not contiguously laid out in memory: neither row-major as C convention nor column-major as Fortran. Many sophisticated array layout patterns correspond to the combined uses of dimtree and type references.

5.2.3 Data Transfer

To represent data transfer from an array of type A to an array of type B, we use the following notation:

$$B \leftleftarrows A.$$

The data transfer involves two real array objects in an actual program and describes the operation to copy every element of a type-A array at location $A[i]$ to an element of a type-B array at $B[i]$. In practice, as the source array is located in hmem and the target array in dmem, the compiler will generate:

`cudaMemcpy`(..., `cudaMemcpyHostToDevice`)

according to the memory types. Other low-level data transfers such as `memcpy` of C, message passing `MPI_Send`/`MPI_Recv`, collectives `MPI_Alltoall` and so on are generated if the memory types and the dimensions conform to other pairing patterns.

As the offset expressions $A[i]$ and $B[i]$ are not equal, the copying cannot be achieved in a single `cudaMemcpy`. A loop is hence needed to cudaMemcpy every 4 contiguous elements from the starting location $A^0[i]$ in the host memory to the starting location $B^0[i]$ in GPU's device memory as a partially contiguous transfer.

Whether the compiler generates one Memcpy command or a loop of Memcpys depends on the array type's contiguity, which the compiler will generate extra code to check.

5.2.4 Threads Arrays

The following type describes two (Pthread) CPU threads:

$$P \mathrel{\widehat{=}} \texttt{pthd}[2].$$

Here `pthd` is called a *thread type*. No element type is required. Other possible thread types include `mpi` for MPI processes and `cuda` for GPU threads. For example, the type

$$M \mathrel{\widehat{=}} \texttt{mpi}[2]$$

describes two MPI processes. A GPU thread array type is two-dimensional: the column dimension indicates the blocks in a grid, while the row dimension indicates the threads in every block. For example, the following array type

$$C \mathrel{\widehat{=}} \texttt{cuda}[N/256][256]$$

describes N GPU threads of which every 256 threads form a CUDA block. Here N can be any C expression (possibly including runtime variables). The compiler will generate code that lexically includes the size expressions. CUDA thread-array dimensions may contain sub-dimensions just like CUDA's grids and blocks.

5.2.5 Distributed Array Types

If an array type has memory or thread type, its references to other array types only affect the offset expressions, but if that is absent, it becomes *mixed* or *distributed* and are often useful in collective data transfers. Consider an array type in ordinary `paged` memory:

$$H \mathrel{\widehat{=}} \texttt{paged float}[2][[2048][4096]]$$

and another type half of its size in device memory $D \mathrel{\widehat{=}} \texttt{dmem float2}\,\#H^1$. The mixed type

$$[\#P][\#D]$$

logically has the same number of elements as H but does not describe array stored in a single memory device. Instead it is distributed over the threads of P and stored in the GPU device memory associated with each CPU control thread.

5.2.6 PCI Scattering

The communication pattern characterized by the type pairing

$$[\#P][\#D] \leftleftarrows H$$

describes data copying from every element at location $H[i][j]$ in ordinary paged hmem to the element at location $D[j]$ in the dmem of the GPU that is associated with the control thread $P[i]$. The above communication pattern uses two parallel CPU threads to scatter hmem data to the dmem of two GPUs.

5.2.7 MPI Scattering

The following communication pattern, on the other hand, collectively scatters the source data to two MPI processes:

$$[\#M][\#H^1] \leftleftarrows H.$$

In practice, a communication pattern may generate different code for synchronous, asynchronous, or one-sided communications.

The communication's mode is determined by the programmer. If that is one-sided, the compiler can generate PGAS code and take advantage of any available runtime optimization (Nieplocha et al. 1996).

5.2.8 Other MPI Collectives

Other MPI collective communications such as Alltoall and Gather have similar a type representation. For example, the following communication pattern describes MPI's Alltoall collective:

$$[[\#H^0][\#M]][\#H^1] \leftrightarrows [[\#M][\#H^0]][\#H^1]$$

where each element at location $H[j][k]$ on process $M[i]$ is copied to the location $H[i][k]$ on process $M[j]$. It effectively swaps the column dimension of H with the distributed "virtual" dimension M. The compiler detects this pattern and generates the `MPI_Alltoall` command based on the fact that the array type H is contiguous and conforms to the semantics of the MPI command.

Unsurprisingly, MPI gathering accords with the converse communication pattern of MPI scattering:

$$H \leftrightarrows [\#M][\#H^1].$$

5.2.9 Non-MPI Collectives

Some PARRAY communication patterns do not correspond to any single standard MPI collective. Consider 2x2 four MPI processes:

$$M' \mathrel{\widehat{=}} \texttt{mpi[2][2]}$$

The following pairing of types describes two separate groups of MPI processes performing Alltoall within each individual group:

$$[[\#M'^0][\#H^0][\#M'^1]][\#H^1] \leftrightarrows [[\#M'^0][\#M'^1][\#H^0]][\#H^1].$$

The pid row dimension M'^1 and the data column dimension H^0 are swapped by the communication.

What gives rise to non-MPI collectives includes not only non-standard inter-process connectivity but also discontiguity in process-to-process communication. In the large-scale 3D FFT algorithm (Chen et al. 2010) developed for turbulence simulation, a distributed array of size up to 14336^3 is transposed over the entire Tianhe-1A GPU cluster with 7168 nodes (see Sect. 5.4). The algorithmic optimization requires discontiguous data transfer between processes to adjust the displacements of elements during communication so that main-memory transposition can then be avoided. The data array of complex numbers (`float2`) on every node has the following type:

$$G \mathrel{\widehat{=}} \texttt{pinned float2 [2][[7168][2]][14336].}$$

The array type for 7168 MPI processes is defined:

$$L \mathrel{\widehat{=}} \texttt{mpi}[7168].$$

The required discontiguous Alltoall communication pattern (with adjusted displacements) corresponds to the following pairing types:

$$[[\#G^1][\#L][\#G^0]][\#G^2] \leftrightharpoons [[\#L][\#G^0][\#G^1]][\#G^2]. \tag{5.1}$$

The types have two dimensions. Only the row dimensions are contiguous. The dimension L is swapped with G^1 instead of G^0. That means the starting locations of communicated data are different from those of the standard Alltoall. As the communication granularity i.e. the size of G^2 already reaches 14×8 KB, its performance should be very close to that of a standard Alltoall, despite the fact that no such displacement-adjusting collective exists.[1]

The above examples have illustrated the advantage of adopting a unifying representational framework such that the communication library need not keep adding *ad hoc* collectives to suit originally unforeseen communication patterns that arise from new applications and emerging architectures.

Because all necessary information has been represented in the types, the generated code is as efficient as the underlying MPI implementation. It is also possible to skip the MPI layer and directly generate Infiniband's IB/verbs invocation with less overheads.

5.2.10 Detecting Contiguity

How does the compiler know a communication pattern corresponds to MPI's Alltoall collective or any other sub-routines? This is achieved by checking the memory/thread types of the dimensions and generating C boolean expressions that can check whether the concerned dimensions are *contiguous* in the sense that the offsets of such a dimension are linearly ordered in memory and located adjacently to each other. Usually contiguity-checking expressions are determined in compile-time and induce no performance overheads. Details of this question are beyond our agenda.

5.2.11 Syntax

Let $e, e_0, e_1, \cdots$ denote index expressions in C (variables allowed), the symbols $U, U_0, U_1, \cdots$ denote array types, $S, S_0, \cdots$ be multi-dimensional types. Array types have the following simple syntax:

[1] This communication pattern is equivalent to a loop of asynchronous Alltoallv collectives followed by a global synchronization. Such implementation is possible owing to more recent MPI development (Kandalla et al. 2011).

$$S ::= [U_0] \cdots [U_{n-1}]$$
$$U ::= e \mid \texttt{disp}\ e \mid S \mid U\#U \mid U^s \mid \texttt{func}(x)\ e.$$

where s is a path sequence of dimension indices to refer a sub-dimension deep in a dimension tree, and $(U^{s_1})^{s_2}$ is considered the same as $U^{s_1 s_2}$. A C expression e describes a 1D array type of size e. The displacement type (`disp` e) describes a dimension of size 1 with offset e. A dimension may have multiple (up to 10 in practice) sub-dimensions. The dimension size of a type reference $U\#V$ coincides with that of U, while V refers to another type that describes the offset expression. A functional offset expression `func`$(x)\ e$ describes a dimension of size 1 with a C offset macro. Functional expressions, as user-defined offset mappings, further strengthen expressiveness but will not be considered in our theoretical analysis.

5.2.12 Informal Rules

Every array-access expression can be decomposed into unary offset expressions called *offset functions*. For example:

$$\begin{aligned} A[[x][y]][z] &= A^{00}[x] + A^{01}[y] + A^{1}[z] \\ &= 8(x \bmod 3) + 4(y \bmod 2) + (z \bmod 4). \end{aligned}$$

That means we only need the definition for every offset function. Another such example is $A^0[x] = 4(x \bmod 6)$. The above semantic definition uses modulo operator. *In the actual implementation the programmer is required to ensure a safe range for every index expression, and the modulo operators are not always generated.*

Consider a type $E \mathrel{\widehat{=}} [2][2]A^0$. The array size is 2×2, but the offset function of E follows that of A^0. The offset function of E^0 is derivable top-down from that of A^0:

$$E^0[x] = A^0[2(x \bmod 2)] = 8(x \bmod 2). \tag{5.2}$$

Then consider the previous example B^0. As its sub-dimensions B^{00} and B^{01} refer to A^{01} and A^{00} respectively, the offset of B^0 is computed bottom-up by decomposing the index into separate indices of its sub-dimensions:

$$\begin{aligned} B^0[x] &= A^{01}[x \operatorname{div} 3] + A^{00}[x] \\ &= 4((x \operatorname{div} 3) \bmod 2) + 8(x \bmod 3). \end{aligned} \tag{5.3}$$

The above two cases are intuitive and common in practice. The general array representation, however, allows more sophisticated cases where a dimension itself does not refer to other types but both its parent dimension and some sub-dimensions contain type references. The adopted rule is to first compute top-down according to case (5.2), and then bottom-up according to case (5.3).

5.2.13 Displacement and Index Range

Displacement is a type notation not mentioned in the previous section. It is useful to shift an offset function. In practice we often use $(n..m) \mathrel{\widehat{=}} (m-n+1)\,\#\,(\texttt{disp}\ n)$ to represent a dimension with a range of indices.

The case study in Sect. 5.4 partitions the 3D data in hmem into two-dimensional slices and use GPU to compute the FFT for every slice separately. The type for a slice in dmem is characterized as

$$Q \mathrel{\widehat{=}} \texttt{dmem float2}\ [N][N],$$

while one of the slices with displacement in hmem is typed as

$$F \mathrel{\widehat{=}} \texttt{pinned float2}\ [\texttt{disp}\ i][N][N].$$

Then the type pairing $Q \leftrightarrows F$ describes contiguous data transfer of size N^2 from hmem to dmem. The starting location of transfer in hmem is $(N^2 * i)$. We may also declare a smaller two-dimensional "window" in the middle of Q:

$$[4..(N-5)\,\#\,Q^0][4..(N-5)\,\#\,Q^1].$$

Such window types are useful in stencil computation. Cyclic displacement is also easily representable.

5.3 Implementation

This section describes how PARRAY is implemented and the rationale behind it.

5.3.1 Data Array Types

PARRAY is implemented as a C preprocessor that generates CUDA, MPI, and Pthread code and a basic library of sub-programs including the general `DataTransfer` command. The preprocessor is detached from the C compiler to maximize cross-platform compatibility and insensitivity to the constant upgrades of hardware and system-level software. That means only directive errors are caught by the preprocessor. C compilation errors are only detectable in the generated code. The following table compares PARRAY notation and the corresponding program syntax.

The following example shows how an array `a` is declared, created, initialized, accessed and freed in the end:

Notation	Program Syntax
A^{01}	`A_0_1`
$\langle A \rangle$	`$dim(A)$`
A[5][2]	`$A[5][2]$`
$A \mathrel{\widehat{=}}$ `pinned float[[3][2]][4]`	`# parray {pinned float[[3][2]][4]} A`
$\leftleftarrows$	`DataTransfer`

```
#parray {pinned float[[3][2]][4]} A
float* a;
#create A(a)
printf("array access: %d\n", a[$A[5][2]$]);
#destroy A(a)
```

It first declares an array type `A` in pinned memory. The command `#create A(a)` then allocates the pinned memory to the pointer. Note that `#parray` only defines an array type. An actual array is a C pointer that allows multiple typing views as long as the array is a pointer of the element type.

Other memory types include `paged` memory that is managed by the operating system only reaching about 60 % the bandwidth of pinned memory for data transfer with GPU device memory (which is denoted by the keyword `dmem`). The keyword `smem` stands for shared memory in GPU, `rmem` for GPU registers (allocated as direct array declaration in GPU kernels), and `mpimem` for MPI-allocated page-lock DMA-able buffer memory. Mellanox and Nvidia's GPU-Direct technology makes `pinned` and `mpimem` interoperable.

The following table lists the actual library calls for memory allocation. Thread types will be explained in Sect. 3.2.

	#create	#destroy	Library
paged	malloc	free	C/Pthread
pinned	cudaMallocHost	cudaFreeHost	CUDA
mpimem	MPI_Alloc_mem	MPI_Free_mem	MPI
dmem	cudaMalloc	cudaFree	CUDA
smem	…shared…		CUDA
rmem			CUDA

It is recommended that all index expressions and data transfers should use array notation (instead of native C notation) so that when an array type is modified, all corresponding expressions and library calls will be updated automatically by the compiler over the entire program. The following code declares a type `B` in dmem by referring to `A`'s dimensions and transfers data between their arrays:

```
#parray {dmem float[[#A_0_1][#A_0_0]][#A_1]} B
float* b;
#create B(b)
```

```
#insert DataTransfer(b, B, a, A){}
```

The sub-program `DataTransfer` automatically detects that the dimensions `A_1` and `B_1` are contiguous, and cudaMemcpy data from the main memory to GPU device memory in a loop of 6 segments, each with 4 floats.

The following table lists the library calls used between memory types. `hmem` refers to `paged`, `pinned` and `mpimem` all in the main memory but allocated for different purposes. Data transfer from or to GPU's shared memory `smem` or GPU's registers `rmem` is always performed element-by-element within a loop.

from\to	hmem	dmem	smem/rmem
hmem	memcpy	cudaMemcpy	
dmem	cudaMemcpy	cudaMemcpy	C loop
smem/rmem		C loop	C loop

5.3.2 Thread Array Types

SPMD is adopted by a wide range of parallel languages. The idea of SPMD programming is that one code is executed on multiple homogeneous parallel threads (or processes), though at any time different threads may be executing different commands of the code.

SPMD alone does not work for heterogeneous parallelism. For example, FFT on a GPU cluster starts multiple MPI process, each of which initiates several threads to control GPUs on that node, and each thread then launches thousands of GPU threads. Another example is GPU-cluster's Linpack code (Fatica et al. 2009), which performs DGEMM on CPU and GPU threads at the same time.

In the existing explicitly parallel languages, the code segments for different processes are declared separately. The interaction between such code segments requires explicit matching synchronization command. The codeblocks of thread arrays, however, are statically nested in the same program context. The control flow may deviate from one thread array whose code is in an outer codeblock to another thread array whose code is in its immediate inner block and return after the execution of the inner block. The compiler will later extract the codeblocks and sequentially stack those from the same thread array to form a separate SPMD code in which matching synchronization and communication commands are automatically inserted. Thus SPMC (or *Single Program Multiple Codeblocks*) is like a compile-time RPC: the control flows travel across different thread arrays. The purpose is to imply the dynamic control flow as much as possible through the static structure of code and helps the programmer "visualize" the interactive pattern of the control flows among different thread arrays. Each thread array is a SPMD unit consisting of multiple homogeneous processes sharing the same code. A SPMC program may consist of multiple *thread arrays*, each as an array of homogenous threads sharing the same codeblock. The codeblocks of different thread arrays are nested in "one single program".

The following example declares and creates an array of 2 CPU threads and then triggers them to run.

```
#parray {pthd[2]} P
#detour P {printf("thread id %d\n", $tid$);}
```

The expression `$tid$` returns the thread id of the current running thread. The following table lists the actual library calls for creating and freeing thread arrays. The code inside a detour can access global variables as it is generated as a global C function or CUDA kernel in the global context.

	`#create`	`#destroy`	Library
`pthd`	`pthread_create`	`pthread_join`	Pthread
`mpi`	`MPI_Comm_split`	`MPI_Intercomm_merge`	MPI
`cuda`			CUDA

The callee codeblock will be extracted by the compiler separately. If in a program there are two detour commands to the same type of thread array, the callee codeblocks as well as the inserted synchronization commands will be *statically* piled up in the generated C function according to their syntactical order in the source program.

5.3.3 Sub-Programming

A sub-program is like a general C macro function in which array types as well as program codes can be passed as arguments, sometimes achieving surprising flexibility. The following simple code, despite its appearance, directly corresponds to a CUDA kernel that performs general copying within the device memory of a GPU.

```
#subprog GPUDataTransfer(t, T, s, S)
   #parray {cuda($elm(T)$* t,$elm(S)$* s)
                [$dim(S)$/256][256] } C
   #detour C(t,s){ t[$T[$tid$]$]=s[$S[$tid$]$]; }
#end
```

The arguments `T` and `S` are input types assumed to have an equal (possibly variable) size that is a multiple of 256. The expressions `$elm(T)$` and `$elm(S)$` extract the element types from the input types. The CUDA thread array `C` declares as many threads as the elements with every 256 threads forming a block. The overall thread id `$tid$` here combines CUDA block id and thread id. The inputs `t` and `s` are passed as actual arguments into the CUDA kernel. Every thread copies an element of array `s` at the location `$S[$tid$]$` to the location `$T[$tid$]$` of array `t`. A sub-program is invoked by the command `#insert`. For example the following code declares and creates two arrays of type `Q` in dmem and contiguously copies elements of array *s* to array *t*.

```
#parray {dmem float[N][N]} Q
float* s; #create Q(s)
float* t; #create Q(t)
#insert GPUDataTransfer(t,Q,s,Q)){}
```

This sub-program' effect is the same as

```
cudaMemcpy(, , , cudaMemcpyDeviceToDevice)
```

as both input types `T` and `S` of the sub-program are contiguous.

With different input types, the sub-program `GPUDataTransfer` may perform a different operation with entirely different generated kernel codes. For example, the following added code effectively performs an out-place array transposition with the row and column dimensions swapped.

```
#parray {dmem float[#Q_1][#Q_0]} R
float* u; #create R(u)
#insert GPUDataTransfer(u,R,s,Q)){}
```

Both data copying and transposition share the same PARRAY code. No existing array representation supports this kind of code reuse. PARRAY sub-programs are insensitive to the layout of the input and output arrays as long as the layout information is described correctly in the declared types. In comparison, a typical function SGEMM for matrix multiplication in the basic math library BLAS has a rather clumsy signature:

```
void sgemm(char transa, char transb,
           int m, int n, int k, ......);
```

where `transa` (or `transb`) is either `'n'` indicating the first array to be row-major or `'t'` being column-major. Other memory layouts are not representable with the BLAS signature. PARRAY's sub-programming mechanism is therefore more flexible than C functions. Like C macros, the price paid for such flexibility is to re-compile a sub-program on every insertion and generate code according to the type arguments.

In fact the general `DataTransfer` command is also a sub-program with the same signature as `GPUDataTransfer`. Its implementation is a series of compile-time conditionals that check the type arguments for structure, memory type and dimension contiguity . The conditional branches eventually lead to various specialized sub-programs like `GPUDataTransfer`.

`GPUDataTransfer` itself too is subject to more specialized optimization. For example, for GPU transposition from Q to R with contiguous column `R_0` and discontiguous `R_1`, shared memory can be used to coalesce both read and write. Other well-known optimization techniques from the CUDA SDK can further minimize bank conflict within shared memory and dmem.

5.4 Case Study: Large-Scale FFT

For *small-scale* FFTs whose data are held entirely on a GPU device, their computation benefits from the high device-memory bandwidth (CUFFT Library Version 2000; Akira and Satoshi 2009; Govindaraju et al. 2008; Nukada et al. 2008; Volkov and Kazian 2008). This conforms to an application scenario where the main data are located on dmem, and FFT is performed many times. Then the overheads of PCI transfers between hmem and dmem are overwhelmed by the computation time.

If the data size is too large for a GPU device or must be transferred from/to dmem every time that FFT is performed, then the PCI bandwidth becomes a bottleneck. The time to compute FFT on a GPU will likely be overwhelmed by data transfers via PCIs. This is the scenario for large-scale FFTs on a GPU cluster where all the data are moved around the entire cluster and between hmem and dmem on every node. Compared to a single node, a cluster will provide multiplied memory capacity and bandwidth. The performance bottleneck for a GPU cluster will likely be either the PCI between hmem and dmem or the network between nodes—whichever has the narrower bandwidth. The fact that GPUs can accelerate large-scale FFTs is surprising, as FFT is extremely bandwidth-intensive, but GPUs *do not* increase network bandwidth.

In our previous work (Chen et al. 2010), we proposed a new FFT algorithm called PKUFFT for GPU clusters. The original implementation uses CUDA, Pthread, MPI and even the low-level infiniband library IB/verbs for performance optimization. That implementation is unportable and specific to a 16-node cluster with dual infiniband cards and dual Tesla C1060 GPUs on each node. To port that code to Tianhe-1A, we first rewrite the code in PARRAY and then re-compile it on the target machine, drastically reducing its length (from 400 lines to 30 lines) while preserving the same depth of optimization. 3D PKUFFT has been deployed to support large-scale turbulence simulation.

Two factors have contributed to the acceleration of FFT with GPUs. Firstly dmem like a giant programmable cache is much larger than CPU cache and hence allows larger sub-tasks to be processed in whole and reduces repeated data transfers between memory and processors. Secondly one major operation of the algorithm requires transposing the entire array, which usually involves main-memory transposition within every node and Alltoall cluster communication. The main optimization of the algorithm (Chen et al. 2010) is to re-arrange and decompose the operation into small-scale GPU-accelerated transposition, large-scale Alltoall communication and middle-scale data-displacement adjustment that is performed during communications. Then the main-memory transposition is no longer needed! The price paid is to use a non-standard Alltoall with discontiguous process-to-process communications (see Sect. 5.2.9).

At source level, porting code from one platform to another platform is straightforward. For simplicity of presentation, the following code is fixed for N-cubic 3D FFTs and requires the GPU-Direct technology (which was not available originally) to use `pinned` memory as a communication buffer. Without GPU-Direct, the main data

and mpimem communication buffer cannot share the same addresses. The variable K is the number of MPI processes.

```
#parray {mpi[K]} L
#detour L(){
 #parray {pinned float2 [N/K][N][N]} G
 #parray {pinned float2 [disp i][N][N]} F
 #parray {dmem float2 [N][N]} Q
 #parray {dmem float2 [#Q_1][#Q_0]} R
 #parray {[[#L][#G_0][#G_1]][#G_2]} S
 #parray {[[#G_1][#L][#G_0]][#G_2]} T
 float2* g; #create G(g)
 float2* gbuf; #create G(gbuf)
 float2* q; #create Q(q)
 float2* qbuf; #create Q(qbuf)
 cufftHandle plan2d;
 cufftPlan2d(&plan2d,N,N,CUFFT_C2C);
 for(int i=0; i<N/K; i++) {
  #insert DataTransfer(q,Q, g,F){}
  cufftExecC2C(plan2d,q,q,CUFFT_FORWARD);
  #insert DataTransfer(gbuf,F, q,Q){}
 }
 #insert DataTransfer(g,T, gbuf,S){}
 cufftHandle plan1d;
 cufftPlan1d(&plan1d,N,CUFFT_C2C,N);
 for(int i=0; i<N/K; i++) {
  #insert DataTransfer(q,Q, g,F){}
  #insert DataTransfer(qbuf,R, q,Q){}
  cufftExecC2C(plan1d,qbuf,qbuf,CUFFT_FORWARD);
  #insert DataTransfer(q,Q, qbuf,R){}
  #insert DataTransfer(gbuf,F, q,Q){}
 }
 #insert DataTransfer(g,S, gbuf,T){}
 cufftDestroy(plan2d);
 cufftDestroy(plan1d);
 #destroy G(g) #destroy G(gbuf)
 #destroy Q(q) #destroy Q(qbuf)
}
```

This code consists of a series of data-transfer operations that we already studied in previous sections. The main 3D complex data are stored in array g of type G. Another array gbuf acts as a buffer. The inner dimension G_2 is contiguous; G_1 is the middle dimension; the outer dimension is a combination of thread dimension L and G_0. Each MPI processes in L contains N/K pages of size N*N. In the first step, every page (with middle and inner dimensions) is transferred to the dmem array q for 2D FFT computation (by calling CUDA library) with results transferred back into

`qbuf`. The following communication over the entire network is the non-standard discontiguous Alltoall communication pattern (5.1) that we studied in Sect. 5.2.9. The communication effectively swaps the outer and middle dimensions, so that the middle dimension is aggregated on each MPI process. Every 2D page of the middle and inner dimensions is transferred to dmem again. Before performing batched 1D FFT on the new middle dimension, we use GPU transposition (see Sect. 5.3.3) to swap the middle and inner dimensions to make the middle dimension contiguous. The original positions of the data are restored after FFT by GPU transposition and communication.

In Fig. 5.1 FFT code is tested on Tianhe-1A using up to 7168 nodes, each with 24 GB main memory, two 6-core Intel Processors and one Tesla Fermi 448-core GPU. The special customized network has 80 Gb/s bandwidth for each node and a fat tree structure for switching. CUDA version is 3.0; CUFFT version is 3.1. For comparison, Intel Cluster MKL (or CMKL) 10.3.1.048 is used on the same cluster but does not use GPU. CMKL is already highly optimized because of the heavy communication load of very-large FFTs. The tests are performed for 3D FFT of different sizes for single-precision C2C forward (with returning communication that restores the data to their original positions). Double-precision FFTs perform at the half speed of single precision (on Fermi as well as data transfers). Figure 5.2 (tested for best supported array sizes) shows that on a large GPU cluster, the GPU-based algorithm significantly outperforms CMKL which does not use GPU. GPU-accelerated FFT also scales better than CPU-based FFT. Figure 5.2 illustrates the scalability in more details. Note that we do not swap the outer and inner dimensions directly, as that will affect the granularity of network communication.

Some FFT algorithms (Pekurovsky 2009) adopt two-dimensional decomposition. This is no longer necessary using PARRAY. On a traditional CPU cluster, the number of MPI processes is usually the number of cores so that MPI utilizes multicore parallelism without the need for OpenMP. That leads to a large number of MPI

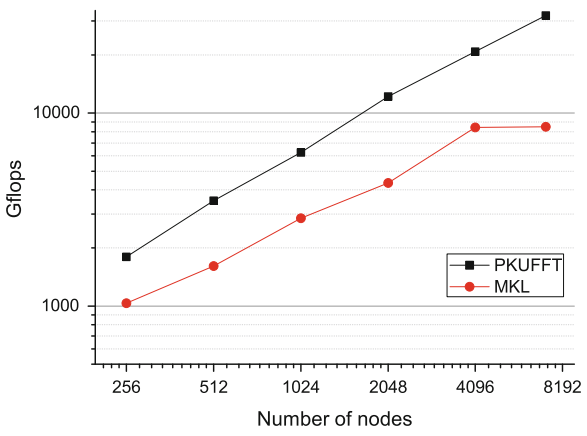


Fig. 5.1 PKUFFT versus intel cluster MKL on Tianhe-1A

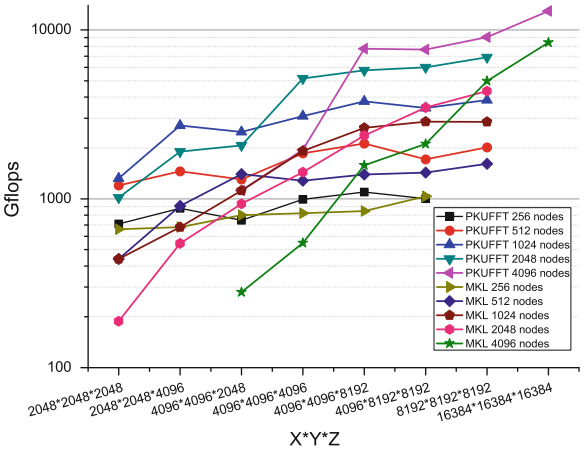


Fig. 5.2 Detailed comparisons of scalability

processes exceeding FFT's dimension size. As we can program each node's internal (multicore and manycore) parallelism, there is no need for over-decomposition that reduces message sizes and performance.

We mainly develop R2C and C2R single-precision 3D FFTs, which are used for large direct numerical simulation of turbulent flows up to the scale of 14336 3D on Tianhe-1A.

5.5 Literature Survey

In this section, we compare our design with other parallel programming interfaces in a table where, for example, "Global" and "Local" denote addressing. The languages considered for comparisons include Chapel (Chamberlain et al. 2007), Co-Array Fortran (CAF) (Numerich and Reid 1998), HMPP (Francois 2010), Hierarchically Tiled Arrays (HTA) (Ganesh et al. 2006), Titanium (Yelick et al. 1998), Stanford PPL (Brown et al. 2011; Chafi et al. 2011), UPC (Zheng et al. 2010), X10 (Charles et al. 2005), ZPL (Chamberlain et al. 2004), Global Arrays (Nieplocha et al. 1996) etc (Fig. 5.3).

These programming interfaces as well as some parallel functional languages (Hains and Mullin 1993) all support some kind of (array) domains. Common dimensional features include Block, Cyclicity, Replication (i.e. indices mapped to replicated value) etc. Such domains are special cases of PARRAY types. CAF also represents multiple cooperating instances of an SPMD program (known as images) through a new type of array dimension called a co-array. Titanium adds several features including multidimensional arrays supporting iterators, subarrays, and copying to Java. ZPL uses a series of array operators to express different access patterns including

	Language	Library	Global	Local	Cluster	Multicore	Manycore	Dimtree/Type Reference
PARRAY	✓			✓	✓	✓	✓	✓
Chapel	✓		✓		✓	✓		
CAF	✓		✓		✓			
HPF	✓		✓		✓			
HMPP	✓			✓		✓	✓	
HTA		✓	✓		✓	✓	✓	
MPI/PVM		✓		✓	✓			
PPL		✓			✓			
Titanium	✓		✓			✓	✓	
UPC	✓		✓		✓			
X10	✓		✓		✓	✓		
ZPL	✓		✓		✓			

Fig. 5.3 Comparisons among parallel programming languages and libraries

translation, broadcast, reduction, parallel prefix operations, and gathers/scatters. In UPC a shared array variable has elements distributed among program instances (or threads). X10 also supports multi-dimensional arrays that can be distributed among cluster nodes.

PARRAY does not directly offer numerical operations between arrays. An operation as simple as matrix multiplication may have different algorithms and implementations on a heterogeneous parallel system. PARRAY is intended to be a performance-transparent layer of abstraction. All effort to lift programming level and hide algorithmic decisions is left to library development of sub-programs.

Among the existing language designs, HTA is perhaps the closest. The claimed points of novelty (1) dimension tree, (2) type references, and (3) thread arrays are not supported by HTA. The improvement is mainly the fact that the array representation in PARRAY is more expressive. The theory part shows a certain algebraic completeness of the representation. Hierarchical tiles arrays assume several default levels (for multicore/cluster parallelism). On the other hand, PARRAY's dimension trees are logical. We believe that hierarchical structure is so important that it should be general and not tied to a specific memory structure.

A large class of languages are PGAS languages. PGAS expects the programmer to think about locality but supports random accesses like shared memory regardless the underlying communication mechanism—essentially something between message passing and variable sharing. PARRAY is not based on any unifying memory model. Instead it is designed to support a variety of communication mechanisms. However, if there exists a well-optimized PGAS library such as Global Arrays, PARRAY can generate code to invoke that library. If the low-level libraries offer both two-sided and one-sided communications, the programmer can use commands `GetDataTransfer` and `PutDataTransfer` to generate one-sided code explicitly.

5.6 Conclusions and Future Work

Our array representation is proposed to unify three forms of parallelism: multicore, manycore and clustering. Other programming ideas often focus on two of them. An advantage of our array representation is its simplicity of semantics and implementation. The source-to-source translation and code generator reach only 2000 lines of C++ code. Basic forms of new data types are intuitive and easy to understand, though some sophisticated types may require more mathematical intuition to handle the inherent complexity of some communication patterns (e.g. the non-standard Alltoall).

Our code generator has been tested on a wide range of other program examples such as matrix operations and stencil computation. In particular we have used FFT in direct numeral simulation of turbulent flows scalable up to 14336^3 single precision. This ongoing experiment (to be reported elsewhere) requires 12 arrays of this size and has reached 36Tflops on Tianhe-1A. Porting the original MPI code to PARRAY for Tianhe-1A took us only five days.

PARRAY only provides abstraction for regular data structures like arrays. Irregular data structures such as trees and graphs must be encoded as arrays to benefit from PARRAY's integrated code generation. The encoding is left to the user or any higher-level software layers/libraries. PARRAY's performance transparency makes sure that any higher-level layers implemented on top of PARRAY will not be performance-wise penalized because of using PARRAY instead of the low-level libraries of its generated code.

The most important thing for a new programming interface is to encourage user acceptance. Training courses have been carried out. Trainees especially those with background in computational sciences respond remarkably well to the new notation. Unlike computer engineers who are more used to language mechanisms like pointer arithmetic, application programmers (e.g. those from oil industry) seem to be more conformable with matrix notation and even find those advanced forms of array types intuitive. For example in stencil computation, a window within a two-dimensional array can be accessed either by moving the pointer to the starting address or using dimensional displacement. Programmers in CS background often prefer pointer arithmetic while many with science backgrounds prefer displacement type. The nature of this interesting difference is perhaps due to their different programming familiarity with C and Fortran, different earlier mathematical education or a combination of the two factors.

In future, PARRAY will support other accelerator devices such as FPGA and Intel MIC and lower-level communication libraries like IB/Verbs. We do not foresee obvious technological hurdles.

Acknowledgments We are grateful to the anonymous referees for their comments that have helped improve the presentation of this paper.

References

Akira N, Satoshi M (2009) Auto-tuning 3D FFT library for cuda GPUs. In: SC'09. ACM, New York, pp 1–10

Brown K et al (2011) A heterogeneous parallel framework for domain-specific languages. In: PACT'11

Chafi H et al (2011) A domain-specific approach to heterogeneous parallelism. In: PPoPP'11

Chamberlain B et al (2004) The high-level parallel language ZPL improves productivity and performance. In: IJHPCA'04

Chamberlain B, Callahan D, Zima HP (2007) Parallel programmability and the Chapel language. IJHPCA 21(3):291–312

Charles P et al (2005) X10: an object-oriented approach to nonuniform cluster computing. In: OOPSLA'05

Chen Y, Cui X, Mei H (2010) Large-scale FFT on GPU clusters. In: ACM International conference on supercomputing (ICS'10), pp 50–59

CUDA CUFFT Library 2009, Version 2.3. NVIDIA Corp.,

Fatica M (2009) Accelerating linpack with CUDA on heterogenous clusters. In: GPGPU'09, June 2009

Francois B (2010) Incremental migration of C and Fortran applications to GPGPU using HMPP. Technical report, hipeac

Ganesh B et al (2006) Programming for parallelism and locality with hierarchically tiled arrays. In: PPoPP'06, pp 48–57

Govindaraju N et al (2008) High performance discrete fourier transforms on graphics processors. In: SC'08, Nov 2008

Hains G, Mullin LMR (1993) Parallel functional programming with arrays. Comput J 36(3):238–245

Hoare CAR (1985) Communicating sequential processes. Prentice Hall, Upper Saddle River

Kandalla K et al (2011) High-performance and scalable non-blocking All-to-All with collective offload on infiniband clusters: a study with parallel 3D FFT. In: ISC'11

Nieplocha JJ, Harrison RJ, Littlefield RJ (1996) Global arrays: a nonuniform memory access programming model for high-performance computers. J Supercomput 10(2):169–189

Nukada A et al (2008) Bandwidth intensive 3-D FFT kernel for GPUs using cuda. In: SC'08, pp 1–11

Numerich R, Reid J (1998) Co-Array Fortran for parallel programming. SIGPLAN Fortran Forum 17(2):1C31

Pekurovsky D (2009) http://www.sdsc.edu/us/resources/p3dfft.php

Volkov V, Kazian B (2008) Fitting FFT onto the G80 architecture. http://www.cs.berkeley.edu/. Accessed May 2008

Yelick K et al (1998) Titanium: a high-performance Java dialect. In: In ACM, pp 10–11

Zheng Y et al Extending unified parallel C for GPU computing. In: SIAM conference on parallel processing for scientific computing

Chapter 6
Practical Random Linear Network Coding on GPUs

Xiaowen Chu and Kaiyong Zhao

Abstract Recently, random linear network coding has been widely applied in peer-to-peer network applications. Instead of sharing the raw data with each other, peers in the network produce and send encoded data to each other. As a result, the communication protocols have been greatly simplified, and the applications experience higher end-to-end throughput and better robustness to network churns. Since it is difficult to verify the integrity of the encoded data, such systems can suffer from the famous pollution attack, in which a malicious node can send bad encoded blocks that consist of bogus data. Consequently, the bogus data will be propagated into the whole network at an exponential rate. Homomorphic hash functions (HHFs) have been designed to defend systems from such pollution attacks, but with a new challenge: HHFs require that network coding must be performed in GF(q), where q is a very large prime number. This greatly increases the computational cost of network coding, in addition to the already computational expensive HHFs. This chapter exploits the potential of the huge computing power of Graphic Processing Units (GPUs) to reduce the computational cost of network coding and homomorphic hashing. With our network coding and HHF implementation on GPU, we observed significant computational speedup in comparison with the best CPU implementation. This implementation can lead to a practical solution for defending the pollution attacks in distributed systems.

Keywords Network coding · Pollution attack · GPU computing

X. Chu (✉) · K. Zhao
Department of Computer Science,
Hong Kong Baptist University,
Hong Kong, China
e-mail: chxw@comp.hkbu.edu.hk

K. Zhao
e-mail: kyzhao@comp.hkbu.edu.hk

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 115
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_6,

6.1 Introduction

In recent years, peer-to-peer (P2P) content distribution applications (e.g., BitTorrent) and video streaming applications (e.g., ppLive) have become popular and constitute more than 30 % of today's Internet traffic. The new coding technique, random linear network coding, is recently adopted by P2P applications (Ahlswede et al. 2000; Koetter and Medard 2003; Ho et al. 2003; Li et al. 2003; Gkantsidis and Rodriguez 2005; Dimakis et al. 2007; Wang and Li 2007), leading to simpler communication protocols, higher throughput, better resilience to network churns, and many more benefits to be discovered. With network coding, the source segments the to-be-distributed content into n data blocks of equal size. Each peer (including the source) sends out encoded data blocks, each of which is a linear combination of the original data blocks. After receiving n linearly independent encoded data blocks, a peer is able to decode the original data blocks by solving n linear equations with n variables. Since it is difficult to verify the integrity of the encoded data, such systems can suffer from the famous pollution attack, in which a malicious node can send bad encoded blocks that consist of bogus data. Consequently, the bogus data will be propagated into the whole network at an exponential rate (Chu and Jiang 2010). To defend against such attacks, homomorphic hash functions (HHFs) have been proposed to provide a mechanism for verifying the integrity of the encoded data blocks received from the network. In a nutshell, HHFs offer a nice property that the hash value of any encoded data block can be derived from the hash values of the original data blocks, based on which we can identify the bad encoded data blocks without decoding them (Gkantsidis and Rodriguez 2006; Li et al. 2006). Hence, it can effectively prevent the propagation of the bogus data blocks.

The theoretical property of HHFs is very attractive to practical P2P applications. Unfortunately, the computational complexity posed by HHF is the stumbling stone to this realization. First, HHF itself is computationally expensive. On a contemporary CPU, say 3.0 GHz Pentium 4 PC, we can only hash hundreds of kilobit per second (Krohn et al. 2004). Second, homomorphic hashing requires extensive modular exponentiation operations over a very larger prime modulus p (e.g., 1024 bit). This requires that the data must be encoded in large finite field, GF(q), where q is a large prime number (e.g., 257 bit). This greatly increases the computational cost of network coding, nearly impractical on CPUs. Our imperative goal is to remove the barrier by reducing the computational cost. The key enabling technologies here are the modern GPUs and the CUDA programming model for non-graphical application development on GPUs.

On the hardware level, recent advances in GPUs open a new era of GPU computing (Owens et al. 2008). For instance, NVIDIA's GTX 280 can achieve 933 GFLOPS of computing power, about 8 times faster than the Intel Harpertown 3.2 GHz CPU. However, using GPU for non-graphic applications has been considered very difficult, mostly due to the limited API support. Nonetheless, the introduction of CUDA programming model makes it easier for software developers to develop non-graphic applications on GPUs (NVIDIA CUDA 2008). In CUDA, GPU is treated as a

dedicated coprocessor to the CPU, and multiple threads based on the same code can run simultaneously on the GPU, working on different data set. With supports from CUDA, it is now possible to implement network coding and HHFs on GPUs. In this chapter, we propose to use GPU to accelerate random linear network coding as well as homomorphic hashing. We designed and developed massively parallel network encoding and decoding algorithms and homomorphic hashing. By carefully applying optimization techniques, we successfully achieved a significant performance boost: 95× speedup for network encoding, 33× speedup for network decoding, and 38× speedup for homomorphic hashing on a contemporary GPU. This makes network coding and HHF a practical solution for pollution attacks in P2P systems.

The rest of the chapter is organized as follows. Section 6.2 provides background information on network coding, homomorphic hashing, the GPU architecture, and the CUDA programming model. Section 6.3 presents the parallel algorithms for random linear network coding in GF(q). Section 6.4 presents the parallel homomorphic hash algorithm. Our experimental results are presented in Sect. 6.5, followed by the conclusions in Sect. 6.6.

6.2 Background and Related Work

This section provides the necessary background knowledge of network coding, homomorphic hash function, the GPU architecture, and the CUDA programming model.

6.2.1 Network Coding

Network coding has been originally proposed in information theory to achieve the optimal throughput in a multicast session (Ahlswede et al. 2000). Since then, it has been applied in various communication networks for better throughput and robustness to network dynamics. The essence of network coding is a paradigm shift to allow coding at intermediate nodes between the source and the receivers in one or multiple communication sessions. The seminal work of network coding has been studied in Ahlswede et al. (2000), Koetter and Medard (2003) and Li et al. (2003), which has shown that a multicast session can achieve the data rate of multicast upper bound if network nodes are allowed to perform coding. The framework of random network coding was proposed in Ho et al. (2003), which makes network coding theory applicable to practical applications. Since then, there are quite a number of proposals to apply network coding in practical systems for performance enhancement. The Avalanche project by Microsoft Research applied random linear network coding in a P2P content distribution application (Gkantsidis and Rodriguez 2005). Similarly, Lava incorporates random linear network coding into a live multimedia streaming system (Wang and Li 2007). Network coding has also been applied in other fields,

such as distributed storage systems (Dimakis et al. 2007) and wireless networks (Katti et al. 2008).

In network coding, each data block is treated as a vector of elements in the finite field, and an encoded block is simply a vector representing the linear combination of a set of data blocks (vectors) with randomly generated coefficients in the finite field. The network coding operations, encoding and decoding, are implemented in finite fields, i.e., prime fields GF(q) or extension fields GF(q^r), where q is a prime number and r is a positive integer. The computational performance of random linear network coding in GF(2^r) has been previously studied in Shojania and Li (2007). In Chu et al. (2008) and Shojania et al. (2009), GPUs have been used to accelerate the performance of network coding in GF(2^r). To the best of our knowledge, this is the first work studying the computational performance of random linear network coding in prime field GF(q), which has a much higher demand of computing power than that of network coding in GF(2^r) (Chu et al. 2009).

6.2.2 Homomorphic Hashing

As discussed in Sect. 6.1, network coding enabled P2P applications are prone to the pollution attacks, in which a malicious peer can easily inject bogus data blocks into an encoded block without being noticed. When network coding is not deployed, peers will receive original data blocks from each other. Hence it is possible to use normal hash functions, such as SHA1, to verify the correctness of a data block by comparing the hash of each received data block to the corresponding hash provided by the source. With network coding, the effect of pollution attack becomes more serious and harder to detect (Gkantsidis and Rodriguez 2006; Li et al. 2006; Yu et al. 2008) for two reasons: First, each bogus block can be encoded with regular data blocks before being propagated in the network. Second, the traditional hash functions, e.g., SHA1, are no longer practical since the encoded blocks received by each peer cannot be predetermined by the source. Some workarounds have been proposed to address these issues. In Gkantsidis and Rodriguez (2006), a cooperative scheme is proposed, in which peers perform probabilistically block verification and inform others when a malicious node has been identified. However, this scheme cannot detect the bogus blocks at the earliest stage and could potentially have false alarms.

To this end, homomorphic hash functions (HHFs) are currently the best solution to address this security issue with network coding. HHFs have the property that the hash value of an encoded block can be constructed by the hash values of the original blocks. In other words, a peer only need to get the hash values of the original blocks from the source, it then can easily verify the integrity of an encoded block immediately after receiving the encoded block. Although the HHFs can theoretically resolve the pollution attack problem, it is technically not practical on today's desktop CPUs. A 3 GHz Pentium 4 CPU can only achieve around a few hundred Kbps of hashing throughput (Krohn et al. 2004). Furthermore, it requires network coding to be performed in GF(q) with large value for q, which makes it computationally

expensive. We have shown that HHFs can be efficiently implemented on contemporary GPUs with data rate of more than 10 Mbps (Chu et al. 2009; Zhao et al. 2009).

6.2.3 GPU Computing and CUDA

GPUs are dedicated hardware for manipulating computer graphics. Due to the huge demand for computing for real-time applications and high-definition 3D graphics, GPUs have been evolved into highly paralleled multi-core processors. The NVIDIA GeForce GTX260 has 24 Streaming Multiprocessors (SMs), and each SM has 8 Scalar Processors (SPs). At any given clock cycle, all SPs of the same SM must execute the same instruction, but can operate on different data. Each SM has four different types of on-chip memory: constant cache, texture cache, registers, and shared memory. The properties of the different types of memories have been summarized in NVIDIA CUDA (2008) and Ryoo et al. (2008). A general optimization principle is that registers and shared memory should be carefully utilized to amortize the global memory latency cost.

The exceptional GPU computing power is very attractive to general-purpose system development. The first generation of GPU computing (namely GPGPU) requires that non-graphics application must be mapped through the graphics application programming interfaces, which is very challenging. In early 2007, one of the major GPU vendors, NVIDIA, announced a new general-purpose parallel programming model, Compute Unified Device Architecture (CUDA) (NVIDIA CUDA 2008), which extends the C programming language for general-purpose application development. Meanwhile, another GPU vendor AMD introduced Close To Metal (CTM) programming model that provides an assembly language for application development (AMD 2006). Intel is also planning to release Larrabee (Seiler et al. 2008), a new multi-core GPU architecture specially designed for GPU computing. Currently, CUDA is the best available programming model, and is the most well accepted model by the research and development community. Since the release of CUDA, it has been used for speeding up a large number of applications (Chu et al. 2008, 2009; Ryoo et al. 2008; Owens et al. 2008; Shojania et al. 2009; Volkov and Demmel 2008; Zhao et al. 2009). For these reasons, we chose to use CUDA in our research. Nevertheless, out algorithms can be easily implemented on other GPU computing models.

In the CUDA model, the GPU is regarded as a coprocessor capable of executing a great number of threads in parallel. A single program consists of *host code* to be executed on CPU and *kernel code* to be executed on GPU. The kernel code is usually computational-intensive, data-parallel and multi-threaded. Threads are organized into *thread blocks*, where each block is associated with one SM. Threads belonging to the same thread block can share data through the shared memory and can perform barrier synchronization. CUDA does not provide any direct synchronization methods between threads that belong to different thread blocks, however. When a thread block terminates, a new thread block can be launched on the vacant SM.

6.3 Parallel Network Coding on GPUs

To facilitate the development of network coding, we have implemented a set of library functions of multiple-precision modular arithmetic on the CUDA platform. These library functions simplify the development of the network coding system and homomorphic hash functions. Our multiple-precision library includes the following functions: comparison, subtraction, modular addition, modular subtraction, multiplication, division, multiplicative inversion, Montgomery reduction, Montgomery multiplication.

Assume the original data to be distributed is divided into n equally sized data blocks $(b_1, b_2, \ldots, b_n)$, where each data block b_i contains m codewords $b_{i,k}$, $k \in \{1, \ldots, m\}$. An encoded block e_j is a linear combination of the n original blocks and it also contains m codewords $e_{j,k}$, $k \in \{1, \ldots, m\}$. The linear relationship between e_j and the original n blocks is described by e_j's *global coefficient vector* $(c_{j,1}, c_{j,2}, \ldots, c_{j,n})$: $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k}$, $k \in \{1, \ldots, m\}$. Obviously the encoding process is a vector-matrix multiplication. A peer can decode the original n data blocks as soon as it has received n linearly independent encoded data blocks $(e_1, e_2, \ldots, e_n)$, by solving the set of linear equations $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k}$, $k \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$. In a P2P application with network coding, a peer receives encoded data blocks from upstream peers, and also creates new encoded data blocks by randomly and linearly combining its received encoded blocks, and then disseminates the new encoded blocks to its downstream peers.

6.3.1 Network Encoding in GF(q)

When network coding is performed in GF(q) where q is a predefined large prime number, the encoding process will creates a sequence of encoded blocks e_j, each of which contains m codewords $e_{j,k}$. e_j is generated based on a random coefficient vector $c_j = (c_{j,1}, c_{j,2}, \ldots, c_{j,n})$: $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k} \bmod q$, $k \in \{1, \ldots, m\}$. Here $c_{j,i}$ are positive 32-bit integers. The encoding process includes two steps: (1) generating the coefficient vector c_j; (2) modular vector-matrix multiplication. The CUDA library provides a very high efficient random number generator using Mersenne Twister method, which can generate tens of millions of random numbers per second using GPU. As the time of generating n random numbers are negligible as compared with the encoding time, we will focus on the vector-matrix multiplication operation, as shown in Fig. 6.1a. We implement the computing of each codeword $e_{j,k}$ by a CUDA thread. Hence encoding a single block requires m threads. Each thread computes a dot product and then performs a modular operation, using our multiple-precision CUDA library. In order to fully exploit the computing power of GPUs, thousands of threads are a normal requirement. Therefore we propose a batched encoding approach for small values of m, which encodes K blocks simultaneously, as shown in Fig. 6.1b. In this case, the number of threads equals mK.

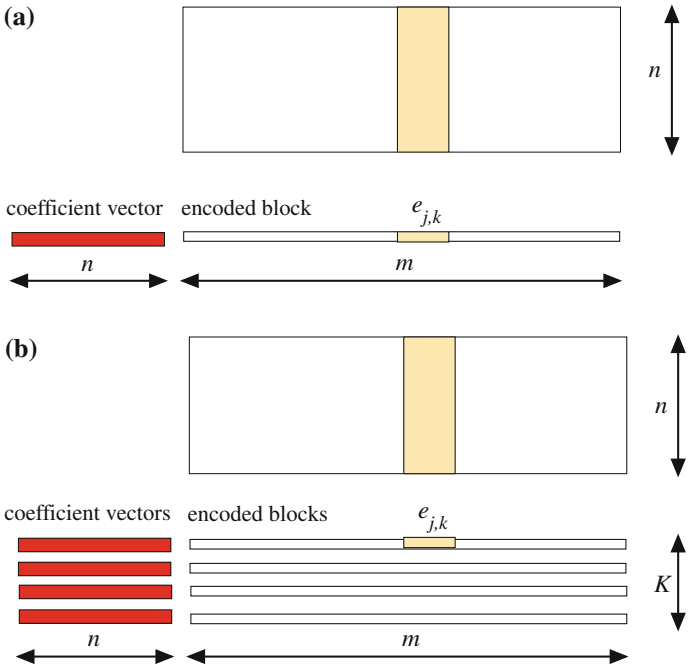


Fig. 6.1 Encoding. **a** Encode a single block. **b** Encode multiple blocks in a batch

6.3.2 Network Decoding in GF(q)

The decoding process includes two steps: (1) matrix inversion; and (2) matrix multiplication. Matrix inversion for floating-point numbers on GPU has been recently studied in Volkov and Demmel (2008). Our problem is very different because we are operating in GF(q). We use Gauss-Jordan elimination for the matrix inversion, which brings a matrix to its reduced row echelon form. There is no stability issue because we are operating in finite field. To overcome the synchronization challenge, our parallel matrix inversion algorithm uses both CPU and GPU, as shown in Table 6.1. The non-parallel parts, e.g., finding the multiplicative inverse, are done by the CPU, whilst the parallel parts, e.g., reducing to row echelon form, are done by GPU.

The matrix multiplication can be implemented in a similar way as the vector-matrix multiplication. Each element in the output matrix is computed by one thread; hence the total number of threads is n^2, which is sufficient to fill the GPU cores since n is normally no less than 64 in practice. Difference from the encoding process, the integer multiplications here are performed between two large integers, since the corresponding values of the coefficients grow as the blocks are being re-encoded at each peer. In this case, a straightforward parallel implementation cannot achieve satisfactory performance due to the global memory latency. GPU's on-chip shared memory can be exploited to amortize the global memory latency, and we propose

Table 6.1 Algorithm of matrix inversion in GF(q)

Algorithm 1 Matrix Inversion in GF(q)

INPUT: An n x n non-singular matrix M, an n x n unit matrix U
OUTPUT: the inverse of M

```
1:  lead ← 0;
2:  row ← n; col ← n;
3:  for ( r = 0 to row - 1)
4:     i ← r;
5:     while M[i, lead] equals 0
6:        i++;
7:     Swap rows i and r of M and U;
8:     t ← multiplicative inverse of M[r, r];  /* on CPU */
9:     Multiply row r of M and U by t;          /* on GPU */
10:    for all rows j except row r of M and U
11:       For M, subtract M[j, lead] multiplied by row r from row j; /* on GPU */
12:       For U, subtract M[j, lead] multiplied by row r from row j; /* on GPU */
13:    end for
14:    lead++;
15: end for
16: return U
```

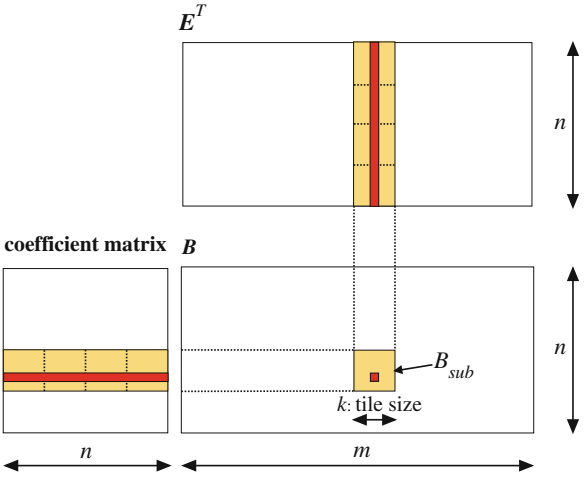


Fig. 6.2 Decoding: tiled matrix multiplication

to use a titled version of matrix multiplication, in which the matrix is divided into a number of sub-blocks (NVIDIA CUDA 2008; Ryoo et al. 2008). As illustrated in Fig. 6.2, the computing of sub-block B_{sub} is done by a thread block. The threads in this block cooperatively load the data from the two tiles in coefficient matrix and E^T into shared memory. These threads compute the partial dot product in shared memory, and then continue with the next tile. The size of the tile should be controlled such that two tiles can be accommodated by the shared memory of a SM.

Table 6.2 Homomorphic hash function parameters

Name	Description	Typical value
λ_p	Discrete log security parameter	1024 bit
λ_q	Discrete log security parameter	257 bit
p, q	Random primes, $\|p\| = \lambda_p$,$\|q\| = \lambda_q$, $q \| p-1$	
m	Number of codewords per data block	512
n	Number of data blocks	128

6.4 Parallel Homomorphic Hashing on GPUs

The homomorphic hash function, $h(\cdot)$, proposed in Krohn et al. (2004) requires a set of hash parameters $G = (p, q, g)$. The parameters p and q are large prime numbers of order λ_p and λ_q chosen such that $q|p-1$. The parameter g is a vector of m numbers, each of which can be written as $x^{(p-1)/q} \bmod p$ where $x \in Z_q$ and $x \neq 1$. The method of creating the parameter set can be found in Krohn et al. (2004). Typical values of the parameters are summarized in Table 6.2. The homomorphic hash of a data block b_i is then calculated as $h(b_i) = \prod_{k=1}^{m} g_k^{b_{i,k}} \bmod p$. The hash values of the original data blocks $(b_1, b_2, \ldots, b_n)$ are $h(b_1), h(b_2), \ldots, h(b_n)$, respectively. Given an encoded data block e_j with global coefficient vector $(c_{j,1}, c_{j,2}, \ldots, c_{j,n})$, the homomorphic hash function $h(\cdot)$ can be shown to satisfy the following condition: $h(e_j) = \prod_{i=1}^{n} h^{c_{j,i}}(b_i) \bmod$ p. This property can be used to verify the integrity of an encoded block, as illustrated in Fig. 6.3. The content publisher first calculates the homomorphic hash values for each of the data blocks. The downloaders need to download a copy of these hash values for the purpose of verifying every single encoded data block.

Homomorphic hashing involves m modular exponentiations and m-1 modular multiplications. The m-1 modular multiplications can be easily parallelized by a regular reduction process. The m modular exponentiations can be very time consuming. We distribute the m modular exponentiations to the GPU processing cores. The challenge is to implement modular exponentiation on GPU in the most efficient way. On the current CUDA platform, integer division and modulo operations are very costly. Therefore we choose to use the Montgomery exponentiation algorithm (as shown in Table 6.3) which can decrease the number of division operations significantly.

We notice that, when applying HHF in network coding enabled P2P applications, the same HHF (i.e., with the same set of parameters), will be used for a large data set such as a whole file or a video streaming session. Under this circumstance, it is possible to speed up the modular exponentiations by precomputation. We extend the existing method by integrating precomputation with Montgomery multiplications, and implement it on CUDA.

To calculate g^e, we first represent the exponent e using radix $b = 2^k : e = \sum_{i=0}^{n-1} a_i b^i$, where $0 \leq a_i < b$ and $a_{n-1} \neq 0$. It is easy to see that $n = \lceil (\lfloor \log_2 e \rfloor + 1)/k \rceil$. Our algorithm requires the precomputation of $Rg^{2^{ki}} \bmod m$ for $0 \leq i \leq n-1$ where

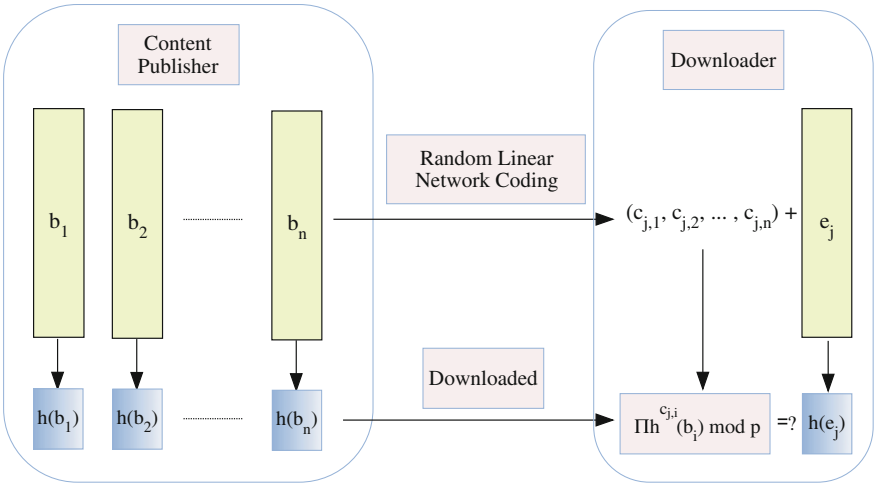


Fig. 6.3 Data verification using homomorphic hashing in network coded P2P applications

Table 6.3 Algorithm of multiple-precision Montgomery exponentiation

Algorithm 2 Multiple-precision Montgomery Exponentiation

INPUT: integer m with n radix b digits and $\gcd(m, b) = 1$, $R = b^n$, positive integer x with n radix b digits and $x < m$, and positive integer $e = (e_t \ldots e_0)_2$.

OUTPUT: $x^e \bmod m$.

```
1: x̃ ← Mont(x, R^2 mod m);
2: A ← R mod m;
3: for ( i from n down to 0)
4:     A ← Mont(A, A);
5:     if e_i == 1
6:     then A ← Mont(A, x̃);
7: end for
8: A ← Mont(A, 1);
9: return A;
```

$R = b^n$. Afterwards, we can use the following algorithm to calculate $g^e \bmod m$ efficiently (Table 6.4).

The above algorithm takes $n + b - 3$ multiplications. For e with 257-bit, the optimal value of b is 16, which takes only 78 multiplications in the worst case, as compared with 512 multiplications required by the binary method used in Algorithm 2. In theory, we can expect a speedup of 6.5 by using this algorithm.

Table 6.4 Algorithm of multiple-precision Montgomery exponentiation with precomputation

Algorithm 3 Exponentiation with Precomputation

INPUT: integers $m, g, e = \sum_{i=0}^{n-1} a_i b^i$, $R = b^n$, and $x_i = Rg^{2^{ki}} \bmod m$ for $1 \le i \le n-1$

OUTPUT: $g^e \bmod m$.

```
1:  A ← R, B ← R ;
2:  for ( j from b − 1 down to 1)
3:    for i from 0 to n − 1
4:      if a_i == j then B ← Mont(B, x_i) ;
5:    end for
6:    A ← Mont(A, B) ;
7:  end for
8:  A ← Mont(A, 1) ;
9:  return A;
```

6.5 Experimental Results

We have implemented the proposed network encoding/decoding and parallel homomorphic hashing algorithms using CUDA. For comparison purpose, we also implemented network coding and homomorphic hash function for CPU in C language, by utilizing the GNU MP arithmetic library, version 4.2.3 (GNU MP 2010). These implementations are running on an Intel Core2 CPU 1.6 GHz. In this study, we only utilize a single core of CPU for comparison. Higher CPU performance can be achieved if all CPU cores are fully utilized. We tested all our algorithms on XFX GTX280 graphic card with an NVIDIA GeForce GTX280, which has 240 processing cores. On GTX280, there are 30 Streaming Multiprocessors (SMs), and each SM has 8 Scalar Processors (SPs), 16384 32-bit registers and 16 KB shared memory.

6.5.1 Performance of Encoding in GF(q)

The throughput of encoding process is shown in Fig. 6.4a in log-scale. In theory, the encoding time complexity is linear to the size of n. This is in accordance with our experimental results. The throughput of network encoding on CPU is very poor: only 10.3 Mbps for $n = 128$. The GPU performance is very impressive: around 800 Mbps can be achieved for $n = 128$ with a small batch size K of 32. We observe that larger batch sizes can lead to better performance, until some threshold value has been met. In our testing environment, $K = 128$ is the optimal setting. The speedup on GPU over CPU has been plotted in Fig. 6.4b, for different batch sizes and n. With a batch size larger than 32, the speedup is greater than 66×. The highest speedup of 95× is obtained when $n = 128$ and $K = 128$.

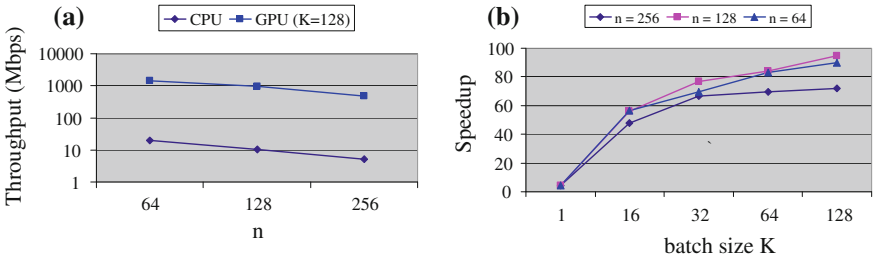


Fig. 6.4 Performance of encoding. **a** Encoding throughput. **b** Speedup over CPU

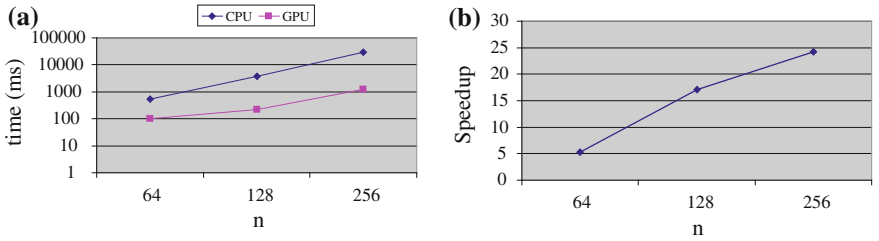


Fig. 6.5 Performance of matrix inversion. **a** Matrix inversion time in *ms*. **b** Speedup over CPU

6.5.2 Performance of Decoding

As mentioned before, the decoding process includes two steps: (1) matrix inversion; (2) matrix multiplication. The time used for matrix inversion is shown in Fig. 6.5a using log-scale, and the speedups on GPU over CPU are plotted in Fig. 6.5b. It is a well known fact that matrix inverse using Gauss-Jordan elimination has a time complexity of $O(n^3)$. Our experimental results on CPU follow this pattern as well. The performance of parallel matrix inversion on GPU is a bit more complicated due to the kernel loading overhead and communication overhead between the CPU and GPU. Figure 6.5b shows that the speedup on GPU grows as the number of blocks, n, increase: 17 for $n = 128$ and 24 for $n = 256$. This is where the benefit of GPU manifests itself, i.e., the overheads are amortized by the increasing parallelism in the computation.

The performance of the matrix multiplication process is shown in Fig. 6.6. As expected, the throughput is much slower than the encoding process. Even so, the GPU can achieve 243 Mbps of throughput for $n = 128$. The speedup on GPU over CPU ranges from 64× to 75× for $n = 64, 128, 256$ respectively.

The performance of the whole decoding process is shown in Fig. 6.7, for $m = 512$. Since the speedup of matrix multiplication is much larger than the speedup of matrix inversion, the speedup of the overall decoding process is limited by the performance of matrix inversion. The overall decoding throughput when $n = 128$ is

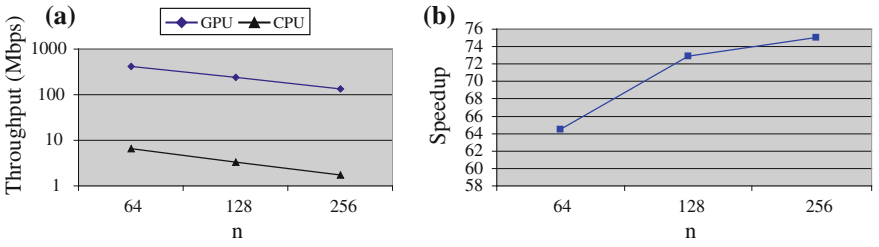


Fig. 6.6 Performance of matrix multiplication. **a** Throughput. **b** Speedup over CPU

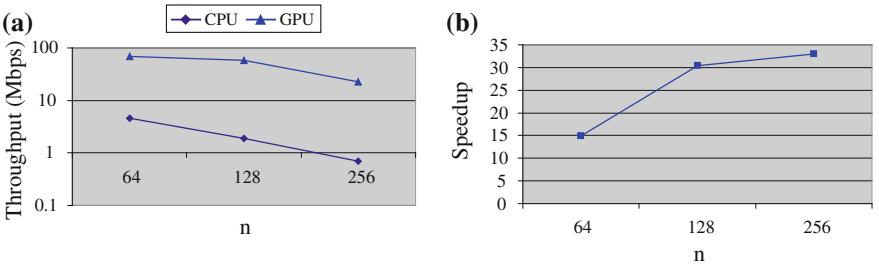


Fig. 6.7 Performance of decoding. **a** Throughput of decoding. **b** Speedup over CPU

58 Mbps which includes the matrix inversion and matrix multiplication. The decoding performance can be further enhanced by using a larger value of m, because the same matrix inverse operation is now used for a larger data volume. The speedup ranges from 15× to 33× for different values of n.

6.5.3 Performance of Homomorphic Hashing

Our CPU version of homomorphic hashing achieved 130 Kbps of throughput, which is relatively lower than the results reported by Krohn et al. (2004) and Gkantsidis and Rodriguez (2006) due to our relatively lower CPU frequency.

The parallel homomorphic hashing uses Algorithm 2 (without precomputation) and Algorithm 3 (with precomputation) to calculate exponentiations. The CUDA architecture requires a large number of threads to hide the memory latency and to fully utilize the computing power. The number of threads per thread block (denoted by ***TB***), and also the number of thread blocks (denoted by ***NB***), are the two main factors that affect the hashing throughput. We plot the throughput for different configurations in Figs. 6.8 and 6.9 for Algorithm 2 and Algorithm 3, respectively. It is easy to observe that more threads per block can generally achieve better throughput. When the number of threads per block is fixed, the throughput can be improved by creating more thread blocks, until some threshold has been reached. Since our GPU has 30

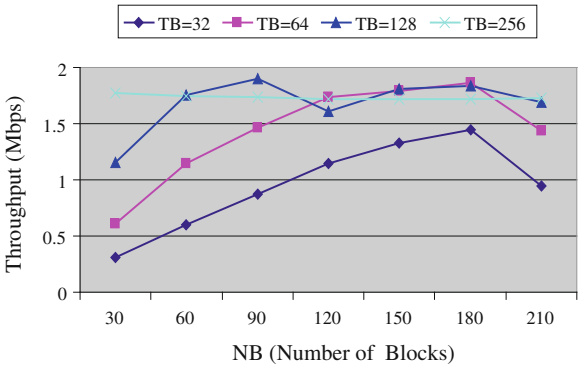


Fig. 6.8 Throughput of homomorphic hashing on GPU

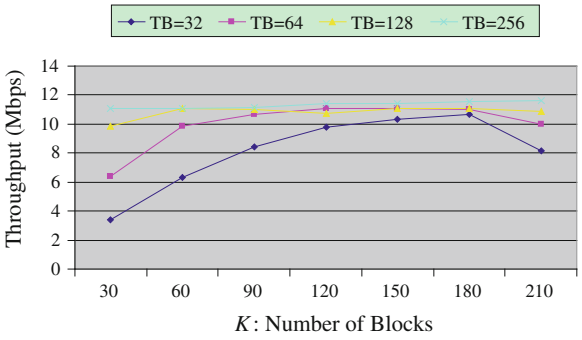


Fig. 6.9 Throughput of homomorphic hashing on GPU with precomputation

SMs, the number of thread blocks should be a multiple of 30. Better throughput can be achieved if the following conditions are satisfied: (1) ***TB*** is a multiple of 32 (i.e., the *warp* size (NVIDIA CUDA 2008; Ryoo et al. 2008)). In CUDA, a *warp* is formed by 32 parallel threads and is the scheduling unit of each SM. If the number of threads in a block is not a multiple of warp size, the remaining instruction cycles will be wasted. (2) ***NB*** is a multiple of 30 (i.e., the number of SMs). Without precomputation, the highest throughput of 1.9 Mbps is obtained when $\boldsymbol{TB} = 128$ and $\boldsymbol{NB} = 90$. With precomputation, we can achieve 11.6 Mbps of throughput, which is in fact more than 38 times of the fastest single core CPU results.

6.6 Conclusions

Network coding has been shown as a powerful technique to enhance the throughput and robustness of P2P systems; and homomorphic hash functions are a supplementary tool for defending the pollution attack. The remaining challenges are the

computational requirement of network coding in prime field and the homomorphic hashing. This chapter demonstrates a practical parallel implementation of network coding and homomorphic hashing using GPUs. Our experimental results show that the computational obstacle of network coding and homomorphic hashing can be overcome by designing efficient parallel algorithms and fully exploiting the computing power of contemporary GPUs that are widely available on today's desktop PCs.

Acknowledgments This work has been partially supported by the Hong Kong RGC under grant GRF HKBU210412, and by HKBU FRG2/09-10/081.

References

Ahlswede R, Cai N, Li SR, Yeung RW (2000) Network information flow. IEEE Trans Inf Theory 46(4):1204–1216

AMD (2006) CTM guide: technical reference manual. http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

Brickell EF, Gordon DM, McCurley KS, Wilson DB (1992) Fast exponentiation with precomputation: algorithms and lower bound. In: Proceedings of EUROCRYPT'92, 1992, pp 200–207

Chu X, Jiang Y (2010) Random linear network coding for peer-to-peer applications. IEEE Netw 24(4):35–39

Chu X, Zhao K, Wang M (2008) Massively parallel network coding on GPUs. In: Proceedings of the 27th IEEE IPCCC, Dec 2008

Chu X, Zhao K, Wang M (2009) Practical random linear network coding on GPUs. In: Proceedings of IFIP networking 2009, Archen, May 2009

Dimakis AG, Godfrey PB, Wainwright MJ, Ramchandran K (2007) Network coding for distributed storage systems. In: Proceedings of IEEE INFOCOM'07, 2007

Gkantsidis C, Rodriguez P (2005) Network coding for large scale content distribution. In: Proceedings of IEEE INFOCOM'05, 2005

Gkantsidis C, Rodriguez P (2006) Cooperative security for network coding file distribution. In: Proceedings of IEEE INFOCOM'06, 2006

GNU MP (2010) Arithmetic library. http://gmplib.org/

Ho T, Koetter R, Médard M, Karger DR, Effros M (2003) The benefits of coding over routing in a randomized setting. In: Proceedings of IEEE ISIT, 2003

Katti S, Katabi D, Balakrishna H, Medard M (2008) Symbol-level network coding for wireless mesh networks. In: Proceedings of ACM Sigcomm'08, Aug 2008

Koetter R, Medard M (2003) An algebraic approach to network coding. IEEE/ACM Trans Netw 11(5):782–795

Krohn M, FreedMan M, Mazieres D (2004) On-the-fly verification of rateless erasure codes for efficient content distribution. In: Proceedings of IEEE symposium on security and privacy, Berkeley

Li S-YR, Yueng RW, Cai N (2003) Linear network coding. IEEE Trans Inf Theory 49:371–381

Li Q, Chiu D-M, Lui JCS (2006) On the practical and security issues of batch content distribution via network coding. In: Proceedings of IEEE ICNP'06, 2006, pp 158–167

NVIDIA CUDA (2008) Compute unified device architecture: programming guide, version 2.0beta2

Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. In: IEEE Proceedings, May 2008, pp 879–899

Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu W (2008) Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of ACM PPoPP'08, Feb 2008

Seiler L et al (2008) Larrabee: a many-core x86 architecture for visual computing. ACM Trans Graph 27(3):1–15
Shojania H, Li B (2007) Parallelized progressive network coding with hardware acceleration. In: Proceedings of the 15th international workshop on quality of service (IWQoS), 2007
Shojania H, Li B, Wang X (2009) Nuclei: GPU-accelerated many-core network coding. In: Proceedings of IEEE INFOCOM'09, Apr 2009
Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: Proceedings of supercomputing'08, Nov 2008
Wang M, Li B (2007) Lava: a reality check of network coding in peer-to-peer live streaming. In: Proceedings of IEEE INFOCOM'07, 2007
Yu Z, Wei Y, Ramkumar B, Guan Y (2008) An efficient signature-based scheme for securing network coding against pollution attacks. In: Proceedings of IEEE INFOCOM'08, Apr 2008
Zhao K, Chu X, Wang M, Jiang Y (2009) Speeding up homomorphic hashing using GPUs. In: Proceedings of IEEE ICC 2009, Dresden, June 2009

Chapter 7
Preliminary Implementation of PETSc Using GPUs

Victor Minden, Barry Smith and Matthew G. Knepley

Abstract PETSc is a scalable solver library for the solution of algebraic equations arising from the discretization of partial differential equations and related problems. PETSc is organized as a class library with classes for vectors, matrices, Krylov methods, preconditioners, nonlinear solvers, and differential equation integrators. A new subclass of the vector class has been introduced that performs its operations on NVIDIA GPU processors. In addition, a new sparse matrix subclass that performs matrix-vector products on the GPU was introduced. The Krylov methods, nonlinear solvers, and integrators in PETSc run unchanged in parallel using these new subclasses. These can be used transparently from existing PETSc application codes in C, C++, Fortran, or Python. The implementation is done with the Thrust and Cusp C++ packages from NVIDIA.

V. Minden (✉)
School of Engineering, Tufts University, Medford, MA 02155, USA
e-mail: victor.minden@tufts.edu

B. Smith
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439-4844, USA
e-mail: bsmith@mcs.anl.gov

M. G. Knepley
Computation Institute, University of Chicago, Chicago, IL 60637, USA
e-mail: knepley@ci.uchicago.edu

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_7,

7.1 Introduction

PETSc (Balay et al. 1997, 2011) is a scalable solver library for the solution of algebraic equations arising from the discretization of partial differential equations and related problems. The goal of the project reported here is to allow PETSc solvers to utilize GPUs with as little change as possible to the basic design of PETSc. Specifically, a new subclass of the vector class has been introduced that performs its operation on NVIDIA GPU processors. In addition, a new sparse matrix subclass that performs matrix-vector products on the GPU was introduced. The Krylov methods, nonlinear solvers, and integrators in PETSc run unchanged in parallel using these new subclasses. These can be used transparently from existing PETSc application codes in C, C++, Fortran, or Python. The implementation uses the Thrust[1] (Bell and Hoberock 2010) and Cusp[2] (Bell and Garland 2010) C++ packages from NVIDIA.

Numerous groups have experimented with sparse matrix iterative solvers on GPUs, for example, Bolz et al. (2003), Feng and Li (2008), Buatois et al. (2007), Cevahir et al. (2009), Bell and Garland (2008, 2009), Baskaran and Bordawekar (2009). The Trilinos package (Baker et al. 2010; Heroux et al. 2005, 2009) already has support for NVIDIA GPUs through its Kokkos package, also using Thrust.

7.2 Sequential Implementation

PETSc consists of a small number of abstract classes: Vec and Mat primarily encapsulate data structures while PC, KSP, SNES, and TS represent algorithmic strategies. By abstract, we mean that each class is defined by a set of operations on the class object, while any data associated with the class object is encapsulated within the object and not directly accessible outside the class. The **Vec** class is used for representing field values, discrete solutions to PDEs, right-hand sides of linear systems, and so forth. PETSc provides a default implementation of Vec that stores the vector entries in a simple, one-dimensional array in memory. Vec uses BLAS 1 for local vectorizable operations when possible and MPI for reduction operations across processes. The **Mat** and **PC** classes do not directly access the underlying array in the vector; instead they call

```
VecGetArray(Vec, double *[])
```

or

```
VecGetArrayRead(Vec, const double *[])
```

to access the local (on process) values of the vector. In general, the **KSP**, **SNES**, and **TS** classes never access **Vec** or **Mat** data directly; rather, they call methods

[1] Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Thrust provides a flexible high-level interface for GPU programming that greatly enhances developer productivity.

[2] Cusp is a library for sparse linear algebra and graph computations on CUDA that uses Thrust.

Table 7.1 Flags used to indicate the memory state of a PETSc CUDA **Vec** object

PETSC_CUDA_UNALLOCATED	Memory not allocated on the GPU
PETSC_CUDA_GPU	Values on GPU are current
PETSC_CUDA_CPU	Values on CPU are current
PETSC_CUDA_BOTH	Values on both devices are current

on the **Vec** and **Mat** objects in order to perform operations on the data. The **PC** class is somewhat special in that many preconditioners are data structure specific. Thus, many **PC** implementations directly access matrix data structures, which in C++ would correspond to a *friend* class.

For this initial implementation of PETSc on GPUs, we have used the following model. PETSc runs in parallel with MPI for communication; and each PETSc process has access to a single GPU, which has its own memory, generally several gigabytes. We introduce a new **Vec** implementation, which we will call a CUDA **Vec**. Each object of this new **Vec** class must potentially manage two copies of the vector data: one in the CPU memory and one in the GPU memory. (We note that on some integrated graphics systems the GPU actually uses the usual CPU memory as its memory; we ignore this for our preliminary work). In order to manage memory coherence, each CUDA **Vec** has a flag that indicates whether space in the GPU memory has been allocated and whether the memory in the GPU, in the CPU, or in both contains the most recent values. The possible flag values are given in Table 7.1. The flag is the only change to the base **Vec** class in PETSc. This was added to the base class rather than the derived GPU-specific **Vec** class because we want to be able to check whether the memory copy is needed, without requiring the extra clock cycles of accessing the derived class for each check.

Two routines are provided,

```
VecCUDACopyToGPU(), VecCUDACopyFromGPU(),
```

that copy vector data down to the GPU memory or up to the CPU memory based on the flag. For example, the method `VecGetArray()` for the CUDA **Vec** copies the values up from the GPU if the flag is PETSC_CUDA_GPU, and sets the flag to PETSC_CUDA_CPU since the user is free to change the vector values. The `VecGetArrayRead()` still performs the copy but sets the flag to PETSC_CUDA_BOTH since the user cannot change the values in the array. For all vector operations performed on the GPU, such as `VecAXPY()`, data will be copied down from the CPU if the flag is PETSC_CUDA_CPU and will be both allocated and copied if it is PETSC_CUDA_UNALLOCATED.

Implementations of the basic vector operations is straightforward. For example, the `VecAXPY()` code is given by the following.

```
ierr = VecCUDACopyToGPU(xin);CHKERRQ(ierr);
ierr = VecCUDACopyToGPU(yin);CHKERRQ(ierr);
try {
  cusp::blas::axpy(*((Vec_CUDA*)xin->spptr)->GPUarray,
            *((Vec_CUDA*)yin->spptr)->GPUarray,alpha);
  yin->valid_GPU_array = PETSC_CUDA_GPU;
  ierr = WaitForGPU();CHKERRCUDA(ierr);
} catch(char *ex) {
  SETERRQ1(PETSC_COMM_SELF, PETSC_ERR_LIB,
                                "CUDA error: \%s", ex);
}
```

For more sophisticated **Vec** methods, such as `VecMAXPY()`, $y = y + \sum_i \alpha_i x_i$, and `VecMDot()`, $\alpha_i = y^T x_i$, the code is more complicated. We unroll loops in order to reuse entries in the y vector. For example, we unroll the outer loop for four vectors. The multiple inner product code, written by using Thrust calls, is given below.

```
for (j=j_rem; j<nv; j+=4) {
  yy0  =  yin[0]; yy1  =  yin[1];
  yy2  =  yin[2]; yy3  =  yin[3];
  ierr =  VecCUDACopyToGPU(yy0);CHKERRQ(ierr);
  ierr =  VecCUDACopyToGPU(yy1);CHKERRQ(ierr);
  ierr =  VecCUDACopyToGPU(yy2);CHKERRQ(ierr);
  ierr =  VecCUDACopyToGPU(yy3);CHKERRQ(ierr);
  try {
    result4 = thrust::transform_reduce(
                thrust::make_zip_iterator(
                  thrust::make_tuple(
          ((Vec_CUDA *)xin->spptr)->GPUarray->begin(),
          ((Vec_CUDA *)yy0->spptr)->GPUarray->begin(),
          ((Vec_CUDA *)yy1->spptr)->GPUarray->begin(),
          ((Vec_CUDA *)yy2->spptr)->GPUarray->begin(),
        ((Vec_CUDA *)yy3->spptr)->GPUarray->begin())),
                thrust::make_zip_iterator(
                  thrust::make_tuple(
            ((Vec_CUDA *)xin->spptr)->GPUarray->end(),
            ((Vec_CUDA *)yy0->spptr)->GPUarray->end(),
            ((Vec_CUDA *)yy1->spptr)->GPUarray->end(),
            ((Vec_CUDA *)yy2->spptr)->GPUarray->end(),
          ((Vec_CUDA *)yy3->spptr)->GPUarray->end())),
```

```
        cudamult4<thrust::tuple<PetscScalar,PetscScalar,
                  PetscScalar,PetscScalar,PetscScalar>,
                 thrust::tuple<PetscScalar,PetscScalar,
                               PetscScalar,PetscScalar> >(),
               thrust::make_tuple(zero,zero,zero,zero),
         cudaadd4<thrust::tuple<PetscScalar,PetscScalar,
                                PetscScalar,PetscScalar> >());
      z[0] = thrust::get<0>(result4);
      z[1] = thrust::get<1>(result4);
      z[2] = thrust::get<2>(result4);
      z[3] = thrust::get<3>(result4);
    } catch(char* ex) {
      SETERRQ1(PETSC_COMM_SELF,PETSC_ERR_LIB,
                                    "CUDA error: %s", ex);
    }
    z    += 4;
    yin  += 4;
  }
```

The CUDA kernel of this operation is given by the following.

```
struct VecCUDAMAXPY4 {
  template <typename Tuple>
  __host__ __device__
  void operator()(Tuple t) {
    /* y += a1*x1 +a2*x2 + a3*x3 +a4*x4 */
    thrust::get<0>(t) +=
                thrust::get<1>(t)*thrust::get<2>(t) +
                thrust::get<3>(t)*thrust::get<4>(t) +
                thrust::get<5>(t)*thrust::get<6>(t) +
                thrust::get<7>(t)*thrust::get<8>(t);
  }
};
```

Note that often the `VecCUDACopyToGPU()` calls simply verify that the vector's flag is PETSC_CUDA_GPU and do not need to copy the data down to the GPU. This is the case during a Krylov solve, where only the results of norm and inner product calls are shipped back to the CPU.

The NVIDIA Cusp software provides a data structure and matrix-vector product operation for sparse matrices in Compressed Sparse Row (CSR) and several other formats. Our initial CUDA **Mat** implementation simply uses the code provided by Cusp. The matrix-vector product code in PETSc then is given by the following.

```
try {
  cusp::multiply(*cudastruct->mat,
                 *((Vec_CUDA *)xx->spptr)->GPUarray,
                 *((Vec_CUDA *)yy->spptr)->GPUarray);
} catch(char* ex) {
  SETERRQ1(PETSC_COMM_SELF,PETSC_ERR_LIB,
                                  "CUDA error: \%s", ex);
}
```

Our primary design goal in this initial implementation was to enable the vector and matrix data to reside on the GPU throughout an entire Krylov solve, requiring no slow copying of data between the two memories.This is now supported for all but one of the Krylov methods in PETSc, including GMRES, Bi-CG-stab, and CG, and several preconditioners including Jacobi and the Cusp Smoothed-Aggregation Algebraic Multigrid. The excluded Krylov method, a variant of Bi-CG-stab that requires only one global synchronization per iteration, actually accesses the vectors directly rather than through the **Vec** class methods (since it requires many operations not supported by the class methods) and hence would need to be rewritten directly in CUDA.

7.3 Parallel Implementation

In the parallel case, there must be communication of vector entries between processes during the computation of the sparse matrix-vector product. In PETSc, for the built-in parallel sparse matrix formats the parallel matrix is stored in two parts: the "on-diagonal" portion of the matrix, A_d, with all the columns associated with the rows of the vector "owned" by the given process, x_d, and the "off-diagonal" portion, A_o, associated with all the other columns (whose vector values are "owned" by other processes, x_o). The sparse matrix-vector product is computed in two steps: $y_d = A_d x_d$, then $y_d = y_d + A_o x_o$. Of course, since A_o has few columns with nonzero entries, most of x_o do not need to be communicated to the given process.

PETSc manages all communication of vector entries between processes via the **VecScatter** object. For the sparse matrix-vector product vector communication, this object is created with a list of global indices indicating from where in the source vector entries are to come from and another list of indices indicating where they are to be stored into a local work vector. The vector communication itself is done in two stages: first a `VecScatterBegin()` copies the vector entries that need to be sent into message buffers, and posts nonblocking MPI receives and sends; then `VecScatterEnd()` waits on the receives and copies the results from the message buffers into the local work vector. If we let $\hat{A}_o$ denote the nonzero columns of A_o and let $\hat{x}_o$ denote the corresponding rows of x_o, then the parallel matrix-vector code is as follows.

```
VecScatterBegin(a->Mvctx, xd, hatxo,
                INSERT_VALUES, SCATTER_FORWARD);
MatMult(Ad, xd, yd);
VecScatterEnd(a->Mvctx, xd, hatxo,
              INSERT_VALUES, SCATTER_FORWARD);
MatMultAdd(hatAo, hatxo, yd, yd);
```

This same code can be used automatically when the A_d and $\hat{A}_o$ matrices are CUDA matrices. The difference from the standard case is that the `VecScatterBegin()` triggers a `VecCUDACopyFromGPU()` of the x_d vector (so that its entries are available in the CPU memory to be packed into the message buffers) and the `MatMultAdd()` triggers a `VecCUDACopyToGPU()` of the vector $\hat{x}_o$ (to move the values that have arrived from other processes down to the GPU memory). Initial profiling indicated that the needed `VecCUDACopyFromGPU()` was taking substantial time. But most entries of the x_d vector are not actually needed by the vector scatter routines, only those values that are destined for other processes that will generally be only a few percent of the values. Thus we have added the following routine, which copies only the needed values.

```
VecCUDACopyFromGPUSome(Vec,
  cusp::array1d<PetscInt,cusp::host_memory> *iCPU,
  cusp::array1d<PetscInt,cusp::device_memory> *iGPU)
```

There are two sets of identical indices, one that resides in the CPU memory and one that lives on the GPU memory, since it would be inefficient to copy the indices between the two memories on each invocation. The constructor for the **VecScatter** determines the required indices and sets them in the two memories. With the addition of this new code the required copy time decreased significantly in the parallel matrix-vector product. This change required a small amount of additional GPU-specific code in the **VecScatter** constructor and `VecScatterBegin()`.

To monitor the movement of data between the two memories, we provided two additional **PetscEvents**, one that tracks the counts and times of copies from the GPU and one for copies to the GPU. This information can be accessed with the usual PETSc `-log_summary` option. Because CUDA calls are, by default, asynchronous, meaning the function calls in the CPU return before the GPU completes the operation, we provide a global flag that forces a wait after each CUDA call until the operation is complete. This is necessary whenever one wants accurate times of the individual phases of the computation. Forcing synchronization appears to cost a few percent of the runtime; in production runs this option is not needed.

7.4 Conclusion and Future Work

We can now run parallel linear solves (with very simple preconditioners) that utilize the GPU for all vector and the matrix-vector product operation. The only vector entries that need to be passed, during the linear solve, between GPU memory and CPU memory are those destined for other processes.

This is preliminary work. Important additional work is needed in several areas.

- Performance evaluation and optimization. We have verified correctness and basic performance of the new code that utilizes the GPUs, but we have not yet done comprehensive studies.
- Matrices with structure. Many applications result in sparse matrices with particular structure, for example, adjacent rows with the same nonzero structure (called i-nodes in PETSc) or made up of small (say, three by three) blocks (here PETSc offers the BAIJ storage format). PETSc has special code that takes advantage of this structure to deliver higher performance. Another format that may be appropriate for GPUs is storage by diagonals. We need to investigate whether any of this structure be utilized on the GPU also to obtain higher performance?
- GPU-based preconditioners. The NVIDIA group is actively developing several of these, and they are easily added as new preconditioners in PETSc by simply deriving new **PC** subclasses that utilize the NVIDIA code.
- GPU-based nonlinear function evaluations. We have a simple, one-dimensional finite difference problem on a structured grid

  `$PETSC_DIR/src/snes/examples/tutorials/ex47cu.cu`

  that uses the Thrust **zip_iterator** to apply a stencil operation. As with the parallel matrix-vector product, the **VecScatter** class is used to manage the communication of ghost point values between processes. More work is needed so that copies of vectors between unghosted and ghosted representations require as few memory copies as possible between GPU and CPU. Various groups are in the process of developing or have already developed implementations of finite-element function evaluations for GPUs (Yokota et al. 2011; Klöckner et al. 2009; Wu and Heng 2004; Komatitsch and Vilotte 1998; Liu and Li 2000; Taylor et al. 2007; Keunings 1995; Abedi et al. 2006; Joldes et al. 2010). These could be used within a PETSc code.
- GPU-based Jacobian evaluations. With GPU-based Jacobian evaluations the entire nonlinear solution process (and hence also ODE integration) could be performed on GPUs without requiring any vector or matrix copies between CPU and GPU memory besides those entries required to move data between processes. This is a difficult task because the sparse matrix data structure is nontrivial and hence the efficient application of the equivalent of `MatSetValues()` on the GPU is nontrivial.

We note that because of the object-oriented design of PETSc it is possible to introduce additional vector and matrix classes ... distinctly different from those discussed

in this paper, that also use the NVIDIA GPUs. In fact, we hope there will be additional implementations to determine those that produce the highest performance.

When considering sparse matrix iterative solvers on GPUs, one must bear in mind that these algorithms are almost always memory-bandwidth limited. That is, the speed of the implementation does not depend strongly on the speed or number of the processor cores but rather on the speed of the memory. Since the best GPU systems have higher memory bandwidth than do conventional processors, one expects (and actually does see) higher floating-point rates with GPU systems; but since the memory bandwidths of GPU systems are only several times faster than those of conventional processors, a sparse matrix iterative solver converted from CPUs to GPUs will be at most only several times faster. Speedups of 100 or more are simply not possible. Additional perspectives on this issue are available in Vuduc et al. (2010).

Acknowledgments We thank Nathan Bell from NVIDIA and Lisandro Dalcin for their assistance with this project. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

Abedi R, Petracovici B, Haber R (2006) A space-time discontinuous Galerkin method for linearized elastodynamics with element-wise momentum balance. Comput Methods Appl Mech Eng 195(25–28):3247–3273

Baker C, Heroux M, Edwards H, Williams A (2010) A light-weight api for portable multicore programming. In: 18th Euromicro international conference on parallel, distributed and network-based processing (PDP), IEEE, pp 601–606

Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhäuser Press, Basel, pp 163–202

Balay S, Brown J, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2011) PETSc Web page. http://www.mcs.anl.gov/petsc

Baskaran M, Bordawekar R (2009) Optimizing sparse matrix-vector multiplication onGPUs. IBM Research Report RC24704, IBM

Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA. NVIDIA corporation, NVIDIA Technical report NVR-2008-004

Bell N, Garland M (2009) Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the conference on high performance computing networking, storage and analysis. ACM, New York, pp 1–11

Bell N, Garland M (2010) The Cusp library. http://code.google.com/p/cusp-library/

Bell N, Hoberock J (2010) The Thrust library. http://code.google.com/p/thrust/

Bolz J, Farmer I, Grinspun E, Schröoder P (2003) Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: SIGGRAPH '03: ACM SIGGRAPH 2003 papers. ACM, New York, pp. 917–924. http://doi.acm.org/10.1145/1201775.882364

Buatois L, Caumon G, Lévy B (2007) Concurrent number cruncher: an efficient sparse linear solver on the GPU. In: Proceedings of the 3rd international conference high performance computing and communications, pp 358–371

Cevahir A, Nukada A, Matsuoka S (2009) Fast conjugate gradients with multipleGPUs. Computational Science-ICCS, Springer, Heidelberg, pp 893–903

Feng Z, Li P (2008) Multigrid on GPU: tackling power grid analysis on parallel simt platforms. In: IEEE/ACM international conference on computer-aided design, ICCAD 2008, pp 647–654

Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, Salinger AG, Thornquist HK, Tuminaro RS, Willenbring JM, Williams A, Stanley KS (2005) An overview of the Trilinos project. ACM Trans Math Softw 31(3):397–423. doi http://doi.acm.org/10.1145/1089014.1089021

Heroux M et al (2009) Trilinos web page. http://trilinos.sandia.gov/

Joldes G, Wittek A, Miller K (2010) Real-time nonlinear finite element computations on GPU-application to neurosurgical simulation. Comput Methods Appl Mech Eng 199:49–52

Keunings R (1995) Parallel finite element algorithms applied to computational rheology. Comp Chem Eng 19(6):647–670

Klöckner A, Warburton T, Bridge J, Hesthaven JS (2009) Nodal discontinuous Galerkin methods on graphics processors. J Comput Phys 228(21):7863–7882. doi http://dx.doi.org/10.1016/j.jcp.2009.06.041

Komatitsch D, Vilotte J (1998) The spectral element method: an efficient tool to simulate the seismic response of 2d and 3d geological structures. Bull Seismol Soc Am 88(2):368–392

Liu R, Li D (2000) A finite element model study on wear resistance of pseudoelastic TiNi alloy. Mater Sci Eng A 277(1–2):169–175

Taylor Z, Cheng M, Ourselin S (2007) Real-time nonlinear finite element analysis for surgical simulation using graphics processing units. In: Proceedings of the 10th international conference on medical image computing and computer-assisted intervention, vol part I. Springer, Heidelberg, pp 701–708

Vuduc R, Chandramowlishwaran A, Choi JMG (2010) On the limits of GPU acceleration. In: HOTPAR: proceedings of the 2nd USENIX workshop on hot topics in parallelism, USENIX

Wu W, Heng P (2004) A hybrid condensed finite element model with GPU acceleration for interactive 3d soft tissue cutting. Comput Animat Virtual Worlds 15(3–4):219–227

Yokota R, Bardhan JP, Knepley MG, Barba L, Hamada T (2011) Biomolecular electrostatics using a fast multipole BEM on up to 512 gpus and a billion unknowns. Comput Phys Commun 182(6):1272–1283. doi:http://10.1016/j.cpc.2011.02.013; http://www.sciencedirect.com/science/article/pii/S0010465511000750

Part IV
Industrial Applications

Chapter 8
Multi-scale Continuum-Particle Simulation on CPU–GPU Hybrid Supercomputer

Wei Ge, Ji Xu, Qingang Xiong, Xiaowei Wang, Feiguo Chen, Limin Wang, Chaofeng Hou, Ming Xu and Jinghai Li

Abstract This chapter serves as an introduction to the supercomputing works carried out at CAS-IPE following the strategy of structural consistency among the physics in the simulated systems, mathematical model, computational software expressing the numerical methods and algorithms, and finally architecture of the computer hardware (Li et al., From multiscale modeling to Meso-science—a chemical engineering perspective, 2012; Li et al., Meso-scale phenomena from compromise—a common challenge, not only for chemical engineering, 2009; Ge et al., Chem Eng Sci 66:4426–4458, 2011). Multi-scale simulation of gas-solid flow in continuum-discrete approaches and molecular dynamics simulation of crystalline silicon are taken as examples, both making full use of CPU-GPU hybrid supercomputers. This strategy is demonstrated to be effective and critical for achieving good scalability and efficiency in such simulations. The software and hardware systems thus designed have found wide applications in process engineering.

8.1 Background

Process engineering is a collective term covering a wide range of industries and disciplines, from traditional chemical, metallurgical and mineral domains, to the fast-growing material, biological, pharmaceutical and cosmetic areas. Despites their apparent diversity, they do share some general activities such as the transformation and utilization of energies and resources, which are fundamental and critical for the whole society. A more intrinsic similarity underlying these activities is the vast scale gap between the products and production equipments in these industries and the multi-scale dynamic structures spanning this gap. For example, the properties and

W. Ge (✉) · J. Xu, Q. Xiong · X. Wang · F. Chen · L. Wang · C. Hou · M. Xu · J. Li
Institute of Process Engineering (IPE), Chinese Academy of Sciences (CAS),
100190 Beijing, China
e-mail: wge@home.ipe.ac.cn

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 143
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_8,

quality of the gasoline we use for our cars is determined by the molecular structures and fractions of its compositions, which is at the scale of 10^{-10}–10^{-9} m, while the reactors for refining gasoline from crude oil, such as the Fluid Catalytic Cracking (FCC) facilities, are typically 50–80 m high.

Therefore, it is not surprising that simulation of such processes has become one of the most demanding area for high performance computing. However, the actual performance of traditional simulation softwares on general purpose supercomputers is, as a whole, not impressive, and sometimes even very frustrating. In some computational fluid dynamics (CFD) simulations on commercial multi-phase reactors, the scalability is limited to dozens of CPU cores albeit more than a quarter million cores are available in modern high-end supercomputers. Even for these cores, the sustainable performance is about 10–20 % of the corresponding peak values.

In principle, this situation is not ascribed to the status of the technology for elemental components at the hardware level, but to the lack of coordination among the models, algorithms and hardwares involved in the simulations. In short, the physical world features multi-scale structures and the computer hardwares are most easily and efficiently organized in a multi-scale manner (at least in terms of their logical architecture). However, the mathematical model and numerical algorithms in traditional simulations only discretize and partition the physical system at a single scale, which incurs excessive long-range and global correlations in the model, and hence data dependence in the algorithm and communications among hardware components in execution. This is the main reason for the low efficiency and poor scalability of traditional simulation softwares in process engineering.

Based on this understanding, systematic multi-scale simulation approaches, from mathematical model to computer hardware, are implemented for gas-solid flow and crystalline silicon. All implementations have reflected the consistency among the physics, model, algorithm and hardware, which are summarized, in a more general sense, by the so-called *EMMS Paradigm* (Li et al. 2009, 2013; Ge et al. 2011).

Currently, the mainstream simulation method for gas-solid flow is the two-fluid model (TFM, Anderson and Jackson 1967; Gidaspow 1994), which treats both the gas and solid phases as continuum. It is considered advantageous for industrial simulations as its computational cost is not necessarily linked to the scale of the system, but to the number of numerical cells which is determined flexibly by the desired resolution. However, due to the intrinsic discrete nature of the solid phase, its constitutive laws as a continuum are not easily obtained, and may not exist at all. Especially, the meso-scale heterogeneity presents below the numerical grid scale proposed great challenges to quantify its statistical behavior and hence the constitutive laws. Therefore, the accuracy of TFM is not satisfactory for engineering purpose in general. On the other hand, direct discrete presentation of the solid phase, though more reasonable and simple, is far beyond the capability of current computing technology, just imaging that an industrial gas-solid reactor may contain trillions of interacting particles and advancing one particle for one time step, typically below milliseconds, may cost hundreds to thousands of flops.

Table 8.1 Specifications of the Mole-8.5 system (Wang et al. 2010, 2012; Ge et al. 2011) (adapted from Li et al. (2013), Dubitzky et al. (2012), Ge et al. (2011))

Peak performance in single precision	2.206 Petaflop/s
Peak performance in double precision	1.103 Petaflop/s
Linpack sustained performance	496.5 Teraflop/s (on 320 nodes)
Megaflop/s per Watt	963.7 (Linpack)
Number of nodes/Number of GPU's (Type)	362/2088 (Tesla C2050)
Top layer	2/0
Middle layer	18/36 (Tesla C2050)
Bottom layer	342/2052 (Tesla C2050)
Total RAM	17.8 Terabyte
Total VRAM	6.5 Terabyte
Total hard disk space	720 Terabyte
Management communication	H3C Gigabit Ethernet
Message passing communication	Mellanox infiniband quad data rate
Occupied area	150 sq.m.
Weight	12.6 ton
Max power	600 kW (computing) + 200 kW (cooling)
Operating system	CentOS 5.4, PBS
Monitor	Ganglia, GPU monitoring
Programming languages	C, C++, CUDA

In recent years, however, developments in many-core computing and coarse-grained discrete modeling begin to show the feasibility of industrial scale discrete solid phase simulation (Xu et al. 2012). Similar to pseudo-particle modeling (Ge and Li 1996, 2003), real solid particles can be presented by much less number of computational particles, whose properties can be measured in simulations and mapped physically to the solid phase (Zhou et al. 2010), which expresses the consistency among the simulated system, the physical model and the numerical method. Evolution of the computational particles features additive and localized operations which are best carried out by many-core processors, such as GPUs, in the highly parallel mode of single-instruction multi-data (SIMD). The gas flow can be solved either by traditional finite difference (FD) or finite volume (FV) methods, or by LBM methods, at scales either above or below the particle scale, which are suitable for CPUs or GPUs, respectively. Thus, the consistency among the *Four Elements* is presented, as summarized in Table 8.5, and the *EMMS Paradigm* can thus be implemented, with a preliminary version found in Ge et al. (2011).

8.2 Physical Model

Although we will focus on the algorithmic and computational aspects of the *EMMS Paradigm*, it is helpful to briefly revisit its physical background and models first. Most gas-solid systems in industries are confined in certain geometries, usually equipment walls, and are operated under steady conditions. The time-averaged steady

state distribution of the flow variables, such as gas and fluid flow velocities and solids concentration, can be predicted with reasonable accuracy by some macro-scale models, such as the global EMMS model (at the reactor level) with some empiric correlations (Ge et al. 2011; Liu et al. 2011). These distributions are then served as the initial conditions for simulating the spatio-temporal evolution of the flow structures in the systems, which basically constitutes the descriptions for the gas phase, the solid phase and their interactions, as introduced below.

The gas phase model below the particle scale is similar to single phase flow, which can be well described by the classical Navier-Stokes (N-S) equation except additional boundary conditions at particle surfaces. Above the particle scale, however, the flow structure induced by the embedded particle may cause the deviation of its effective properties (e.g., viscosity and pressure), from pure gas, and significant nonlinearity is found. Correlations for these properties can be obtained in direct numerical simulations (DNS) based on the N-S equation or Boltzmann equation. Coarse-grained LBM may provide another basis for the modeling of the gas phase, where the partial occupation of the solid phase and different permeabilities are allowed (Wang et al. 2012). With the introduction of multi-relaxation time (MRT) and large-eddy simulation (LES), and proper smoothing of the boundary configuration, the method may sustain high velocity and pressure different for the lab-scale reactor simulation (Yu et al. 2006). In all these attempts, the compressibility of gas phase can be increased to facilitate the numerical methods without affecting the accuracy very much.

The solid phase can be described either as a continuum or a discrete material. For higher resolution, the discrete description is preferred, and in order to reduce computational cost, coarse-graining of the real solid particles or description of their collective behavior is desirable. Several approaches are followed for this purpose:

Coarse-grained particles: In this approach, we try to simulate a much smaller number of elements to present the same statistical behavior of a huge number of real particles. To achieve this equivalence, the simulated particles will be, in general, more dissipative (with lower restitution) as compared to real particles, so as to maintain the energy balance, and more elastic to accommodate deformability, and less frictional to keep fluidity. The time step for these coarse-grained particles can be much larger than real solids, which further improve its efficiency. Usually, number dependence of the constitutive laws sets in when the particle number is small enough, which caps the extent of such coarse-graining.

Particle parcels: On the other hand, we may try to approximate the behavior of a swarm of particles as a single one, vividly called a parcel. Such parcels have continuous interactions with their neighbors, in a manner much more complicated than single particles, so as to account for the deformation and the exchanges of mass and momentum between the parcels. Smoothed particle hydrodynamics for the solid phase (Xiong et al. 2011) may present a framework model for the parcels with rational basis, but adjustments to its particle properties are necessary.

Particle clusters: In gas-solid systems, the particle distribution is very heterogeneous. Most particles aggregate to form islands in the gas flow field with few particles (the

so-called dilute phase). Such particle clusters can be taken as natural discrete entities for simulation purpose, and it can be larger than the coarse-grained particles or particle parcels discussed above. However, the shapes of clusters are usually very complicated and deformable, which have to be simplified drastically. The energy-minimization multi-scale (EMMS) model (Li et al. 1988; Li and Kwauk 1994), from which the EMMS paradigm is developed, can be employed as a rational basis for determining the effective size of the clusters.

Grid based approaches: Some (partially) grid-based approaches also possess particulate nature and can be used for the simulation of the solid phase. Particle in cell (PIC, Harlow 1988) methods is a hybrid Euler-Lagrange description of fluid flow, where fluid is tracked as a collection of mass carriers, statistics on these carriers are then performed via a Eulerian grid, and the continuum equations are solved on the grid numerically with the statistical data, which give the flow field. The velocities of the mass carriers are then interpolated from the grid values and their positions are updated individually, and so forth. As the solid phase is intrinsically discrete, PIC for the solids may be proven to be more reasonable (Li et al. 2012). In fact, PIC is similar to SPH expect it is partly grid-based. That means, similar difficulties will be faced, such as the collapse of particles at high concentration gradient. Insertion of a DEM core may also be helpful for this method, or otherwise, the method can be switched to DEM or parcel based methods when certain concentration or concentration gradient limits are met.

Note that, we have also listed in Table 8.5 a continuum model for the solid phase, that is, considering the solid phase as highly compressible gas with collisional cooling. However, as the numerical method for simulating such gas is explicit and lattice-based, it is algorithmically similar to particle methods with fix neighborhood. Therefore, the whole framework of the implementation is still of the continuum-particle type. The high non-linearity of the state equation of the solid phase, that is, the dramatic increase of the solid phase stress near minimum fluidization voidage, may present a difficulty.

The gas and the solid phases are coupled by the interfacial forces, mainly the drag between them. For uniform suspension of the particles, the drag can be well predicted by semi-empirical correlations, such as the Wen and Yu (1966) equation linking the drag with local slip velocity and particle concentration. Under more general conditions, the EMMS model or similar approaches (Xu et al. 2007) should be used to account for the effect of non-uniformity in the gas and/or solid phases.

8.3 Numerical Methods and Algorithm

For the physical models described above, the corresponding numerical methods can be selected or developed, and then software algorithms are designed for these methods with considerations to the computing hardware available. We will discuss the

numerical methods for the gas and solid phases, respectively, and then the major types of algorithms they can share.

8.3.1 Gas Phase Simulation

Accurate numerical methods must reflect the nature of the physical model. The gas phase in most gas-solid systems is nearly incompressible, that means flow at one location is affected by other locations simultaneously. Implicit methods are, therefore, more accurate for the gas phase because it can reflect this global dependence. However, this dependence is also expressed in its algorithm, which is boiled down to the solving of linear equation sets featuring sparse matrixes. Low computation to data accessing rate, global data dependence and hence poor scalability are the major challenges for efficient implementation of this method on massive parallel computers. Multi-core CPUs with large shared memory coupled with message passing interface (MPI) is suitable for these algorithms as explicit data communication can be minimized. But as the communication inevitably increases non-linearly with the number of CPUs involved, it is desirable to use coarse grid for the gas phase, so as to reduce the computational cost. In this regard, meso-scale models considering the distribution of gas flow in the grids and the appropriate drag law is critical for maintaining reasonable accuracy.

When high resolution of the gas phase is required, explicit methods may become more favorable, since no global data dependence and iterations are involved, and updating of the data at each grid point requires only data in neighboring grids, which allows virtually unlimited weak scalability and hence spatial scale. But this is at the price of much finer grid and time step to recovery the physical global dependence at larger spatial-temporal scales. Weak compressibility is assumed in the model, which may introduce further errors to the model, especially for the pressure distribution. These prices are paid off only when the system is large enough. Therefore, LBM and explicit FD or FV methods are more suitable for resolving the gas phase at the scale comparable or smaller than the solid entities (particles, parcels or clusters). One get-around may be provided by a modification to the physical picture of flow. At relatively high particle concentration (e.g., above 1 %), the mass of the flow is mainly carried by the solid phase, and hence the actual density distribution of the gas phase becomes less important to the flow of the mixture, as long as they provides a similar flow distribution and drag force. In this case, the compressibility of the gas phase can be increased artificially, bring the Mach number to the range of about 0.3–0.5, to validate the use of explicit numerical schemes for compressible flow. Adjustments to the drag coefficients are required to maintain the same level of inter-phase frictions. These explicit methods are intrinsically suitable to GPUs or other single-instruction multi-data (SIMD) manycore processors, which are highly parallel in computation and largely localized in memory access.

For implementation of these methods, open source or commercial software, such as Fluent (http://www.ansys.com) can be used besides development from scratch. With its user interface, we can exchange particle data with the cells of the software through files. To speedup the process, we may start multiple Fluent processes in a domain-decomposition mode, which also communicate through files. To accelerate file reading and writing, virtual disks can be installed in the memory. And most importantly, the amount of data exchanged between the solid and gas phase should be minimized. In principle, only cell averaged voidages and velocities should be included.

8.3.2 Solid Phase Simulation

Particle methods can be employed for solid phase on different coarse-graining levels with similar numerical methods and algorithms. The interactions between the particles are processed as the numerical integration of the forces between neighboring particles, which is pairwise additive and explicit, and the interactions are organized through a neighbor detecting process and followed by the updating of the particle positions. Though interactions may present the most time-consuming part of the algorithm, neighbor detection is usually the most complicated part and is critical to the efficiency of the algorithm. Cell-list and neighbor-list algorithms are the two mainstream approaches for this part, which are suitable for fast changing and more stable neighborhood, respectively. All procedures of the particle methods can be implemented on GPUs with higher speed as compared to CPUs, but extensive optimizations is necessary to reach best performance.

Note that, explicit numerical methods for continuum models are computationally a simplified form of the particles methods, where the complicated neighbor detecting process is not needed anymore. Highest performance can be achieved on GPUs with these methods, if the operations on the grid data are computationally intensive. As the solid phase is highly compressible, continuum description solved by explicit FD or discrete kinetic method (DKM) can be most efficient, though not the most accurate in general, and it is still fit well into the continuum-particle implementation of the *EMMS Paradigm*. On the other hand, the PIC method presents a hybrid continuum-particle method, where particles do not interact pairwisely, but collectively via the grids, which is also applicable to this implementation.

8.3.3 General-Purpose Particle Simulator

As we know from the discussions above, discrete particle simulation can be employed in different forms for both gas and solid phases. In a more general background, it also covers a variety of systems and processes, such as granular flow (Liu et al. 2008),

emulsions (Gao et al. 2005), polymers (Xu et al. 2010) and proteins (Ren et al. 2011), foams (Sun et al. 2007), micro-/nano-flows (Chen et al. 2008), crystals (Hou et al. 2012) and reaction-diffusion processes. The efficiency and scalability of discrete simulation was demonstrated repeatedly in these works, and the common nature of discrete methods that leads to these advantages, namely additivity and locality, is also recognized (Ge et al. 2011; Ge and Li 2000, 2002). Here additivity refers to the interactions between the particles which can be processed independently at the same time and then sum up to give the resultant force on a particle. It ensures that parallel computing can be carried out at a very fundamental level of the algorithm, that is, fine-grain parallelism. On the other hand, the locality refers to the fast decay of the strength of such interactions, so that only local interactions should be considered rigorously. It provides the parallelism at a larger scale and the weak scalability of the algorithm.

This common nature enables us to develop a general-purpose platform for particle methods at different coarse-graining levels (Ge and Li 2000, 2002; Tang et al. 2004; Wang et al. 2005), from atoms and molecules at micro-scale to boulders at macro-scale, and from real particles to more complicated discrete entities representing particle clusters. With these methods, the full range of phenomena in process engineering, from atoms to apparatus, can be simulated, the general structure, main modules and functions of the platform are summarized in Fig. 8.1.

8.3.4 GPU Implementation of the Particle Simulator

This platform for particle simulation was originally developed for CPU-based massive parallel systems. With the development of GPGPU and its programming environment, the time is ripe for transplanting the platform to CPU+GPU hybrid systems. Although other approaches, like the implicit PDE solver for the gas, have been tried with encouraging success (Wang et al. 2010), particle simulation is, in a broader sense, more suitable for GPU implementation. As detailed in Ge et al. (2011), the cell-list and neighbor-list schemes are combined in our GPU implementation, where cell list is employed to traverse all elements and find their interacting neighbors which are then put into their neighbor list. When putting the particles into cells, one thread is preferably assigned with one particle. Thanks to the atomic functions supported by Nvidia C2050 GPUs, one cell can contain several particles, but the write conflict, occurred when multiple threads write to the global memory can be avoided. The neighbor list thus generated for each particle is stored in a two dimensional array in the global memory of the GPU. In this way, although memory redundancy is unavoidable, coalesced global memory access is achieved. When generating the neighbor list, one block corresponds to one cell with each particle in it assigned to a different thread to speedup the computation. The particle information of the local and neighboring cells are buffered in the shared memory to reduce the global memory

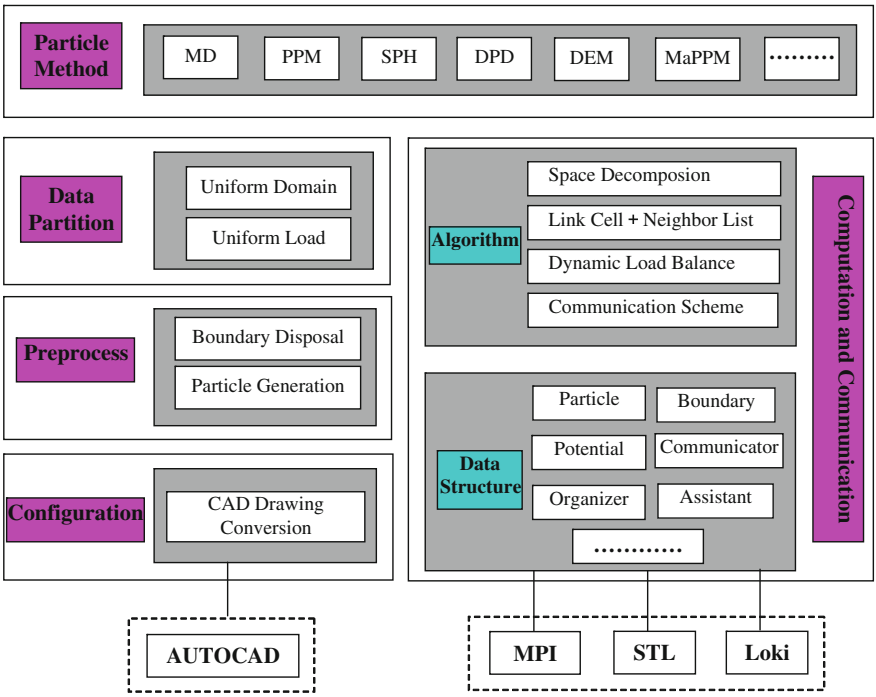


Fig. 8.1 General algorithmic platform for discrete simulation (Tang et al. 2004)

access. The overall flow chat for the general algorithm is show in Fig. 8.2 reproduced from Ge et al. (2011).

For simpler cases of fixed neighbors and for processing the interactions after the neighbors are listed, similar algorithms can be shared, also with explicit finite difference or finite volume methods, lattice-based methods and MD methods for condensed materials at low temperature. They are usually very efficient for GPUs, due to their spatial locality, natural parallelism and explicit schemes.

Though extensive optimizations are required to implement the various interactions between the discrete elements on GPUs, our emphasis has been on the effective use of the device memory bandwidth, since it is common to most methods, and it is especially important for methods with low ratios of computational operations to memory access. For best performance, the data in registers and local memories should be reused as much as possible, and storing and loading of the data to global memory should aligned and coalesces. LBM may serve as a typical example for memory bounded applications on GPUs and interesting readers are referred to our recent publications (Ge et al. 2011; Xiong et al. 2012; Li et al. 2012).

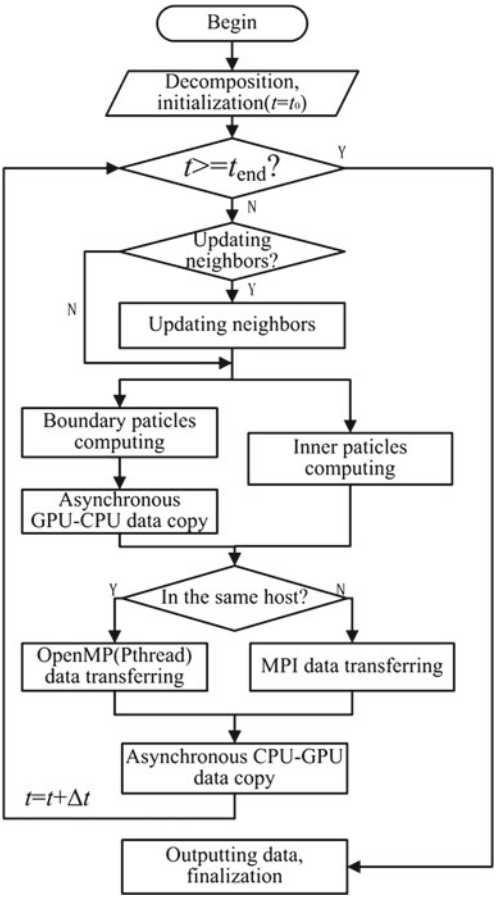


Fig. 8.2 General purpose particle simulation algorithm on multiple GPUs (adopted from Ge et al. 2011)

8.4 Hardware Development

With the development and extension of the EMMS model to different areas and the expression of the common nature of different discrete methods under the same algorithmic framework, a general multi-scale computing mode was established (Chen et al. 2009; Ge et al. 2011; Li et al. 2012) for typical complex system in process engineering. In this mode, the system is discretized on different levels. On the top and middle levels, long range interactions or correlations are treated by imposing stability conditions, which gives the global and local distribution of variables at the statistically steady state with relatively low computational cost; While on the middle and bottom levels, local interactions among the discrete elements are treated explicitly based on these distributions, reproducing the dynamic evolution of the system in detail. Taking advantage of the fast

Fig. 8.3 The Mole-8.5 system at IPE, CAS (Photo by Xianfeng He) (adopted from Li et al. 2013; Ge et al. 2011; Dubitzky et al. 2012)

distribution process, development of system behavior from the artificial initial condition to the steady state, which is of little interest to engineering practice, can be bypassed almost completely, and hence speedup the simulation considerably (Ge et al. 2011; Liu et al. 2011, 2012).

However, with traditional CPU-based supercomputers, no significant advantage of this computing mode can be demonstrated because the interactions and motion of the particles are processed with very limited parallelism as compared to its full potential. The advent of GPU computing, facilitated by CUDA^{TM}, introduced new means to implement this mode. As GPUs typically contain hundreds of relatively simple stream processors operated in the SIMD mode, they have a good balance, for discrete simulation, between the complexity of the arithmetic or logic operations that can be carried out by a stream processor and the number of parallel threads they can run. The communication among multiple GPUs may present an imperfection, as for the moment it has to resort to the PCIE bus and CPUs, or even the inter-node network, with limited bandwidth and considerable latency. However, weak scalability is still warranted for most discrete simulations.

The Mole-8.5 system (Wang et al. 2010, 2012; Ge et al. 2011; Li et al. 2013) at IPE, pictured in Fig. 8.3, is the first supercomputer using NVIDIA Tesla C2050 GPU boards in the world, reaching 1PFlops peak performance in double-precision. It was established to provide a customized hardware that can taking full advantage of the

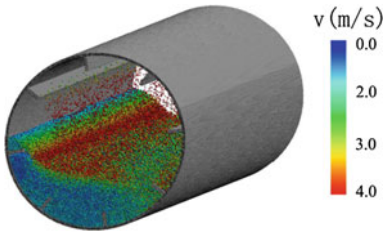


Fig. 8.4 Snapshot from the simulation of the industrial scale rotary drum (adapted from Xu et al. 2011)

CPU-GPU hybrid architecture to implement the multi-scale computing mode based on EMMS model and discrete simulations. It features a three-layer structure with increasing number of GPUs per node at lower layers, as specified in Table 8.1. We demonstrate that this design is economically profitable for most discrete simulations though it may not give good results for Linpack tests.

8.5 Applications

The multi-scale computing mode introduced above has been applied to a wide range of processes in chemical and metallurgical engineering, molecular biology and renewable energy, either for industrial designing and optimization, or for purely scientific exploration. Even a full H1N1 viron in vivo can be simulated on the molecular level at a speed of 0.77 ns per day (Xu et al. 2011). We will give some further examples below.

8.5.1 Quasi-Realtime Simulation of Rotating Drums

To demonstrate how discrete particle simulation can be accelerated by GPU or many-core computing, we carried out a DEM simulation on the granular flow in rotating drums which are widely used in process industries (Xu et al. 2011a). When a simple interaction model for smooth particles is used, each C2050 GPU can process at most about 90 million particle updates per second, about two orders faster than the serial code on CPUs. And when an industrial scale rotating drum, 13.5 m long and 1.5 m in diameter with nearly 10 million centimeter particles (a segment of the drum is show in Fig. 8.4) are simulated on 270 GPUs with message passing interface (MPI), nearly realtime speed can be achieved (Xu et al. 2011a) even when a more comprehensive tangential interaction model was added.

Table 8.2 Outline of the multi-scale approach to DNS of gas-solid suspension

System components	Physical model	Numerical method	Software algorithm	Hardware
Gas phase	Continuum (Boltzmann equation)	Lattice Boltzmann (fine grid)	Regular, explicit & local lattice operations	Linked many-core (e.g. GPU)
Solid phase	Discrete particles (Newton)	Integration of ordinary differential equations	List & arithmetic operations	Shared memory multi-core (e.g. CPU)

8.5.2 Direct Numerical Simulation of Gas-Solid Suspension

When gas-solid systems are simulated, the multi-scale computing mode can be fully exemplified (Ge et al. 2011; Xiong et al. 2012). For DNS, the consistency from the simulated system to computing hardware is detailed in Table 8.2. In this method, we have carried out the largest scale DNS of gas-solid systems so far (Ge et al. 2011; Xiong et al. 2012), which contains more than 1 million solid particles with 1 billion lattices for the gas phase in 2D, and 100 thousand particle with 500 million lattices in 3D. Some of the results are shown in Figs. 8.5 and 8.6. Some 20–60 folds speedup is obtained when comparing one GPU with one CPU core.

8.5.3 Euler-Lagrangian Simulation of Gas-Solid Fluidization

DNS of gas-solid flow has revealed unprecedented details of the flow field which is important for the establishing larger scale models for industrial applications (Xu et al. 2012). However, its direct application in industry is very limited. Most industrial simulations have employed TFM which treat both gas and solid phases as continuum and follows a Euler-Euler frame of description. This is certainly insufficient in terms of accuracy but was previously the only feasible method due to computational cost. Now with our multi-scale computing mode, a Euler-Lagranian method with less computation than DNS and higher resolution than TFM can be employed for industrial simulations (Xu et al. 2012). As detailed in Table 8.3, the solid particles (either real or coarse-grained) are still tracked one by one as in DNS, which is the Lagrangian part, but the gas flow is resolved at a scale much larger than the solid particles using continuum-based finite volume method, which constitutes the Eulerian part. With GPU computing for the Lagrangian part, its speed can be comparable to traditional TFM simulation on CPUs (Xu et al. 2012).

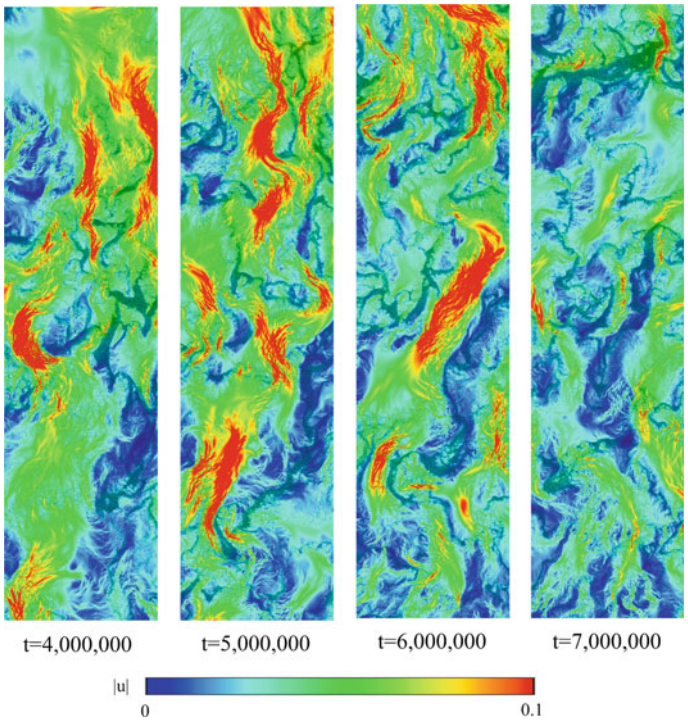


Fig. 8.5 Snapshot from 2D DNS of gas-solid suspension (adopted from Ge et al. 2011; Xiong et al. 2012)

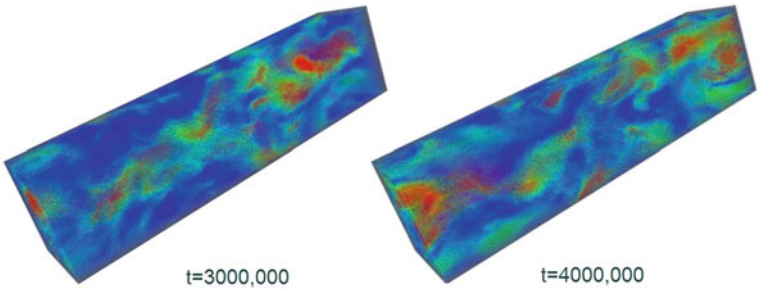


Fig. 8.6 Snapshot from 2D DNS of gas-solid suspension (adopted from Ge et al. 2011; Xiong et al. 2012)

Table 8.3 Outline of the multi-scale approach to Euler-Lagrangian simulation of gas-solid flow

System components	Physical model	Numerical method	Software algorithm	Hardware architecture
Solid phase	Particles (Newton)	ODE integration	List & arithmetic operations	Linked many-core (e.g., GPU)
Gas phase	Continuum (N-S)	PDE solver (Simple)	Sparse matrix operations	Shared memory multi-core (e.g., CPU)

Table 8.4 Outline of the multi-scale approach to atomistic simulation of crystalline silicon

System components	Physical model	Software algorithm	Hardware architecture
Bulk atoms (majority)	Regular lattices with fixed neighbors	Massive and intensive computing in SIMD-style, single precision allowed	Shared memory many-core (e.g., GPU)
Interface/defects/ dopants (minority)	Irregular lattices with dynamic neighbors	Less but more complicated computing in MIMD-style, double precision required	Linked multi-core (e.g., CPU)

8.5.4 Atomistic Simulation of Crystalline Silicon

The multi-scale computing mode can be used in areas other than multi-phase flow. One example is the atomistic simulation of crystalline silicon and its surface reconstruction (Hou and Ge 2011; Hou et al. 2012), which is of special interest to the photovoltaic and IC industrials (Hou et al. 2012). As explained in Table 8.4, the features of CPUs and GPUs, respectively, are best utilized in this mode. As a result, for bulk simulation, we have obtained 1.87 Petaflops (single precision) sustained performance on the Tianhe1A supercomputer (www.Top500.org/lists/2010/11), which has 7168 Nvidia M2050 GPUs. That is, the simulation using the multi-body Tersoff potential has reached 25.3 % of its peak performance. In fact, the instruction throughput and memory throughput on a single GPU approached 80 %. When coupled with 86016 CPU cores, the more complicated simulation on surface reconstruction also reached Petaflops sustainable performance (1.17 Petaflops in single precision plus 92 Teraflops in double precision). More than 1000 billion atoms were simulated in this case, which links atomistic behavior with macro-scale material properties.

Table 8.5 Simulation of gas-solid flow in the EMMS paradigm

Simulated system		Physical model		Numerical method		Software algorithm		Hardware architecture
Macro-scale distribution	Global Local	EMMS global EMMS local		Nonlinear equation set Multi-objective optimization		Iteration; GA/ANN/PSO		Single node: CPU+multi-GPU
Meso-scale evolution	Inter-phase	Drag/flow distribution		Interpolation/mapping		Reduce/broadcast		CPU+ inter-layer communication
	Gas phase	Continuum	Above particle scale (incompressible)	Partial differential equation: Implicit finite difference (volume)		Sparse matrix operations		Multi-CPU: Intra-node—shared memory; Inter-node—neighborhood communication
			Below particle scale	Explicit finite difference (volume)	LBM		Particle Interaction & updating	Multi-GPU Additive operation
			Above particle scale (compressible)		Discrete kinetics			Regular adjacent communication
Micro-scale evolution	Solid phase	Discrete	Coarse grain	Ordinary differential equation		Particle searching		Multi-GPU Additive operation
			Soft parcel Deformable cluster	Numerical integration				Irregular local communication
		Hybrid	PIC	Hybrid		Hybrid		Hybrid

8.6 Conclusions

In summary, structural consistency among the hardware, software, model and the system to be simulated is critical for the high efficiency of supercomputing. The continuum-discrete implementation of the so-called EMMS paradigm can take full advantage of the CPU-GPU hybrid computing mode and unprecedented simulation results on multi-phase systems or even beyond have been or can be obtained in this paradigm. The prospects of simulating industrial scale multi-phase systems at almost realtime with reasonable accuracy and resolution, or in short, virtual process engineering is not remote considering the dramatic development of both the hybrid computing mode and its hardware developments.

Acknowledgments We thank all members of the EMMS group at IPE for their long term collaboration and support on this work. This work is sponsored by National Natural Science Foundation of China under the Grant no. 20821092, Ministry of Finance under the Grant no. ZDYZ2008-2, Chinese Academy of Sciences under the Grants nos. KGCX2-YW-124 and KGCX2-YW-222. We also thank Nvidia for sponsoring the CUDA Center of Excellence (CCOE) at IPE.

References

Anderson TB, Jackson R (1967) A fluid mechanical description of fluidized beds: equations of motion. Ind Eng Chem Fundam 6:527–539

Chen F, Ge W, Wang L, Li J (2008) Numerical study on gas-liquid nano-flows with pseudo-particle modeling and soft-particle molecular dynamics simulation. Microfluid Nanofluid 5:639–653

Chen F, Ge W, Guo L, He X, Li B, Li J, Li X, Wang X, Yuan X (2009) Multi-scale HPC system for multi-scale discrete simulation-development and application of a supercomputer with 1 Petaflops peak performance in single precision. Particuology 7:332–335

Dubitzky (2012) Large-scale computing techniques for complex system simulations. Wiley

Gao J, Ge W, Hu G, Li J (2005) From homogeneous dispersion to MicellesA molecular dynamics simulation on the compromise of the hydrophilic and hydrophobic effects of sodium dodecyl sulfate in aqueous solution. Langmuir 21:5223–5229

Ge W, Li J (1996) Pseudo-particle approach to hydrodynamics of gas-solid two-phase flow. In: Kwauk M, Li J (eds) Proceedings of the 5th international conference on circulating fluidized bed. Science Press, Beijing, pp 260–265

Ge W, Li J (2000) Conceptual model for massive parallel computing of discrete systems with local interactions. Comput Appl Chem 17(5): 385–388. (Chinese)

Ge W, Li J (2002) General approach for discrete simulation of complex systems. Chin Sci Bull 47(14):1172–1175

Ge W, Li J (2003) Macro-scale phenomena reproduced in microscopic systems: pseudo particle modeling of fluidization. Chem Eng Sci 58:1565–1585

Ge W, Wang W, Yang N, Li J, Kwauk M, Chen F, Chen J, Fang X, Guo L, He X, Liu X, Liu Y, Lu B, Wang J, Wang J, Wang L, Wang X, Xiong Q, Xu M, Deng L, Han Y, Hou C, Hua L, Huang W, Li B, Li C, Li F, Ren Y, Xu J, Zhang N, Zhang Y, Zhou G, Zhou G (2011) Meso-scale oriented simulation towards virtual process engineering (VPE)-The EMMS paradigm. Chem Eng Sci 66(19):4426–4458

Gidaspow D (1994) Multiphase flow and fluidization: continuum and kinetic theory description. Academic Press, Boston

Harlow FH (1988) PIC and its progeny. Comput Phys Comm 48:1–10

Hou C, Xu J, Wang P, Huang W, Wang X, Shen G, Ge W, He X, Guo L, Li J (2012) Petaflops molecular dynamics simulation of crystalline silicon on Tianhe-1A. Int J High Perform Comput (In print) Doi:10-1177/1094342012456047

Hou C, Xu J, Ge W, Wang P, Huang W, Wang X (2012) Efficient GPU-accelerated molecular dynamics simulation of solid covalent crystals. Comput Phys Comm Accepted

Hou C, Ge W (2011) GPU-accelerated molecular dynamics simulation of solid covalent crystals. Mol Simul 38(1):8–15

Li J, Kwauk M (1994) Particle-fluid two-phase flow: the energy-minimization multi-scale method. Metallurgical Industry Press, Beijing, P. R. China

Li J, Tung Y, Kwauk M (1988) Multi-scale modeling and method of energy minimization in particle-fluid two-phase flow. In: Basu P, Large JF (eds) Circulating fluidized bed technology II. Pergamon Press, Toronto, pp 89–103

Li F, Song F, Benyahia S, Wang W, Li J (2012) MP-PIC simulation of CFB riser with EMMS- based drag model. Chem Eng Sci 82(12):104–113

Li J, Ge W, Kwauk M (2009) Meso-scale phenomena from compromise - a common challenge, not only for, chemical engineering arXiv:0912.5407

Li J, Ge W, Wang W, Yang N, Liu X, Wang L, He X, Wang X, Wang J, Kwauk M (2013) From multiscale modeling to Meso-science - a chemical engineering perspective. Springer (In print), Berlin

Liu X, Ge W, Li J (2008) Non-equilibrium phase transitions in suspensions of oppositely driven inertial particles. Powder Technol 184:224–231

Liu Y, Chen J, Ge W, Wang J, Wang W (2011) Acceleration of CFD simulation of gas-solid flow by coupling macro-/meso-scale EMMS model. Powder Technol 212:289–295

Liu X, Guo L, Xia Z, Lu B, Zhao M, Meng F, Li Z, Li J (2012) Harnessing the power of virtual reality. Chem Eng Prog 108(7):28–33

Ren Y, Gao J, Xu J, Ge Wei, Li Jinghai (2011) Key factors in chaperonin-assisted protein folding. Particuology 10(1):105–116

Sun Q, Ge W, Huang J (2007) Influence of gravity on narrow input forced drainage in 2D liquid foams. Chin Sci Bull 52:423–427

Tang D, Ge W, Wang X, Ma J, Guo L, Li J (2004) Parallelizing of macro-scale pseudo-particle modeling for particle-fluid systems. Sci China Ser B Chem 47(5):434–442

Wang X, Ge W, He X (2010) Development and application of a HPC system for multi-scale discrete simulation-Mole-8.5. In: International supercomputing conference. Hamburg, Germany

Wang X, Ge W (2012) The Mole-8.5 supercomputing system. In: Vetter JS (ed) Contemporary high performance computing: from petascale toward exascale. Taylor and Francis, Boca Raton

Wang X, Guo L, Ge W, Tang D, Ma J, Yang Z, Li J (2005) Parallel implementation of macro-scale pseudo-particle simulation for particle-fluid systems. Comput Chem Eng 29:1543–1553

Wang J, Xu M, Ge W, Li J (2010) GPU accelerated direct numerical simulation with SIMPLE arithmetic for single-phase flow. Chin Sci Bulletin 55:1979–1986

Wang L, Zhang B, Wang X, Ge W, Li J (2012) Lattice Boltzmann based discrete simulation of gas-solid fluidization. Chin Sci Bull, Accepted

Wen CY, Yu YH (1966) Mechanics of fluidization. Chem Eng Progr Symp Ser 62:100–111

Xiong Q, Deng L, Wang W, Ge W (2011) SPH method for two-fluid modeling of particle-fluid fluidization. Chem Eng Sci 66:1859–1865

Xiong Q, Li B, Zhou G, Fang X, Xu J, Wang J, He X, Wang X, Wang L, Li J (2012) Large-scale DNS of gas-solid flows on Mole-8.5. Chem Eng Sci 71:422–430

Xu M, Ge W, Li J (2007) A discrete particle model for particle-fluid flows with considerations of sub-grid structures. Chem Eng Sci 62:2302–2308

Xu J, Ren Y, Ge W, Yu X, Yang X, Li J (2010) Molecular dynamics simulation of macromolecules using graphics processing unit. Mol Simul 36:1131–1140

Xu J, Wang X, He X, Ren Y, Ge W, Li J (2011) Application of the Mole-8.5 supercomputer: probing the whole influenza virion at the atomic level. Chin Sci Bull 56(20):2114–2118

Xu J, Qi H, Fang X, Lu L, Ge W, Wang X, Xu M, Chen F, He X, Li J (2011a) Quasi-real-time simulation of rotating drum using discrete element method with parallel GPU computing. Particuology 9:446–450

Xu M, Chen F, Liu X, Ge W, Li J (2012) Discrete particle simulation of gas-solid two-phase flows with multi-scale CPU-GPU hybrid computation. Chem Eng J 207–208:746–757

Yu H, Luo L-S, Girimaji SS (2006) LES of turbulent square jet flow using an MRT lattice Boltzmann model. Comput Fluids 35:957–965

Zhou G, Ge W, Li J (2010) Smoothed particles as a non-Newtonian fluid: a case study in Couette flow. Chem Eng Sci 65:2258–2262

Chapter 9
GPU Best Practices for HPC Applications at Industry Scale

Peng Wang and Stan Posey

Abstract Current trends in high performance computing (HPC) are moving towards the availability of several cores on the same chip of contemporary processors in order to achieve speed-up through exploiting the potential of fine-grain parallelism in applications. The trend is led by graphics processing units (GPUs) which have recently been developed exclusively for computational tasks as massively-parallel co-processors to conventional x86 CPUs. Since the introduction in 2006 of the NVIDIA Tesla GPU and CUDA programming environment, the HPC community has achieved noted performance gains across a broad range of application software. In particular, various scientific research disciplines within computational physics and chemistry have reported performance levels as high as two orders of magnitude over current quad-core CPUs. During 2010 an extensive set of new HPC architectural features were offered in the third generation Tesla and CUDA (codenamed Fermi), giving engineering disciplines a similar opportunity to expand use of GPUs for applications relevant to industry modeling and simulation. Similar to the scientific research community, practical applications in industry observe constant growth in model fidelity, but parallel efficiency of commercial software and job completion times also become important factors behind decisions on model size and scale, and level of physics features to include. This work examines algorithmic development best practices, and performance results of application software for the Tesla Fermi architecture in modelling and simulation examples relevant to industry-scale HPC practice. Included are GPU implementations of computational structural mechanics (CSM) and computational fluid dynamics (CFD) software that support mechanical product design in manufacturing industries. Specifically, the critical requirements of memory optimization and storage formats are discussed for grid-based direct solvers

P. Wang (✉) · S. Posey
NVIDIA, Santa Clara, CA, USA
e-mail: penwang@nvidia.com

S. Posey
e-mail: sposey@nvidia.com

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 163
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_9,

that appear in CSM and for highly irregular sparse matrices that require iterative solver schemes in CFD.

9.1 Introduction

The continual increase in CPU speeds is limited due to power and thermal constraints for processors with multiple CPU cores. To achieve boosts in performance without increasing clock speeds parallelism must be developed. This parallelism can come in the form of task parallelism, data parallelism, or even a combination of the two. Common methods for implementing parallelism are explicit message-passing using an MPI library for either distributed or shared memory systems, and OpenMP for shared memory systems. A hybrid method is also possible with OpenMP used on multi-core and multiprocessor nodes and MPI used among the nodes.

Although parallel applications that use multiple cores are a well established technology in computational science and engineering (CSE), a recent trend towards the use of GPUs to accelerate CPU computations is now common. In this heterogeneous computing model the GPU serves as a co-processor to the CPU. The need for high performance and the parallel nature of CSE problems has led GPU designers to create current designs with hundreds of cores. Today GPUs and software development tools are available for implementing more general applications that use the GPU for applications such as CSM, CFD and others where computations are needed to be completed as fast as possible.

Much work has recently been focused on GPUs as devices that can produce a very high FLOPS (floating-point operations per second) rate if an algorithm is well-suited for the device. There have been several studies illustrating the performance gains that are possible by using GPUs, but a modest number of commercial CSM and CFD software has yet to make full use of them, and those that have demonstrate nominal overall gains of $2\times$ to $3\times$ over multi-core CPUs. This is mostly due to the current focus of linear equation solvers for GPUs rather than complete implementations, which will come in progressive stages.

The GPU was originally designed for graphics and the majority of this computation involves computing the individual color for each pixel on a screen. If we consider each pixel as similar to a quadrilateral element or cell, computing the pixel colors will be similar to the computations on a structured mesh. There is a very regular, orderly data access pattern with neighbors easily computed by simple offsets of indices. However, the majority of commercial CSM and CFD software use some form of an unstructured mesh often with hexahedrals and tetrahedrals in 3D. These meshes lead to an irregular and somewhat disorderly data access pattern which is not particularly well suited to the memory system of a GPU.

Through a careful process of analyzing the relation between finite elements and and/or finite volumes and vertices, and taking advantage of the large amount of available processor performance on a GPU, techniques can be applied that partitions and sorts the mesh in such a way that the data access pattern becomes much more

regular and well-ordered. The pre-processing of the mesh connectivity is a one-time step performed just before the main computation begins and can require a negligible amount of compute time while significantly increasing the performance of the equation solver.

Shared memory is an important feature of the GPU and is used to avoid redundant global memory access among threads within a block. The GPU does not automatically make use of shared memory, and it is up to the software to explicitly specify how shared memory should be used. Thus, information must be made available to specify which global memory access can be shared by multiple threads within a block. For structured grid based solvers, this information is known up-front due to the fixed memory access pattern of such solvers, whereas the memory access pattern of unstructured grid based solvers is data-dependent.

Algorithm design for optimizing memory access is further complicated by the number of different memory spaces the application must take into consideration. Unlike a CPU the memory accesses are under the full and manual control of the developer. There are several memory spaces on the GPU which in turn is connected to the CPU memory. Different memory spaces have different scope and access characteristics: some are read-only, some are optimized for particular access patterns. Significant gains (or losses) in performance are possible depending on the choice of memory utilization.

Another issue to be considered for GPU implementation is that of data transfers across the PCI-Express bus which bridges the CPU and GPU memory spaces. The PCI-Express bus has a theoretical maximum bandwidth of 4–8 GB/s depending on whether it is of generation 1 or 2. When this number is compared to the bandwidth between the GPU's on-board GDDR5 memory and the GPU multi-processors (up to 141.7 GB/s), it becomes clear that an algorithm that requires a large amount of continuous data transfer between the CPU and GPU will unlikely achieve good performance.

For a CSM or CFD solver, the obvious solution is to limit the size of the domain that can be calculated so that all of the necessary data can be stored in the GPU's main memory. Using this approach, it is only necessary to perform large transfers across the PCI-Express bus at the start of the computation (geometry) and at the end (final solution). High-end NVIDIA GPUs offer up to 6 GB of main memory, sufficient to store all the data needed by most commercial parallel engineering software, so this restriction is not a significant limitation.

Over the past 10 years, CSM and CFD software has become increasingly reliant on clusters of multiprocessors to enable more detailed simulations within design time frames. For this reason, the scalability of a solver across multiple servers is equally important as its single-processor performance. A potential problem with increasing single-processor performance by an order of magnitude is that the multi-processor performance suffers since the time required to exchange boundary information remains roughly constant. When operating in parallel across multiple GPUs, some boundary information must be transferred across the PCI-Express bus at the end of each time step. However, with implementation of a low surface-to-volume ratio in mesh partitioning, this data transfer need not be a bottle-neck.

The relative performance of processors versus memory over the past few decades has extended to more than three orders of magnitude. CPUs have gone to great lengths to bridge the gap between processor and memory performance by introducing instruction and data caches, instruction level parallelism, and so forth. And although GPUs offer a different approach in terms of hiding memory latency because of their specialization to inherently parallel problems, the fact remains that processor performance will continue to advance at a much greater rate than memory performance. If we extrapolate out, without any fundamental changes in memory, processors will become infinitely fast relative to memory, and performance optimization will become solely an exercise in optimizing data movement.

9.2 Direct Sparse Solvers

Direct sparse solvers are widely used in the discipline of computational structural mechanics (CSM) for simulations that deploy implicit schemes. Multi-frontal direct sparse solver techniques are the most common algorithms for commercial software. In a multi-frontal solver, an assembly tree is built for the sparse matrix, whose nodes are dense matrices. The factorization of the sparse matrix is based on the factorization of those dense matrices (fronts) and their assembly into super-nodes. Most of the runtime of a direct sparse solver is spent in the factorization of the frontal matrices and their assembly.

The typical strategy of developing a direct sparse solver for the GPU is to offload those dense matrix operations to the GPU, which typically can be very efficient for a massively parallel GPU architecture. Other solver operations such as the matrix assembly, tree transversal, and forward/backward solve can all stay on CPU, since they are difficult to parallelize on the GPU and typically require only a small fraction of the total solution time.

For application to a symmetric matrix, one requirement is that the matrix is stored in a packed format but high performance GPU kernels like DGEMM (double precision matrix-matrix multiply) require a regular 2D data layout. There are in general two approaches to manage this requirement. First, one can perform a data format transformation to transfer the packed format to a block data structure, and the dense matrix GPU kernels can then be applied to each block. Since dense matrix GPU kernel are of $O(n^3)$ operations and format transformations of $O(n^2)$, there would not be significant overhead.

As a second approach, one could rewrite the dense matrix GPU kernels to handle a packed data format directly. The complexity for this would require an understanding of the correct index mapping for this packed storage format, otherwise the kernel would behave largely the same as in the first approach.

Since matrix fronts are copied to the GPU one-by-one for computation and then copied back to the CPU afterwards, there are two potential performance bottlenecks. First, for finite element models with many “holes” and/or thin shells, the fraction of runtime for small fronts will increase. Generally, a GPU requires a dense matrix to

be larger than ∼1K to reach peak performance. For many small matrices, this could require a significant fraction of the factorization phase and the overall performance could be well below the peak GPU performance.

The solution to a more general direct solver GPU approach that could benefit a broader range of model geometries is to use the concurrent kernel feature introduced in the Fermi architecture. Since the matrix fronts on the same assembly tree level can be processed independently, their GPU kernels can run concurrently on the GPU, and thereby fully utilizing the hardware for peak performance. Another potential problem is overhead associated with CPU-GPU data transfers. Knowing that fronts on the same assembly tree level can be processed independently, one could overlap CPU-GPU data transfer of one front with the kernel computation of another front, which is a supported feature on Fermi GPUs.

9.2.1 Example: LS-DYNA Direct Solver

An example of a GPU-accelerated direct solver for CSM is provided with LS-DYNA commercial software based on the finite element method and developed by Livermore Software Technology Corporation, with headquarters in Livermore, CA, USA. LS-DYNA is a multi-purpose structural and fluid analysis software for high-transient, short duration structural dynamics, and other multiphysics applications. Considered one the most advanced nonlinear finite element programs available today, LS-DYNA has proved an invaluable simulation tool for industry and research organizations who develop products for automotive, aerospace, power-generation, consumer products, and defense applications, among others. Complexities arise from simulations in these industries since they often require predictions of surface contact and penetration, models for loading and material behavior, and accurate failure assessment.

Performance and parallel efficiency for LS-DYNA is dependent upon many elements of a system's architecture and the geometry and conditions of the simulation. Structural simulations in LS-DYNA often contain a mix of materials and finite elements that can exhibit substantial variations in computational expense, which may create load-balance complexities. The ability to efficiently scale to a large number of processors is highly sensitive to load balance quality of computations. This study of LS-DYNA performance does not yet consider multi-node distributed parallel, and instead investigates the parallel efficiency that GPUs can provide for single node computations.

GPU development of LS-DYNA has been limited to implicit modeling at the time of this paper, and specifically the use of a multi-frontal sparse direct solver that usually requires out-of-memory solution processing. This occurs because the stiffness matrix that must be factorized is typically much larger than the allowable memory for a particular server or set of cluster nodes. The model for the study however is small enough to reside in-memory and is comprised of more than 500 K DOFs. The geometry is a set of concentric cylinders and a description of the model and conditions is provided in Fig. 9.1.

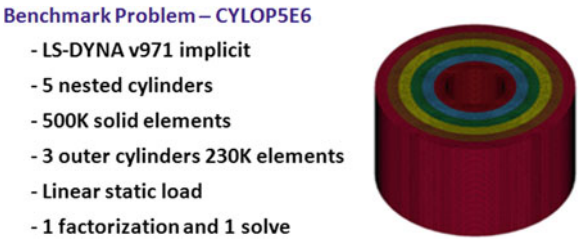


Fig. 9.1 Description of the LS-DYNA model for GPU performance tests

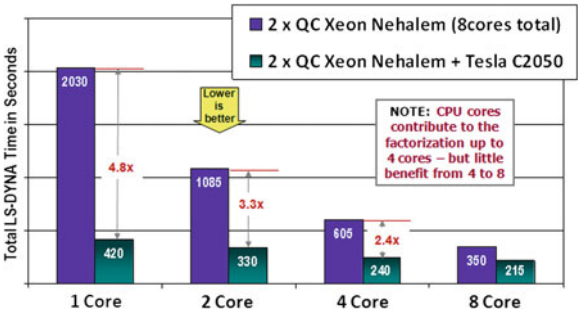


Fig. 9.2 Performance results of LS-DYNA wall time for model CYLOP5E6

The system used for the study contained an Intel Xeon 5500 Nehalem based CPU with dual sockets of 4 cores each for a total of 8 cores, and a Tesla C2050 GPU with 448 cores and 3 GB of main GPU memory. For the CYLOP5E6 model the matrix is symmetric with a rank of 760320, and its diagonal and lower triangle contain 29213357 non-zero entries. After reordering with METIS, it required 7.104E+12 operations to factor the matrix, and the resulting factored matrix contains 1.28E+09 entries.

The total time required for LS-DYNA including factorization of the matrix is shown in Fig. 9.2, as a function of the number of cores employed, both with and without the GPU. The dual socket Nehalem host sustains 10.3 GFLOPS when using one core, and 59.7 GFLOPS when using all eight. When the GPU is used, the benchmark performs 6.57E+12 operations or 92 % of the total required and sustained 98.1 GFlop/s peak. Overall performance with the GPU improves to 61.2 GFLOPS when one core is used, and 79.8 GFLOPS with all 8.

The results demonstrate that a GPU could substantially accelerate LS-DYNA when comparing wall clock times. In the case of 1 core operating with the GPU, the speedup is nearly 5×, but note that the optimal use of the combined CPU + GPU solution is at 4 cores of the CPU. At this level the acceleration is 2.4× and provides a total job time of just 240 s. This time can be lowered slightly to 215 s with the use of the 4 remaining cores, but is not an optimal use.

9.3 Iterative Sparse Solvers

Iterative sparse solvers are widely used in the discipline of computational fluid dynamics (CFD) for simulations that deploy implicit schemes. For CFD iterative solvers are most commonly owing to their computational and storage efficiency. Implementation of an iterative solver such as a conjugate gradient method on GPU would consist mainly of a sparse-matrix-vector-multiply (SpMV) kernel, and a few additional BLAS-1 kernels.

Because of an SpMV kernel's low arithmetic intensity, it is highly memory-bound, and therefore optimization of the memory access pattern to achieve peak GPU memory-bandwidth is the primary consideration. Even though the data access pattern is irregular for general SpMV, with careful re-design of data structures for the sparse matrices, it is possible for the SpMV kernel to run very close to the peak of the GPU memory bandwidth.

Currently a very successful GPU data structure for sparse matrix is the Ellpack format (ELL). For the ELL format, one uses a 2D regular storage space for the matrix nonzero elements and their column indices and the row size of the 2D storage would just be the matrix row size. For the column size, because of the regular storage requirement, the size would be that of the largest number of nonzero row elements for the matrix. With the ELL format, the GPU kernel would use one thread per row. In this way, the 2D regular storage enables fully coalesced access to the matrix elements. In addition, since neighboring rows have generally a good chance of reusing the same vector elements, binding the vector to texture cache has been shown to lead to up to a 30 % performance increase.

These techniques for SpMV on a GPU can reach very close to the memory-peak, which would deliver ∼15 GFLOPS Tesla C2050 GPU. For each SpMV operation using ELL format, the flop count is 2 (1 add and 1 multiply), and gives an arithmetic intensity 0.1 when divided by the number of 20 bytes loaded, (8 bytes for matrix elements, 8 bytes for vector elements, and 4 bytes for column indices). Therefore when SpMV is computing at close to the memory bandwidth peak of around 150 GB/s for the Tesla C2050 (with ECC turned off), the flop rate would be 150*0.1 or 15 GFLOPS.

One possible concern with the ELL format is that its column size needs to be the largest row size in the matrix. This could potentially lead to a large waste of storage space. The solution to this problem is to use a hybrid (HYB) format. The idea of the HYB format is to simply set the size of the ELL format column to be the typical number of non-zero rows in the matrix. This technique also requires use of another addition storage format, such as the coordinate (COO) format, in order to store the exceptional elements in rows that exceed the typical number of non-zeros.

For the HYB technique, most of the matrix elements will be in ELL format, so the use of COO format would not reduce performance significantly. The HYB format with carefully chosen parameters, can achieve a good balance between performance and storage efficiency. The additional parameters in HYB looks like an additional complication, but in its implementation, one could design it in a way to request the HYB format parameters at runtime.

With an efficient SpMV kernel, an iterative method such as a conjugate gradient can be implemented fully on a GPU, however preconditioners are usually required. All contemporary CFD iterative solvers use a preconditioner to speed-up the convergence rate of a solver, and this would also need to be implemented on the GPU for overall performance. The difficulty of this implementation varies significantly depending on the specific preconditioner used. For example, a GPU-efficient Jacobi preconditioner would be very simple to implement, however a highly sequential preconditioner such as an incomplete Cholesky scheme, would be difficult for good parallel efficiency. This trade-off may motivate a redesign of a particular preconditoner in order to run massively parallel on a GPU.

9.3.1 Example: AcuSolve Iterative Solver

An example of GPU-accelerated iterative solver for CFD is provided with AcuSolve commercial software based on the finite element method and developed by ACUSIM, with headquarters in Mountain View, CA, USA. AcuSolve is based on the Galerkin Least-Squares (GLS) finite element method. GLS is a higher-order accurate, yet stable formulation that uses equal order nodal interpolation for all variables, including pressure. The method is specifically designed to maintain local and global conservation of relevant quantities under all operating conditions and for all unstructured mesh types. AcuSolve utilizes an efficient iterative solver for fully coupled pressure-velocity equation systems which like most commercial CFD has sparse matrix-vector (SpMV) multiplies as its primary operation.

SpMV is of singular importance in sparse linear algebra. In contrast to the uniform regularity of dense linear algebra, sparse operations encounter a broad spectrum of matrices ranging from the regular to the highly irregular. Exploiting the tremendous potential of throughput-oriented GPU processors for sparse operations requires that a solver expose substantial fine-grained parallelism and impose sufficient regularity on execution paths and memory access patterns. Optimizing SpMV for GPUs is qualitatively different than SpMV on latency-oriented multi-cores. Whereas a multi-core SpMV kernel needs to develop 4 or more threads of execution, a many-core implementation must distribute work among thousands or tens of thousands of threads. Many-core GPUs will often demand a high degree of fine-grained parallelism because, instead of using large sophisticated caches to avoid memory latency, they use hardware multithreading to hide the latency of memory accesses.

The linear equation solver in AcuSolve was examined for GPU acceleration for a relatively simple case of 80,000 elements for an S-Duct geometry. AcuSolve utilizes a hybrid parallel MPI/OpenMP approach with an iterative GMRES solver. An execution profile of AcuSolve for the S-Duct case appears in Fig. 9.3 and demonstrates the dominant feature of SpMV operations as 57 % of the total execution time. As described in a previous section, even a complete port of SpMV to the GPU will still leave 43 % of execution time on the CPU and limit the overall effective speed-up to $\sim 2\times$.

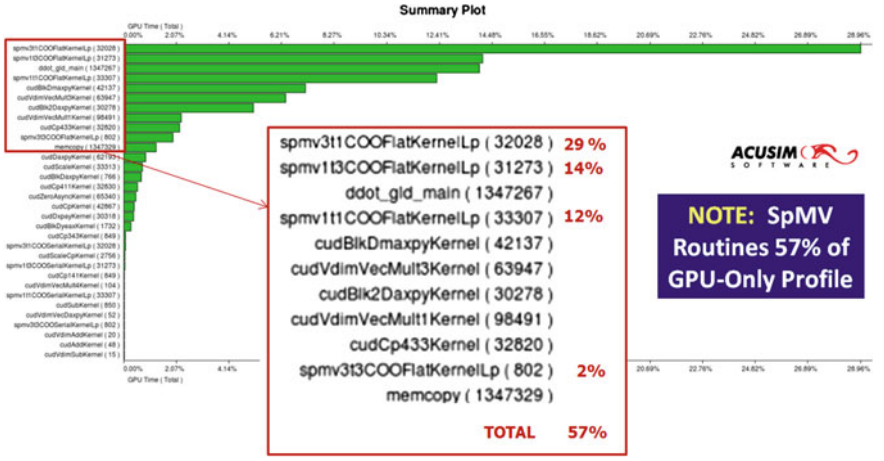


Fig. 9.3 Execution profile of AcuSolve and the S-Duct case of 80,000 DOF

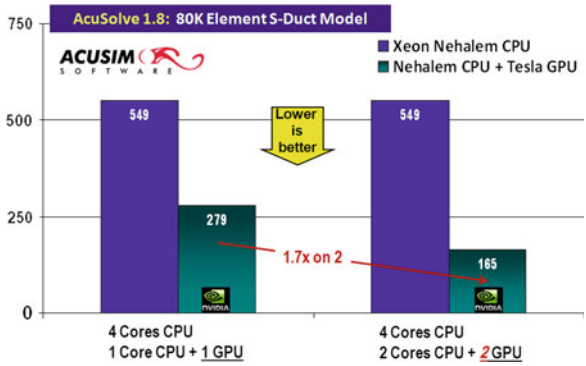


Fig. 9.4 Results for AcuSolve 1.8 on multi-GPUs versus Quad Core Xeon CPU

Performance results for the S-Duct case are shown in Fig. 9.4 and feature a comparison between a quad-core Xeon 5500 series CPU (Nehalem) and a single NVIDIA Tesla C2050 (Fermi). The speed-up of the 1 core + GPU is ~2× and this improves to 3.3× with the use of 2 GPUs owing to the distributed memory parallel implementation in AcuSolve. ACUSIM has additional GPU development projects underway to improve the performance for future releases.

9.4 Conclusion

Increased levels of parallel processing for industry-relevant software by utilizing GPUs in an HPC environment is enabling much larger and complex simulations to be addressed in product development workflows. As CSM and CFD simulation

requirements continue to grow such as the need for transients, high-resolution, and multi-scale simulations that are all heavy in numerical operations, hybrid parallel systems of CPUs and GPUs will become an essential HPC technology.

The heterogeneous nature of such high fidelity simulations and their HPC resource usage will continue to grow the requirements for balanced GPU-based server nodes within distributed memory clusters. It was demonstrated that substantial CSM and CFD performance gains can be achieved by using the latest GPU technology, the NVIDIA Tesla 20-series by acceleration of applications on CPUs. Based on these trends, we can expect that GPU acceleration will be a meaningful HPC trend in the future of scientific computing and the future of scientific computing and engineering modeling and practice.

References

ACUSIM Corporation and AcuSolve: www.acusim.com

Andrew C, Fernando C, Rainald L, John W In: 19th AIAA computational fluid dynamics, June 22–25, San Antonio, Texas

Brandvik T, Pullan G (2009) An accelerated 3D Navier-Stokes solver for flows in turbomachines. In: Proceedings of GT2009 ASME turbo expo, (2009) power for land, sea and air, June 8–12, Orlando, USA

Kodiyalam S, Kremenetsky M, Posey S (2007) Balanced HPC infrastructure for CFD and associated multidiscipline simulations of engineering systems. In: Proceedings of the 7th Asia CFD conference 2007, Nov 26–30, Bangalore, India

LS-DYNA User's Manual Version 971, Livermore Software Technology Corporation, Livermore, CA, 2007

Lucas R, Wagenbreth G, Davis D (2007) Implementing a GPU-enhanced cluster for large scale simulations. In: I/ITSEC, Orlando, FL, USA

Michalakes J, Vachharajani M (2008) GPU acceleration of numerical weather prediction. Parallel Process Lett 18(4):531–548

NVIDIA Corporation (2008) NVIDIA CUDA compute unified device architecture 2.0 programming guide

Palix Technologies, LLC (2010) http://www.palixtech.com. Advanced numerical design solver and solver white paper

Chapter 10
Simulation of 1D Condensing Flows with CESE Method on GPU Cluster

Wei Ran, Wan Cheng, Fenghua Qin and Xisheng Luo

Abstract We realized the space-time Conservation Element and Solution Element method (CESE) on GPU and applied it to condensation problem in a 1D infinite length shock tube. In the present work, the CESE Method has been implemented on a graphics card 9800GT successfully with the overlapping scheme. Then the condensation problem in 1D infinite shock tube was investigated using the scheme. The speedup of the condensation problem with the overlapping schemes is 71× (9800GT to E7300). The influence of different meshes on the asymptotic solution in an infinite shock tube with condensation was studied by using the single GPU and GPU cluster. It is found that the asymptotic solution is trustable and is mesh-insensitive when the grid size is fine enough to resolve the condensation process. It is worth to mention that the peak value of computing reaches 0.88 TFLOPS when the GPU cluster with 8 GPUs is employed.

Keywords GPU cluster · CESE method · Shock tube · Condensation

10.1 Introduction

Historically, the CFD has apace developed due to the rapid increase of CPU performance and the swift reduction of the hardware price. However, with the limitation of physics, the CPU's performance can't be improved easily by raising clock frequency as ever (Kish 2002). To increase the computational ability of CPU, a large and expensive cache is integrated and many-core system has been employed (Geer 2005). The small scale problems of CFD can be fulfilled by a PC of multi-core CPU with shared memory. For a large scale problem, however, a PC with a few cores can't offer enough computational capability and the cluster with many CPU

W. Ran (✉) · W. Cheng · F. Qin · X. Luo
Department of Modern Mechanics, University of Science and Technology of China, 230026 Hefei, P.R. China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 173
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_10,

cores is needed. Nevertheless, the memory bottleneck, which appears in the form of bandwidth limitation and fetching latency, has restricted the performance of the many-core system. At the meantime, graphics process unit (GPU), having recently turned into general-purpose programmable units, is the first to abandon expensive caches and combat latency by massive parallelism instead (Kirk and Hwu 2010), (NVIDIA 2009). Now, the GPU has widely used in many fields of scientific computing. Preis et al. has implemented GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model (Preis et al. 2009). Klöckner et al. has used CUDA to accelerate discontinuous Galerkin methods (Klöckner et al. 2009). Corrigan et al. realized CFD solvers on GPU using an unstructured grid (Corrigan et al. 2009).

We will present the work of implementation of the space-time Conservation Element and Solution Element (CESE) (Chang 1995) method on GPU and GPU cluster with CUDA. The space-time CESE method, originally proposed by Chang (1995), has many unique features comparing with conventional methods such as finite volume, finite difference and finite element. It offers a new approach to solve conservation laws like Navier-Stokes or Euler equations. It can obtain highly accurate numerical solutions for flow problems involving discontinuities (e.g. shocks and contact surfaces), vortices, acoustic waves, boundary layers and chemical reactions (Yu et al. 2009), (Cheng et al. 2010). To accurately calculate a complex flow field, a fine mesh with enormous grid points is employed normally, which demands a large amount of computing resource. The requirement of huge computing resource is not only needed in 2D or 3D problems, but also in some 1D problems for example the simulation of the asymptotic behavior in an infinite shock tube with homogeneous condensation (Cheng et al. 2010). It is expected that the GPU accelerated CESE method developed here could greatly reduce the calculating time and, therefore, meet the requirement of huge computing resource in these problems.

The rest of this paper is organized as follows: Sect. 10.2 is the overview of the CESE method. The implementation of the CESE method on GPU and GPU cluster will be dedicated in Sect. 10.3. Section 10.4 will present the simulation results and the computational performance. Finally conclusions are summarized in Sect. 10.5.

10.2 Overview of the Method

According to Yu and Chang (1997), the explicitly treatment of stiff source terms is easily achieved. First, the governing equation is a little different from the Euler equation. A source term vector is appended in it, which can be expressed as:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = \mathbf{S}.$$

U, F and **S** are:

$$\mathbf{U}=\begin{pmatrix}\rho\\ \rho u\\ \rho E\\ \rho g\\ \rho Q_2\\ \rho Q_1\\ \rho Q_0\end{pmatrix};\ \mathbf{F}=\begin{pmatrix}\rho u\\ \rho u^2+\rho\\ (\rho E+\rho)\,u\\ \rho g u\\ \rho Q_2 u\\ \rho Q_1 u\\ \rho Q_0 u\end{pmatrix};\ \mathbf{S}=\begin{pmatrix}0\\ 0\\ 0\\ 4\pi\rho_l(Jr_c^3+3\rho Q_2\overline{\frac{dr}{dt}})/3\\ Jr_c^2+2\rho Q_1\overline{\frac{dr}{dt}}\\ Jr_c+\rho Q_0\overline{\frac{dr}{dt}}\\ J\end{pmatrix},$$

where Q_0, Q_1 and Q_2 stand for finite numbers of moments of the size distribution function which describe the conservation of the liquid phase. g is the liquid mass fraction. is the density of liquid water. J is the nucleation rate. r_c is the critical radius and $\overline{\frac{dr}{dt}}$ is the averaged droplet growth/shrinkage rate. Here we used the liquid mass fraction g that is directly related to the third-order moment Q_3, $g=4\pi\rho_l Q_3/3$ (Luo et al. 2007).

With the explicit treatment of source terms, the governing equation is first divided into a homogeneous and an inhomogeneous part (Yu and Chang 1997) as:

$$\frac{\partial \mathbf{U}^{hom}}{\partial t}+\frac{\partial \mathbf{F}}{\partial x}=0,$$
$$\frac{\partial \mathbf{U}}{\partial t}=\mathbf{S}(\mathbf{U}^{hom}),$$

where $\mathbf{U}^{hom}$ is the solution of the homogeneous part. The homogeneous part is an Euler equation which can be calculated by the CESE Method. The governing equation solved by the CESE method can be written to the difference form as:

$$u_{mt}+f_{mx}=0.$$

Define that:

$$\mathbf{A}=\frac{\partial f_m}{\partial u_k},$$

the intermediate variable f_m and its time derivative f_{mt} at time n and point j are given as:

$$(f_m)_j^n=A_j^n(u_m)_j^n,$$
$$(f_m t)_j^n=A_j^n(f_m)_j^n.$$

The a-α scheme of the CESE method is:

$$(u_m)_j^n=\frac{1}{2}[(u_m)_{j-1/2}^{n-1/2}+(u_m)_{j+1/2}^{n-1/2}+(c_m)_{j-1/2}^{n-1/2}-(c_m)_{j+1/2}^{n-1/2}],$$

$$(u_{mx})_j^n=\frac{|vxr_j^n|^\alpha vxl_j^n+|vxl_j^n|^\alpha vxr_j^n}{|vxr_j^n|^\alpha+|vxl_j^n|^\alpha},$$

where

$$(c_m)_j^n = \frac{dx}{4}(u_{mx})_j^n + \frac{dt}{dx}(f_m)_j^n + \frac{(dt)^2}{4dx}(f_{mt})_j^n,$$

$$vxl_j^n = -\frac{(u_m)_{j-1/2}^{n-1/2} - (u_m)_j^n + (dt/2)(u_{mt})_{j-1/2}^{n-1/2}}{dx/2},$$

$$vxr_j^n = +\frac{(u_m)_{j+1/2}^{n-1/2} - (u_m)_j^n + (dt/2)(u_{mt})_{j+1/2}^{n-1/2}}{dx/2},$$

with dt and dx being the time and the space step, respectively. $(c_m)_j^n$ is an intermediate variable. vxl_j^n and vxr_j^n are the left and right gradients of $(u_m)_j^n$ respectively.

The explicit treatment of source terms is simple and its computation is local reliant. The physical model is given in Smolders (1992).

10.3 Implementation

As described above, the simple interdependency of the difference scheme makes the CESE method be easily parallel implemented. Consider a problem with a total number of grid points of NG. NG also represents the total threads in a Grid. Let NT denote the number of Threads in a Block with $NT = 2^T,\ 0 < T < 9$ (usually, $T = 7$), then the number of Blocks NB can be determined by $NB = NG/NT$.

10.3.1 The Overlapping Scheme

The number of threads in a Block in the overlapping scheme is one more than that the grid points needed. This method is based on a principle similar to cache. Cache reads more data nearby than that the program actual needs in memory. As shown in Fig. 10.1 and Algorithm 1, total number of threads is $NT + 1$ which means Shared Memory will read $NT + 1$ vectors of $(u_m)_j^{n-1/2}$ and $(u_{mx})_j^{n-1/2}$ from Device Memory. Then $NT + 1$ of $(f_m)_j^{n-1/2}$, $(f_{mx})_j^{n-1/2}$ and $(c_m)_j^{n-1/2}$ are calculated. With these $NT + 1$ of $(u_m)_j^{n-1/2}$ and $(c_m)_j^{n-1/2}$, 2^n of $(u_m)_j^n$ and $(u_{mx})_j^n$ can be calculated. This scheme avoids the communication between Blocks apparently. In fact, the communication has finished before calculation, where the overlapping corresponds between two adjacent Blocks. The source code of the kernel function is listed in Appendix A.

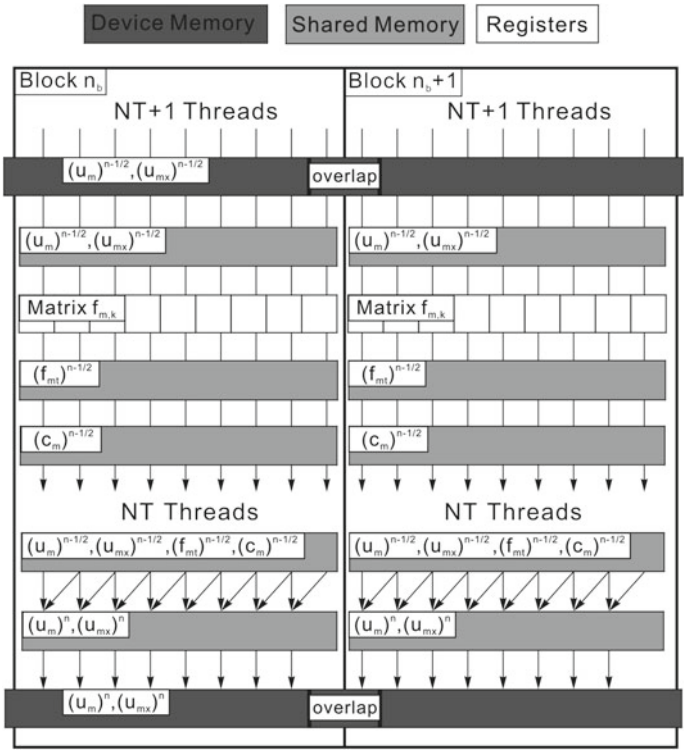


Fig. 10.1 Main computation procedures of the overlapping scheme for the CESE Method

10.3.2 Implementation on GPU Cluster

For a large scale problem, computational ability of single GPU is not enough. Thus GPU cluster is considered. Implementation of CESE method on GPU cluster is also achieved in our study by using the MPI. As depicted in Fig. 10.2, in the application of GPU cluster, MPI threads are used to start and control Devices. The calculation on each Device is the same as the single GPU application. What's difference is the communication between Devices, which must be processed to ensure the correctness of results.

In the application of GPU cluster, bandwidth between Devices is relatively low. Bandwidth in Device is above 100 GB/s (coalescence accessing), and 8 GB/s between Device and Host (PCIE-X16), while between Devices is 10 GB/s (RAM) or 1.25 GB/s (InfiniBand). It is obvious that the bottleneck is in the communication between Devices. To tackle this bottleneck problem, the frequency of communication between Devices must be limited. Two methods are employed to reduce the frequency through low bandwidth channel. First one is packing and unpacking data in Device. With this method, packing and unpacking time of data is less than doing it on Host because

the bandwidth on Device is much higher than it on Host. This method also reduces frequency of communication between Device and Host. The second method is setting a large size buffer to reduce the frequency of communication between Devices.

Require: Calculation scale NG, a Grid of $NB \times 1 \times 1$,
a Block of size $[NT + 1] \times 1 \times 1$;
Require: Input: $(u_m)^{n-1/2}$, the flow field at time $n - 1/2$; $(u_{mx})^{n-1/2}$,
the spatial gradient of $(u_m)^{n-1/2}$;
Ensure: Output: $(u_m)^n$, $(u_{mx})^n$, the flow field at time n;
Allocate $u_m, u_{mx}, f_m, f_{mt}, c_m$ in Shared Memory.
Allocate coefficients of matrix $\mathbf{A}$ in Register Memory.
Load $(u_m)^{n-1/2}$, $(u_{mx})^{n-1/2}$ from Device Memory into u_m, u_{mx}.
for $id \in [0, NT + 1)$ **parallel do**
$(f_m)_{id}^{n-1/2} \Leftarrow \mathbf{A} \times (u_m)_{id}^{n-1/2}$.
$(f_{mt})_{id}^{n-1/2} \Leftarrow \mathbf{A} \times (f_m)_{id}^{n-1/2}$.
$(c_m)_{id}^{n-1/2} \Leftarrow \frac{dx}{4} \times (u_{mx})_{id}^{n-1/2} + \frac{dt}{dx} \times (f_m)_{id}^{n-1/2} + \frac{(dt)^2}{4dx} \times (f_{mt})_{id}^{n-1/2}$.
end for
—— Barrier+Memory Fence ——
Allocate vxl, vxr in Register Memory.
for $id \in [0, NT)$ **parallel do**
if $id < NT$ **then**
All calculations use Shared Memory.
$(u_m)_{id}^{n} \Leftarrow \frac{1}{2} \times [(u_m)_{id}^{n-1/2} + (u_m)_{id+1}^{n-1/2} + (c_m)_{id}^{n-1/2} - (c_m)_{id+1}^{n-1/2}]$.
$vxl \Leftarrow [(u_m)_{id}^{n-1/2} - (u_m)_{id}^{n} + \frac{dt}{2} \times ((f_{mt})_{id}^{n-1/2})]/\frac{dx}{2}$.
$vxr \Leftarrow [(u_m)_{id+1}^{n-1/2} - (u_m)_{id}^{n} + \frac{dt}{2} \times ((f_{mt})_{id+1}^{n-1/2})]/\frac{dx}{2}$.
$(u_{mx})^{n} \Leftarrow (vxl^2 \times vxr + vxr^2 \times vxl)/(vxl^2 + vxr^2)$.
end if
end for
Load $(u_m)^n, (u_{mx})^n$ from Shared Memory into Device Memory.

Algorithm. 1. Algorithm of overlapping scheme for the CESE Method

As implied above, the overlapping scheme can gain higher computing performance. Here we use the multiple threads' overlapping scheme to set the buffer. With

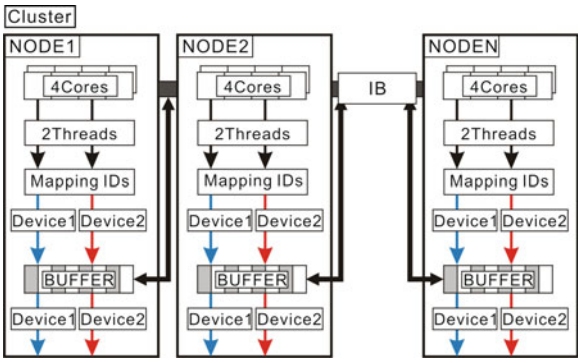


Fig. 10.2 MPI based CUDA, one MPI thread controls one CUDA Device

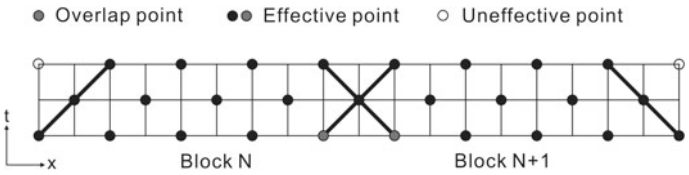


Fig. 10.3 Multiple threads' overlap: effective point decrease as time step accumulating

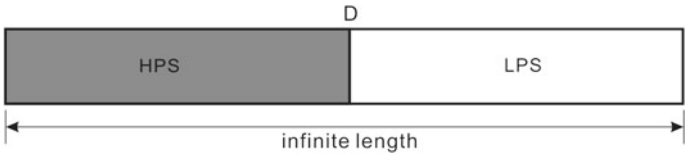


Fig. 10.4 The shock tube with both ends opened. The tube is infinite long and is divided into the HPS and the LPS by the diaphragm D

this method, if we set 1 times more size of buffer, the frequency of communication between Devices will decrease to a half. However, multiple threads' overlap will reduce the effective points, which is shown in Fig. 10.3. Thus, the balance between data loss and communication frequency reducing should be well considered. The best situation is that the buffer size is 1/8 of total data size in our test.

10.4 Example and Results

Condensation problem in an infinite length shock tube is simulated in this section. Initially a shock tube is equally divided into a high pressure section (HPS) and a low pressure section (LPS) by a diaphragm D, as shown in Fig. 10.4.

The initial pressure is 1.0 bar in the HPS and 0.3 bar in the LPS, respectively. The initial temperatures of both HPS and LPS are 295 K. The gas in the tube is humid nitrogen with an initial saturation ratio of 0.8. An initial mesh with $dx = 0.05\,\text{mm}$ and a time step $dt = 0.2\,\mu s$ are set.

The problem of condensation in an infinite length shock tube has an asymptotic solution after an infinitely long time (Cheng et al. 2010). Now, GPU is applied to prove the numerical solution implied in Smolders (1992) is mesh-independence. The overlapping scheme is chosen for the following computation because it is fast. The calculation method is to double the space step and time step when the shock transmits to the end of tube and then the length of shock tube is doubled (Cheng et al. 2010). With the method of doubling dx and dt, 5 different scales of grid ranging from 2^{12} to 2^{16} are employed in calculation.

The result in global view is presented in Fig. 10.5a. As depicted in Fig. 10.5b, all the results are nearly the same at 20 years and approach to the theoretical solution. Although the space step is expanded ranging from 10^8 to 10^7 meters, the changes of

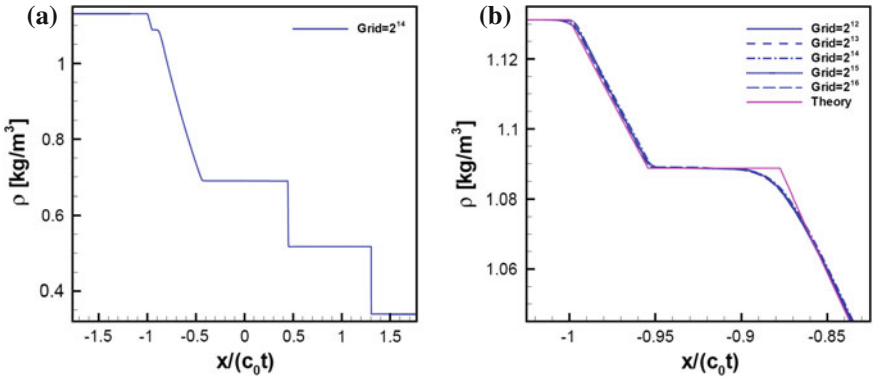


Fig. 10.5 Condensation problem in an infinite shock tube at t = 20 years (single card). **a** Density evolution, $Grid(NG) = 2^{14}$. **b** Asymptotic solution with different grids at the time 20 years

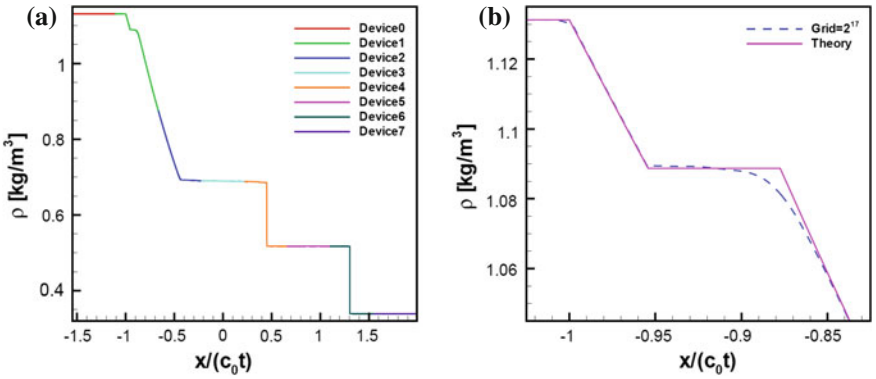


Fig. 10.6 Condensation problem in an infinite shock tube at t = 20 years, $Grid(NG) = 2^{17}$ (multi-card). **a** Density evolution. **b** Asymptotic solution with different grids at the time 20 years

space step's length have little effect on the final results at the same evolution time. So we put forward that the theoretical solution is mesh-independence.

The acceleration ratio of single card computing is 71. The GPU used is the NVIDIA graphics card 9800GT and the CPU is Intel CPU Core Dual E7300. The peak value of GPU is about 75 GFLOPS. Both the CPU and GPU's key parameter are listed in Appendix B.

For a large scale problem, computational ability of single GPU is not enough. Thus GPU cluster is considered. Implementation of CESE method on GPU cluster is also achieved in our study.

Here we present the results of condensation problem in an infinite shock tube which is computed on GPU cluster. Figure 10.6a shows the global view of the density evolution at t = 20 years. Figure 10.6b illustrates the platform in the asymptotic solution.

For this case, the peak value of GPU cluster is 0.88 TFLOPS (using 8 C1060 GPU cards), and the time consumed on communication between Devices is lower than 0.1 %.

10.5 Conclusion

We presented a GPU accelerated version of the CESE method and the explicit treatment of source terms. The new approach of GPU acceleration with CUDA was applied and we have gained good computational performance increase with an old fashion graphics card 9800GT which only supports singe precision float number. To optimize the code performance, Shared Memory is employed—the overlapping scheme computes with Shared Memory. The principle of Overlapping scheme can be used in other similar algorithm. The scheme also achieved good acceleration ratio in the problem of condensation in shock tube which is 71. We have also implemented the CESE method on GPU cluster and gained peak value of 0.88 TFLOPS (using 8 C1060 GPU cards). With the simulation results of single GPU and GPU cluster, we proved that the asymptotic solution of condensation problem in an infinite length shock tube is mesh-independence. In the future work, we will use the GPU and CUDA to solve 2 and 3D problems with the CESE Method.

Acknowledgments This research was carried out with the support of the National Natural Science Foundation of China under grant 10972214.

Appendix A

Here we list out the code of kernel function:

```
__global__ void kernel_q(float * q11,float *qx11,int j1, float qdt,float qdx,float dtx, float hdt,float hdx,float
cp,float cv,float l_0,float l_1,float rv)
{
    const unsigned int bx = threadIdx.x;
    const unsigned int dx = (Thread_Num-1)*blockIdx.x+bx;

    float
        w2,w3,w4,f21x,f22x,f24x,f31x,f32x,f33x,f34x,a1,a2,a3,a4,cpx,cvx,ga;
    float
        vxl,vxr;
    int i;

    __shared__ float q1[nvarb][Thread_Num];
    __shared__ float qx1[nvarb][Thread_Num];
    __shared__ float qt1[nvarb][Thread_Num];
    __shared__ float s1[nvarb][Thread_Num];
    __shared__ float qns[Thread_Num];

    for(i=0;i<nvarb;i++)
    {
        q1[i][bx]=q11[i*nxd+dx];
        qx1[i][bx]=qx11[i*nxd+dx];
    }
    __syncthreads();

    w2=q1[1][bx]/q1[0][bx];
    w3=q1[2][bx]/q1[0][bx];
    w4=q1[3][bx]/q1[0][bx];

    cpx=cp-w4*l_1;

    cvx=cv+w4*(rv-l_1);
    ga=cpx/cvx;

    a1 = ga - 1.0;
    a2 = 3.0 - ga;
    a3 = a2/2.0;
    a4 = 1.5*a1;
    f21x= -a3*w2*w2;
    f22x=a2*w2;
    f24x=a1*l_0;
    f31x=a1*w2*w2*w2-ga*w2*w3-a1*l_0*w2*w4;
    f32x=ga*w3-a4*w2*w2+a1*l_0*w4;
    f33x=ga*w2;
    f34x=a1*l_0*w2;

    qt1[0][bx]= -qx1[1][bx];
    qt1[1][bx]= -(f21x*qx1[0][bx]+f22x*qx1[1][bx]+a1*qx1[2][bx]+f24x*qx1[3][bx]);
    qt1[2][bx] =-(f31x*qx1[0][bx]+f32x*qx1[1][bx]+f33x*qx1[2][bx]+f34x*qx1[3][bx]);

     for (i=3;i<nvarb;i++)
     {
         qt1[i][bx]=-(-q1[i][bx]*w2/q1[0][bx]*qx1[0][bx]+q1[i][bx]/q1[0][bx]*qx1[1][bx]+
         w2*qx1[i][bx]);
     }
```

```
    s1[0][bx] = qdx*qx1[0][bx] + dtx*(q1[1][bx] + qdt*qt1[1][bx]);
    s1[1][bx] = qdx*qx1[1][bx] + dtx*(f21x*(q1[0][bx] + qdt*qt1[0][bx])+f22x*(q1[1][bx] + qdt*qt1[1][bx])
                + a1*(q1[2][bx] + qdt*qt1[2][bx])+f24x*(q1[3][bx] + qdt*qt1[3][bx]));
    s1[2][bx] = qdx*qx1[2][bx] + dtx*(f31x*(q1[0][bx] + qdt*qt1[0][bx])+f32x*(q1[1][bx] + qdt*qt1[1][bx])
                + f33x*(q1[2][bx] + qdt*qt1[2][bx])+f34x*(q1[3][bx]+ qdt*qt1[3][bx]));

    for (i=3;i<nvarb;i++)
    {
        s1[i][bx] = qdx*qx1[i][bx] + dtx*(-q1[i][bx]*w2/q1[0][bx]*(q1[0][bx]+qdt*qt1[0][bx])+
                    q1[i][bx]/q1[0][bx]*(q1[1][bx]+qdt*qt1[1][bx])+w2*(q1[i][bx] + qdt*qt1[i][bx]));
    }
    __syncthreads();

    if(bx<Thread_Num-1)
    {
        for (i=0;i<nvarb-1;i++)
        {
            qns[bx] = 0.5*(q1[i][bx] + q1[i][bx+1] + s1[i][bx] - s1[i][bx+1]);
            vxl= (qns[bx] - q1[i][bx] - hdt*qt1[i][bx])/hdx;
            vxr = (q1[i][bx+1] + hdt*qt1[i][bx+1] - qns[bx])/hdx;
            qx11[i*nxd+dx+j1] = (vxl*vxr*vxr + vxr*vxl*vxl)/(vxl*vxl+ vxr*vxr + 1.0e-26);
            q11[i*nxd+dx+j1]=qns[bx];
        }

        qns[bx] = 0.5*(q1[6][bx] + q1[6][bx+1] + s1[6][bx] - s1[6][bx+1]);
        vxl= (qns[bx] - q1[6][bx] - hdt*qt1[6][bx])/hdx;
        vxr = (q1[6][bx+1] + hdt*qt1[6][bx+1] - qns[bx])/hdx;
        qx11[6*nxd+dx+j1] = (vxl*sqrtf(fabs(vxr)) + vxr*sqrtf(fabs(vxl)))/(sqrtf(fabs(vxl))+
                            sqrtf(fabs(vxr))+ 1.0e-26);
        q11[6*nxd+dx+j1]=qns[bx];
     }
}

extern "C"
void launch_cesekernel (dim3 BLOCKN, dim3 THREADN,float * q11,float *qx11,int j1, float qdt,float
qdx,float dtx, float hdt,float hdx,float cp,float cv,float l_0,float l_1,float rv)
{
            kernel_q<<<BLOCKN,THREADN>>>(q11,qx11, j1, qdt, qdx, dtx,  hdt, hdx, cp, cv, l_0, l_1, rv);
}
```

Appendix B

Table B.1 Key facts of Intel CPU Core Dual E7300:

	Intel CPU Core Dual E7300
Number of cores	2
L2 Cache	2 MB
Clock rate	2.66 GHz

Table B.2 Key facts of NVIDIA graphics card 9800GT:

	NVIDIA graphics card 9800GT
Number of streaming multiprocessors	14
Number of streaming processors	112
Global memory	1 GB
Shared memory per multiprocessor	16 KB
Register memory per multiprocessor	8192*4 B
Clock rate	1.50 GHz
Compute capability	1.1

Table B.3 Key facts of NVIDIA computing card Tesla C1060:

	NVIDIA computing card Tesla C1060
Number of streaming multiprocessors	30
Number of streaming processors	240
Size of global memory	4 GB
Shared memory per multiprocessor	16 KB
Register memory per multiprocessor	16384*4 B
Clock rate	1.296 GHz
Compute capability	1.3

References

Chang SC (1995) The method of space-time conservation element and solution element-A new approach for solving the Navier-Stokes and Euler equations. J Comput Phys 119:295–324

Cheng W, Luo X, Yang J, Wang G (2010) Numerical analysis of homogeneous condensation in rarefaction wave in a shock tube by the space-time CESE method. Comput Fluids 39:294–300

Cheng W, Luo X, van Dongen MEH (2010) On condensation-induced waves. J Fluid Mech 651:145–164

Corrigan A, Camelli F, Löner R, Wallin J (2009) Running unstructured grid based CFD solvers on modern graphics hardware, in: 19th AIAA computational fluid dynamics. American Institute of Aeronautics and Astronautics, Inc., San Antonio, Texas, USA

Geer D (2005) Chip makers turn to multicore processors. Computer 38(5):11–13

Kirk D, Hwu W (2010) Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers, Morgan Kaufmann

Kish L (2002) End of Moore's law: thermal (noise) death of integration in micro and nano electronics. Physics Letters A 305(3–4):144–149

Klöckner A, Warburton T, Paul JB, Hesthaven J (2009) Nodal discontinuous Galerkin methods on graphics processors. J Comput Phys 228:7863–7882

Luo X, Wang M, Yang J, Wang G (2007) The space-time CESE method applied to phase transition of water vapor in compressible flows. Comput and Fluids 36:1247–1258

NVIDIA Corporation, NVIDIA CUDA Programming Guide, Version 2.3.1, NVIDIA Corporation (2009)

Preis T, Virnau P, Paul W, Schneider JJ (2009) GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. J Comput Phys 228:4468–4477

Smolders HJ (1992) Non-linear wave phenomena in a gas-vapour mixture with phase transition. Ph.D. thesis, Eindhoven University of Technology, Eindhoven

Yu ST, Chang SC (1997)Treatments of stiff source terms in conservation laws by the method of Space-Time Conservation Element/Solution Element, AIAA paper, 0435 1997
Yu S-TJ, Yang L, Lowe RL, Bechtel SE (2009) Numerical simulation of linear and nonlinear waves in hypoelastic solids by the CESE method. Wave Motion 47:168–182

Chapter 11
Two-Way Coupled Sprays and Liquid Surface: A GPU-Based Multi-Scale Fluid Animation Method

Guijuan Zhang, Gaojin Wen and Shengzhong Feng

Abstract GPU-based fluid animation is a hot topic in many applications such as films, cartoons and games. As the flow phenomena contain highly complex behaviors and rich visual details, it is necessary to explore the intrinsic multi-scale property in fluid animation. In this paper, we present a multi-scale fluid animation method on GPU. Our method is designed to animate fluid details of grid and sub-grid scale with high efficiency. In our method, the motion of liquid surface is obtained by solving Navier-Stokes equations and Level Set equation while the dynamics of fluid sprays are dominated by SPH solution. The interaction between liquid surface and sprays is modeled by a two-way coupling algorithm which can be executed efficiently on GPU. From the results of the experiments, we can reach the conclusion that the proposed GPU based acceleration method can improve the processing speed of the multi-scale fluid animation significantly while getting interesting details.

Keywords GPU · Fluid animation · Two-way coupling · Multi-scale

11.1 Introduction

Fluid animation is one of the most desirable techniques in computer graphics since it is widely used in films and real-time computer games. These applications often require that the fluid animation results have high degree of realism and can be generated efficiently. In fact, it is rather difficult to achieve such a goal only by a single fluid model. Therefore, exploring the multi-scale property of fluid motion and modeling it on GPU architecture is of great importance.

G. Zhang (✉) · G. Wen · S. Feng
Center of High Performance Computing, Institute of Advanced Computing and Digital Engineering, Shenzhen Institutes of Advanced Technology, Chinese Academy of Science, 518055 Shenzhen, China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_11,

In fluid animation world, researchers often use physically-based methods to capture full 3D fluid details (Enright et al. 2002; Losasso et al. 2008; Wojtan et al. 2010; Brochu et al. 2010). Provided with nice results, these methods suffer from the disadvantage of huge memory usage and long computation time. Although GPU-based methods (Li et al. 2003; Liu et al. 2004; Zhao 2008) can improve the efficiency significantly, they scarcely model the multi-scale features in fluid motion.

In this paper, we present a GPU-based multi-scale fluid animation method. Our method can animate liquid details of grid and sub-grid scale with high efficiency. We use 3D Navier-Stokes equations to compute fluid velocity field, and adopt the Level set equation to model free surface motion. The dynamics of sprays are dominated by SPH solution. We also provide an efficient two way coupling method on GPU to describe the interaction between liquid surface and sprays. Experiments show that our method can improve the efficiency while exhibiting visual pleasing animation results.

11.2 Related Work

In the literature of fluid animation, researchers often use grid-based methods to capture the details of liquid surface (Enright et al. 2002). Based on the basic framework presented in Enright et al. (2002), researchers modeled solid-fluid coupling (Carlson et al. 2004), multiphase flow (Losasso et al. 2006), surface tension (Wang et al. 2005) and so on. However, the numerical dissipation in these methods damages fluid details obviously. To combat numerical dissipation and maintain as much details as possible, explicit method that uses mesh-models to represent liquid surface is developed (Wojtan et al. 2010; Brochu et al. 2010). Because both the implicit and explicit methods are grid-based, the final results crucially depend on grid resolution.

Another group method to animate fluid is using particles. In general, particle methods can be classified into two categories: simple particle system and particle-interaction system. Reeves (1983) presented a simple particle system for animating smoke. To model the interaction between particles, Moving Particle Semi-implicit method (MPS) as well as Smoothed Particle Hydrodynamics (SPH) are introduced into fluid animation community. In MPS method, Premoze et al. ensured incompressibility by solving the Poisson equation on particles (Premoze et al. 2003). Müller et al. applied SPH model to animate fluid surfaces interactively (Müller et al. 2003). To reduce the number of particles during animation process, Adams et al. (2007) presented an adaptive sampling algorithm for SPH model. In Losasso et al. (2008), Losasso et al. coupled SPH and PLS to obtain fluid details of both liquid surface and water sprays.

With the support of the latest graphics card, Crane et al. (2007) animated various fluid phenomena including smoke, water and fire in real-time. Geiss (2007) presented a GPU-based Marching Cube algorithm that could be used to extract liquid surface in fluid simulation. In addition, GPU-based SPH fluid simulation is also studied (Amada et al. 2004). To exploit the massive computational power of GPUs, Harada et al.

presented a method that can search for neighboring particles on GPUs efficiently (HARADA et al. 2007). The demo "Cascades" (Lin 2007) used a particle system to simulate a real-time waterfall entirely on GPU. Zhang et al. (2008) also developed an adaptive SPH method on GPU to speed up computation.

11.3 Overview

The goal of our method is to produce multi-scale fluid details efficiently given the initial and boundary conditions. To achieve this goal, we present a framework that consists of three parts. As illustrated in Fig. 11.1, the first part captures fluid surface motion using PLS method while the second part animates liquid sprays according to SPH model. As for the third part, we present a two way coupling algorithm on GPU to compute fluid velocity field for both liquid surface and water sprays. We will give the details of each part in the following sections.

11.4 Physically-Based Surface and Sprays Simulation

11.4.1 Free Surface Motion

In water animation, the most visually interesting part is the liquid surface. Since the surface undergoes frequent topological changes, we use particle level set method (PLS) (Enright et al. 2002) in this paper. The method represents the motion of liquid surface according to level set equation and corrects the numerical errors by particles.

In PLS method, the water surface is defined as the zero iso-surface of a signed distance function ϕ implicitly. It evolves under the level set equation

$$\phi_t + \boldsymbol{u} \cdot \nabla \phi = 0, \tag{11.1}$$

where $\boldsymbol{u}$ is the fluid velocity field. Eq. (11.1) can be solved by semi-Lagrange method [ELF05].

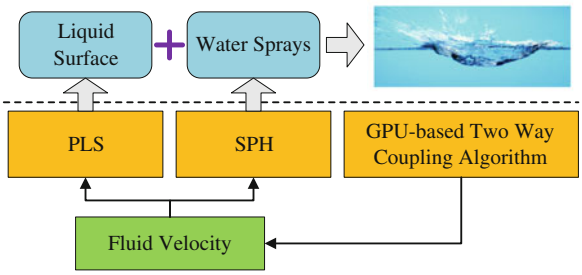


Fig. 11.1 The framework of our method

Besides the evolution of ϕ, two sets particles that located near the water surface are used to detect and correct numerical errors. These particles are passively advected by

$$\frac{d\boldsymbol{x}_p}{dt} = \boldsymbol{u}_p,$$

where $\boldsymbol{x}_p$ is the particle position and $\boldsymbol{u}_p$ is the fluid velocity interpolated at $\boldsymbol{x}_p$. When particles escape to the other side of interface, errors are detected. The local level set function of the escaped particle is then defined as

$$\phi_p(\boldsymbol{x}) = s_p\left(r_p - \left|\boldsymbol{x} - \boldsymbol{x}_p\right|\right),$$

where s_p denotes the particle's sign and r_p is the particle's radius.

11.4.2 Particle-Based Sprays Animation

We use SPH model to simulate water sprays. In SPH model, field quantities can be computed according to the values carrying by particles. For example, the quantity X at location $\boldsymbol{x}$ can be interpolated by all its neighboring particles

$$X(\boldsymbol{x}) = \sum_p m_p \frac{X_p}{\rho_p} W(x, h),$$

where p iterates over all particles, m_p represents mass, ρ_p is the density and X_p is the value at particle position $\boldsymbol{x}_p$. The function $W(r, h)$ is a smoothed, radically-symmetric kernel function with core radius h. It is often defined as

$$W(\boldsymbol{x}, h) = \begin{cases} c\left(1 - \left\|\boldsymbol{x} - \boldsymbol{x}_p\right\|^2 / r_p^2\right) & when \ \left\|\boldsymbol{x} - \boldsymbol{x}_p\right\|^2 \le r_p^2 \\ 0 & otherwise \end{cases},$$

where c is the normalization constant, r_p is radius. Figure 11.2 gives the basic framework of sprays animation algorithm in this paper. Note that the Possion equation in step 4 is built upon two way coupling method that will be detailed in the following sections.

11.5 Two-Way Coupling Method on GPU

In physically-based liquid animation, the most time-consuming step is to obtain fluid velocity field via solving inviscid form of the Navier-Stokes equations

$$\nabla \cdot \boldsymbol{u} = 0 \tag{11.2}$$

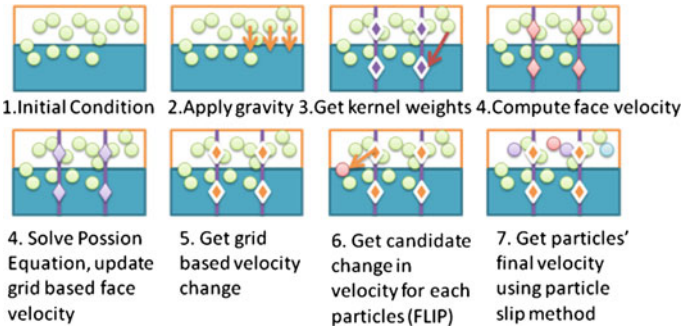


Fig. 11.2 Sprays animation using SPH model

$$\boldsymbol{u}_t = -\boldsymbol{u}\cdot(\nabla \boldsymbol{u}) - \frac{1}{\rho}\nabla p + \boldsymbol{f}, \tag{11.3}$$

where $\boldsymbol{u}$ is the fluid velocity field, $\boldsymbol{\rho}$ is the density, p denotes pressure and $\boldsymbol{f}$ is the external force field. Since both the models of liquid surface and water sprays allow for the enforcement of incompressibility and targeting particle number density, we couple the two pressure solver into a single Poisson equation similar to (Losasso et al. 2008). As a result, the mass conservation equation is modified as

$$\frac{1}{n_r}\frac{Dn_r}{Dt} + \nabla\cdot\boldsymbol{u} = 0, \tag{11.4}$$

where $D/Dt = \partial/\partial t + \boldsymbol{u}\cdot\nabla a$, n_r denotes the number of particles per region.

According to the two-way coupling method, we compute a unified velocity field for both liquid surface and spray model. We also implement this part on GPU to improve efficiency.

11.5.1 Two-Way Coupled Velocity Solution

Suppose the velocity $\boldsymbol{u}^n$ has already been resolved at time t and we will compute the fluid velocity $\boldsymbol{u}^{n+1}$ at a later time $t + \Delta t$. The process of solving fluid velocity field can be divided into three steps: adding force, advection and projection.

For the first step, we add external force (e.g., gravity) to the velocity field: $\boldsymbol{u}_1 = \boldsymbol{u}^n + \Delta t\boldsymbol{f}$. The next step solves for the effect of the advection term $-\boldsymbol{u}\cdot(\nabla \boldsymbol{u})$. We adopt Stam's Semi-Lagrange advection method here. It requires two steps to compute the intermediate velocity field $\boldsymbol{u}^*(x)$. First, tracking each fluid cell in the grid back in time to get its previous position; second, copying the corresponding velocity value at previous location to $\boldsymbol{u}^*(x)$.

After obtaining $\boldsymbol{u}^*(x)$, we will compute the final velocity field $\boldsymbol{u}^{n+1}$ at time $t+\Delta t$ by a two-way coupling method. According to Eq. (11.3), we get

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^* - \frac{\Delta t}{\rho}\nabla p. \tag{11.5}$$

Taking divergence of both sides of Eq. (11.5) and substituting $\nabla\cdot\boldsymbol{u}^{n+1}$ from Eq. (11.4) yields

$$\nabla^2 p = \frac{\rho}{\Delta t}\nabla\cdot\boldsymbol{u}^* + \frac{1}{n_r}\frac{Dn_r}{Dt}, \tag{11.6}$$

where $\frac{1}{n_r}\frac{Dn_r}{Dt}$ can be approximated by $\frac{1}{n_r^{k+1}}\left(\frac{n_r^{k+1}-n_r^k}{\Delta t}\right)$. Equation (11.6) is a Possion equation for pressure p and can be converted into a linear system finally. After solving p, $\boldsymbol{u}^{n+1}$ can be computed by Eq. (11.5).

11.5.2 Implementation and Performance Considerations

We implement the three steps on GPU. The data storage and boundary conditions are also considered. Corresponding kernel functions are given in the appendix.

11.5.2.1 Data Structure

When solving for $\boldsymbol{u}^{n+1}$, nine 3D arrays are required: velocity field $\boldsymbol{u}^n$,$\boldsymbol{u}^{n+1}$, pressure$p$, particle number n_r and external force field $\boldsymbol{f}$. To save as much memory as possible, we reuse one velocity array as temporary storage for p and let $\boldsymbol{f}$ be the constant gravity field. Therefore, only seven 3D arrays are required in global memory. To achieve high global memory access efficiency, we consider the memory coalescing by converting a 3D array into a 1D array. A cell (x, y, z) has an index $x + NX\ (y + zNY)$ where NX and NY are the size of 3D arrays in x and y dimension respectively. Since a grid in CUDA is organized as a two dimensional array of blocks, each block process all slices of 3D arrays in z direction.

11.5.2.2 Boundary Conditions

Boundary conditions are very important for fluid animation because they provide a reasonable means to handle the interaction among liquid, solid and the air. Let $\boldsymbol{u} = \boldsymbol{u}_N + \boldsymbol{u}_T$, where $\boldsymbol{u}_N$ is the velocity along the normal direction of solid wall and u_T is the tangential velocity. We set $\boldsymbol{u}_N = \boldsymbol{u}_{solid}\cdot\widehat{\boldsymbol{n}}$ in order to prevent the liquid flowing into solid. The tangential velocity uses free slip boundary condition so that $\boldsymbol{u}_T$maintains unchanged. As for pressure term, we set Newmann boundary condition for solid cells and $p = 0$ for all air cells. Figure 11.3 shows an example.

To implement this section on CUDA, we also need a 3D array *tag* to record the cell type. We use a single type for internal solid cells, liquid cells, and air cells respectively. In addition, cells at different side of solid surface such as left, right,

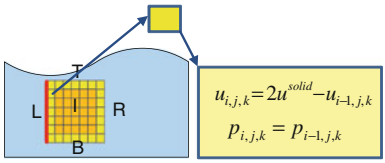


Fig. 11.3 Boundary conditions

bottom, top, back, and front also have different type values. Similar to the other 3D arrays, *tag* is also converted to 1D array for memory coalescing. We give a CUDA example of setting boundary condition for pressure term in the appendix. It can be extended to velocity term easily.

11.5.2.3 Velocity Update

As introduced in Sect. 11.5.1, three steps are required to get the final velocity field: adding force, advection and projection. Implementing the first two steps on CUDA is straightforward that we only need to consider how to locate liquid cells according to block ID and thread ID as shown in the appendix. Here, we detailed the third step only.

The projection step involves solving the Possion Eq. (11.6) at first and computing the final fluid velocity field according to Eq. (11.5). We use CUDA-based Jacobi's iterative method to solve the Possion equation. As for Eq. (11.6), we get

$$\begin{aligned} &-p_{i-1,j,k}-p_{i,j-1,k}-p_{i,j,k-1}+6p_{i,j,k}-p_{i+1,j,k}-p_{i,j+1,k}-p_{i,j,k+1} \\ &=-\Delta x\frac{\rho}{\Delta t}\left(u_{i+1,j,k}-u_{i,j,k}+v_{i,j+1,k}-v_{i,j,k}+w_{i,j,k+1}-w_{i,j,k}\right) \\ &+\frac{1}{n_r^{k+1}}\left(\frac{n_r^{k+1}-n_r^k}{\Delta t}\right). \end{aligned} \tag{11.7}$$

The right side of Eq. (11.7) is a known value so it finally forms a linear system $A\boldsymbol{p}=\boldsymbol{b}$. The coefficient matrix A is a symmetric, positive definite and sparse matrix. There are no more than six non-zero elements in each row of A. Given the initial solution $\boldsymbol{p}^0=(0,\cdots,0)$, the Jacobi method obtains the solution after $n+1$ iterations as

$$p_i^{n+1}=\frac{1}{a_{ii}}\left(b_i-\sum_{j\neq i}a_{ij}p_j^n\right),$$

where p_i^{n+1} is the ith component of the solution $\boldsymbol{p}$, b_i is the ith component of $\boldsymbol{b}$, and a_{ij} is an element of matrix A.

We use one block to process all slices of $\boldsymbol{p}$ in z direction. Since the Jacobi method will take several iterations, we defined a shared memory variable to store the temporal values of $\boldsymbol{p}$. The iteration will stop until

$$\max_{1 \le i \le n_p} \left| p_i^k - p_i^{k-1} \right| \le \varepsilon,$$

where ε is a small positive number which equals to $1e-6$ in this paper. After solving $\boldsymbol{p}$, we compute the final fluid velocity according to Eq. (11.5).

11.6 Results

We have applied our fluid animation method to produce interesting fluid behaviors. Our experiment is separated into two parts: testing the performance of fluid velocity solution and exhibiting the fluid animation results. All the results are gathered on a PC with 2.6 GHz Intel Core 2 Duo CPU, 2 GB memory, and GeForce 8800 GT graphics card. We use CUDA 2.0 for the implementation. The results show that our approach can produce desirable fluid behaviors while improving the efficiency significantly.

11.6.1 Performance Analysis

In this experiment, the grid size is $128 \times 51 \times 128$. The number of fluid cells increases while adding water source during this process. We sample 300 frames and record the computational time of both CPU and GPU based version as shown in Fig. 11.4. From the Fig. 11.4a, b, we can see that the CUDA-based version improves the efficiency significantly. Figure 11.4c shows the speedup gained from the example. The speedup of CUDA based version increases when the number of fluid cells grows. The maximal speedup in the sampled frames is 33.0.

In general, the efficiency of both CPU and GPU based version may fluctuate due to many factors. For example, when we use different scene-models and initial conditions for fluid animation, the results are far from each other. In our experiments, we find that different efficiency stems from two main reasons. First, the efficiency varies according to the number of liquid cells during animation. Second, the Possion equation is often ill-conditioned and requires more iteration steps in some frames.

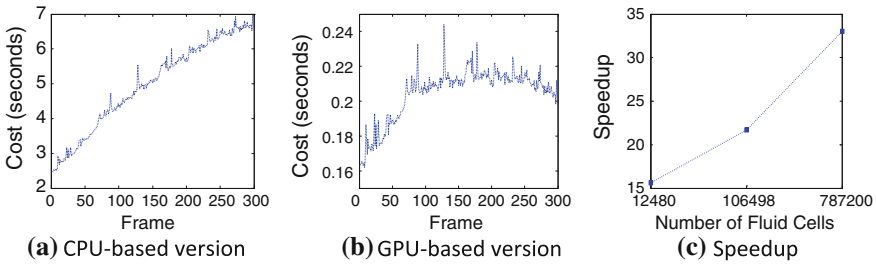


Fig. 11.4 Performance of the velocity solution. **a** CPU-based version. **b** GPU-based version. **c** Speedup

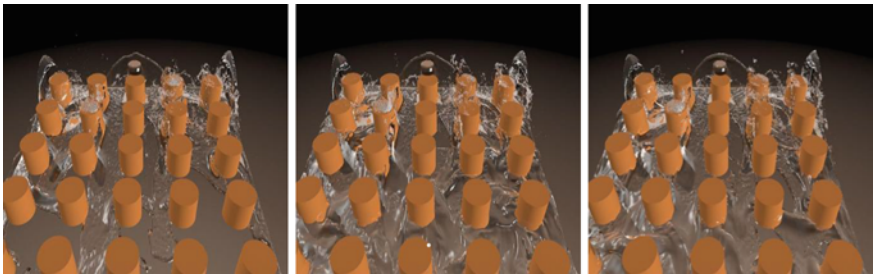


Fig. 11.5 A sequence of fluid animation result

11.6.2 Animation Results

Figure 11.5 shows a sequence of fluid animation results. In this example, the water is poured into a tank whose size is $1 \times 1 \times 0.4$. In order to exhibit as much fluid details as possible, we put 25 cylinders in the tank. The height of each cylinder is 0.1 while the radius equals to 0.05. The water source is positioned at (0.1, 0.175, 0.5) and the source velocity is set to $(-2.4, -2.0, 0)$. From Fig. 11.5, we can see the liquid surface and water sprays apparently near the cylinders.

11.7 Discussion and Conclusions

We have developed a GPU-based two way coupling method for fluid animation in this paper. The method solves for fluid motion with details of grid and sub-grid scale efficiently. We use physically-based liquid animation model and solves for fluid velocity field according to two-way coupling method. Since it is the most time-consuming process, we also implement the two way coupling algorithm on GPU. Experiments show that the proposed GPU based acceleration method can improve the efficiency of multi-scale fluid animation significantly.

This paper only implements the most time-consuming step of fluid animation on GPU. In fact, physically-based fluid animation method is a very complex model that includes many equations and algorithms such as the Level Set equation, SPH model, Fast Marching algorithm and Marching Cubes et al. So as for future work, we would like to extend the above algorithms on GPU to further improve the efficiency. In addition, optimizing the method presented in this paper according to the hardware properties of GPU card is another meaningful work.

Acknowledgments This work is supported by the National High-Tech Research and Development Plan of China under Grant Nos. 2006AA01A114, 2007AA120502, and Shenzhen Innovation Technology Program under Grant No. SY200806300211A.

Appendix

```
   //add gravity
   __global__ void add_force_kernel (float *x, float dt)
   {
 int i = threadId.x + blockId.x * blockDim.x;
    int j = threadId.y + blockId.y * blockDim.y;
     for(int k=0; k<NZ; k++)
      x[IX(i,j,k)] += dt*gravity; }
//set boundary condition for pressure
     __global__ void set_pressure_bnd_kernel(int i, int j, int k, int *tag, float *x)
     {
     if (SOLID_INTERNAL == tag[IX(i,j,k)])      x[IX(i,j,k)]=0;
     else if (SOLID_LEFT == tag[IX(i,j,k)])                  x[IX(i,j,k)] = x[IX(i+1,j,k)];
     else if (SOLID_RIGHT == tag[IX(i,j,k)])    x[IX(i,j,k)] = x[IX(i-1,j,k)];
     else if (SOLID_BOTTOM == tag[IX(i,j,k)])  x[IX(i,j,k)] = x[IX(i,j+1,k)];
     else if (SOLID_TOP == tag[IX(i,j,k)])                   x[IX(i,j,k)] = x[IX(i,j-1,k)];
     else if (SOLID_BACK == tag[IX(i,j,k)])     x[IX(i,j,k)] = x[IX(i,j,k+1)];
     else if(SOLID_FRONT == tag[IX(i,j,k)])     x[IX(i,j,k)] = x[IX(i,j,k-1)];
     else if((tag[IX(i,j,k)]& SOLID_BOTTOM) && (tag[IX(i,j,k)]&SOLID_BACK))
                x[IX(i,j,k)] = 0.5f*( x[IX(i, j , k+1)] + x[IX(i, j+1, k)]);
     else if((tag[IX(i,j,k)]&SOLID_BOTTOM) && (tag[IX(i,j,k)]&SOLID_FRONT))
                        x[IX(i,j,k)] = 0.5f*( x[IX(i, j , k-1)]+x[IX(i, j+1 , k)] );
     else if((tag[IX(i,j,k)]&SOLID_TOP) && (tag[IX(i,j,k)]&SOLID_BACK))
                        x[IX(i,j,k)] = 0.5f*( x[IX(i, j, k+1  )]+x[IX(i, j-1, k )]);
     else if((tag[IX(i,j,k)]&SOLID_TOP)&&(tag[IX(i,j,k)]&SOLID_FRONT))
                        x[IX(i,j,k)] = 0.5f*( x[IX(i, j, k-1)]+x[IX(i, j-1, k)]);
     else if((tag[IX(i,j,k)]& SOLID_LEFT) && (tag[IX(i,j,k)]& SOLID_BACK))
                        x[IX(i,j,k)] =  0.5f*(x[IX(i, j, k+1 )]+x[IX(i+1, j, k )]);
     else if((tag[IX(i,j,k)]& SOLID_LEFT )&& (tag[IX(i,j,k)]& SOLID_FRONT))
                        x[IX(i,j,k)] = 0.5f*(x[IX(i, j, k-1)]+x[IX(i+1  , j, k)]);
     else if((tag[IX(i,j,k)]& SOLID_RIGHT )&& (tag[IX(i,j,k)]&SOLID_BACK))
                        x[IX(i,j,k)] = 0.5f*(x[IX(i, j, k+1  )]+x[IX(i-1, j, k    )]);
     else if ((tag[IX(i,j,k)]& SOLID_RIGHT) &&(tag[IX(i,j,k)]&SOLID_FRONT))
                        x[IX(i,j,k)] = 0.5f*(x[IX(i, j, k-1)]+x[IX(i-1, j, k)]);
     else if((tag[IX(i,j,k)] & SOLID_LEFT)&&(tag[IX(i,j,k)] & SOLID_BOTTOM))
                        x[IX(i,j,k)] = 0.5f*(x[IX(i, j+1 , k)]+x[IX(i+1, j, k)] );
     else if((tag[IX(i,j,k)] & SOLID_LEFT)&&(tag[IX(i,j,k)] & SOLID_TOP))
                        x[IX(i,j,k)] = 0.5f*(x[IX(i , j-1, k)]+x[IX(i+1 , j, k)] );
     else if((tag[IX(i,j,k)] & SOLID_RIGHT)&&(tag[IX(i,j,k)] & SOLID_BOTTOM))
                        x[IX(i,j,k)] = 0.5f*(x[IX(i, j+1  , k)]+x[IX(i-1, j    , k)] );
     else if((tag[IX(i,j,k)] & SOLID_RIGHT)&&(tag[IX(i,j,k)] & SOLID_TOP))
                        x[IX(i,j,k)] = 0.5f*( x[IX(i, j-1, k)]+x[IX(i-1, j, k)] );
     else if((tag[IX(i,j,k)] & SOLID_LEFT) && (tag[IX(i,j,k)] &SOLID_BOTTOM) &&
(tag[IX(i,j,k)] & SOLID_BACK))
```

```
            x[IX(i,j,k)] = (x[IX(i+1,j,k)]+x[IX(i,j+1,k)]+x[IX(i,j,k+1)])/3.0f;
    else if((tag[IX(i,j,k)] & SOLID_LEFT) && (tag[IX(i,j,k)] &SOLID_BOTTOM) &&
(tag[IX(i,j,k)] & SOLID_FRONT))
            x[IX(i,j,k)] = (x[IX(i+1,j,k)]+x[IX(i,j+1,k)]+x[IX(i,j,k-1)])/3.0f;
    else if((tag[IX(i,j,k)] & SOLID_LEFT) && (tag[IX(i,j,k)] & SOLID_TOP ) &&
(tag[IX(i,j,k)] & SOLID_BACK ))
            x[IX(i,j,k)] = (x[IX(i+1,j,k)] + x[IX(i,j-1,k)] +x[IX(i,j,k+1)])/3.0f ;
    else if((tag[IX(i,j,k)] & SOLID_LEFT) && (tag[IX(i,j,k)] &SOLID_TOP) &&
(tag[IX(i,j,k)] & SOLID_FRONT))
            x[IX(i,j,k)] = (x[IX(i+1,j,k)] +x[IX(i,j-1,k)] +x[IX(i,j,k-1)])/3.0f;
    else if((tag[IX(i,j,k)] & SOLID_RIGHT) && (tag[IX(i,j,k)] & SOLID_BOTTOM) &&
(tag[IX(i,j,k)] & SOLID_BACK))
            x[IX(i,j,k)] = (x[IX(i-1,j,k)]+x[IX(i,j+1,k)]+x[IX(i,j,k+1)])/3.0f;
    else if((tag[IX(i,j,k)] & SOLID_RIGHT) && (tag[IX(i,j,k)] &SOLID_BOTTOM) &&
(tag[IX(i,j,k)] & SOLID_FRONT))  x[IX(i,j,k)] = (x[IX(i-
1,j,k)]+x[IX(i,j+1,k)]+x[IX(i,j,k-1)])/3.0f;
    else if((tag[IX(i,j,k)] & SOLID_RIGHT) && (tag[IX(i,j,k)] & SOLID_TOP ) &&
(tag[IX(i,j,k)] & SOLID_BACK ))
             x[IX(i,j,k)] = (x[IX(i-1,j,k)]+x[IX(i,j-1,k)]+x[IX(i,j,k+1)])/3.0f;
    else if((tag[IX(i,j,k)] & SOLID_RIGHT) && (tag[IX(i,j,k)] &SOLID_TOP) &&
(tag[IX(i,j,k)] & SOLID_FRONT))
            x[IX(i,j,k)] = (x[IX(i-1,j,k)]+x[IX(i,j-1,k)]+x[IX(i,j,k-1)])/3.0f;}
//advect for the x-component of velocity field
    __global__ void advectu (int *tag, float * d, float * d0, float * u, float* v, float *w, float
dt )
    {
            int   i0, j0, i1, j1, k0, k1;
            float x, y, z, s0, t0, q0, s1, t1, q1, dt0;
            dt0 = dt/h;
            int i = threadId.x + blockId.x * blockDim.x;
            int j = threadId.y + blockId.y * blockDim.y;
            int type;
            for(int k=0; k<NZ; k++)
            {
                 type  = tag[IX(i,j,k)]
            if (LIQIUID_CELL==type)
            {
               x = i-dt0*u[IU(i,j,k)]; y = j-dt0*v[IU(i,j,k)]; z = k-dt0*w[IU(i,j,k)];
               if (x<0.5f) x=0.5f; if (x>NX+1.5f) x=NX+1.5f; i0=(int)x; i1=i0+1;
               if (y<0.5f) y=0.5f; if (y>NY+0.5f) y=NY+0.5f; j0=(int)y; j1=j0+1;
               if (z<0.5f) z=0.5f; if (z>NZ+0.5f) z=NZ+0.5f; k0=(int)z; k1=k0+1;
               s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1; q1 = z-k0; q0 = 1-q1;
               d[IU(i,j,k)]=q0*(s0*(t0*d0[IU(i0,j0,k0)]+t1*d0[IU(i0,j1,k0)])
                   +s1*(t0*d0[IU(i1,j0,k0)]+t1*d0[IU(i1,j1,k0)]))+q1*(s0*(t0*d0[IU(i0,j0,k1]
               +t1*d0[IU(i0,j1,k1)])+s1*(t0*d0[IU(i1,j0,k1)]+t1*d0[IU(i1,j1,k1)]));
            }
```

```
        __syncthreads();
        set_vel_bnd_kernel(1,i,j,k,tag,d);}}
 //Jacobi method for solving Possion equation
 __global__ void jacobi(float *x, float *x0, int *tag, float a, float c, float epcl)

{
        int tx = threadId.x; int ty = threadId.y;
            int bx = blockId.x; int by = blockId.y;
        int i = tx + bx * blockDim.x;
        int j = ty + by * blockDim.y;
        int index = tx + by * blockDim.x;
        __shared__ float temp_x[NUMBER_THREAD];
            int type;
        while(error>epcl) {
        for (int k=0; k<NZ; k++){
        type = tag[IX(i,j,k)];
        if(LIQIUID_CELL==type){
           temp_x[index] = (x0[IX(i,j,k)] + a*(x[IX(i-1,j,k)]+x[IX(i+1,j,k)]+x[IX(i,j-
        1,k)]+x[IX(i,j+1,k)]+x[IX(i,j,k-1)+x[IX(i,j,k+1)]]))/c;
        __syncthreads();
        x[IX(i,j,k)] = temp_x[index];
        __syncthreads();}}
        set_pressure_bnd_kernel(i,j,k,tag,x);
        if(tx ==0 && ty ==0 && bx == 0 && by ==0) error = 0;
        __syncthreads();
        float x_dis = (x0[IX(i,j,k)]-x[IX(i,j,k)])*(x0[IX(i,j,k)]-x[IX(i,j,k)]);
            atomicExch(&error,x_dis);
        __syncthreads();
        }
}
```

References

Adams B, Pauly M, Keiser R, Guibas LJ (2007) Adaptively sampled particle fluids. ACM Trans Graph 48–48* (Proceedings of SIGGRAPH)

Amada T, Imura M, Yasumoto Y, Yamabe Y, Chihara K (2004) Particle-based fluid simulation on gpu. In: ACM workshop on general-purpose computing on graphics processors, pp 204–211

Brochu T., Batty C., Bridson R. (2010) Matching fluid simulation elements to surface geometry and topology. ACM Trans Graph 29:1–9 (Proceedings of SIG-GRAPH)

Carlson M, Mucha PJ, Turk G (2004) Rigid fluid: animating the interplay between rigid bodies and fluid. ACM Trans Graph 23(3):377–384 (Proceedings of SIGGRAPH)

Crane K, Llamas I, Tariq S (2007) Real-time simulation and rendering of 3D fluids. In: Nguyen H (ed) GPU gem 3. Addison Wesley, New York

Enright D, Marschner S, Fedkiw R (2002) Animation and rendering of complex water surfaces. ACM Trans Grap 21:736–744 (Procedings of SIGGRAPH)
Geiss R (2007) Generating complex procedural terrains using the gpu. In: Nguyen H (ed) GPU gem 3, Addison-Wesley, Reading, pp 7–37
Harada T, Koshizuka S, Kawaguchi Y (2007) Smoothed particle hydrodynamics on GPUs. In Proc Comput Graph Intl 63–70
Li W, Fan Z, Wei X, Kaufman A (2003) Gpu-based flow simulation with complex boundaries. Technical Report, 031105, Computer Science Department, SUNY at Stony Brook
Lin N (2007) Special effect with Geforce 8 series hardware. Game Developer Conference Shanghai
Liu Y, Liu X, Wu E (2004) Real-time 3d fluid simulation on gpu with complex obstacles. In: Proceedings of the 12th Pacific conference computer graphics and applications, pp 247–256
Losasso F, Shinar T, Selle A, Fedkiw R (2006) Multiple interacting liquids. ACM Trans Graph 25(3):812–819 (Proceedings of SIGGRAPH)
Losasso F, Talton J, Kwatra N, Fedkiw R (2008) Two-way coupled sph and particle level set fluid simulation. IEEE Trans Vis Comput Graph 14(4):797–804
Müller M, Charypar D, Gross M, (2003) Particle-based fluid simulation for interactive applications. In: SCA '03 Proceedings of the (2003) ACM SIGGRAPH/Eurographics symposium on computer animation (Aire-la-Ville. Switzerland), Eurographics Association, pp 154–159
Premoze S, Tasdizen T, Bigler J, Lefohn A, Whitaker R (2003) Particle-based simulation of fluids. Comput Graph Forum 22(3):401–410
Reeves WT (1983) Particle systems|a technique for modeling a class of fuzzy objects. ACM Trans Graph 2(2):91–108
Wang H, Mucha PJ, Turk G (2005) Water drops on surfaces. ACM Trans Graph 24(3):921–929
Wojtan C, ThÄurey N, Gross M, Turk G (2010) Physics-inspired topology changes for thin fluid features. ACM Trans Graph 1–8 (Proceedings of SIGGRAPH)
Zhao Y (2008) Lattice Boltzmann based PDE solver on the GPU. Visual Comput 24(5):323–333
Zhang Y, Solenthaler B, Pajarola R (2008) Adaptive sampling and rendering of fluids on the gpu. In: Volume and point-based graph, 2008 pp 137–146

Chapter 12
High Performance Implementation of Binomial Option Pricing Using CUDA

Yechen Gui, Shenzhong Feng, Gaojin Wen, Guijuan Zhang, Yanyi Wan and Tao Liu

Abstract Binomial tree model is often used for option pricing in the financial market. According to this method, it is rather expensive to obtain high accurate option price. Although existing methods running on CPU clusters have improved the efficiency significantly, there is still a great gap between the real performance and the desired. In this paper, we parallelize this model on CUDA to further improve the efficiency. We optimize our method according to principles of memory hierarchy and extend it to support multiple GPUs. Experiments on single Tesla C1060 GPU chip show an average of 285× speedup compared to the result on single CPU node. Furthermore, for the data size of 64 K, GPU performance has reached 315 Gflops, which outperforms the earlier version on the Sun station by a factor of about 100×. The maximum performance reached with 108 GPU nodes is 30 Tflops.

Keywords Binomial tree · Option pricing · CUDA · GPU

12.1 Introduction

Binomial tree model is a commonly used method for valuing options in the financial market (Hull 2005). The model is efficient when the number of involved options is small. In general, only a few milliseconds are needed for computing the price of a single option on modern CPU. However, if the number of options grows, the computation becomes very expensive. This issue often occurs when revaluing a large number of real-time options with live data-feeds, or embedding pricing models into a Monte-Carlo simulation. Therefore, improving the efficiency of binomial tree model is crucial.

Y. Gui (✉) · S. Feng · G. Wen · G. Zhang · Y. Wan · T. Liu
Center of High Performance Computing, Institute of Advanced Computing and Digital Engineering, Shenzhen Institutes of Advanced Technology, Chinese Academy of Science, 518055 Shenzhen, China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 201
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_12,

Related work focuses on implementing the binomial tree model on CPU clusters. Although they show hundreds times speedup, the overall efficiency is still unsatisfied (Gerbessiotis 2004; Thulasiram 2002). For example, when computing a single option with 32 K time steps, the algorithm proposed by Gerbessiotis (2004) only achieves 1.5 % of peak performance on a single node of Pentium cluster. This ratio degrades to 0.8 % on 16 nodes. The method proposed by Zubair and Mukkamala (John and Mohammad 2010; Mohammad 2008) achieved 1484 Mflops for 64 K time steps on 8 UltraSPARCIIIi processors by considering the memory hierarchy. Note that it is not necessary to keep 64 k time steps since the binomial model has sustained good convergence when the time step is larger than 1 k (Mehmet 2007). Also, the CPU clusters are very expensive to build and it is inconvenient to optimize corresponding methods.

Another group method adopts programmable GPUs to compute option price. For example, *GPU gemsIII* shows how to implement Monte Carlo methods on Geforce 8800 (http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html). In Matthew and Jike (2009), the authors proposed a GPU-based market value-at-risk estimation algorithm. In addition, Mehmet (2007) and Qiwei and David (2009) also provided GPU-based solution for binomial option pricing. However, the former method can only process small-scale Europe options while the latter doesn't optimize for high efficiency.

In this paper, we propose an algorithm based on CUDA architecture for large-scale real-time pricing. Our method outperformed the algorithm running on sun station (John and Mohammad 2010) by a factor of about 100 and also outperformed the algorithm in Qiwei and David (2009) by a factor of 4.

The rest of paper is organized as follows: Sect. 12.2 reviews the binomial model for American option pricing; Sect. 12.3 discusses how to parallelize the model on GPU and introduced some optimizations; Sect. 12.4 presents the results and analysis. Finally, we give our conclusion in Sect. 12.5.

12.2 The Binomial Option Pricing Model

An option contract is a financial instrument that gives its holder the right to buy or sell a financial asset at a specified price (which is referred to as strike price) on or before the expiration date. Under certain simplifying assumptions, we can statistically model the asset's future price fluctuations and estimate the value of option using the asset's current price and its volatility by establishing a binomial tree. This tree model derives the value of an option at time $t = 0$ (that is, now) by stepping "backward" from time $t = T$ in a discrete number of time steps N, where T is the expiration date (Fig. 12.2).

Without loss of generality, we consider in particular the pricing of an American put option. We adopt the standard notations for describing the option, i.e. denoting the current price of the underlying asset by S, its strike price by K, its expiration time by T, and the volatility of the asset by v. In addition, let the interest rate be r, and the

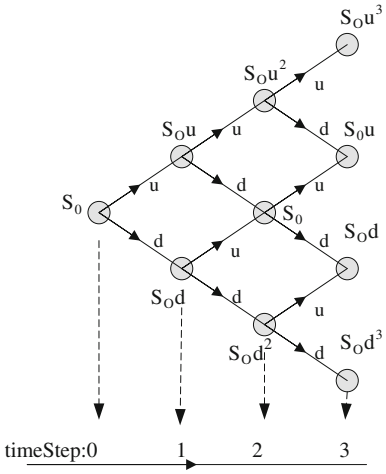


Fig. 12.1 Asset price fluctuations in a three-step tree

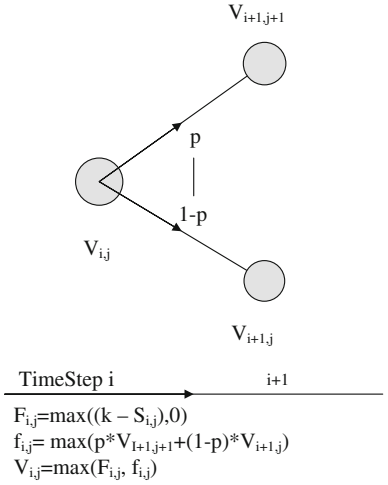


Fig. 12.2 The valuation of a put option at one step

smallest time interval for the value-variation in the model be Δt. The computation process is described below:

1. Establish a binomial tree of length $n = T/\Delta t$ to trace the fluctuations of the asset price S. We identify the jth node at time step i by the index (i, j) where $0 < j < i$, $0 < i < n$, and set step 0 as the root node. The asset price S(i, j) at each node during iteration is given by Eq. (2.1), in which u and d indicate the fraction by which this asset price can go up or down respectively during one time interval Δt. Figure 12.1 shows this process by a three-step tree.

$$S_{i,j} = S_0 u^{i-j} d^j = S_0 u^{i-2j} 0 \leq j \leq i \tag{2.1}$$

2. Iteratively compute option prices $V_{i,j}$ at time step i by using the value pricing at time step $i+1$ by Eqs. (2.2)–(2.4). Note that Eqs. (2.3) and (2.4) are computed for the consideration of early exercise while pricing an American option, in which $F_{i,j}$ is the option payoff at the current step.

$$f_{i,j} = e^{-r\Delta t}[pV_{i+1,j+1} + (1-p)V_{i+1,j}] \tag{2.2}$$
$$F_{i,j} = \max(k - S_{i,j}, 0) \tag{2.3}$$
$$V_{i,j} = \max(F_{i,j}, f_{i,j}) \tag{2.4}$$

Also, the pseudo probabilities p in Eq. (2.2) is given by Cox et al. (1979) as Eq. (2.5).

$$\begin{aligned} u &= e^{\sigma\sqrt{\Delta t}} \\ d &= e^{-\sigma\sqrt{\Delta t}} \\ p &= \frac{e^{(r-q)\Delta t} - d}{u - d} \end{aligned} \tag{2.5}$$

12.3 GPU Implementation Using CUDA

12.3.1 A Brief Description of GPU

The new generation of GPUs adopts the unified shader architecture CUDA and has evolved into a highly multithreaded, many-core processors with tremendous computational power.

Taking Telsa C1060 illustrated in Fig. 12.3 as an example. The GPU is composed of 30 multiprocessor units (indicated by gray box) with 8 processors in each unit (indicated by small orange boxes). Each multiprocessor unit can process parallel groups of threads concurrently, called warps, and the whole device has a parallel data cache memory of 16 KB size that can be shared within the multiprocessor unit. All the processors can be driven by CUDA API (http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html), by which a GPU kernel can be launched, and the grid of a thread block and threads can be allocated.

12.3.2 GPU Implementation Using CUDA

Since GPU runs on Single-Instruction, Multiple-thread (SIMT) execution mode (Nvidia 2009), and pricing for one option do not interfere with the pricing for other options, we set our kernel grid as a two-dimension grid, in which each block will

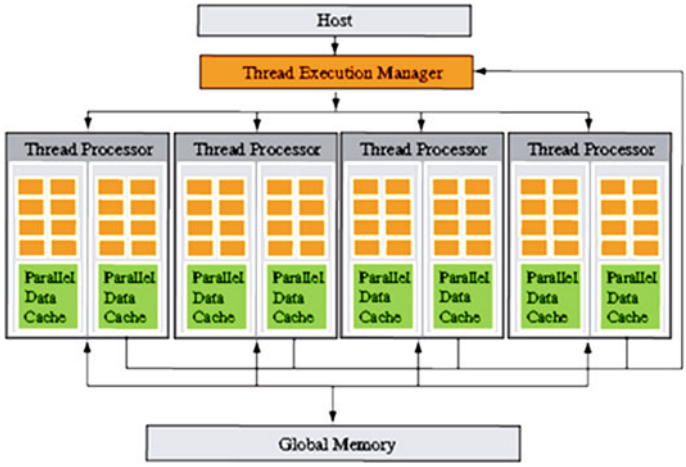


Fig. 12.3 GPU architecture, the thread processors is indicated in *orange* (http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html)

process a single binomial tree and price for an option, while the number of thread blocks is the same as that of options. That is, every thread has only to address two data elements and then process the operation explained in Eq. (2.5) at each time step.

As accessing shared memory is around 300–400 times faster than accessing the global memory (http://www.mcs.anl.gov/research/projects/mpi/) on a GPU chip, the straightforward approach is to fit the whole tree into the shared memory and fit all local variables into the register file. However, this approach is not efficient and valid for two reasons:

1. The CUDA architecture introduced the term *warps* for a multiprocessor to manage and schedule the large amount of threads. A warp is a group of 32 threads that run concurrently on GPU and get scheduled by a *warp scheduler* for execution. When one step function of accessing the same memory address is executed between different warps, the order of threads scheduling might cause read-after-write hazards as shown in Fig. 12.4, where data V(i,j) in the array which depends on the valuation of $V(i-1, j-1)$ in warps: k' has been updated to $V(i-1, j)$ by warps: k, leading the step function executed by thread: $j-1$ in warps: k to generate the wrong result.
2. The size of the shared memory on a GPU chip is only 16 KB and the maximum dimension of one thread block is 512, loading all nodes at one time step into the shared memory for reduction is impractical, especially when the binomial tree is very large and other extra memory usage is needed. Meanwhile, overly reliance on the extra shared memory can act as a constraint on GPU occupancy (Matthew and Jike 2009) and might results in fewer active blocks shared by different multiprocessors, which forces some multiprocessors to be idle during thread synchronization and during device memory read/write.

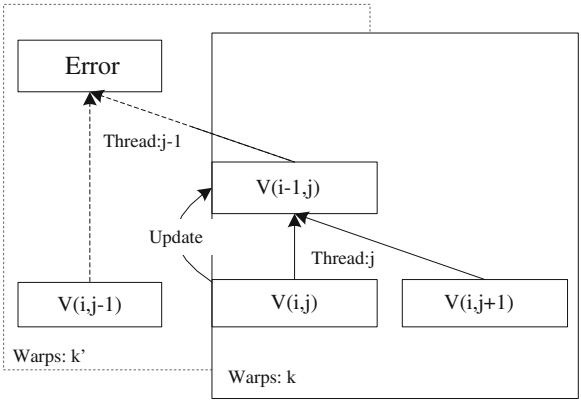


Fig. 12.4 The read-after-write hazard among shared memory at the same time step

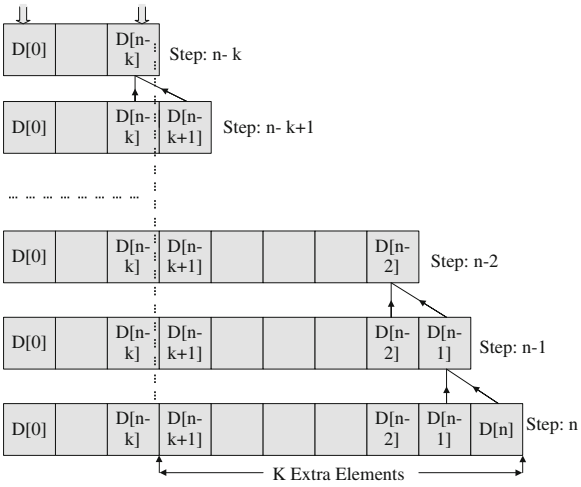


Fig. 12.5 The relation between time steps and the data size

To address the first problem, we use double-buffering in the shared memory. We introduce two indexes, pIn and pOut, to record the current process. They are initialized to 1 and 0, respectively, and swap their values every round for later iterations.

For the second problem, we partition the whole tree into several sub-parts. It can be seen from Fig. 12.5 that if we need to reduce k steps from D[0] to D[n − k] , an extra of k elements from D[n − k] must be loaded in. Therefore, for all the nodes residing in global memory, we could first load an sub-binomial tree of cache size into the shared memory and then reduce them by k steps just as Fig. 12.6 shows, then repeat this process for the next sub tree until all the nodes at the same step has been reduced. Next we perform synchronization and put the results into the global

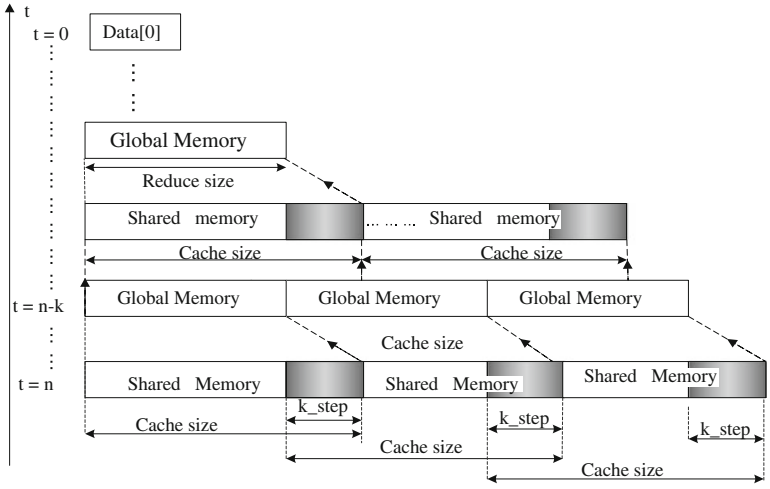


Fig. 12.6 The reducing procedure of the whole tree

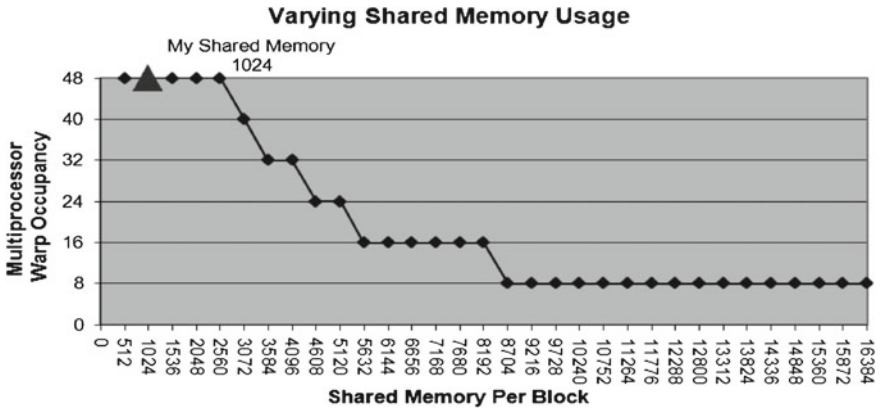


Fig. 12.7 Theretical multiproccessor warp occupancy of Telsa C1060 (Resource usage is indicated by the *gray triangle* on the graph, which is 1024)

memory. We iteratively carry out the described procedure for the next k steps until reaching D[0] at time step $t = 0$.
Further optimizations:

1. By using CUDA GPU Occupancy Calculator (http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html), we can trace the GPU occupancy which is the theoretical ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. This ratio is determined by GPU resource usage such as shared memory used per block and register per thread. Our final choice and theoretical GPU Warp Occupancy is indicated in the Fig. 12.7. Besides, this

calculator can also help to cover latency during global memory loads that are followed by a__syncthreads() in one block.
2. Once the parameters u, d, p indicated by Eq. (2.4) have been computed on the device, we load them into GPU constant memory for higher efficiency since constant memory is cached and these variables need not be updated during the whole computation.

12.3.3 Multi-GPU Implementation

Since GPU kernel must be launched through CPU, We use MPI (http://www.mcs.anl.gov/research/projects/mpi/) to activate multi-CPU threads so that this approach could be mapped onto multi-GPUs. To exploit the efficient utilization of GPU resources on other nodes, we make three assumptions:

1. The amount of the options is proportional to the number of nodes, and when the number of nodes is 108, the total amount of the option to be valued is 108* 16 K.
2. All request for option valuations are received simultaneously.
3. For every option, we price it from the expiration date to three months later with the time interval 0.0027, which approximates one day.

12.4 Experimental Results

The performance results are summarized in Sects. 12.4.1 and 12.4.2, respectively. We use the performance of the algorithm on CPU as the comparison benchmark.

12.4.1 GPU Implementation on Single Node

Our GPU implementation on singe node is run on a Tesla C1060 chip with 16 KB shared memory. The implementation on CPU runs on Intel 2.2 GHz Core2Due processor. The overall performance is shown in Table 12.1, where option/second represent the number of option priced per second and the speedup is the ratio computed by CPU_time divided by GPU time. In particular, Fig. 12.8 indicates that the GPU implementation offers an 143–370 times acceleration as the number of option increases. Not surprisingly, the GPU performance roars up from 122.5 Gflops with 1024 options to 315.10 Gflops with 64 K options since more active blocks will be activated and computation became more intensive. However, when Peak (%) is taken into consideration, we find that even when the number of options has reached up as 65535, the GPU peak performance has only got up to 33.8 %, which might due to the frequent data transfer between shared memory and global memory every k steps

Table 12.1 Performance on Tesla C1060(options/second is options being priced in every second, the total number of time steps is 1024)

Options	1024	2048	4096	8192	16384	32768	65535
CPU_time (s)	25.20	50.53	101.10	202.2	404.3	805.88	1615.9
GPU_time (s)	0.18	0.24	0.37	0.64	1.17	2.24	4.38
Speed up	143.8	211.7	271.70	316.10	345.3	360.6	369.2
Options/second	5848	8580	11004	12808	13958	14621	14980
Gflops	122.53	180.43	234.43	269.36	293.5	307.5	315.10
Peak (%)	13.2	19.3	24.8	28.9	31.5	33.0	33.8

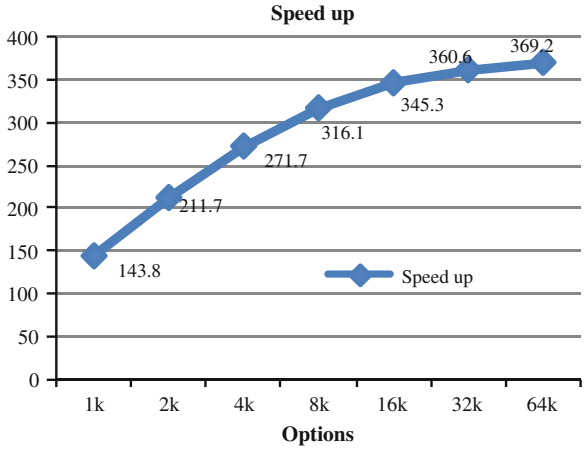


Fig. 12.8 Speedup ratio obtained by GPU

mentioned before. In general, 8580 options/second for the data size of 2 K has already satisfied the requirements of real-time business processing engine (Fig. 12.9).

We have also conducted experiments to see the impact of the size of shared memory on the performance. The result is shown in Table 12.2, where the total time step was set as 4 k to ensure that GPU performance is of high sensibility to the communication delay. We have observed a maximum performance of the theoretical peak for the shared memory size of about 1024, which is about 38 % no matter how many time steps (indicated as step_iter in the table) are reduced once. This satisfies the result simulated by the GPU Occupancy Calculator and as explained before, proper usage of shared memory might help overlapping between data load latency and time cost by synchronizing inside the thread block.

Furthermore, since binomial model is a numerical approximation method, we investigate its precision as well as the variation of GPU performance with different number of steps. The result is shown in Table 12.3. In this table, we find that as the step number increases, GPU performance also increases substantially because of more parallelization. At the same time, precision has also been improved. In reality,

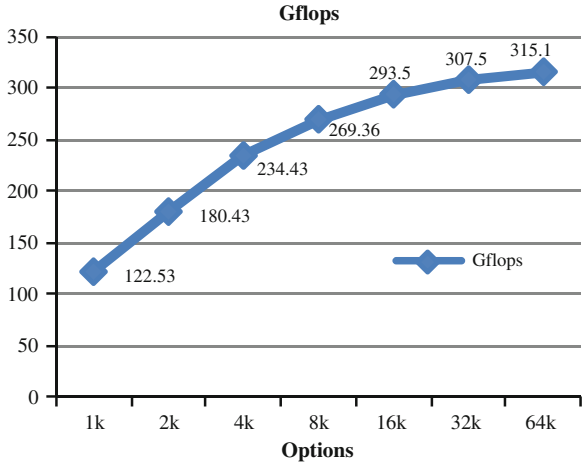


Fig. 12.9 Gflops rating for this algorithm

Table 12.2 The influence of shared_data_size upon the peak performance (%) of GPU (the amount of options is 64 k, the total time steps is 4 k, step_iter is the time steps reduced in the cache each time)

Shared_data_size (bytes)	256 (%)	512 (%)	1024 (%)	2048 (%)
Step_iter = 8	33.5	35.8	36.4	33.2
Step_iter = 16	30.3	35.2	37.3	35
Step_iter = 32	21.1	31.4	36.1	35.3

Table 12.3 The influence of the number of time steps upon the GPU performance and the accuracy (the cache_data size is 1 k, the total options are 64 k)

Time_steps	128	256	512	1024	2048	4096
Gflops	109.51	199.04	279.76	321.56	340.06	348.20
Peak (%)	11.7	21.3	30	34.4	36.4	37.3
Precision (10^{-6})	2.91	5.93	11.9	18.1	18.2	18.2

it is very rare for step number to be greater than 1024, and there is a trade-off between the time cost and the precision. Note that precision is computed compared with CPU like follows, where v_i is the price computed on GPU, r_i is the price computed on CPU:

$$MeanError = \sum_{i=1}^{n} |v_i - r_i| / r_i$$

Table 12.4 Multi-GPUs performance for American option pricing

GPU cards	Performance (Gflops)
4	1425.48
8	2835.30
16	5617.84
32	11023.93
60	17738.67
84	24308.71
96	27793.77
108	30930.97

12.4.2 Multi-GPU Implementation

The overall result of multi-GPU implementation is shown in Table12.4. As data size increases, GPU performance has raised from 1425.48 Gflops on 4 Tesla C1060 to 30930.97 Gflops on 108 Tesla C1060s, which demonstrates a fairly good scalability on multi-GPUs.

12.5 Conclusion

This paper describes a parallel binomial tree model for valuing American options based on GPUs. The key features include using GPU memory hierarchy and binomial tree partitioning. The implementation on single GPU runs 300 times faster than its corresponding result running on CPU for the 8 k data size and the GPU performance has reached up to 315 Gflops for 64 k data size. Besides, it also presents a good scalability when it is mapped on multi-GPUs.

For the future work, we want study the GPU implementations with higher utilization ratio of GPU resources, as compared with the current value of around 38 %. Task Scheduling can be studied to enhance performance. In addition, we can extend the binomial tree models to trinomial tree models.

Acknowledgments This work is supported by the National High-Tech Research and Development Plan of China under Grant Nos. 2006AA01A114, 2007AA120502, and Shenz-hen Innovation Technology Program under Grant No. SY200806300211A.

Appendix

```
//CUDA-interp-MPI Code for multi-GPU American options pricing of the
expiring date varying from one month to three month
CUTThread* threadID = (CUTThread*)malloc(sizeof(CUTThread)*GPU_N);
//Get options needed to be priced for each GPU and recieved input data for
optionSolver[][]……
MPI_Recv(options, GPU_N*opt_N*sizeof(Option_data)/sizeof(float),
MPI_FLOAT, MASTER, mtype, MPI_COMM_WORLD, &status);
mygputime = clock();
//Start CPU thread for each GPU and launch GPU kernels by solverThread
for(ig = 0; ig < GPU_N; ig++)
threadID[ig] = cutStartThread((CUT_THREADROUTINE)solverThread,
&optionSolver[ig]);
cutWaitForThreads(threadID, GPU_N);
//when CPU threads is initilized, launch GPU kernels
static CUT_THREADPROC solverThread(TOptionPlan *plan){
        //Init GPU
        cutilSafeCall( cudaSetDevice(plan->device) );
        int count = 0;
        for (float t = plan->T_begin; t < plan->year_Count; t += plan-
>delta_T )
        {binomial_GPU(plan->optionData,plan-
>callValue+count*opt_N,opt_N,t);
                count++;
        }
        //Shut down this GPU
        cudaThreadExit();
        CUT_THREADEND;
}
```

```
//compute the current price for S on GPU
__device__  float expiryValue_current(float S,float X,float vDt,int CurrentStep ,int
i)
{
        real d = S * __expf(vDt*(2.0 * i - CurrentStep))-X;
        return (d >0)? d : 0;
}
//data transfer between global memory and shared memory on GPU,end_Index is
the maximum index of cache_Data,callValueBuf_A stands for the array residing in
shared_memory.v_base is the current address on Global memory
if(tid <=max_Index)
{buf_Out = 0;
buf_In = 1;
callValueBuf_A[buf_Out* CACHE_DATA + tid] = optionsPresent[v_base + tid];
}
//data reducing in the shared memory starting from the maximum index at the
begginning time step,which is indicated as begin_Index and the maximum
index ,which is indicated as end_Index at the last step in the k step iteration
j = begin_Index – 1;
if(j>= end_Index)
{
        __syncthreads();
        //double buffering
        buf_Out = 1 - buf_Out;
        buf_In = 1 - buf_Out;
        if ( tid <=  j)
                callValueBuf_A[buf_Out * CACHE_DATA + tid] = max(
                pu * callValueBuf_A[buf_In* CACHE_DATA + tid + 1] +pd
*callValueBuf_A[buf_In* CACHE_DATA + tid ],expiryValue_current( S , X , vDt ,
step_temp,c_base + tid));
        j--;
        step_temp--;
   }
```

References

Cox JC, Ross SA, Rubinstein M (1979) Option pricing: a simplified approach. J Financ Econ 7:229-263

Gerbessiotis AV (2004) Architecture independent parallel binomial tree option privaluations. Parall Comput 30(2):301–316

Hull J (2005) Options, futures, and other derivatives, 6th edn. Prentice Hall, New Jersey

John ES, Mohammad Z (2010) Cache-optimal algorithms for option pricing. ACM Trans Math Softw 37(1):7:1–7:30 (Article 7)

Matthew D, Jike C (2009) Acceleration of market value-at-risk estimation, In WHPCF ’09: Proceedings of the 2nd workshop on high performance computational finance pp 1–8

Mehmet H (2007) A comparison of lattice based option pricing models on the rate of convergence. Appl Math Comput 184:649–658

Mohammad Z, Ravi M (2008) High performance implementation of binomial option pricing. In: ICCSA, Part I. LNCS, vol5072. pp 852–866

Nvidia (2009) CUDA C programming guide 3.0. NVIDIA Corp

Qiwei J, David B (2009) Exploring reconfigurable architectures for tree-based option pricing models. ACM Trans Reconfigurable Technol Syst 2(4):21:1–21:17 (Article 21)

Thulasiram RK (2002) Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives. In: Proceedings of the 4th international workshop on high performance scientific and engineering computing with applications, Vancouver, Canada, IEEE Computer Society. Los Alamiatos pp 306–313

Chapter 13
Research of Acceleration MS-Alignment Identifying Post-Translational Modifications on GPU

Zhai Yantang, Tu Qiang, Lang Xianyu, Lu Zhonghua and Chi Xuebin

Abstract MS-Alignment is an unrestrictive post-translational modification (PTM) search algorithm with an advantage of searching for all types of PTMs at once in a blind mode. However, it is time-consuming, and thus it could not well meet the challenge of large-scale protein database and spectra. We use Graphic Processor Unit (GPU) to accelerate MS-Alignment for reducing identification time to meet time requirement. The work mainly includes two parts. (1) The step of Database search and Candidate generation (DC) consumes most of the time in MS-Alignment. We propose an algorithm of DC on GPU based on CUDA (DCGPU). The data parallelism way is partitioning protein sequences. We adopt several methods to optimize DCGPU implementation. (2) For further acceleration, we propose an algorithm of MS-Alignment on GPU cluster based on MPI and CUDA (MC_MS-A). The comparison experiments show that the average speedup ratio could be above 26 in the model of at most one modification and above 41 in the model of at most two modifications. The experimental results show that MC_MS-A on GPU Cluster could reduce the time of identifying 31173 spectra from about 2.853 months predicted to 0.606 h. Accelerating MS-Alignment on GPU is applicable for large-scale data requiring for high-speed processing.

Z. Yantang (✉) · T. Qiang · L. Xianyu · L. Zhonghua · C. Xuebin
Supercomputing Center of Computer Network Information Center,
Chinese Academy of Sciences, Tianjin, China
e-mail: zyt0303@163.com

T. Qiang
e-mail: tuqiang@sccas.cn

L. Xianyu
e-mail: lxy@sccas.cn

L. Zhonghua
e-mail: zhlu@sccas.cn

C. Xuebin
e-mail: chi@sccas.cn

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 215
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_13,

13.1 Introduction

Reliable identification of post-translational modifications (PTMs) is key to understanding biological functions of proteins (Dekel et al. 2005; Seungjin et al. 2008; Stephen et al. 2005). Most MS/MS database search algorithms perform a restrictive search that can only take into account a few types of PTMs and ignore all others. Dekel Tsur etc. proposed an unrestrictive PTM search algorithm MS-Alignment, one advantage of which is its searching for all types of PTMs at once in a blind mode (Dekel et al. 2005; Seungjin et al. 2008). However, it has high time complexity and low identification speed, and thus it couldn't meet the demand of processing even large-scale protein database and spectra in a short time.

The development of high performance computing (HPC) makes a significant contribution to accelerating scientific computing. The programmable Graphic Processor Unit (GPU) has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth (NVIDIA 2009). GPU is especially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity (NVIDIA 2009). GPU well speeds up many applications that process large datasets in the field of image rendering. With enhancing programmability on GPU, it is applied to general-purpose computing breaking through the field of image rendering (John et al. 2007). With the development of GPU, general-purpose computing on GPU evolves as a studying focus in recent years (John et al. 2007; Wu 2004). That GPU-based HPC is applied to computational biology also becomes a development trend and many GPU-based algorithms and softwares in bioinformatics come into being (Liu et al. 2009; Lukasz and Witold 2009; Michael et al. 2007; Svetlin and Giorgio 2008).

NVIDIA introduces Compute Unified Device Architecture (CUDA) which is a general-purpose parallel computing architecture with a new parallel programming model and instruction set architecture. CUDA leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU (NVIDIA 2009). With the CUDA programming model that focuses on data parallelism, one can achieve both high-performance and high-reliability in their applications (David and Hwu 2010).

13.2 Research of MS-Alignment Algorithm

13.2.1 Description of MS-Alignment

Modified Protein Identification Problem (Dekel et al. 2005)
Input: A database of proteins *DB*, an experimental spectrum *S*, and a parameter *k* capping the number of modifications.

Output: A modified peptide with the best match Match(S, $\overline{P_B}$) to the spectrum S that is at most k modifications away from a peptide P that appears in the database (namely, P is a substring of some protein in the database).

MS-Alignment mainly has the following steps: Preprocessing, Tag Generation, Database Search and Candidate Generation (DC), P-value Computation, of which DC is the key step. Computing match score of any candidate to the spectrum S (hereinafter "computing match score") is a significant part of DC. MS-Alignment includes two program modules: the module with k equaling one (hereinafter "Mod = 1") and the module with k equaling two (hereinafter "Mod = 2"), and the methods of the two modules are different.

1. The Method of Computing Match Score in Mod = 1 (Zhai 2010; Zhai et al. 2010b)

The match score Match(S, $\overline{P_{i,i'}}$) of candidate $\overline{P_{i,i'}} = p_i \ldots \hat{p}_j \ldots p_{i'}$ to S consists of three parts: score of a prefix, score of a PTM and score of a suffix (see Eq. 13.1).

$$\begin{aligned}&\text{Match}(S, \overline{P_{i,\,i'}}) = \\ &\text{PrefixScore}(i,\, j-1) + \text{PTMScore}(\Delta,\, p_j) + \text{SuffixScore}(j+1,\, i')\end{aligned} \tag{13.1}$$

where $i \le j \le i'$,

$$\text{PrefixScore}(i,\, j-1) = \begin{cases} \sum_{l=i}^{j-1} \text{MassScore}(\text{mass}(p_i \ldots p_l)) & \text{if } j > i \\ 0 & \text{if } j = i \end{cases}$$

$$\text{SuffixScore}(j+1, i') = \begin{cases} \sum_{l=j+1}^{i'} \text{MassScore}(\text{PM(S)} - \text{mass}(p_l \ldots p_{i'})) & \text{if } j < i' \\ 0 & \text{if } j = i' \end{cases}$$

PM(S) is the mass of the spectrum, which is equal to the mass of the peptide that generated the spectrum. PTMScore(Δ, p_j) ≤ 0 is a penalty for having a modification Δ on the amino acid p_j. MassScore (v) is a scoring function for every mass v.

2. The Method of Computing Match Score in Mod = 2 (Zhai 2010; Zhai et al. 2010a,b).

A database peptide $P = p_1 \ldots p_n$ is a sequence of amino acids and $F = f_1 \ldots f_m$ is the set of spectral peaks. Define a product $P \otimes F$ as $p_n \times f_m$ two-dimensional matrix D, and define directed graphs with vertices corresponding to elements in the matrix and edges corresponding to pairs of vertices (i, j) and (i', j') with $i \le i'$ and $j \le j'$. The score of a vertex is the score of the path ending with this vertex. A vertex (i, j) to be scored will be directed by the vertex (i_{prev}, j_{prev}) having be scored with an edge which has the top score. A peptide can be formed by the sequence of amino acids corresponding to a path in the matrix as a prefix and the following sequence of amino

acids as a suffix, whose match score is the sum of scores of the path, the suffix and PTM. Peptides whose scores are greater than a threshold are marked as candidates. Equation 13.2 illustrates match score of peptide $P_{i',j} = p_{i'} \ldots p_j$ $(i' \leq i \leq j)$ to a spectrum S.

$$\begin{aligned} &\text{Match}(S, P_{i',j}) = \\ &D(p_{i'}, f_k) + \text{SuffixScore}(p_{i+1}, p_j) + \text{PTMScore}(\Delta, p_{i+1}) \end{aligned} \tag{13.2}$$

where $p_i \in P$ and $f_k \in F$.

MS-Alignment computed elements in D by row using dynamic programming. The recursion formula is illustrated in Eq. 13.3.

$$\begin{aligned} &D(p_i, f_k) = \text{MassScore}(\text{mass}(f_k)) + \max\{ \\ &\quad \text{StartScore}(p_i); \\ &\quad \max\{D(p_{i-1}, f_b) + \text{score}(e) | \\ &\qquad e \in \{\text{edges that end with} f_k \text{ and whose masses equal mass}(p_i)\}, \\ &\qquad f_b \text{ is the starting point of e}\}; \\ &\quad \max\{D(p_{i-2}, f_b) + \text{MassScore}(1/2\text{mass}(e)) + \text{score}(e) | \\ &\qquad e \in \{\text{edges that end with} f_k \text{ and whose masses equal mass}(p_{i-1}p_i)\}, \\ &\qquad f_b \text{ is the starting point of e}\}; \\ &\quad \max\{D(p_{i-3}, f_b) + \\ &\qquad \text{MassScore}(1/3\text{mass}(e)) + \text{MassScore}(2/3\text{mass}(e)) + \text{score}(e) | \\ &\qquad e \in \{\text{edges that end with} f_k \text{ and whose masses equal mass}(p_{i-2}p_{i-1}p_i)\}, \\ &\qquad f_b \text{ is the starting point of e}\}; \\ &\quad \max\{D(p_{i'}, f_0) + \text{PrefixScore}(p_{i'}, p_i) | 0 < i' \leq i\}; \\ &\quad \max\{D(p_{i'}, f_0) + \text{PrefixScore}(p_{i'}, p_i) + \text{PTMScore}(\Delta, p_i) | 0 < i' \leq i\}; \\ &\} \end{aligned} \tag{13.3}$$

13.2.2 Performance Analysis of MS-Alignment

Let N_s be the number of spectra, let N_p be the number of peptides of protein database, let m_p be the number of amino acids of a peptide, let m_s be the number of spectral peaks, let L be the length of a candidate and let M_d be the number of PTM types at an amino acid. We conclude that the average time complexity of Mod $= 1$ is $O(N_s \cdot N_p \cdot m_p \cdot L \cdot M_d)$ and the average time complexity of Mod $= 2$ is $O(N_s \cdot N_p \cdot m_p \cdot m_s \cdot L \cdot M_d)$ (Zhai 2010; Zhai et al. 2010a,b).

The parent mass of a spectrum is an important factor of convergence, which L is closely related to. In general, the larger L the higher the parent mass whereas the smaller L the lower the parent mass. Computation time of MS-Alignment is proportional to the number of spectra, the size of database, and the parent mass of a spectrum. That of MS-Alignment(Mod $= 2$) is proportional to the number of spectral peaks additionally (Zhai 2010; Zhai et al. 2010a,b)

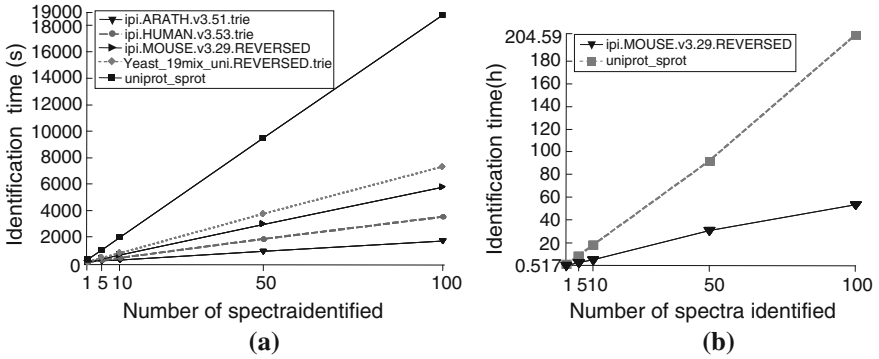


Fig. 13.1 **a** The identification time of MS-Alignment(Mod = 1) by the number of spectra. **b** The identification time of MS-Alignment(Mod = 2) by the number of spectra

We use the following computing environment CE1: CPU: Intel Xeon E5410 2.33 GHz, Memory: 8 GB, moreover, we use the following MS/MS and protein databases.

- MS/MS: HISTONE2H_1D provided by PhD Sheng Quanhu in Shanghai Institutes for Biological Sciences (SIBS) of Chinese Academy of Sciences (CAS).
- Protein databases: ipi.MOUSE.v3.29.REVERSED and Yeast_19mix_uni. REVERSED are provided by Sheng Quanhu; ipi.ARATH.v3.51, ipi.HUMAN. v3.53 and uniprot_sprot are downloaded at European Molecular Biology Laboratory-European Bioinformatics Institute (EMBL-EBI).

We get the computation time of MS-Alignment by inputting different number of spectra and different databases, and we get statistics by computing the average of N results (see Eq. 13.4).

$$T = \frac{1}{N}\sum_{i=1}^{N} t_i \tag{13.4}$$

Figure 13.1a, b respectively illustrate the computation time of MS-Alignment in Mod = 1 and Mod = 2 in CE1 (Zhai 2010; Zhai et al. 2010b). The average parent mass of the spectra used to test Mod = 1 is 1125.826 Da and that of the spectra used to test Mod = 2 is 1130.808 Da. According to the growth trend in the above figures, we forecast the computation time of MS-Alignment identifying another 31173 mass spectra whose average parent mass is 1409.374 Da (see Table 13.1) (Zhai et al. 2010b). It can be concluded that MS-Alignment is time-consuming and it could not well meet the challenge of large-scale data.

Using the same statistic method as Eq. 13.4 we get that the computation time of the step of DC accounts for more than 99 % of the total time of MS-Alignment. Therefore, we could improve the cost performance of MS-Alignment by reducing the time of DC.

Table 13.1 The predicted computation time of MS-Alignment identifying 31173 mass spectra

Algorithm	Protein database	Predicted time
MS-Alignment(Mod = 1))	ipi.MOUSE.v3.29.REVERSED	26.074 days
	uniprot_sprot	2.853 months
MS-Alignment(Mod = 2)	ipi.MOUSE.v3.29.REVERSED	2.370 years
	uniprot_sprot	9.074 years

13.3 Research and Implementation of DCGPU Algorithm

13.3.1 DCGPU Algorithm Design

GPU is only co-processor not central processing unit now. Explicitly transferring data across the PCI Express bus between host memory and device memory similar to the "ping-pong" operation is most frequently used, although zero-copy is available.

Referring to InsPecT (Stephen et al. 2005) implementing the MS-Alignment algorithm, we propose the DCGPU (DC on GPU based on CUDA) algorithm implementing the step of DC on single GPU. Data partition method is as follows:

- Denote the protein database as a single sequence Q.
- Partition Q into equal-length sequences (QG) to serially compute on GPU, since $Q$is too long to compute on GPU in one time.
- Partition QG into equal-length sequences (QB) in blocks.
- Partition QB into shorter sequences in threads and share some amino acids between threads.

DCGPU includes two different algorithms: DCGPUM1 (DCGPU for Mod = 1) and DCGPUM2 (DCGPU for Mod = 2) since Mod = 1 and Mod = 2 are different. The following describes them:

1. DCGPUM1 Algorithm (Zhai 2010; Zhai et al. 2010b)

Let BS be the number of threads per block and let GS be the number of blocks per grid. The length of a sequence computed in a thread is set to 32 based on the statistics in (Ari 2008; Sheng et al. 2000) and features of CUDA programming.

- Host Site:

(1) Initialize;
(2) Add 31 asterisks in front of Q (asterisk denotes separator in protein database, whose mass is positive infinite); set ReadP which is read pointer to Q to zero;
(3) Transfer $GS*BS+32$ length of sequence begging with ReadP from host memory to device memory;
(4) Call GPU kernels to compute and then transfer candidates from device memory to host memory;
(5) Update read pointer: ReadP+ $=GS*BS$; if ReadP isn't larger than the length of Q, loop back to (3), otherwise go on;
(6) Get and output best matches; free resources and finish.

- GPU Site:

(1) The first kernel kernel_1 computes mass and score of every prefix of every peptide in Q. One-dimensional block and one-dimensional thread are specified. Let tx be threadIdx.x that refers to the x-index of a thread and let bx be blockIdx.x that refers to the x-index of a block. For simplicity, we will refer to a thread as Thread_{tx} and a block as Block_{bx} since y-index is zero. Every block loads $BS + 30$ length of sequence from global memory to shared memory. Block_{bx} reads the sequence from bx*BS to $(bx + 1)$*$BS + 29$. Therefore, a grid needs GS*$BS + 30$ length of sequence from global memory. Using Eq. 13.5, a thread computes mass and score of every prefix P_{prefix} of peptide $P = p_i \ldots p_{i+30}$ begging with amino acid p_i in Q, and P_{prefix} also begins with p_i. Store results to global memory.

$$\begin{cases} \text{PrefixMass}(p_i, p_j) = \text{PrefixMass}(p_i, p_{j-1}) + \text{mass}(p_j) \\ \text{PrefixScore}(p_i, p_j) = \text{PrefixScore}(p_i, p_{j-1}) + \text{MassScore}(\text{PrefixMass}(p_i, p_j)) \end{cases} \tag{13.5}$$

(2) The second kernel kernel_2 computes mass of every suffix of every peptide in Q and computes match score of peptides to spectrum S. One-dimensional block and one-dimensional thread is specified as well. Every block load $BS+31$ length of sequence from global memory to shared memory. Block_{bx} reads the sequence from bx*BS to $(bx + 1)$*$BS + 30$. Hence, a grid needs GS*$BS + 31$ length of sequence from global memory. Using Eq. 13.6, a thread computes mass and score of every suffix P_{suffix} of peptide $P = p_{j'-31} \ldots p_{j'}$ ending with amino acid $p_{j'}$, and P_{suffix} also ends with $p_{j'}$.

$$\begin{cases} \text{SuffixMass}(p_{i'}, p_{j'}) = \text{SuffixMass}(p_{i'+1}, p_{j'}) - \text{mass}(p_{i'}) \\ \text{SuffixScore}(p_{i'}, p_{j'}) = \text{SuffixScore}(p_{i'+1}, p_{j'}) + \text{MassScore}(\text{SuffixMass}(p_{i'}, p_{j'})) \end{cases} \tag{13.6}$$

Then we can get match score of a peptide to S by summing previous prefix score, PTM score and suffix score (see Eq. 13.1). Store candidates and matches to global memory.

2. DCGPUM2 Algorithm (Zhai 2010; Zhai et al. 2010a,b)

- Method in host site is similar to that of DCGPUM1.
- GPU Site:

(1) The first kernel kernel_1 computes mass and score of every prefix of every peptide in Q, which is similar to that of DCGPUM1.

(2) The second kernel kernel_2 computes mass and score of every suffix of every peptide in Q, which is similar to that of DCGPUM1. Every block loads BS+30 length of sequence from global memory to shared memory. Block_{bx} reads the sequence from bx*BS to $(bx + 1)$*$BS + 29$. Therefore, a grid needs GS*BS+30 length of sequence. Every thread computes mass and score of every suffix P_{suffix} of peptide $P = p_{j'-30} \ldots p_{j'}$ ending with amino acid $p_{j'}$ using Eq. 13.6, and P_{suffix} also ends with $p_{j'}$. Store results to global memory.

(3) The third kernel kernel_3 fills D using Eq. 13.3. D could be taken as a two-dimensional table, where top left coordinates are smaller than lower right coordinates. MS-Alignment memorizes the solutions to sub problems in D and fills it by rows like other dynamic programming algorithms. MS-Alignment also uses a "two-dimensional" table PT to memorize paths. Rows in D correspond to amino acids in a sequence while columns correspond to peaks of a spectrum. However, the data access pattern to fulfill global memory coalescing requirement is not natural. Traversing one thread per column is needed and the data accessed by threads in a half-warp need to be adjacent to one another in the horizontal direction, not vertical (Sain-Zee et al. 2008). Therefore, D is transposed in global memory, where rows correspond to peaks of a spectrum while columns correspond to amino acids in a sequence. While filling a column, to compute an element, a thread needs masses and scores of prefixes computed in kernel_1 and some top left elements in D. The element has data dependency on those in the top left of D. Thus, data dependency not only occurs within a block but also occurs among different blocks. Global barrier synchronization is needed after an element is computed. D and PT are all in global memory.

(4) The fourth kernel kernel_4 traverses D, computes match scores and gets candidates. A thread traverses in a column and computes match score of a peptide to S using Eq. 13.2. While an element is traversed, a PTM at this amino acid corresponding to the element is taken into account and masses and scores of suffixes computed in $\text{kernel}_2$are needed. Store candidates and matches to global memory.

13.3.2 Optimization Implementation of DCGPU

Hardworking optimization could give performance boost to CUDA programs and there are multiple methods for CUDA programming optimization. However, some trade off is needed in optimization.

Memory bandwidth may be bottleneck of CUDA program performance. The effective bandwidth can vary by an order of magnitude depending on access pattern for each type of memory (NVIDIA 2009).

We use multiple methods to optimize implementation of DCGPU as follows (Zhai 2010; Zhai et al. 2010a,b):

1. Coalesced global memory access

- Amino acids of a sequence are orderly stored in global memory in the type of char which accounts for one byte. Thread_{tx} access the tx-th amino acid.
- Using the type of struct to store mass and score of a prefix/suffix, and use the alignment specifier __align__(8) for the compiler to enforce the size and alignment requirements. Thus, mass and score are stored in an 8-byte word in global memory which can be read in a single instruction. Mass and score can also be stored in the type of the built-in vector type int2.

- DCGPUM2 needs to use address value which accounts for 64 bits in 64-bit computer. We use the built-in vector type uint2 to store the right 32 bits and the left 32 bits.
- If one-dimensional arrays in global memory are taken as two-dimensional tables, one thread traverses per column and threads access data adjacent to one another in the horizontal direction.

2. Using page-locked host memory

One of benefits to use page-locked host memory is that bandwidth between host memory and device memory is higher. Although the bandwidth is even higher if in addition host memory is allocated as write-combining, reading from write-combining memory is prohibitively slow (NVIDIA 2009). Therefore, we use page-locked host memory not write-combining host memory.

We use it to store peptide sequences, candidates and other data, which account for about 25 MB in total. The codes using page-locked host memory are as follows:

```
cudaMallocHost( (void **)&h_BufferChar, size_BufferChar );
cudaFreeHost( h_BufferChar );
```

The performance of using page-locked host memory in DCGPUM2 has about 32.37 % improvement. While not using it, the bandwidth from host memory to device memory is 1.63 GB/s and the bandwidth from device memory to host memory is 1.94 GB/s. However, while using it, the two bandwidths respectively increase to 2.47 and 5.11 GB/s.

But page-locked host memory is a scarce resource. In addition, by reducing the amount of physical memory available to the operating system for paging, allocating too much page-locked host memory will reduce overall system performance (NVIDIA 2009).

3. Using texture memory

Some read-only tables are searched in kernels, which texture memory is fit for. In DCGPUM2, kernels randomly access the tables of edges in D. These tables have the form of a sparse matrix and then are stored in texture in the representation of Compressed Row Storage. Masses of amino acids are also stored in tow-dimensional texture memory bound to CUDA array supporting the clamp addressing mode. The codes using texture memory are as follows:

```
// storing edges in D
// texture reference declaration
texture<BackEdge_Two,1,cudaReadModeElementType>
texRef_BackEdge_Buffer_;
// bind texture to global memory
cudaMalloc((void **)&d_BackEdge_Buffer_, size );
cudaMemcpy(d_BackEdge_Buffer_,h_BackEdge_Buffer_,size,
cudaMemcpyHostToDevice);
cudaBindTexture(0, texRef_BackEdge_Buffer_, d_BackEdge_Buffer_);
// fetch texture element
BackEdge_Two edge = tex1Dfetch(texRef_BackEdge_Buffer_, edgeIndex);
// unbind texture
```

```
cudaUnbindTexture(texRef_BackEdge_Buffer_);
cudaFree(d_BackEdge_Buffer_);
d_BackEdge_Buffer_ = NULL;
// storing masses of amino acids
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindSigned);
cudaMallocArray(&cuArray_AminoMass, &channelDesc, width, height);
cudaMemcpyToArray(cuArray_AminoMass, 0, 0, PeptideMass+64, sizeof(int) *27, cudaMemcpyHostToDevice);
texRef_AminoMass.addressMode[0] = cudaAddressModeClamp;
cudaBindTextureToArray(texRef_AminoMass, cuArray_AminoMass, channel Desc);
```

The performance of using texture memory in DCGPUM2 has about 30.94 % improvement. The disadvantages of using it are: it does not support triple types (such as float3) and user-defined struct, and the components of texture elements should equal those of return values of texture references (Zhang 2009).

4. Using shared memory and avoiding bank conflicts

Some data in global memory is shared among threads in the kernels of DCGPU. We use shared memory to load and store the data shared among threads, thus avoiding frequently global memory access. The types in shared memory account for four bytes and thread ID corresponds to access index, which can avoid bank conflicts. The performance of using texture memory in DCGPUM2 has about 4.9 % improvement. However, shared memory is a scarce resource. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch. Therefore, shared memory should be used at the key point.

5. Reducing data transfer from device memory to host memory

Optimization should minimize data transfer between host memory and device memory. DCGPUM1 transfers candidates from global memory to host memory and transferring all candidates is inadvisable. Instead, to decrease data transfer, DCGPU adopts the way like reduction which compares candidates of two threads and keeps the one having higher match score. In addition, a thread compares to the thread beyond a certain stride instead of the adjacent thread, which is for bank conflict-free in shared memory. The comparing codes are as follows:

```
if(tx<256 && PeptideScore[tx]<PeptideScore[index = tx+256])
{ … }
__syncthreads();
if(tx<128 && PeptideScore[tx]<PeptideScore[index = tx+128])
{ … }
__syncthreads();
…
if(tx<8 && PeptideScore[tx]<PeptideScore[index = tx+64])
{ … }
__syncthreads();
```

6. Using kernel launch for global barrier synchronization

The kernel_3 of DCGPUM2 has data dependency and needs global barrier synchronization. CUDA has three barrier functions: __syncthreads(), __threadfence_block() and __threadfence(). The first two can be used to coordinate communication between the threads of the same block while the third one can be used to coordinate communication between all threads of a grid. Kernel launch can also be used for global barrier. These two ways can all be used in DCGPUM2:

(1) Loop in the kernel and use __threadfence() at the end of the loop as follows:

```
__global__ void d_SeekMatch2PTM_k_FillDTable(…)
{
   for(NodeIndex=0; NodeIndex<NodeCount; NodeIndex++)
   {
   DTable[CellIndex] = final_DTable;
   __threadfence();
   }
}
```

(2) Loop outside the kernel and use kernel launch as follows:

```
   for(NodeIndex=0; NodeIndex<NodeCount; NodeIndex++)
   {
      kernel<<<GRID_SIZE, BLOCK_SIZE>>> (…);
   }
```

The performance of the latter is 8.46 % better than that of the former.

7. Reducing the use of control flow instructions

Any flow control instruction (if, switch, do, for, while) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge, that is, to follow different execution paths (NVIDIA 2009). For complete search, any sequence of not longer than 32 is computed in DCGPU. The sequences got by the threads from 1 to 31 aren't longer than 32 initially: the sequence p_1 ending with p_1 is assigned to thread_1, the sequence p_1p_2 ending with p_2 is assigned to thread_2 …and the sequence $p_1p_2 \ldots p_{30}p_{31}$ ending with p_{31} is assigned to thread_{31}. It will make threads load unbalance and needs control instruction such as if to diverge different conditions. Therefore, we add 31 asterisks in front of Q to form the sequence $p'_1p'_2 \ldots p'_{30}p'_{31}$. Then, the sequence $p'_1p'_2 \ldots p'_{30}p'_{31}p_1$ is assigned to thread_1, the sequence $p'_2p'_3 \ldots p'_{31}p_1p_2$ is assigned to $\text{thread}_2 \ldots$ and the sequence $p'_{31}p_1 \ldots p_{30}p_{31}$ is assigned to thread_{31}. Similarly, we append asterisks after Q if the final sequence is not long enough.

8. Merging kernels

DCGPUM1 could have three kernels: kernel_1 computes masses and scores of prefixes, kernel_2 computes masses and scores of suffixes and kernel_3 computes match scores and generates candidates. If so, masses and scores of suffixes computed by kernel_2 should be stored in global memory and then they are read from global memory by kernel_3. Data-reusing is bad, global memory is needed to transfer data and

kernel launch causing synchronization doesn't facilitate thread concurrent execution. Therefore, we merge the latter two kernels to one kernel $kernel_2$.

13.4 Research of MC_MS-A Algorithm

Two kinds of data can be partitioned in MS-Alignment: spectrum partition is the way of coarse-grained data parallelism while sequence partition is the way of fine-grained data parallelism. We will further accelerate MS-Alignment by spectrum partition on GPU cluster and sequence partition on GPU.

P_InsPecT parallels MS-Alignment using MPI on CPU cluster (Tu 2009). Based on it and DCGPU algorithm, we propose the MC_MS-A (MS-Alignment on GPU cluster based on MPI and CUDA) algorithm implementing MS-Alignment on GPU cluster. The algorithm description is as follows (Zhai 2010; Zhai et al. 2010b):

1. Spectra are identified independently, thus are partitioned in MPI processes to identify, which calls DCGPU to compute the step of DC. Master-slave model is adopted. The master process mainly scatters and gathers data while slave processes mainly compute cooperating with GPUs.
2. The main flow of a slave process:

(1) Preprocess data.
(2) Get an available GPU:
   (a) Split MPI_COMM_WORLD into sub communicators by host names.
   (b) Compute the number of available GPUs which are NVIDIA GPUs with not less than compute capability 1.3 (The GPUs showing compute capability 9999.9999 don't support CUDA and thus are kicked out).
   (c) Get an available GPU by sub process ID in the sub communicator. A process gets only one GPU. For the sake of simplicity, we let the process having top ID be the master process. If a process ID is larger than the total number of available GPUs, it couldn't acquire an available GPU and couldn't compute using GPU. We will refer to a process acquiring an available GPU as computing process.
(3) Computing process identifies the spectrum received from the master process. If the spectrum index received from the master process is larger than the total number of spectra, the computing process exits.
(4) Computing process sends results to the master process. Loop to (3).

3. The main flow of the master process:

(1) Preprocess data; sort spectra indices by mass in descending order.
(2) Send a spectrum index to every computing process.
(3) Receive results from a computing process and then send a spectrum index to the computing process.
(4) Loop to (3) if spectra are not computed completely, otherwise go on.

(5) Compute P-values of spectra.
(6) Output the final results and then exit.

13.5 Performance Analysis of GPU Accelerated Algorithm

13.5.1 Performance of DCGPU Algorithm

We refer to MS-Alignment algorithm on single GPU as CUDA_MS-A (MS-Alignment based on CUDA). We test CUDA_MS-A using the experimental data and statistic method in 2.2 and using the following computing environment: CPU: Intel Xeon E5410 2.33 GHz, Memory: 8 GB, GPU: NVIDIA Tesla C1060 1.44 GHz and CUDA Compiler: nvcc 2.3.

We get the speedup ratio of the step of DC in Mod = 1 on single GPU versus single CPU and the speedup ratio of the whole flow of MS-Alignment in Mod = 1 on single GPU versus single CPU (see Fig. 13.2) (Zhai 2010; Zhai et al. 2010b). Inputting different number of spectra or different scale database, the speedup ratios of Mod = 1 are different. The speedup ratio of the step of DC is a little larger than that of the whole flow. While inputting more spectra and larger database, the performance improvement is apparent. The speedup ratio of DC is above 28 and that of the whole flow is above 26, and the identifying power in Mod = 1 of a Tesla C1060 GPU can be equivalent to that of 26 2.33 GHz CPUs.

We also get the two kinds of speedup ratios in Mod = 2, which are respectively illustrated in Fig. 13.3 (Zhai 2010; Zhai et al. 2010a,b). Inputting different number of spectra or different scale database, the two speedup ratios of Mod = 2 are also different and the former speedup ratio is larger than the latter speedup ratio. The speedup ratios are more than 30 and they are above 41 while larger database is inputted. The identifying power in Mod = 2 of a Tesla C1060 GPU can be equivalent to that of 41 2.33 GHz CPUs.

The average time complexity of CUDA_MS-A is the same as that of MS-Alignment. Data parallelism is primary cause of performance improvement of CUDA_MS-A.

The computation time of DC is directly proportional to the number of spectra and the computation time of DCGPU is also directly proportional to the number of spectra. Thus, the ratio of the former time to the latter time—the speedup ratio of the step of DC on single GPU versus single CPU—isn't directly related by the number of spectra. Similarly, the speedup ratio of the whole flow on single GPU versus single CPU isn't directly related by the number of spectra.

Similarly, the two kinds of speedup ratios are not directly influenced by database size.

We predict the time of CUDA_MS-A identifying 31173 spectra (see Table 13.2) (Zhai 2010).

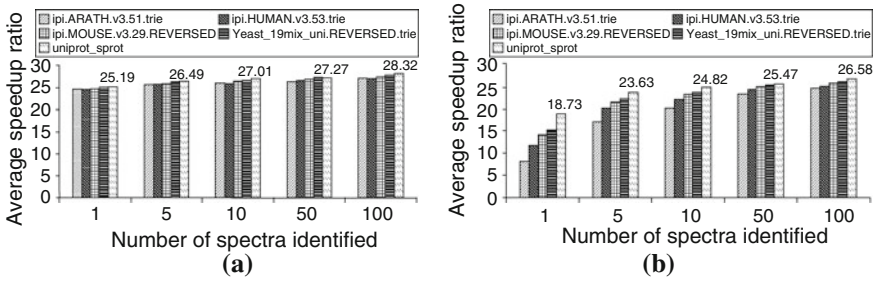


Fig. 13.2 **a** The average speedup ratio of the step of Database search and Candidate generation in Mod = 1 on single GPU versus single CPU. **b** The average speedup ratio of the whole flow of MS-Alignment in Mod = 1 on single GPU versus single CPU

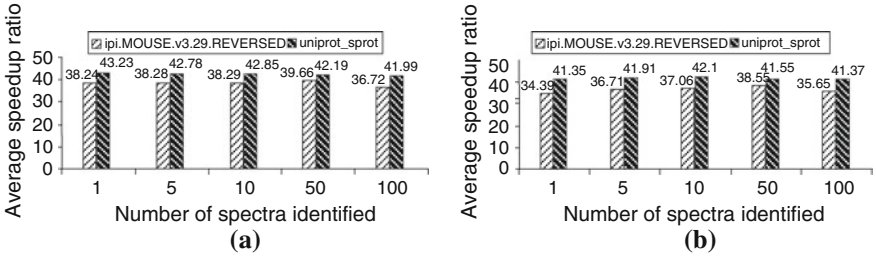


Fig. 13.3 **a** The average speedup ratio of the step of Database search and Candidate generation in Mod = 2 on single GPU versus single CPU. **b** The average speedup ratio of the whole flow of MS-Alignment in Mod = 2 on single GPU versus single CPU

Table 13.2 The predicted computation time of CUDA_MS-A identifying 31173 mass spectra

Algorithm	Protein database	Predicted time
CUDA_MS-A(Mod = 1)	ipi.MOUSE.v3.29.REVERSED	0.998 days
	uniprot_sprot	3.161 days
CUDA_MS-A(Mod = 2)	ipi.MOUSE.v3.29.REVERSED	0.810 months
	uniprot_sprot	2.726 months

13.5.2 Performance of MC_MS-A

We test the performance of MC_MS-A on the GPU Cluster of Supercomputing Center of CAS (SCCAS). The hardware collocation of computing nodes of the GPU Cluster is illustrated in Table 13.3. Every node connects with each other by 4×DDR InfiniBand and Gigabit Ethernet.

The time of MC_MS-A identifying 31173 spectra in the GPU Cluster is illustrated in Table 13.4 and the change trend by the number of GPUs is illustrated in Fig. 13.4 (Zhai 2010; Zhai et al. 2010b). It can be seen that the identification time is

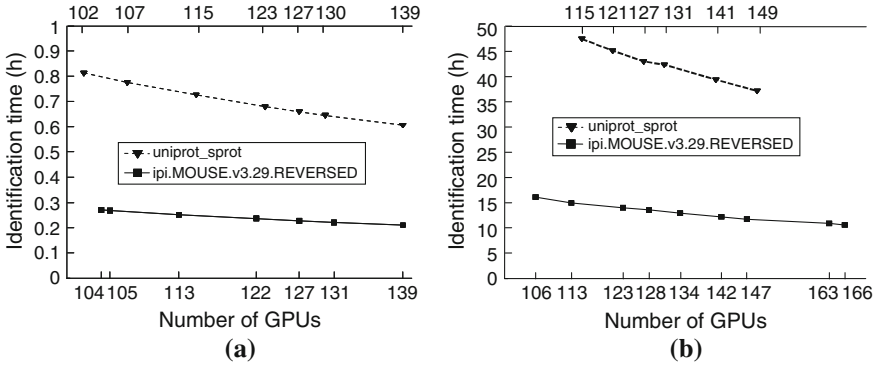


Fig. 13.4 **a** The time of MC_MS-A(Mod = 1) identifying the 31173 mass spectra on the GPU Cluster by the number of GPUs. **b** The time of MC_MS-A(Mod = 2) identifying the 31173 mass spectra on the GPU Cluster by the number of GPUs

Table 13.3 The hardware collocation of computing nodes of the GPU Cluster of SCCAS

Computing nodes	Hardware collocation	
AMD node	CPU	AMD Phenom 9850 Quad-core 2.5 GHz, L2: 4*256 KB, L3: 4 MB
(number: 18)	GPU	3 Tesla C1060 1.30 GHz
	Memory	8 GB
Intel node	CPU	Intel Xeon E5410 Quad-core 2.33 GHz, L2: 2*6 MB
(number: 72)	GPU	2 Tesla C1060 1.44 GHz
	Memory	8 GB

Table 13.4 The time of MC_MS-A identifying the 31173 spectra on the GPU Cluster

Algorithm	Protein database	GPU number	Time (h)
MC_MS-A(Mod = 1)	ipi.MOUSE.v3.29.REVERSED	139	0.210
	uniprot_sprot	139	0.606
MC_MS-A(Mod = 2)	ipi.MOUSE.v3.29.REVERSED	166	10.556
	uniprot_sprot	149	37.188

roughly inversely proportional to the number of GPUs if their computing powers are approximately equal.

The real computation time of MS-Alignment or CUDA_MS-A identifying many spectra will be very long (see Tables 13.1 and 13.2), so it's hard to test. Thus, we couldn't get the speedup ratio or efficiency in further of MC_MS-A identifying a few orders of magnitude spectra.

We indirectly analyze the efficiency of MC_MS-A identifying a few orders of magnitude spectra (Zhai 2010; Zhai et al. 2010b). Table 13.5 illustrates the time proportion of DCGPU to MC_MS-A and the time proportion of MPI communication to MC_MS-A. We compute the proportions of the time difference of the master

Table 13.5 The time proportions of MC_MS-A

Algorithm	Protein database	Prop1(%)	Prop2(%)	Prop3(%)
MC_MS-A(Mod = 1)	mouse	83.836	7.745	1.014
	sprot	91.068	2.639	0.599
MC_MS-A(Mod = 2)	mouse	98.567	0.175	0.516
	sprot	99.339	0.039	0.463

mouse ipi.MOUSE.v3.29.REVERSED, *sprot* uniprot_sprot
Prop1 the time proportion of DCGPU to MC_MS-A
Prop2 the time proportion of MPI communication to MC_MS-A
Prop3 the maximum proportion of the time difference of the master process and computing processes to the time of the master process

process and computing processes to the time of the master process, whose maximums are illustrated in Table 13.5. It can be seen that in executing time, GPU computing is in the majority while communication and waiting is in the minority, and the time difference of the master process and computing processes is small. Therefore, we conclude that MC_MS-A has good efficiency while identifying a few or even more orders of magnitude spectra.

13.6 Conclusions

We firstly analyzed MS-Alignment algorithm and concluded that it could not well meet the identification time requirement of large-scale spectra and protein database although it has an advantage of blind search. Another feature of MS-Alignment is that different spectra and peptide sequences do the same tasks, which makes data parallelism possible. We then proposed the DCGPU algorithm implementing the step of DC on single GPU. Based on P_InsPecT and DCGPU algorithm, we proposed the MC_MS-A algorithm implementing MS-Alignment on GPU cluster. Experimental results show that DCGPU algorithm can greatly accelerate MS-Alignment algorithm and MC_MS-A on GPU cluster can reduce identification time very much and can meet the speed requirement of identifying large-scale data.

We adopted several methods to optimize implementation of DCGPU. However, the algorithms based on GPU aren't optimized sufficiently. We try to further optimize them as follows. Streams could make data transfer and kernel execution concurrently and thus could keep overall time down (NVIDIA 2009). Transfer data in block and enlarge the amount of data transferred at a time. With respect to the GPU Cluster, the processes getting no GPU could get involved in computation using MS-Alignment algorithm as well as the computing processes, which will implement heterogeneous parallel computing.

MS-Alignment algorithm is a method of database search via mass spectrometry. Thus, using GPU-based HPC to accelerate MS-Alignment brings a new thought to mass data processing in Computational Proteomics based on MS/MS.

GPU is specialized for compute-intensive, highly parallel computation. CUDA's parallel programming model could make the many cores of GPU bring into full play and could maintain a low learning curve for programmers familiar with standard programming language such as C. However, CUDA has disadvantages. CUDA could only be used on NVIDIA GPUs. CUDA programs need hardworking optimizations and some factors need to be traded off in optimizations. CUDA couldn't deal with complex data dependency well, which may degrade performance.

Acknowledgments This work was supported by CAS grant KGGX1-YW-13 and Computer Network Information Center of CAS grant CNIC_ZR_09005. We are grateful to Professor Wu Jiarui of SIBS for directing our research work and to PhD Sheng Quanhu for providing mass spectra and protein databases. This research was supported in part by the National High Technology Research and Development Program of China 2006AA01A116 and Major Research Equipment Development Project of Ministry of Finance ZDYZ2008-2. The protein databases of ipi.ARATH.v3.51, ipi.HUMAN.v3.53 and uniprot_sprot were downloaded at EMBL-EBI website.

References

Ari MF (2008) Algorithms for tandem mass spectrometry-based proteomics. Ph.D. thesis, University of California, San Diego

David K, Hwu W-M (2010) ECE 498AL: Applied Parallel Programming. http://courses.ece.illinois.edu/ece498/al/

Dekel T, Stephen T, Ebrahim Z et al (2005) Identification of post-translational modifications via blind search of mass-spectra. Nat Biotechnol 23:1562–1567

John DO, David L, Naga G et al (2007) A survey of general-purpose computation on graphics hardware. Comput Graph Forum 26:80–113

Liu Y, Douglas LM, Bertil S (2009) CUDASW++: optimizing Smith-Waterman sequence database searched for CUDA-enabled graphics processing units. BMC Res Notes 2:73

Lukasz L, Witold R (2009) An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: 2009 IEEE international symposium on parallel and distributed processing, pp 1–8

Michael CS, Cole T, Arthur LD et al (2007) High-throughput sequence alignment using graphics processing units. BMC Bioinformatics 8:474

NVIDIA Corporation (2009) NVIDIA CUDA Programming Guide Version 2.3.1. http://www.nvidia.cn/object/cuda_get_cn.html

Sain-Zee U, Melvin L, Sara SB et al (2008) CUDA-Lite: reducing GPU programming complexity. In: Languages and compilers for parallel computing: 21th international workshop (LCPC), 2008, pp 1–15

Seungjin N, Jaeho J, Heejin P et al (2008) Unrestrictive identification of multiple post-translational modifications from tandem mass spectrometry using an error-tolerant algorithm based on an extended sequence tag approach. Mol Cell Proteomics 7:2452–2463

Sheng Q, Xie T, Ding D (2000) De novo interpretation of MS/MS spectra and protein identification via database searching (in Chinese). Acta Biochim Biophys Sin 32:595–600

Stephen T, Shu H, Ari F et al (2005) Inspect: fast and accurate identification of post-translationally modified peptides from tandem mass spectra. Anal Chem 77:4626–4639

Svetlin AM, Giorgio V (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinform 9:S10

Tu Q (2009) Research on parallelization and optimization of protein post-translational modifications software InsPecT (in Chinese). Master thesis, Computer Network Information Center of Chinese Academy of Sciences

Wu E (2004) State of the art and future challenge on general purpose computation by graphics processing unit (in Chinese). J Softw 15:1493–1504

Zhai Y (2010) Research and implementation of using GPU to accelerate MS-alignment for identification of post-translational modifications (in Chinese). Master Thesis, Computer Network Information Center of Chinese Academy of Sciences

Zhai Y, Tu Q, Lang X et al (2010a) Research of CUDA-based acceleration of MS-alignment for identification of post-translational modifications (in Chinese). Appl Res Comput 27:3409–3414

Zhai Y, Tu Q, Lang X et al (2010b) Research of using GPU to accelerate MS-alignment algorithm to identify protein post-translational modifications (in Chinese). J Comput Res Dev (in review)

Zhang S, Yanli C (2009) CUDA of GPU high performance computing (in Chinese). China Water-Power Press, Beijing

Part V
Chemical Physical Applications

Chapter 14
GPU Tuning for First-Principle Electronic Structure Simulations

Yue Wu, Weile Jia, Lin-Wang Wang, Weiguo Gao, Long Wang and Xuebin Chi

Abstract With increasing demands on hardware in quantum chemistry calculations, modern Graphical Processing Units (GPUs) have great potential meeting the resources of high performance computing. In this paper we investigate the possibility to accelerate the planewave pseudopotential code PEtot on CUDA architecture. In particular, we execute two most time consuming steps, i.e., the nonlocal projections and FFT transformations on GPU with careful implementations to reduce the data exchanges between the CPU and the GPU. Our experience for the molecule with as many as 512 atoms is also shown.

14.1 Introduction

Ab initio electronic structure calculations based on density functional theory (DFT) (Kohn and Sham 1965) has become the work horst for many material science simulations. Recently, it has also become the major tool in quantum chemistry calculations. In large computer centers where the statistics data are available, the DFT

This work is supported in part by NSF of China (11071047), Science and Technology Commission of Shanghai Municipality (09ZR1401900).

Y. Wu · W. Gao
School of Mathematical Science, MOE Key Laboratory for Computational Physical Sciences, Fudan University, Shanghai 200433, China

W. Jia (✉) · L. Wang · X. Chi
Supercomputing Center, Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China
e-mail: jiawl@sccas.cn

L.-W. Wang
Lawrence Berkeley National Laboratory, One Cyclontron Road, Berkeley, CA 94720, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 235
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_14,

based calculations accounts for 75 % of all the computer times allocated to material science simulations (Wang 2006). While, there are many different numerical methods to carry out the DFT calculations, including the atomic basis set method, the Muffin-tin method, real space grid method, and the plane wave basis set method, the most mature and widely used method is the plane wave (PW) method. The PW method expands the single particle electron wave functions in PW basis set, with a kinetic energy cut off E_{cut} to select the PW basis functions. It uses pseudopotentials (Kleinman and Bylander 1982) to soft the potential and the wave functions, hence the need of the number of basis functions. Compared to other methods, it has the advantages of having a single parameters E_{cut} to control the accuracy of the calculation, and the variational principle of the total energy with regard to the PW coefficients. The modern PW-DFT codes use iterative methods to diagonalize the single particle Hamiltonian H by repeated applications of $H\psi_i$ which can be written as:

$$H\psi_i = \Big\{ -\frac{1}{2}\nabla^2 + V(r) + \sum_{R,l} |\phi_{R,l}\rangle\langle\phi_{R,l}| \Big\}\psi_i. \tag{14.1}$$

Here ψ_i is the wave function, the $\{\phi_{R,l}\}$ are the nonlocal potential projection operators which for each atom R and angular moment l, the potential $V(r)$ consists of the Hartree potential, the local pseudopotential and exchange-correlation potential (Martin 2004)

$$V(r) = V_{Hartree}(r) + V_{loc}(r) + V_{xc}(r).$$

Taking the advantage of the fact that the kinetic energy operator is diagonal in reciprocal space (the planewave coefficient space) while the potential $V(r)$ is diagonal in real space, the codes use inverse FFT to transform the wave function from the reciprocal space to real space, multiply the potential, then a forward FFT to transform the wave function back to reciprocal space to carry out $H\psi$, see Fig. 14.1.

One of the challenges of PW-DFT calculation is to speed up the simulation for a given system. This is particularly true for *ab initio* molecular dynamics (MD) simulations, where hundreds of thousands of steps are needed to simulate the dynamics of the system for hundreds of picoseconds. The current state of the art PW-DFT (e.g., by the VASP code) can typically reach 1 min per 1–2 fs MD step. Due to the stagnation of the CPU clock speed, this is reached by parallelization in three levels: the parallelization in planewave coefficients (also to be called parallelization in reciprocal Q-space) with N_Q processors; parallelization in the wave function $\{\psi_i\}$ index i with N_b processor groups; the parallelization in the k-point with N_k processor groups. Thus the total number of processors are $N_p = N_Q N_b N_k$. However, for

Fig. 14.1 Application of single particle Hamiltonian to the wave functions

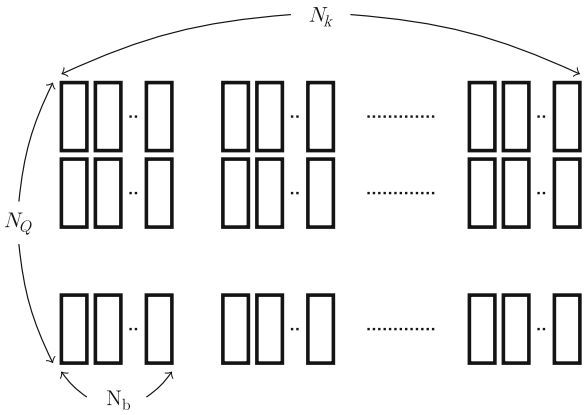


Fig. 14.2 Three-level parallelization of the wavefunctions

systems with a few hundred to a thousand atoms, numerical tests for the parallel codes implemented under MPU indicate that the maximum number of processor N is about a thousand, and it fails to scale beyond that. This set the above limit of 1 min per MD step. It makes *ab initio* MD simulation beyond a few ps extremely difficult.

In recently years, the merge of graphic processing unit (GPU) for scientific computing has attracted great attention in the scientific community. GPU can offer hundred times speed up compared with CPU while maintaining low electricity consumption (Volkov and Demmel 2008). The GPU has been proved successful for many material science simulations (Ufimtsev and Martinez 2008), including the classical MD, the quantum Monte Carlo simulations. However, its applicability to the PW-DFT calculation is still under intensive study. The important issues are the overall strategy of the GPU acceleration, e.g., which parts can be calculated by GPU, and which parts to be kept in CPU calculation; the data structure and memory manipulation, which part to be stored in CPU memory, and which part to be transferred into the GPU global memory. In this paper, we will take the PEtot code (http://hpcrd.lbl.gov/~linwang/PEtot/PEtot.html), which is a planewave pseudopotential code, and test different GPU strategies to accelerate the calculation. The PEtot code has the aforementioned three level parallelizations, it has both the norm conserving pseudopotential and ultrasoft pseudopotential, and it can move the atoms for structure optimization. In the current paper, we will use the norm conserving pseudopotential, and focus on the wave function iteration subroutine which uses the all-band (block) conjugate gradient algorithm to improve the accuracy of the wave functions $\{\psi_i\}$ for a given Hamiltonian H. The wave function ψ_i is kept in reciprocal space as the coefficients $\{C_i(G)\}$:

$$\psi_i(r) = \sum_G C_i(G) \exp(-iG \cdot r) \tag{14.2}$$

Table 14.1 Pure floating point computation time of `zgemm` and `cublas_zgemm`

	Computational time
`zgemm`	37s
`cublas_zgemm`	4.3s

The projection operator $\{\phi_{R,l}\}$ can be represented in real space r or reciprocal space G. In the current test, we keep them in reciprocal space (Fig. 14.2).

There are two major steps which consume a lot of time. The first is the nonlocal projection to $\{\phi_{R,l}\}$. For large system, this is a $O(N^3)$ operation. Since we are using block version of the conjugate gradient, the Eq. 14.1 is carried out for many wave functions $\{\psi_i\}$ simultaneously. As a result, the nonlocal potential projection calculation is a matrix to matrix multiplication, which can be calculated using BLAS-3 library. One of our test in this paper is to used BLAS-3 library of the GPU to carry out this calculation. The second step is the FFT. In a G-parallelized version ($N_G > 1$), the wave function coefficients $C_i(G)$ are distributed among different CPUs and nodes. During the FFT, the data has to be transferred among those CPUs. It will be extremely difficult to speed up such FFT using GPU with fragmented data pieces to be communicated with MPI among different CPU, then to be transferred to GPU. In the current paper, we will test the calculation with $N_G = 1$, but $N_b \gg 1$. Doing this, we can do all a single 3D FFT within one GPU, and take the advantage of the GPU FFT library. The issue here is how to transfer the data into the GPU global memory, and match the data structure of the $C_i(G)$, which is within a sphere, to the GPU FFT data structure which is a full 3D grid (Table 14.1).

The rest of this paper is organized as follows. In Sect. 14.2, we avoid redundant data communication by placing the nonlocal projector into the GPU in the entire diagonalization step. Then in Sect. 14.3, we introduce the strategy of packing the big box data into small box which substantially reduces the data exchange between the CPU and the GPU. We show the performance test in Sect. 14.4 and make conclusion remarks in Sect. 14.5.

14.2 Nonlocal Projector Implementation

14.2.1 Using GPU BLAS-3

The level-3 BLAS computation is an extremely time consuming part in the nonlocal projector implementation. Especially, the routine `zgemm` (matrix-matrix multiplication on complex data) is called frequently. More specifically, the `zgemm` called by the routine computing $H\psi_i$ takes nearly 50 % of the total computating time. Driven by the motivation to accelerate the computing speed of this part, we replaced the original level-3 blas routines with GPU implementations (Table 14.2).

Table 14.2 Total computation time using different mx

Total computational time	$mx = 20$ (s)	$mx = 200$ (s)	$mx = 500$ (s)
CPU BLAS	18.12	77.9	224.5
PEtot CUBLAS	12.8	34.2	74.2

Table 14.3 FFT Performance of grid size 128*128*128

	Time (s)	Percentage of total (%)
CPU mapping	0.05006	57.094
Memcpy (CPU to GPU)	0.006431	7.335
Expansion	0.004605	5.252
Inverse FFT	0.008892	10.142
Multiply with $V(r)$	0.001075	1.226
Forward FFT	0.008631	9.844
Fetch small cube	0.001484	1.693
Memcpy (GPU to CPU)	0.006501	7.415
Total	0.087679	

The Compute Unified Device Architecture (CUDA) (http://developer.nvidia.com/object/cuda.html) has huge computation power and can be highly efficient in performing data-parallel tasks. CUBLAS is a BLAS library on CUDA architecture. Reports from NVIDIA illustrate that CUBLAS has significant floating point arithmetic speed up over the general BLAS library. Thus we tried using the power of CUBLAS in the PEtot codes and got quite a bit speed up (Table 14.3).

14.2.2 Placing the Projection Wave Functions Inside the GPU

What all the programmer should take into consideration when developing GPU programs is to avoid the memory communication between CPU and GPU as much as possible, since this step is time consuming and will definitely do harm to the total computing efficiency.

The standard flow of calling a CUBLAS routine contains the following five steps:

1. Alloc GPU memory.
2. Copy the data stored in the CPU memory to the GPU memory.
3. Call the BLAS routine to do the computation.
4. Copy the data stored in the GPU memory back to the CPU memory.
5. Free the GPU memory.

The memory communication consuming part is Steps 2 and 4. If we replace all the CPU BLAS routines with the above standard flow, too much unnecessary copy will take place. There are two strategies we adopt in practice. Firstly, we avoid

unnecessary copy in Steps 2 and 4. Before calling a `cublas_zegemm` routine, we only initialize the data of the multiplier matrix, and after computation, only the result matrix is transferred to the CPU memory. Secondly, the data copy in Step 2 can be further reduced if a matrix is frequently used as a multiplier but there is no change to be done on it. Thus we just do initialization on this kind of matrix once, then it is used to compute matrix-matrix multiplication from begin to end. In our case, after we initialize the projection wave function in the CPU at the very beginning, we do one GPU alloc and set matrix to GPU memory, then we place the projection wave function inside the GPU all the time and it doesn't affect the computation of `cublas_zgemm` at all. In this way we save much computing time.

14.3 FFT Implementation

14.3.1 Spherical FFT

In the plane wave pseudopotential calculation, we carry out different operations in reciprocal space and real space. This dual-space representation requires us to do frequent FFT between them to construct the charge density and transform the potential terms. It is due to the FFT algorithm, the plane wave calculation becomes attractive. Generally the ratio of calculations to communications in FFT is the order of $\log N$ where N is the grid dimension, which makes it critical to PW-DFT code. A scalable FFT is implemented for the PW-DFT code (Canning 2008).

The PEtot FFT is different from the standard FFT in that a sphere of points instead of a standard grid is used in the reciprocal space. It is consisted of three sets of 1d FFT in the x, y, z direction with two transposition of data between each 1d FFT. In order to save communication, PEtot FFT takes advantage of the fact that diameter of sphere is usually half the size of the real space grid, and performs 1d FFTs only on non-zero part of the sphere data. This greatly reduces the amount of computation and data communication. In the GPU FFT implementation, we have derived this methodology and made further effort to make FFT run more efficiently on GPU.

14.3.2 Sphere to Small Box

Excellent accelerations are achievable for computational intensive tasks and bandwidth intensive tasks with data locality on CUDA device. However, the complex memory hierarchy places many difficulties for communication intensive tasks in using many-core parallel systems. For communication intensive tasks such as FFT, CUDA acceleration is hard because the bottleneck lies in the PCI-express between main memory and GPU device memory. CUFFT is an Nvidia FFT library on CUDA architecture. According to our test, 20 times speedup can be achieved compared to

FFTW without counting the CPU and GPU communication time. However, if we take CPU and GPU communication time into account, the speedup is quite limited. Our test on Tesla C1060 shows that more than 60 % of total time is consumed at data transfer between CPU and GPU (see Fig. 14.3). For double precision FFT, CUFFT has less than 2 times of speedup compared with FFTW.

The PEtot FFT is a special FFT. Its data in the reciprocal space is a sphere and expand into a big box of data in the real space. Then the real space data will multiply with potential $V(r)$ to construct the charge density and transform from real space back to reciprocal space. By moving charge density construction and forward FFT into the CUDA device, computation inside GPU was increased over data communication between CPU and GPU. The grid size in the real space is usually two times of the diameter of the sphere in the reciprocal space, so we take advantage of this fact by sending non-zero data into the GPU global memory. However, a sphere data is hard for inside GPU expansion. Thus a small box of data is used on the CPU. The original sphere data in reciprocal space is periodic and the center is (1, 1, 1) in the CPU. We need to map the data into small box like Fig. 14.4.

After 3d inverse FFT, multiply with potential $V(r)$ and forward FFT, we need to map the data back from small box of data into big box of data. Potential $V(r)$ is not changed during all FFT calculations, thus we only need to transfer it from CPU to GPU before the first FFT.

14.3.3 GPU FFT Implementation

For large scale FFT on GPU cluster, a substantial speedup can be achieved with sustained high GPU bandwidth, processing larger subtasks and matrix transposition during data transfer (Chen et al. 2010; Maimaitijiang et al. 2009). PEtot FFT scale, which is usually like $128 \times 128 \times 128$, is rather small for multiple-GPU solution. Deep memory hierarchy and data transfer limit of PCI-express makes FFT hard to speed up on CUDA device. In the PEtot FFT GPU implementation, two factors contribute to better performance: firstly data transfer of only the non-zero data reduced the data

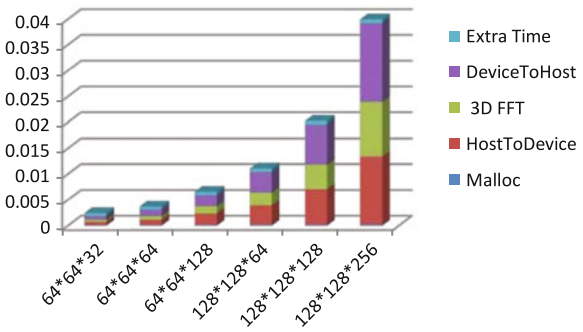


Fig. 14.3 Ratio of CUFFT computation time and communication time

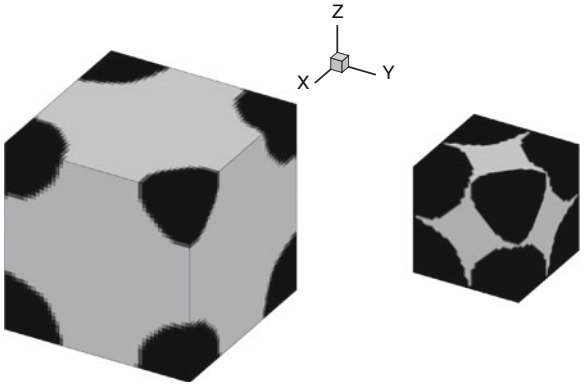


Fig. 14.4 Mapping from big box of data to small box of data

transfer time between CPU and GPU; secondly moving more computation into the GPU device for one memory copy between CPU and GPU.

The details of each step in our GPU PEtot FFT are (see Fig. 14.5):

1. The CPU processor pad out the ends of each of the z-column of g vector for form a small box, which will contain all the non-zero coefficients of the reciprocal space data.
2. The small box of data is transferred from main memory to GPU global memory.
3. Allocate space for a large box of data and initialize big box data all zeros. Expand the small box of data into each corner of the big box to make it exactly like PEtot FFT.
4. Perform a 3d inverse FFT on the big box of data using double precision CUFFT to transform from reciprocal space to real space.
5. Construct the charge density by multiplying the $V(r)$ data with each point of the big box. The $V(r)$ data was transferred into the GPU global memory only once because $V(r)$ is read-only data.
6. Perform a 3d forward FFT on the big box of data with double precision CUFFT to transform from real space to reciprocal space.
7. Fetch the data from each corner of the big box to build a small box of data.
8. Copy the small box of data from GPU global memory to main memory.
9. The CPU processor maps the small box of data to the sphere data.

In the total 9 steps of the GPU PEtot FFT, the expansion and mapping back between real space data and reciprocal space data are limited only on the effective data. The first and the last steps are performed on the CPU. These steps are avoidable in the future if we use small box of data instead of sphere data on the CPU.

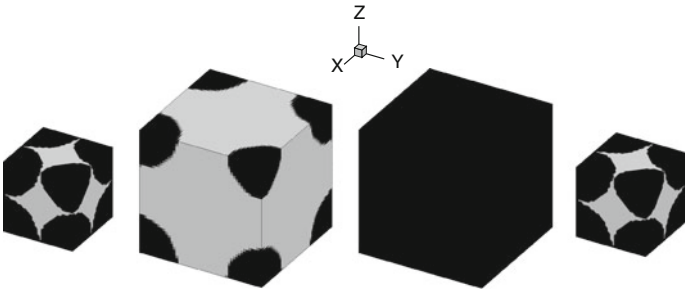


Fig. 14.5 Data mapping in the GPU PEtot FFT

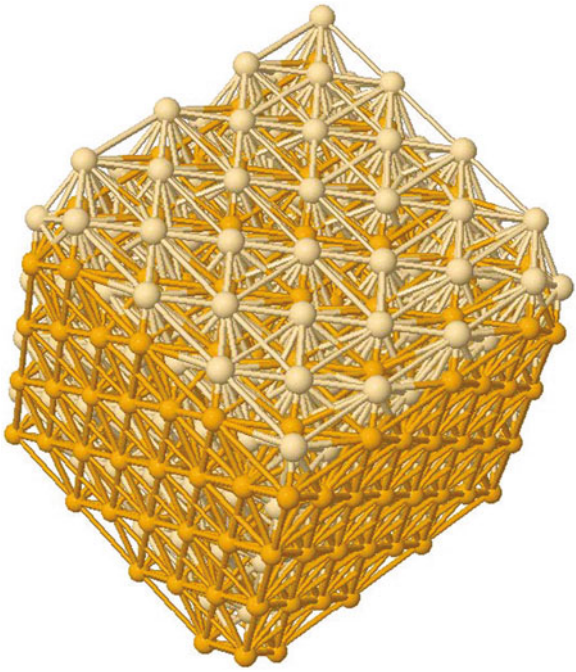


Fig. 14.6 512-atom CdSe quantum dot

14.4 Performance Test

To evaluate the accelerated PEtot, we ran the PEtot code on the Tesla GPU cluster at Supercomputing Center of Chinese Academy of Sciences (SCCAS). Each node is equipped with one Intel Xeon E5410, 8 Gigabyte of main memory and two C1060 Tesla GPU card. Firstly we tested the performance of PEtot calling CUBLAS. We ran a CdSe system of 512 atoms in the diamond structure (Fig. 14.6) on 8 nodes using 16 CPUs and 16 GPUs. In this case we set $N_Q = 16$, $N_b = 1$, $N_k = 1$.

Firstly, we tested pure floating point computational time of `zgemm` and `cublas_zgemm`, i.e. regardless of the time spent on transferring data between GPU and CPU. In this test case we were to multiply a 3072×500 matrix with a 500×28685 matrix in the $H\psi_i$ function.

From the above we can conclude that compared with CPU, GPU has extremely significant speed up in floating point arithmetic.

In the next stage, we tested the overall performance improvement using `cublas_zgemm`. We embedded the routine into the PEtot codes according to the strategies described in the previous section. Except for the `zgemm` in $H\psi_i$, we also replace the `zgemm` in other routines such as `dot_product` which is used to compute the dot-product of the wave function. Here we list the total computation time. We tested three cases with $mx = 20, mx = 200$, and $mx = 500$. Here the parameter mx decides the number of states to be calculated. We ran the system on 32 nodes using 64 CPUs and 64 GPUs.

It is showed that for large mx we can save two-thirds of the total computing time. We also find that the speed up ratio grows with the dimension of the problem. This may because that considering the trade-off between accelerating floating point arithmetic and consuming time in communication between nodes, dealing with large scale of data use GPU gets to better performance. It sounds good because in the field of HPC the ability to handle large scale of data is one of the main issues.

Then we tested the performance of PEtot FFT. We set $N_Q = 1$, $N_b = 16$, $N_k = 1$. The test case is the same 512 atoms system. In this test case, $128 \times 128 \times 128$ size FFT is used. The GPU PEtot FFT has 4 times of overall speedup compared with original PEtot FFT. It has 6.3 times of speedup compared with FFTW standard grid FFT.

In the PEtot GPU FFT, it takes 57 % of total time to map between sphere data and small box of data on CPU. In the future, the CPU mapping part can be avoided by using small box of data, which means that 8 times of speedup can be achieved compared with PEtot FFT.

14.5 Conclusion Remarks and Future Work

With different settings, we showed that both nonlocal projection and FFT transformation can be speeded up by a factor about 4. However, in the setting of $N_Q = 1$, each node keeps the whole nonlocal projection operators $\{\phi_{R,l}\}$ which leads to high memory demand. There is also a lot of communication when forming the Gram matrix in the diagonalization step in this case. We will address these issues in the future.

Finally, there is an issue we have not touched in the current paper, but will be tackled later is the orthogonalization, and diagonalization among $\{\psi_i\}$ and the projection of the conjugate gradient search direction $P_i = H\psi_i - \langle\psi_i|H|\psi_i\rangle\psi_i$ from $\{\psi_i\}$. All these require the cross product operation like $P_i|\psi_j\rangle$, which is particularly challenging for a $N_b \gg 1$ parallelization, because the wave functions of different

i are resided in different CPU processors and nodes. Calculation of this part might not be able to use GPU, but careful CPU implementations of such operations are necessary, and algorithms which minimize the use of such steps will be beneficial.

A. CUDA Source Code for CUBLAS

```
c     Etotalcalc.f
      ......................
      call cublas_alloc(nref_tot*mg_nx, 16, cu_wqmask)
      call cublas_set_matrix(mg_nx, nref_tot, 16,
     &  wqmask, mg_nx, cu_wqmask, mg_nx)
      ......................
      call CG_AllBand(...,cu_wqmask)
      ......................
      call cublas_free(cu_wqmask)
      ......................

c     CG_Allband.f
      ......................
      do 3000 nint2=1,nline
         ..................
         call Hpsi_comp_AllBandBP(pg_m,pgh_m,
     &           nblock_band_mx,ilocal,vr,workr_n,kpt,1,
     &           spg_m,sumdum_m,iislda,cu_wqmask)
         ..................
3000  continue
      ......................

c     Hpsi_comp_AllBandBP.f
      ......................
      call cublas_alloc(nblock*mg_nx, 16, cu_wg)
      call cublas_alloc(nblock*nref_tot, 16, cu_sumy_m)
      call cublas_set_matrix (mg_nx, nblock, 16, wg,
     &     mg_nx, cu_wg, mg_nx)
      call cublas_zgemm('c', 'n', nref_tot, nblock, ng_n, cc1, cu_wqmask,
     &     mg_nx, cu_wg, mg_nx, cc0, cu_sumy_m, nref_tot)
      call cublas_get_matrix (nref_tot, nblock, 16, cu_sumy_m,
     &     nref_tot, sumy_m, nref_tot)
      call cublas_free(cu_wg)
      call cublas_free(cu_sumy_m)
      ......................
```

B. CUDA Source Code of Data Mapping for CUFFT

```
__global__ void fetchCube(cufftDoubleComplex* odata,cufftDoubleComplex* idata,
                          int diff, int  half, int longedge, int shortedge)
```

```
{
    /*  fetch the small cube from the big box of data. */
    __shared__ cufftDoubleComplex tile[LEN];

    int index_out = blockIdx.x * shortedge + threadIdx.x;
    int xNew = threadIdx.x + ( threadIdx.x / half ) * diff;
    int yNew = blockIdx.x + ( blockIdx.x / half ) * diff;
    int index_in = xNew + yNew * longedge;

    tile[threadIdx.x] = idata[index_in];
    __syncthreads();

    odata[index_out] = tile[threadIdx.x];
}
```

References

Canning A (2008) Scalable parallel 3d FFTs for electronic structure codes. In: Palma J, Amestoy P, Daydé M, Mattoso M, Lopes J (eds) High performance computing for computational science—VECPAR 2008, vol 5336. Springer, Berlin, pp 280–286

Chen Y, Cui X, Mei H (2010) Large-scale FFT on GPU clusters. In: Proceedings of the 23rd international conference on supercomputing

Kleinman L, Bylander DM (1982) Efficacious form for model pseudopotentials. Phys Rev Lett 48(20):1425–1428

Kohn W, Sham LJ (1965) Self-consistent equations including exchange and correlation effects. Phys Rev 140(4):1133–1137

Maimaitijiang Y, Wee HC, Roula A, Watson S, Patz R, Williams RJ (2009) Evaluation of parallel FFT implementations on GPU and multi-core PCs for magnetic induction tomography. In: IFMBE Proceedings, vol 25/4. Springer, Berlin, pp 1889–1892

Martin RM (2004) Electronic structure: basic theory and practical methods. Cambridge University Press, Cambridge

Ufimtsev IS, Martinez TJ (2008) Graphical processing units for quantum chemistry. Comput Sci Eng 10(6):26–34

Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing IEEE Press. Piscataway, pp 1–11

Wang L-W (2006) A survey of codes and algorithms used in nersc material science allocations. LBNL Report 61051, Lawrence Berkeley National Laboratory

Chapter 15
Nucleation and Reaction of Dislocations in Some Metals and Intermetallic Compound TiAl

D. S. Xu, H. Wang and R. Yang

Abstract The shear deformation in selected metals and intermetallic compound TiAl under different conditions was investigated using molecular dynamics (MD) simulation with many-body interatomic potentials. The atomic-scale details of the dislocation nucleation were simulated with the GPU implementation of our Para MD program. For the homogeneous nucleation in perfect lattice, as the lattice strain increases, strain localization occurs during which the strain condenses gradually on a few lattice planes where the nucleation is finally achieved. The dislocations on different slip planes in the same slip system can react with each other if the slip planes are only a few interplanar-spacings apart, forming a variety of defects including vacancies, interstitial atoms and their clusters, small dislocation loops, etc., depending on the distance between the slip planes and the characteristics of the reacting dislocations. It was shown that the C1060 GPU has a substantial acceleration on MD simulation compared with conventional CPU, indicating therefore a method to reduce the total cost of MD simulations; however, for ultra-large atomic systems, the relatively small memory capacity on C1060 hinders further increase of the simulation size beyond a few million atoms, therefore expansion of the GPU memory, which meanwhile reduces the communication burden given the same simulation size, rather than the number of cores is more important for such simulations.

15.1 Introduction

Advanced technology needs better materials, for example, designing of jet engine with higher efficiency needs materials working at higher temperature and with good reliability and long life. TiAl intermetallic alloys are promising materials for the

D. S. Xu (✉) · H. Wang · R. Yang
Institute of Metal Research, Chinese Academy of Sciences,
72 Wenhua Road, Shenyang 110016, China
e-mail: dsxu@imr.ac.cn

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 247
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_15,

turbine blades in jet engine, due to its high strength to weight ratio and retention of strength and modulus at high temperature. However, in the last few decades, the room temperature brittleness hinders their applications (Clemens and Kestler 2000; Xu et al. 2008). It is believed that the ductility and plastic deformation of metallic materials are mainly governed by dislocation nucleation, movement and reactions among themselves and with other obstacles. In the last 60 years, large amount of work has been done to investigate the dislocation behavior with the help of surface trace analysis and transmission electron microscopy (Nabarro 1997; Xu et al. 2009). However, due to the limitation of temporal and spatial resolution of the various experimental methods, the atomic details as to how the dislocations were nucleated and how the dislocations reacted when they are in close contact are still lacking.

In addition to the mobility of single dislocations, the interaction between dislocations is believed to have strong effect on the deformation behavior and therefore the mechanical properties. Among the various dislocation structures, the dipolar configuration is one of the most important classes and needs atomic detail simulation, since the strain field around a dislocation was largely canceled by the dislocation with opposite sign in a dipole. TEM observation is not appropriate for this problem because both the contrast and the spatial resolution is not high enough; in effect, the dislocation dipole can not be discerned as the height of the dipole reduces to a few nanometers. Further more, due to the complication of the strain field around the core of dislocation dipole, the detailed core structure can not be revealed due to the image shift around the core of dipole (Veyssiere 2006). In terms of dislocation mechanism it is not well understood why the compound TiAl formed from two ductile metals Ti and Al shows very strong room temperature brittleness.

In the last 10 years, MD simulation has been employed as an effective means of understanding the atomic behavior of the dislocation in a controlled deformation mode (Bulatov and Cai 2006). Due to the atomic resolution and the small time steps of the simulation, the length and time scales for the MD simulation are limited, and large-system simulation for a longer time is always desired by investigators, but needs both computer power and time. As the simulation box increases in size, large scale parallel method has to be used, and this will incur the cost of a large number of CPU cores and associated power consumption, to the effect that simulation of a short process may even cost more than a TEM session of experimental observation. Acceleration of the MD simulation process using a GPU is one of the viable options to reduce computing costs.

In this paper, different fcc metals such as Al and Cu, as well as TiAl, an ordered intermetallic compound based on the fcc structure, are investigated and the nucleation and reaction behaviors of dislocations are compared to gain fundamental understanding of the dislocation movement and defect formation by dislocation reaction during deformation.

15.2 GPU Implementation and Deformation Simulation

15.2.1 GPU Implementation of ParaMD

In order to exploit the computing power provided by the new type GPU, the Parallel Molecular Dynamics program, ParaMD, developed in the authors' group was modified so that it can be used for large-scale and relatively longer time atomic simulations of the deformation behavior of metallic materials. ParaMD is operative on both CPU and GPU, with the main program structure shown in Fig 15.1.

GPU implementation is realized with the cutting-edge feature provided by the PGI compiler to translate the CPU-oriented Fortran codes into GPU programs through direct use of the acceleration directives. A segment of the code from the potential subroutine is shown in Fig. 15.2, exemplifying the use of the directives.

15.2.2 Evaluation of ParaMD with GPU Support

The performance of ParaMD on GPUs is evaluated and compared with three other popular MD codes, namely Amber, NAMD and LAMMPS. ParaMD tests are launched on Opteron 2431 CPUs with six cores each. The machines are interconnected by a 20G InfiniBand switch. The condition for ParaMD is specifically 20,000 atoms in each core and 800 atoms across the boundary under spatial decomposition; for the other MD codes cited here the readers are referred to the corresponding webpages. As shown in Fig. 15.3a, 4 C1060 GPUs run at a satisfactory acceleration ratio of 3.72, very close to that of 16 Opteron2431 CPUs. Such a conclusion is comparable to that from NAMD (Fig. 15.3c), while LAMMPS does generate better result (Fig. 15.3d). In the present case, the data exchange between CPU and GPU in the same machine is still frequent, hence slowing down the calculations.

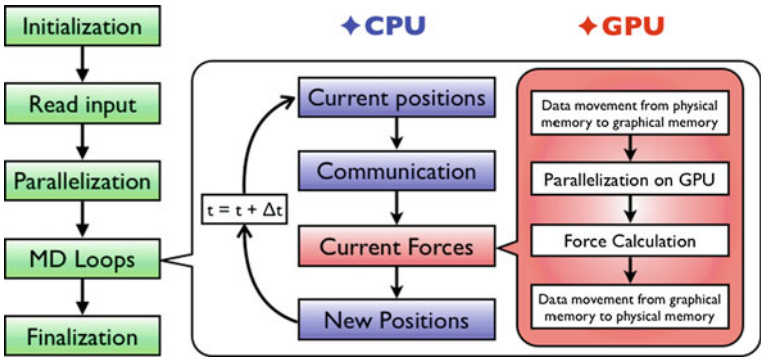


Fig. 15.1 Schematic of the main flow chart of ParaMD

```
CALL MPI_BARRIER(MYCOMM, ierr)
CALL SYNC_AF()
CALL MPI_BARRIER(MYCOMM, ierr)


!$acc region &
!$acc copyin(af(1:Local_To_Global(-1,MYID))) &
!$acc copyout(fxd(1:Local_To_Global(0,MYID)),&
!$acc         fyd(1:Local_To_Global(0,MYID)),&
!$acc         fzd(1:Local_To_Global(0,MYID)),&
!$acc         epd(1:Local_To_Global(0,MYID)), &
!$acc         s11,s12,s13,s21,s22,s23,s31,s32,s33) &
!$acc local(il,ig,j,jg,jl,sxij,syij,szij,rxij,ryij,rzij,&
!$acc       r2ij,rij,uxij,uyij,uzij,fxij,fyij,fzij,vij, &
!$acc       temp0,temp1,temp2,ddtemp)

! Zero the stress terms
s11 = 0.0
s12 = 0.0
s13 = 0.0
s21 = 0.0
s22 = 0.0
s23 = 0.0
s31 = 0.0
s32 = 0.0
s33 = 0.0
            •
            •
            •
            •
            •

   !$acc end region

!$acc end data region
```

Fig. 15.2 Part of the potential subroutine with the acceleration directives

15.2.3 MD Simulation Setup

Molecular dynamics simulations are carried out for the constant strain-rate loading and high temperature annealing situation using EAM type interatomic potentials for Al (Mishin et al. 1999), Cu (Mishin et al. 2001) and TiAl (Mishin et al. 2001; Zope and Mishin 2003). Simulation cells containing up to 1.5 million atoms were employed, with periodic boundary condition in all three directions. Shear deformation was imposed along ⟨110⟩ direction on {111} planes for both perfect and dislocation containing crystals. The simulations were carried out at different temperatures, from 100 K to close to the melting point of each material. The shear strain rate ranged from 10^6 to 5×10^{10} s^{-1}.

15.3 Simulation Results and Discussion

15.3.1 Homogeneous Nucleation of Dislocations

15.3.1.1 Dislocation Nucleation in Al and Cu

Shear deformation of Al, Cu and TiAl was carried out at different strain rates ranging from 10^6 to 5×10^{10} s^{-1}, at temperatures from 100 to 600 K, on (111) plane, along

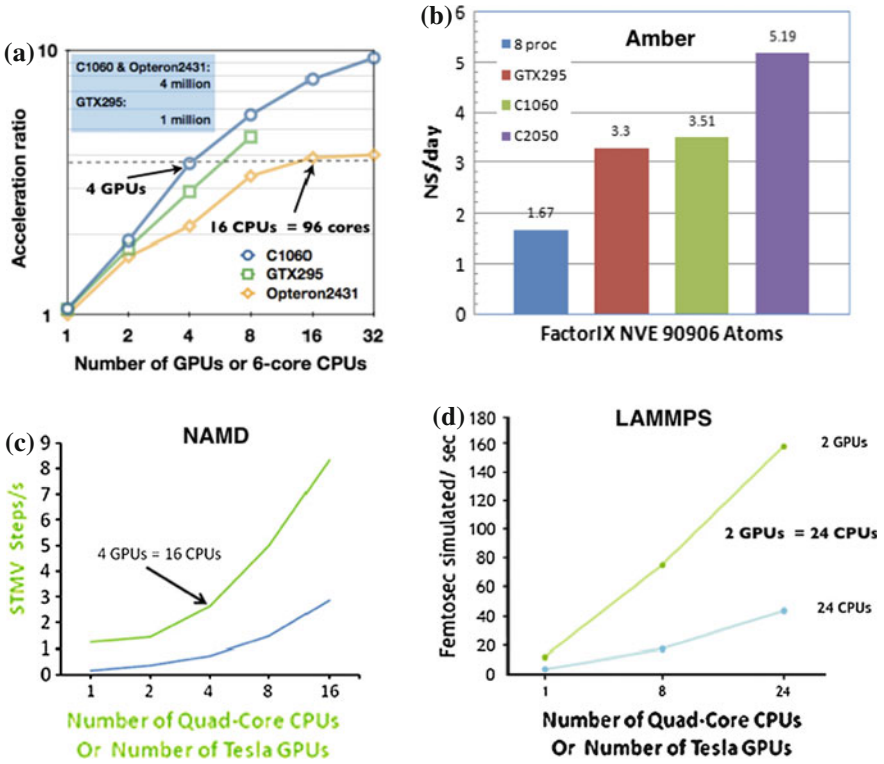


Fig. 15.3 Acceleration of ParaMD and three other MD codes over different numbers of CPU cores

the [1$\underline{1}$0] direction for fcc metals and [10$\underline{1}$] and [$\underline{1}$01] directions in TiAl for the superdislocation nucleation. Multiple nucleations appeared in high speed shearing. Dozens of loops formed in the simulation box at a shearing rate of $5 \times 10^{10}\ s^{-1}$, each being a single partial dislocation with very limited range of movement, since the distance between the different loop nuclei is very small, which limited the motion of each partial. Due to the high density of nuclei, limited strain is available for each loop, and only single partial can be generated at such an ultra-high speed deformation. The Burgers vector of the Shockley partial is 1/6[1$\underline{2}$1].

At the strain rate of $5 \times 10^{9}\ s^{-1}$, fewer loops formed at the same time, and most of the loops are full dislocations with Burgers vector of 1/2[1$\underline{1}$0] in Al, a snapshot from the simulation being shown in Fig. 15.4a. It can be seen the second partial loop is nucleated when the first partial loop grows up to about 4 to 5 nm in size. The partial separation width ranges from 1 to 3 nm depending on the character of the segment of the dislocation. When the shear strain rate is below $10^{8}\ s^{-1}$, only one loop is formed in each simulation. Its long axis is oriented along the 30° direction, parallel to the Burgers vector of the first partial. As the loop grows, the difference

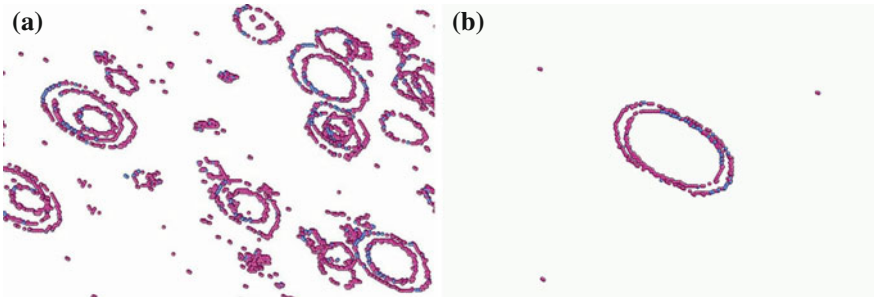


Fig. 15.4 Dislocation loop nucleation in Al sheared along [1$\underline{1}$0] direction on the (111) plane at strain rate of **a** $5 \times 10^9\,s^{-1}$ and **b** $10^8\,s^{-1}$. The AtomEye is used for the display of the atomic configuration, with atoms colored according their local coordination number; atoms with normal coordination number 12 are made invisible (Li 2003)

between the widths of the stacking faults, i.e., the partial separation along different direction, increases, with that along the 30° dislocation the smallest, while that in the 120° direction is the largest.

As for Cu, the nucleation shows a similar trend, with the separation between the two partials much larger than in Al, due to the lower stacking fault energy of the Cu from the interatomic potential. Apart from that, the orientations of the inner loop and the outer loop show some difference, especially at the initial stage of the nucleation.

15.3.1.2 Asymmetry of Superdislocation Nucleation in TiAl

TiAl is a compound with an $L1_0$ structure ordered on the basis of fcc metals. It is face centered tetragonal having alternate layers of Ti and Al atoms along c direction. Due to the $L1_0$ ordering, the three ⟨110⟩ directions are no longer equivalent, and the shear deformation along the different ⟨110⟩ directions will result in different dislocation structure. Along any ⟨110] direction (here the notation introduced in Hug et al. (1988) is used), the shear will produce ordinary dislocations, while along any ⟨101] will cause the so called superdislocation to nucleate. Our simulations show that, the superdislocation shear is not symmetric when shearing in the two opposite senses; the dissociations decomposed differently depending on the sense of the shear stress. Figure 15.5 shows the nucleation and growth process when sheared along two different senses. It can be observed that each superdislocation is composed of four partials (only 2 shown in Fig. 15.5a), therefore the Burgers vectors are twice that of the ordinary dislocations, which will cause different lattice friction for the dislocation movement; due to the correlated movement of the four partials, larger stress can be imposed on the nearby dislocations, and more reactions between dislocations can be achieved when dislocations from different sources meet. Further more, after nucleation, due to the difference in the fault energy on different atomic planes, the dislocation may undergo different reconfiguration and cross slip partially. This would

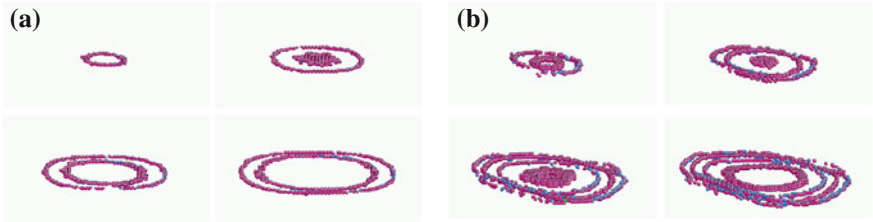


Fig. 15.5 Super-dislocation loop nucleation and growth in TiAl when sheared along **a** [10$\underline{1}$] and **b** [$\underline{1}$01] direction on the (111) plane. In both cases the superdislocation decomposed into four partials, but with different separations between each partial when sheared along different directions, only two partials being shown in **a**

increase the work hardening rate of TiAl and will exhaust the dislocation source sooner, and therefore is considered one of the factors influencing the anomalous increase in strength at high temperature and the room temperature brittleness of the compound (Hug et al. 1988).

15.3.2 Reaction Between Dipolar Dislocation and Debris Formation

15.3.2.1 Dislocation Reaction and Defect Formation During Shear Deformation

The nucleated dislocation loops expand upon further shear, and they interact with each other when coming closer, forming various structures depending on the relative positions of the approaching loops. The loops on the same slip plane would simply annihilate when they meet, since the meeting segments should have the opposite sign, while those from different slip planes would form dipoles of different heights, and further reaction may happen generating many vacancy tubes, mostly at 60° angle to the shearing direction. Although transformed dipoles of various heights were observed ranging from 1d to 3d, the major type of defects are due to the 1d dipole transformation, most often forming vacancy tubes. Due to the high density of nuclei at the beginning, the density of the vacancy tube is much higher for faster shearing compared with shearing at lower rate. There are also similar numbers of interstitial type dipoles forming small interstitial loops. These loops are very mobile in the lattice and have a high probability of annihilation with vacancy tubes if they meet on the same or nearby slip planes; otherwise, the interstitial loop may be pinned by the vacancy tube and vibrate around it. More details can be found in Xu et al. (2009).

These loops or vacancy rods can also react with mobile dislocations passing by on the nearby slip planes. As a result of the reactions, the dislocation becomes jogged at different points along the dislocation line, making the movement of the dislocation

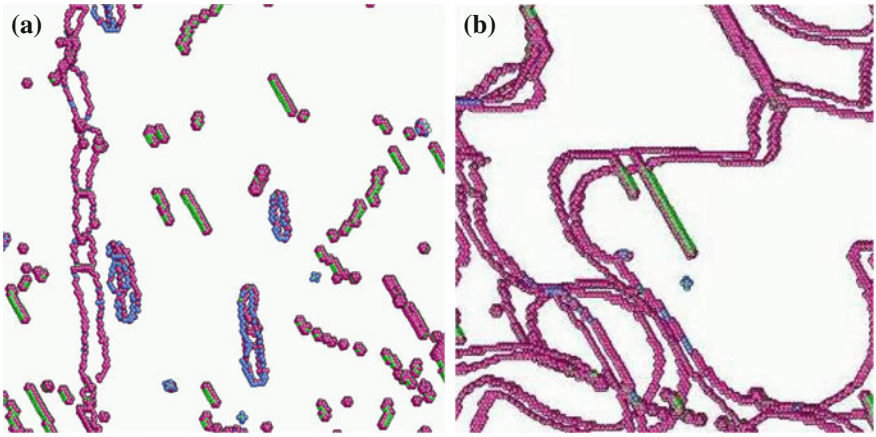


Fig. 15.6 **a** Various defects formed after different reactions. The *short bars* in *green* are the vacancy rows, while *rings* marked *blue* are interstitial loops. The long dislocation with two partials is heavily jogged after reaction. **b** Formation process of the vacancy tube by the movement of dislocations dragging two jogs

increasingly difficult, especially at locations with the presence of the debris. For the debris with large mobility, such as interstitial dipole loop, the moving dislocation can react with it if they are on the same slip plane; otherwise the moving dislocation may sweep the small loops in the wake of the dislocation motion, and make a clean channel for the future dislocation motion.

For the formation of vacancy rows and interstitial loops from dislocations in Al when sheared along the [110] direction, two types of mechanisms are observed in our simulation: one is the two-dislocation mechanism by which the two dislocations meet and form a vacancy row, as shown in our former MD simulation (Xu et al. 2009); the other is concerns the movement of a jog-containing dislocation, the dragging of jogs in one direction would form vacancy rows, while the opposite motion create interstitial clusters, the former being shown as green rods and the latter as blue loops in Fig. 15.6.

15.3.2.2 Formation of SFT and Other Debris from Dipole at High Temperature

Dislocation dipole is a type of configuration frequently found in metals and intermetallic compounds. Due to the attraction between the dislocations with the opposite signs, some stable configurations can be formed when two dislocations come from different slip planes meet. At high temperature, a dipole may undergo further transformation even without shearing. Figure 15.7 shows the different kind of defects formed after high temperature annealing of the dipoles with their height from 1 to 4 d_{111}. Our MD simulations show that the migration barrier for vacancy along the line

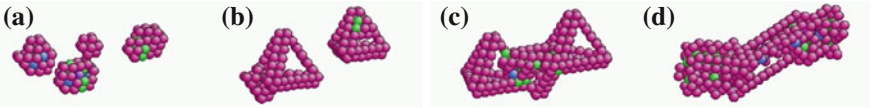


Fig. 15.7 Transformation of dipoles with different height: **a** 1d; **b** 2d; **c** 3d and **d** 4d

of the dipole cores is greatly reduced, the atoms can move quickly and the local diffusion promotes the formation of small stacking fault tetrahedron (SFT). The smallest SFT containing only three vacancies can move via shape change. It can grow by absorbing single vacancy, vacancy cluster, via a ledge mechanism, the details of which have been published recently as a series of papers (Wang et al. 2011a, b, c). SFT may influence dislocation motion either by long range elastic interaction, or, most strongly, by their reactions with mobile dislocations forming various defects along the dislocation line.

The stability of the debris formed was estimated by defect energy calculation after static relaxation. The energy of the faulted dipole, SFT and dipoles along various directions were compared; as shown in our recent work, the faulted dipole and SFT both have lower energy than the dipole, showing the stability of these kinds of defect at ambient temperature (Wang et al. 2008, 2009). This is in agreement with the experimental results of SFT formation at room temperature, high strain rate deformation in Al (Kiritani et al. 1999), and the stable perfect SFT found in Au, Ni, Cu (Singh et al. 2004).

15.3.3 Implications to Mechanical Properties

15.3.3.1 Breaking of Dipole by Shear Deformation

For edge dipoles, the 45° configuration is the low energy stable configuration. Shearing of a dipole-containing crystal may break the dipole into independent dislocations, the critical stress for this being called breaking stress, which is a function of the dipole height. Figure 15.8 shows the results of our MD calculation and the estimation from elasticity theory. The interaction of dislocations in a dipole is strongly dependent on the height and distance; below a certain height, dipole can react directly upon shear deformation, forming complicated structure which will not only change their mobility, but also hinder the motion of other dislocations. Typical debris may include faulted dipole, the combination of differently oriented faulted dipole, smaller dislocation loops of different types, etc.

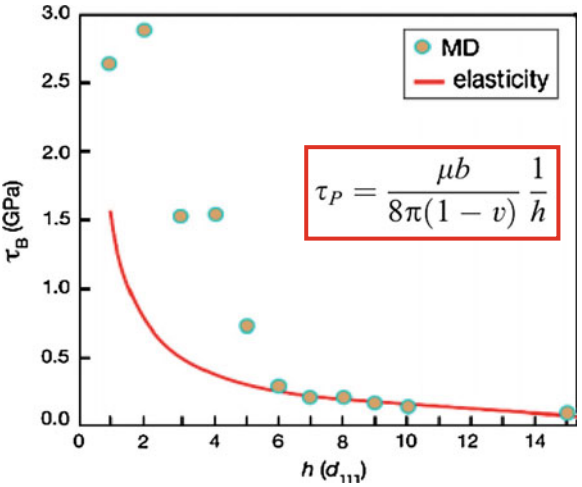


Fig. 15.8 Comparison of MD and elastic estimation of breaking stress. The breaking stress calculated from MD is much higher for 1d to 5d

15.3.3.2 Dipole Transformation by Shear Straining

Dipoles with different character may have different energies (Wang et al. 2009); therefore their stability varies, and they behave differently under shear straining. The pure screw dipole is usually easy to be cross annihilated, while the edge type may change their character upon shearing. Along 60°, the dipoles are prone to transforming to faulted dipole in fcc systems. The breaking stress depends strongly on the structure of the dipole; as is shown in Fig. 15.9, the edge type dipole is easily broken by shear, while after multi-passage shear by the dislocations in the dipole, the originally straight dipole becomes curved and finally transforms into a zigzagged configuration with each segment being a faulted dipole, as shown in Figs. 15.9b, c. The breaking of such a transformed dipole needs much larger stress, as shown in Fig. 15.9a.

In a deformed metal, especially fatigue deformed one, the patches of dislocation may finally turn into a dislocation wall, which mostly contains dipolar dislocations. In such a case, detailed understanding of the behavior of the dipole would enrich our knowledge of the crack formation and fracture of the materials.

15.3.3.3 Critical Height for Dipole Annihilation

It was generally believed that dipoles lower than a certain height will simply annihilate and disappear in the lattice and will not affect further deformation. Most models for dislocation dynamics simulation have just deleted the dislocation segment when a low-height dipole was formed in order to reduce the amount of calculation. Our

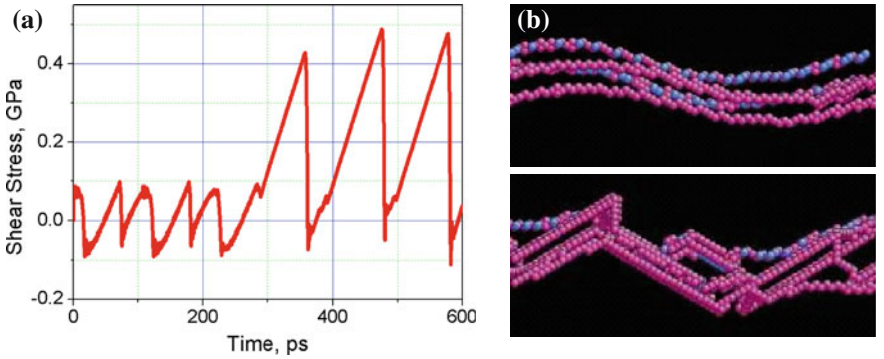


Fig. 15.9 **a** Stress-strain curve of a dipole during shear deformation. The dipole breaking stress increased after several passages. The configuration after several passage shear (**b**) transformed into **c** when the breaking stress increased

recent MD simulation shows that there exists no critical height below which small-height edge dislocations and 60° dipoles would annihilate and their debris simply disappears. Various structures can result from shearing dipoles under an applied stress including faulted dipoles and zigzagged structures containing 60° dipole segments connected by junctions. These defect structures are believed to contribute to work hardening and to act as nucleation sites for dislocation entanglement and self-patterning under single slip (Wang et al. 2008).

15.4 Conclusions

(1) MD simulation on GPU now exhibits a substantial acceleration, with a speed boost of >10 times compared to a core of the advanced CPU.
(2) Dislocation or superdislocation nucleates homogeneously in fcc metal and TiAl by shearing in $\langle 110]$ or $\langle 101]$ direction, and the core structures are identified.
(3) MD simulation shows that during shear deformation dislocation debris in the form of point defects such as vacancies, self-interstitial rows and independent dipole loops can be formed by reaction of dislocations within the same slip system.
(4) With increasing temperature, edge dipoles transform into a series of complex structures, including zigzagged faulted dipoles and truncated and perfect SFTs, which may form obstacles to dislocation motion and affect the ductility of metals.

Acknowledgments The support from the Ministry of Science and Technology of China (2011CB604104), the National Science Foundation of China (50911130367 and 50631030), the Ministry of Finance (ZDYZ2008-2-A12), and the Chinese Academy of Sciences (INFO-115-B01) is gratefully acknowledged.

References

Bulatov VV, Cai W (2006) Computer simulations of dislocations. Oxford University Press, Oxford
Clemens H, Kestler H (2000) Processing and applications of intermetallic gamma-TiAl-based alloys. Adv Eng Mater 2(9):551–570
Hug G, Loiseau A et al (1988) Weak-beam observation of a dissociation transition in TiAl. Philos Mag A 57(3):499–523
Kiritani M, Satoy Y et al (1999) Anomalous production of vacancy clusters and the possibility of plastic deformation of crystalline metals without dislocations. Philos Mag Lett 79:797–804
Li J (2003) AtomEye: an efficient atomistic configuration viewer. Model Simul Mater Sci Eng 11(2):173–177
Mishin Y, Farkas D et al (1999) Interatomic potentials for monoatomic metals from experimental data and ab initio calculations. Phys Rev B 59(5):3393–3407
Mishin Y, Mehl MJ et al (2001) Structural stability and lattice defects in copper: Ab initio, tight-binding, and embedded-atom calculations. Phys Rev B 63(22):16
Nabarro FRN (1997) Fifty-year study of the Peierls-Nabarro stress. Mater Sci Eng A Struct Mater Prop Microstruct Process 234:67–76
Singh BN, Golubov SI et al (2004) Review: Evolution of stacking fault tetrahedra and its role in defect accumulation under cascade damage conditions. J Nucl Mater 328:77–87
Veyssiere P (2006) The weak-beam technique applied to the analysis of materials properties. J Mater Sci 41(9):2691–2702
Wang H, Xu DS et al (2009) The transformation of narrow dislocation dipoles in selected fcc metals and in γ-TiAl. Acta Mater 57:3725
Wang H, Xu DS et al (2011a) The formation of stacking fault tetrahedra in Al and Cu: I. Dipole annihilation and the nucleation stage. Acta Materialia 59(1):1–9
Wang H, Xu DS et al (2011b) The formation of stacking fault tetrahedra in Al and Cu: II. SFT growth by successive absorption of vacancies generated by dipole annihilation. Acta Materialia 59(1):10–18
Wang H, Xu DS et al (2011c) The formation of stacking fault tetrahedra in Al and Cu: III. Growth by expanding ledges. Acta Materialia 59(1):19–29
Wang H, Xu DS et al (2008) The transformation of edge dislocation dipoles in aluminium. Acta Materialia 56(17):4608–4620
Xu DS, Wang H et al (2008) Molecular dynamics investigation of deformation twinning in γ-TiAl sheared along the pseudo-twinning direction. Acta Materialia 56:1065–1074
Xu DS, Wang H et al (2009) Point defect formation by dislocation reactions in TiAl. IOP Conf Series: Mater Sci Eng 3:012024
Zope RR, Mishin Y (2003) Interatomic potentials for atomistic simulations of the Ti-Al system. Phys Rev B 68(2):024102

Part VI
Geophysical and Fluid Dynamical Application

Chapter 16
Large-Scale Numerical Weather Prediction on GPU Supercomputer

145.0 TFlops with 3990 GPUs on TSUBAME 2.0

Takayuki Aoki and Takashi Shimokawabe

Abstract In order to drastically shorten the runtime of a weather prediction code ASUCA developed by the JMA (Japan Meteorological Agency) for the purpose of the next-generation weather forecasting service, the entire parts of the huge code are re-written for GPU computing from scratch. By introducing many optimization techniques and several new algorithms, very high performance of 145 TFlops has been achieved with 3990 GPUs on the TSUBAME 2.0 supercomputer. It is quite meaningful to show that the GPU supercomputing is really available for one of the major applications in the HPC field.

16.1 Introduction

Weather forecasting is an indispensable parts in our daily lives and business activities, needless to say, for natural disaster preventions. The atmosphere has a very thin thickness to compare with the Earth diameter. In the previous atmosphere code, the force balance between the gravity and the pressure gradient in the vertical direction and we call them hydrostatic models. Recently it is widely recognized that the vertical dynamical processes of the water vapor should be taken into consideration in cloud formations. Three-dimensional non-hydrostatic model describing up-and-down movement of air have been developed in the weather research.

For the weather simulations, initial data are produced by the data assimilation dealing with many kinds of observed data and simulation results based on four-dimensional variational principle. Since the weather phenomena are chaotic, a predictable duration is less than several days for one set of the initial data and the jobs run sequentially by updating the initial data.

T. Aoki (✉) · T. Shimokawabe
Global Scientific Information and Computing Center,
Tokyo Institute of Technology, Tokyo, Japan

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 261
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_16,

In recent years, it is highly demanded to forecast detailed weathers such as unexpected local heavy rain, and the high resolution non-hydrostatic models is desired to be carried out with fine-grained grids.

16.2 GPU Computing for Weather Simulations

The computational heavy load is required to run the high-resolution weather models. A next-generation atmosphere simulation model WRF (Weather Research and Forecasting) is a world standard code developed at the NCAR (National Center for Atmospheric Research), UCAR (University Corporation for Atmospheric Research) and so on in the United States and supported by worldwide researchers. The WRF has been scored 50 TFlops on the current fastest supercomputer in the world.

Numerical weather models consist of a dynamical core and physical processes. In the dynamical core, forecast variables such as winds, atmospheric pressure and humidity are calculated by solving fluid dynamics equations. The physical processes strongly depend on parametrizations related to such microphysics as condensation of water vapor, cloud physics, and rain. In the computation of dynamics core, memory access time is a major part of the elapsed time to compare with floating point calculations and it is hard to get close to the peak performance in any computers. On the other hand, the physical processes include some computationally intensive parts demanding high performance of floating point calculations.

Graphics Processing Units (GPUs) have been developed for the rendering purpose of computer graphics. The request for high-level computer visualization makes the GPUs to have high performance of floating point calculation and wide memory bandwidth. Recently, exploiting GPUs for general-purpose computing, i.e. GPGPU, has emerged as an effective technique to accelerate many applications. After CUDA (CUDA 2010) was released by NVIDIA as the GPGPU-programming framework in 2006, it allows us to use the GPU easily as an accelerator. In the area of high performance computing (HPC), it has been reported that a lot of successful applications in Computational Fluid Dynamics (CFD), Fast Fourier Transform (FFT), molecular dynamics, astrophysics, bio-informatics and so on ran on GPUs dozens time faster than on a conventional CPU.

In the numerical weather prediction, it was reported that a computationally expensive module of the WRF model was accelerated by means of a GPU (Michalakes and Vachharajani 2008; Linford et al. 2009). These efforts, however, only result in a minor improvement (e.g., 1.3 × in Michalakes and Vachharajani 2008) in the overall application time due to the partial GPU porting of the entire code. Since the other functions (subroutines) run on the CPU and all the variables were allocated on the CPU main memory, it is necessary to transfer the data between the host CPU memory and the GPU video memory through the PCI Express bus every GPU kernel-function call. They reported that the acceleration for the microphysics module itself achieved a twenty-fold speedup. While physical processes are composed of small and relatively independent modules and able to be easily replaced with other mudules,

a dynamical core computes time integration of interdependent forecast variables. Thus, GPU porting of parts of the dynamical core does not contribute improvement of performance.

As the successor to TSUBAME 1.2, the TSUBAME 2.0 supercomputer, which is equipped with more than 4000 GPUs, has started operating in November 2010 and has become the first petascale supercomputer in Japan. Since TSUBAME 2.0 provides most part of its computing performance by GPUs, it is a key issue to achieve high performance on GPU in many applications. In the article, we show the process porting an operational weather prediction code to the GPU on TSUBAME 2.0 and demonstrate the performance for a practical operation size.

16.3 Next-Generation Weather Prediction Code ASUCA

ASUCA (Asuca is a System based on a Unified Concept for Atmosphere) is a next-generation high resolution mesoscale atmospheric model being developed by the Japan Meteorological Agency (JMA)(Ishida et al. 2010). The ASUCA succeeds the Japan Meteorological Agency Non- Hydrostatic Model (JMA-NHM) as an operational non-hydrostatic regional model at the JMA.

First, we have implemented the dynamical core as the first step toward developing the full GPU version of the ASUCA. In the ASUCA, a generalized coordinate and flux-form non-hydrostatic balanced equations are used for the dynamical core. The time integration is carried out by a fractional step method with the horizontally explicit and vertically implicit (HE-VI) scheme (Skamarock and Klemp 1994). One time step consists of short time sub-steps and a long time step. The horizontal propagation of sound waves and the gravity waves with implicit treatment for the vertical propagation are computed in the short time step with the second-order Runge-Kutta scheme. The long time step is used for the advection of the momentum, the density, the potential temperature and the water substances, the Coriolis force, the diffusion and other effects by physical processes with the third-order Runge-Kutta method. The above matters are almost same as those employed in the WRF model. In the present ASUCA, the physical core is still being developed and a Kessler-type warm-rain model has been implemented for the cloud-microphysics parameterization describing the water vapor, cloud water, and rain drops.

16.4 Single GPU Implementation and Performance

Although our final destination is to develop the multi-GPU version of ASUCA, we start from the single GPU implementation and show its performance. Figure 16.1 illustrates the computational flow diagram. In the beginning of the execution, the host CPU reads the initial data from the input files onto the host memory, and then transfers them to the video memory on the GPU board. The GPU carries out all the

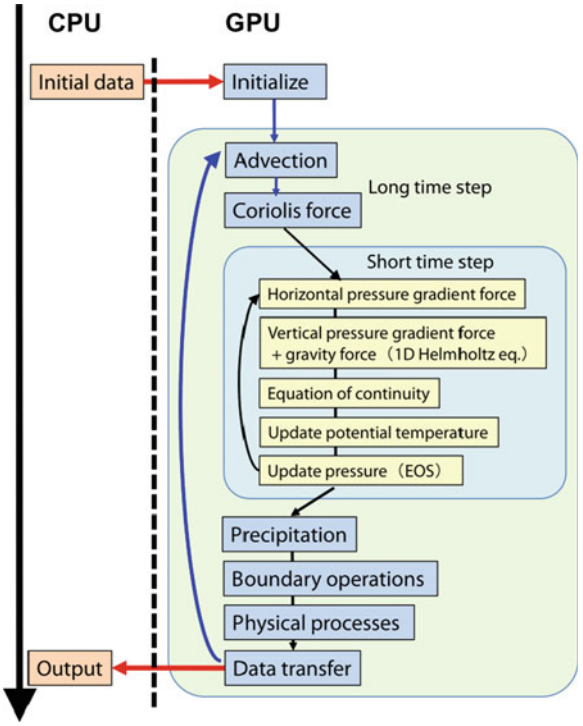


Fig. 16.1 Computational flow diagram of the ASUCA. All the modules inside the short time-step and the long time-step loops are executed on the GPU

computational modules inside the short time-step and long time-step loops. When the forecast data are completed on the GPU, the minimal data are transferred to the host CPU memory to reduce the communication between CPU and GPU.

16.4.1 Optimizations

In order to improve the performance on the GPU, we have introduced several optimizations in implementing the code in CUDA. We focus on two components as examples: (a) the advection computation (b) the 1-dimensional Helmholtz-type equation for the pressure.

(a) Implementation of the advection computational modules

The 3-dimensional advection computation is strongly memory bound and it is very effective to reduce the access to the video memory (called global memory in CUDA) in order to improve the performance. We make use of the shared memory as a software-managed cache, which is shared among threads in a block in the CUDA programming. For a given grid size (nx, ny, nz) of the computational domain, the

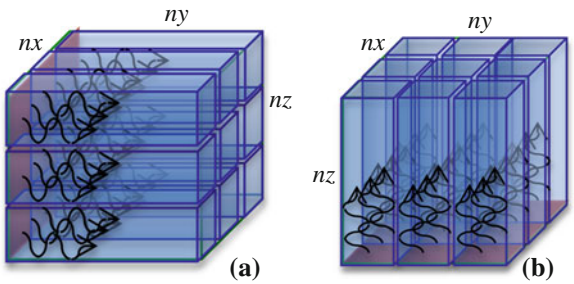


Fig. 16.2 Configuration of the CUDA blocks and marching directions of the threads. **a** Advection computation and **b** Helmholtz-type pressure equation

GPU kernel function is invoked with (nx/64, nz/4, 1) blocks with (64, 4, 1) threads. The z direction in physical space is mapped to the y direction in the CUDA code. Each thread specifies a point (x, z) and calculate the advection equation on the grid point from $j = 0$ to $j = ny - 1$ marching in the y direction as shown in Fig. 16.2a. The block size of (64, 4, 1) threads is derived from the performance optimization.

The discretization of the advection equation has a four-point stencil in each direction on the mesh. Each block holds an array of $(64 + 3) \times (4 + 3)$ elements on the shared memory. When a block computes a xy plane of the computational domain, the variable data on the global memory copied to the shared memory to be shared among threads in the block. On the other hand, the stencil access in the y-direction is closed in the thread by marching the computation in the y-direction. The variable data of the global memory are stored in the registers (temporal variables). When we compute the j+1-th plane, the data have been already stored in the registers from the global memory in computing j-th plane. In our implementation, we copy the data in the registers to the shared memory and reuse them without same accesses to the global memory (Fig. 16.3).

(b) Implementation of the Helmholtz-type pressure equation

Due to the HE-VI splitting, the pressure equation reduces to 1-dimensional Helmholtz-type elliptic equation in the vertical (z) direction. The discretization of the equation is expressed by a tri-diagonal matrix. It is possible to apply the TDMA algorithm to solve the matrix; however we have to calculate the elements sequentially in the z-direction. Figure 16.2b shows the data parallel by marching the sequential calculation in the z-direction.

16.4.2 Performance

Since the ASUCA is being developed in FORTRAN language at the JMA, the GPU code has to be developed from scratch in CUDA. Before implementing the ASUCA on GPU, we re-wrote the Fortran ASUCA code to C/C++ language because we change the element order of the 3-dimensional array to improve the memory access

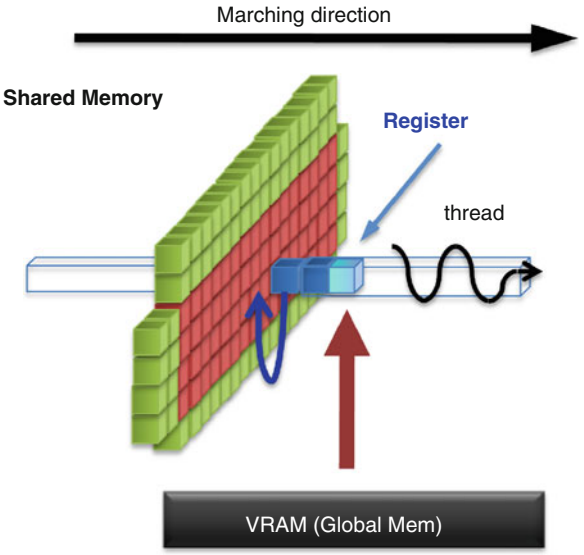


Fig. 16.3 Usage of the shared memory and the register to reduce the access to the global memory

performance of the GPU computing. In order to measure the performance of floating-point operations on a GPU, we count the number of floating-point operations of the CPU-based ASUCA by running it on a CPU with a performance counter provided by PAPI (Performance API) (Browne et al. 2000). By using the counts and the GPU elapsed time, the performance of the GPU computing is evaluated.

In all the cases, we fixed the grid number nx $= 256$ and nz $= 48$ of the computational domain and varied the number ny 32 to 208. The performance was measured in both single- and double- precision floating-point calculation using a NVIDIA Tesla M2050 in TSUBAME 2.0. The results of the GPU performance are shown in Fig. 16.4. We achieved 49.1 GFlops in single precision for $256 \times 208 \times 48$ mesh on a single GPU. In the double precision, the performance has about half the single precision. As references, the performances on the Intel CPU (Xeon X5670 (Westmere-EP) 2.93 GHz 6 core x2: total 12 cores and 1 core) are plotted on the same graph. The original FORTRAN code was compiled by the Intel ifort compiler. It is found that the single GPU performance achieved a six-fold speedups in comparison with 2 sockets of the CPU performance of the Intel Xeon X5670 in double precision.

16.5 Multi-GPU Computing

The GPU Tesla M2050 card on TSUBAME 2.0 has only 3-GB on-board video memory, which can hold up to a grid of size $256 \times 208 \times 48$ in running the ASUCA. For large-size problems, it is necessary to use multiple GPUs beyond the video memory on a single GPU. The current operation for the weather forecast at the JMA utilizes the grid of size $721 \times 577 \times 50$.

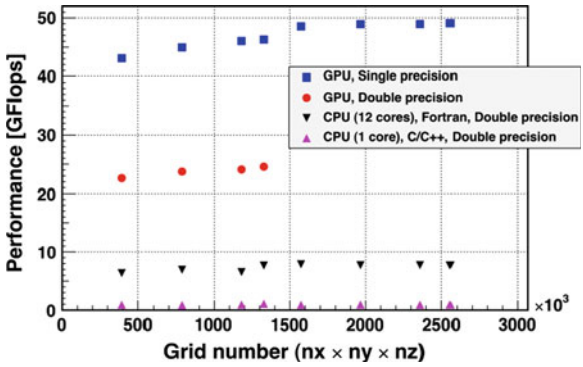


Fig. 16.4 Single GPU performance on a NVIDIA Tesla M2050 of TSUBAME 2.0 and CPU performance of the Intel Xeon X5670

We decompose the given computational domain in both the x- and y-directions (2-D decomposition) and allocate the sub domain to each GPU since the z-directional mesh size is relatively small. Because GPUs cannot directly access to the data stored on the global memory of other GPUs, the host CPUs are used to bridge GPUs for the exchange of the boundary data between the neighbor GPUs. The process is composed of the following three steps: (1) the data transfer from GPU to CPU using the CUDA runtime library, (2) the data exchange between nodes with the MPI library, and (3) the data transfer back from CPU to GPU with the CUDA runtime library.

In the case of multi-GPU computing, the data communication time with the neighbor GPUs is not ignored in the total execution time. The overlapping technique with the computation is available to hide the communication costs and achieves better performance with a large number of GPUs (Shimokawabe et al. 2010).

16.5.1 Performance of Multi-GPU Computing

Each node of TSUBAME 2.0 has three NVIDIA GPU Tesla M2050 attached to the PCI Express bus 2.0 × 16, two QDR Infiniband and two sockets of the Intel CPU Xeon X5670 (Westmere-EP) 2.93 GHz 6-core. The nodes are connected to the fat-tree interconnection with 200 Tbps bi-section bandwidth. Each GPU handles the domain of $256 \times 108 \times 48$ in double precision and $256 \times 208 \times 48$ in single precision, respectively.

The multi-GPU performance of ASUCA on TSUBAME 2.0 is shown in Fig. 16.5. Using 3990 GPUs, we achieved an extremely high performance of 145.0 Tflops for the domain of $14368 \times 14284 \times 48$ in single precision. The double precision performance is 76.1 for the domain of $10336 \times 9988 \times 48$. It is confirmed to maintain a good weak scalability. To compare with the CPU performance, the performance of 3990 is compatible with 3990 × 50 CPU cores.

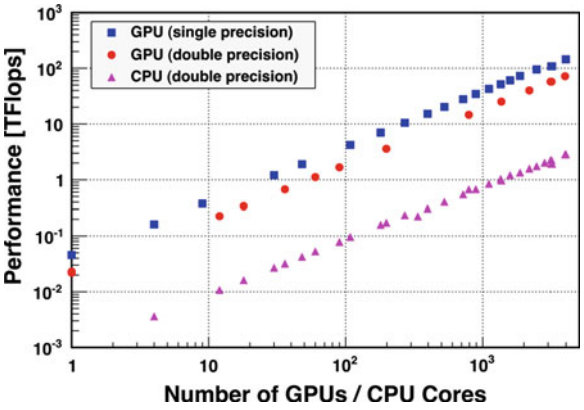


Fig. 16.5 Multi-GPU performance of ASUCA on TSUBAME 2.0 comparing with the CPU

Figure 16.6 demonstrates the real case of the ASUCA operation with both the real initial and the boundary data used for the current weather forecast at the JMA. This simulation was performed with a $4792 \times 4696 \times 48$ mesh with horizontal mesh resolution of 500 m using 437 GPUs of TSUBAME 2.0 in single precision.

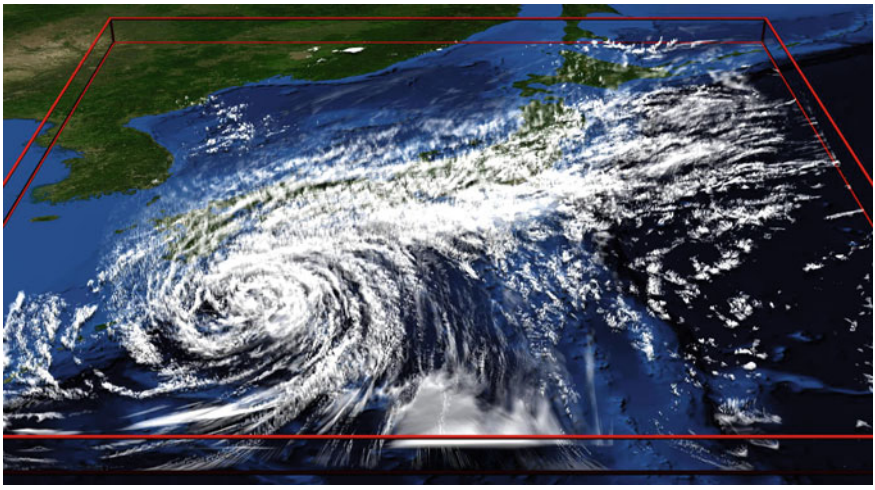


Fig. 16.6 ASUCA real operation to describe a typhoon with $4792 \times 4696 \times 48$ mesh using 437 GPUs of TSUBAME 2.0

16.6 Summary

The full GPU implementation of the next-generation, production weather code ASUCA was carried out on the TSUBAME 2.0 supercomputer in the Tokyo Institute of Technology. The GPU offers not only extremely huge computational performance but also big advantages in respect of the cost and power consumption. It is really meaningful that the numerical weather prediction, one of the typical applications for a practical purpose, is provided as a successful example of the GPU supercomputing. In China and other countries, large-scale GPU computing on supercomputers is about to begin. TSUBAME 2.0 in the Tokyo Tech Global Scientific Information and Computing Center (GSIC) is the first over-petaflops supercomputer in Japan. In the GPU supercomputing era, it is expected that research communities becomes large and active to achieve great outcomes on GPU supercomputers.

Acknowledgments We would like to thank Chiashi Muroi, Junichi Ishida and Kohei Kawano at the Japan Meteorological Agency the for providing the original ASUCA code and helping us develop the GPU version. We are grateful to Satoshi Matsuoka, Toshio Endo, Akira Nukada and Naoya Maruyama at the Tokyo Institute of Technology for helping us to use TSUBAME 2.0. This research was supported in part by the Global Center of Excellence Program "Computationism as a Foundation for the Sciences" and KAKENHI, Grant-in-Aid for Scientific Research (B) 19360043 from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan, and in part by the Japan Science and Technology Agency (JST) Core Research of Evolutional Science and Technology (CREST) research program "ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies".

References

Skamarock WC, Klemp JB, Dudhia J, Gill DO, Barker DM, Duda MG, Huang XY, Wang XY, Powers JG (2008) A description of the advanced research WRF version 3, National Center for Atmospheric Research

Bland AS, Kendall RA, Kothe DB, Rogers JH, Shipman GM (2009) Jaguar: the world's most powerful computer. In: 2009 CUG Meeting, pp 1–7

CUDA programming guide 3.2 (2010) http://developer.download.nvidia.com/compute/cuda/32/toolkit/docs/CUDA C Programming Guide.pdf, NVIDIA

Michalakes J, Vachharajani M (2008) GPU acceleration of numerical weather prediction. In IPDPS: IEEE, pp 1–7

Linford JC, Michalakes J, Vachharajani M, Sandu A (2009) Multi-core acceleration of chemical kinetics for simulation and prediction. In SC '09: Proceedings of the conference on high performance computing networking, storage and analysis, ACM, New York, pp 1–11

Ishida J, Muroi C, Kawano K, Kitamura Y (2010) Development of a new nonhydrostatic model "ASUCA" at JMA, CAS/JSC WGNE reserch activities in atomospheric and oceanic modelling

Skamarock WC, Klemp JB (1994) Efficiency and accuracy of the Klemp-Wilhelmson time-splitting technique. Mon Weather Rev 122:2623

Micikevicius P (2009) 3D finite difference computation on GPUs using CUDA. In GPGPU-2: Proceedings of 2nd workshop on general purpose processing on graphics processing units, ACM, New York, pp 79–84

Browne S, Dongarra J, Garner N, Ho G, Mucci P (2000) A portable programming interface for performance evaluation on modern processors. Int J High Perform Comput Appl 14(3):189–204

Shimokawabe T, Aoki T, Muroi C, Ishida J, Kawano K, Endo T, Nukada A, Maruyama N, Matsuoka S (2010) An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In SC '10: Proceedings of the conference on high performance computing networking, storage and analysis, ACM in press, New York

Chapter 17
Targeting Atmospheric Simulation Algorithms for Large, Distributed-Memory, GPU-Accelerated Computers

Matthew R. Norman

Abstract Computing platforms are increasingly moving to accelerated architectures, and here we deal particularly with GPUs. In Norman et al. (2011), a method was developed for atmospheric simulation to improve efficiency on large, distributed-memory machines by reducing communication demand and increasing the time step. Here, we improve upon this method to further target GPU-accelerated platforms by reducing GPU memory accesses, removing a synchronization point, and clustering computations. The modified code ran more than two times faster than the original in some cases even though more computations were required, demonstrating the importance of improving memory handling on the GPU. Furthermore, we discovered that the modification also has a near 100 % hit rate in fast, on-chip L1 cache and discuss the reasons for this. Finally, we remark on further potential improvements to GPU efficiency.

17.1 Introduction

17.1.1 Multi-Scale Aspects of Atmospheric Models

The numerical integration of a set of conservation laws for mass, momentum, and energy forms the dynamical core (hereafter, dycore) of an atmospheric model. Alongside the dycore are physics packages representing phenomena that either fall below truncation scales after discretization or are otherwise absent from the conservation

This submission was written by the author(s) acting in (his/her/their) own independent capacity and not on behalf of UT-Battelle, LLC, or its affiliates or successors..

M. R. Norman(✉)
Oak Ridge National Laboratory, National Center for Computational Sciences,
Oak Ridge, TN, USA
e-mail: normanmr@ornl.gov

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 271
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_17,

laws. The dycore is also responsible for transporting variables used by the physics, a challenging task when many species are involved. A coupling determines interactions between the dycore and physics, much of which is multi-scale (Williamson 2007). The dycore typically dominates run time for larger problem sizes, sets the grid spacing for physics packages (with notable exceptions, e.g., Khairoutdinov et al. 2005), and sets the stage for overall parallel efficiency. This is even more the case because physics routines currently operate only in vertical columns which are not currently parallelized in a distributed memory manner.

The notion of multi-scale applies to dycore computations both spatially and temporally. Temporally, all atmospheric dycores must respect the coupling of fast and slow dynamics. Acoustic velocity in the lower atmosphere is roughly $350\,\mathrm{ms}^{-1}$ while wind speed maximizes at roughly $100\,\mathrm{ms}^{-1}$ in mid-latitude jet streaks. The more normative wind speed is really only order $10\,\mathrm{ms}^{-1}$. Respecting the influence of fast modes while simulating on the temporal scale of slower modes is non-trivial. Ideally, a Newton–Krylov (NK) implicit method (Evans et al. 2009; Knoll and Keyes 2004) would be preferred as it achieves non-linearly convergent coupling of all scales. At the core of a NK method is the linear Krylov solver, containing a series of domain-spanning matrix-vector multiplies. These, however, require significant data movement between separate memory spaces in a distributed memory environment—both between system nodes and between GPU and main memories within nodes. Thus, such methods often exhibit insufficient parallel efficiency to warrant using the order 10^4 nodes available on modern computers. One may use a smaller node count, but then the throughput is not up to standard with needs of climate. Implicit methods with a sufficiently effective preconditioner may prove useful for some weather applications, which require a much lower simulation throughput. But for now, we simulate time-explicitly, resolving at least nominally all temporal modes of dynamics captured by the spatial grid.

Explicit integration methods (e.g. Durran 1991; Klemp et al. 2007; Lin and Rood 1997) can operate with only local communication in parallel, allowing for much better parallel efficiency. However, they must use restricted time steps to respect fast modes, increasing the time to solution. Also, explicit coupling of fast and slow scales is rarely high-order and usually zeroth-order. A common method of handling fast modes for atmospheric models at current is to linearize and cycle the terms relevant to acoustic propagation multiple times in a time step with cheap methods while the more significant terms are handled to high-order accuracy on a larger time step (a philosophy usually called split-explicit sub-cycling in atmospheric modeling). In explicit methods, the manner in which fast modes are incorporated into the overall integration must respect computational efficiency as well as accuracy. This was the main purpose of Norman et al. (2011) (hereafter, NNS11): resolving fast waves faithfully in an explicit model while improving parallel efficiency by reducing communication burden. Since jet streaks and sound waves only differ by at most a factor of 4 in velocity for global models, the situation is not considered particularly stiff. Therefore, the acoustic time step is used, and a focus on improved parallel efficiency marks the study in NNS11. Here, we continue the efforts of NNS11 with particular emphasis on GPUs and report current experiences.

17.1.2 Efficiency on Distributed-Memory, GPU-Accelerated Machines

Traditionally, large time steps and low computation give optimal efficiency. At low to moderate levels of parallelism, semi-implicit semi-Lagrangian algorithms (Staniforth and Cote 1991) are designed toward this end. On large machines, however, parallel efficiency can be difficult to achieve due to communication burden. In many applications, the problem size is increased for implicit methods to give a good computation-to-communication ratio. The time to solution for realistic atmospheric simulations, however, is infeasibly large at these sizes. For larger distributed memory computers, split-explicit integration techniques improve parallel efficiency via local communication (Cullen and Davies 1991; Gassmann 2005; Klemp et al. 2007). (Note that we are not comparing overall run time or accuracy but only parallel efficiency.) Split-explicit methods still suffer from frequent synchronization and communication points interrupting relatively small pockets of computation, and they likewise cannot scale to a large number of processes.

GPU acceleration gives additional efficiency constraints, mostly having to do with memory handling. Data transfers between host memory (that is, main system memory) and device memory (that is, GPU memory) suffer lower bandwidth than those within GPU memory. Thus, transfers between host and device memories must be minimized and/or overlapped with computation. Second, within a kernel (a function specifying the work of an arbitrary GPU thread), slow global memory (GPU DRAM) accesses must be minimized and reused. Also, accesses should be organized such that the compiler can coalesce transfers for efficient bus usage, transporting as much data at a time as possible. The general efficiency paradigm for using global memory is to store data in local fast reservoirs of on-chip memory and *reuse* it.

Another dominant issue is so-called device occupancy. Kernels should be frugal enough with local (on chip, per multiprocessor) resources that hundreds of threads can share a multiprocessor. This aids in hiding memory accesses by switching threads while one is waiting on memory. The number of threads should be set to a multiple of the available hardware resources to ensure that the entire GPU is not waiting on a single multiprocessor or thread to finish. The parallelism must also be sufficiently fine grained so as to specify enough threads to fill the entire device. Another constraint to balance in is register pressure. The fewer the registers available per thread, the slower the application. There are also other efficiency issues beyond these such as reads and writes accessing the same shared memory banks.

17.1.3 Targeting Algorithms to the Hardware

In this paper, we discuss our experiences with modifying a new explicit method to improve efficiency on GPUs. We also encourage the reader to reconsider their computational path toward scientific goals even from the point of the numerical algorithm, giving hardware utilization and numerical assumptions equal weight. For

instance, less accurate algorithms that improve hardware utilization can allow refinement in the discretization, potentially leading to a superior solution when refined. While there are large costs involved in changing the core of most codes, the reach of simulation capabilities at petascale and exascale likely outweighs development cost. Also, many of the algorithmic choices one would make numerically to respect GPU efficiency constraints also improve efficiency on distributed memory architectures (e.g., infrequent synchronization, clustered computation).

As mentioned earlier, though split-explicit finite volume methods promise locality and improved parallel efficiency, the communication burden has much room for improvement. This is one reason for recent investment into low communication Galerkin methods (Giraldo and Restelli 2008; Taylor et al. 2007; Nair et al. 2009; Taylor et al. 1997) which offer potential to span order 10^4 separate memory spaces efficiently, giving a better time to solution for large problem sizes. Algorithms with excessive memory accesses and synchronization with relatively little computation in between do not play to the strengths of a GPU device either. In NNS11, a new finite volume approach was investigated to allow larger time steps while reducing the typical finite volume communication frequency. Here, we give an example of tailoring that algorithm specifically for GPU-accelerated hardware. Our goals are keep the large time steps and reduced communication of NNS11 while increasing compute intensity and data reuse.

17.2 Numerical Methods

17.2.1 Equation Set

In NNS11, a two-dimensional, compressible, non-hydrostatic model is used (essentially the Euler system of equations) which explicitly conserves mass, momentum, and a thermodynamic variable of meteorological relevance called potential temperature. A Cartesian rectangular grid is used for spatial discretization. The equation set is as follows:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathbf{H}(\mathbf{U})}{\partial z} = \mathbf{S} \tag{17.1}$$

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho w \\ \rho\theta \end{bmatrix}, \ \mathbf{F}(\mathbf{U}) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u w \\ \rho u\theta \end{bmatrix}, \ \mathbf{H}(\mathbf{U}) = \begin{bmatrix} \rho w \\ \rho w u \\ \rho w^2 + p \\ \rho w\theta \end{bmatrix}, \ \mathbf{S}(\mathbf{U}) = \begin{bmatrix} 0 \\ 0 \\ -\rho g \\ 0 \end{bmatrix} \tag{17.2}$$

where ρ is the density, u is the horizontal wind, w is the vertical wind, p is the pressure, and θ is the potential temperature which is related to the actual temperature, T, by $\theta = T\left(p_0/p\right)^{R_d/c_p}$. The equation set is closed by the equation of state: $p = C_0\left(\rho\theta\right)^{\gamma}$ where the constant C_0 is defined by: $C_0 = R_d^{\gamma} p_0^{-R_d/c_v}$. The constants

are $\gamma = c_p/c_v \approx 1.4$, $R_d = 287\,\mathrm{J\,kg^{-1}\,K^{-1}}$, $c_p = 1004\,\mathrm{J\,kg^{-1}\,K^{-1}}$, $c_v = 717\,\mathrm{J\,kg^{-1}\,K^{-1}}$,and $p_0 = 10^5\,\mathrm{Pa}$. Note that when reduced to primitive form (i.e., conserving ρ, u, w, and p), this is identical to the traditional Euler system that conserves energy. Thus, these are equivalent in continuous form. This equation set allows one to simulate the horizontal-vertical interactions in the atmosphere in a simpler 2-D Cartesian setting with test cases that are relevant to atmospheric interests and also offering a numerically challenging testbed for new methods.

17.2.2 Numerical Discretization

We use an explicit finite volume method to solve these equations because we found its flexibility in time discretization and spatial reconstruction to be conducive to our goals. In the finite volume method, the set of conservation laws is first integrated in space over an arbitrary cell domain (the union of cell domains fills the physical domain). Then, the Gauss divergence theorem is applied to the flux divergence integrals, removing the need for spatial derivatives of the fluxes to exist. In the final form, the local time evolution of cell-averaged state variables depends on the flux vectors specified at cell boundaries. Integrating directly in time places the equation set in fully discrete form in which the state variables are updated from time step to time step based on the time-averaged flux through bounding interfaces. It is this time-averaged flux over a time step that forms the central computation of the NNS11 method.

To compute the time-averaged fluxes in an explicit manner, a generalized Riemann solver that allows time steps larger than CFL $= 1$ is used. The CFL (or Courant-Friedrichs-Lewy) number is $CFL = c_{max}\Delta t\left(\Delta x\right)^{-1}$ where c_{max} is the maximum wave speed admitted by the equation set, Δt is the time step, and Δx is the grid spacing. Most time-explicit methods cannot simulate stably beyond CFL $= 1$. This solver could be thought of as a flux vector analog of the solver presented in Ahmad and Linderman (2007), itself being an adaptation of the f-waves solver in Leveque (2002). It can stably simulate at larger time steps though the maximum stable time step past CFL $= 1$ still depends on the flux Jacobian gradient. Any spatial reconstruction can be used with the method as well. We used the WENO (Weighted Essentially Non-Oscillatory) interpolation method of Capdeville (2008), and because of its expense we pre-computed the reconstructions, sampling them when an interpolation was needed. The choice to pre-compute turned out to be a suboptimal choice for both CPUs and GPUs, which will be demonstrated in the next section. For details on the integration method, please see NNS11.

17.2.3 Reducing Memory Accesses and Synchronization

Here we create a rough map of the memory dependence in updating the state variables in a cell in one direction. Consider n to be the number of variables per cell, and c to

be the ceiling of the CFL number. For a single cell, $5n$ floats (floating point values) are retrieved from main memory to compute the WENO reconstructions (because of a 5-cell stencil), and $5n$ floats are then written back to memory (five moments per reconstruction). Next, for an interface, $5nc$ floats are read in from memory to compute the flux from reconstructions, and n floats are then written to memory. Finally, to update a cell, $2n$ values (fluxes from bounding interfaces) are read from memory, and n values per cell back to memory. Thus the volume of memory accesses (reads and writes) needed by one cell is roughly $(14 + 5c)n$ per cell in a given direction.

We propose a modification of the method in NNS11 which would reduce the volume of memory accesses. Suppose the WENO reconstructions are not pre-computed, and instead standard polynomial interpolation is computed on-the-fly. Hyperdiffusion (even-ordered derivatives of fourth order and above) could be applied after the fact to damp oscillations. We would then need $(4 + c)n$ values read in from memory to compute the fluxes with n values written back out to memory. Then, $2n$ values are read in to compute the update, and n values are written back out in the update. For sixth-order hyperdiffusion or lower, $7n$ cells must be read in, and then n cells must be written back to update state variables. This gives total memory requirements of $(16 + c)n$ per cell per direction.

For CFL values of 1, 2, and 3, the modification reduces memory accesses per cell by 11 %, 15 %, and 34 %, respectively. Without the post-hoc hyperdiffusion (e.g., WENO reconstructions are pre-computed on a per-block basis), for CFL values of 1, 2, and 3, memory accesses would be reduced by 53 %, 58 %, and 62 %, respectively.

Though this analysis is not exactly representative of true memory access counts, it demonstrates how a minor modification in the algorithm reduces memory dependence. In addition, this modification removes the synchronization point between the reconstruction and flux kernel launches, clustering more computation between memory accesses.

17.3 Results

We refer to the method of NNS11 as the "Original" formulation and the modification to reduce memory accesses given in Sect. 17.2.3 as the "Modified" formulation. We use only the rising thermal test case of NNS11 herein and focus only on computational aspects, leaving accuracy to future publication. For specification of these test cases and for results from the Original formulation, please refer to NNS11.

17.3.1 Implementing the Multi-GPU Code

In coding for GPUs, we chose the CUDA for C language by Nvidia. The principal efficiency constraint for memory handling with GPUs on current architectures is that data should be communicated between main memory and GPU memory as little as

possible. For this reason, all resident data (reconstructions, fluxes, and state variables) remains in the GPU memory at all times. Data is transferred between system and GPU memory spaces only for file I/O and halo exchanges. Halo exchange consume negligible run time because the data volume is small.

There were five functions in the Original formulation's C code that required conversion to CUDA kernels: reconstruction of state variables, computation of interface fluxes, updating of cell-mean state variable based on fluxes, application of the gravity source term, and maintenance of boundary values. Note that even if a kernel gains little to no speedup, conversion is still worthwhile to avoid memory transfers over the PCI Express bus. We show C code from one of our simpler functions in Listing 1 and corresponding CUDA code in Listing 2 to illustrate the simplest loop transformation guaranteeing all of the work will be done regardless of the kernel launch specifications. This transformation does not permit efficient use of shared memory in general because there is no longer a guaranteed unique thread-to-cell mapping. This means that one cannot know a priori in writing the kernel how to manage the layout of data from global memory into shared memory.

Listing 3 gives an alternative CUDA kernel that gives a better context for using shared memory since there must be a unique cell-to-thread mapping for it to work properly. This kernel, however, requires that the block and grid dimensions specified during kernel launch be large enough to ensure that there are enough threads to span all of the cells. We do not make use of shared memory at the moment because of complications when the block size is not an exact multiple of the cells or interfaces.

On the machine used to run tests for this study, there are two Nvidia GTX 480 cards with the GF100 architecture containing 16 streaming multiprocessors (SMs) with 32 streaming processors each. The GTX 480 disables one of the multi-processors to give 480 cores total per card. Each card has 1.5 GB of DRAM aided by a 768 kB L2 cache. Each SM has 32×2^{10} registers available to spread among resident threads and 64 kB of on-chip memory which can be partitioned either into 16 kB of L1 cache and 48 kB of shared memory or 48 kB of shared memory and 16 kB of L1 cache. We currently give preference to the L1 cache since we are not yet using shared memory.

To spread the problem over two GPUs on the same machine with CUDA, one must map one GPU per host (CPU) code thread. This leaves two options for host parallelization: OpenMP and MPI. An OpenMP code usually parallelizes only the dense sections of computation which would benefit from threading. Because of the unique host thread to GPU device mapping required by CUDA, however, it is not yet clear to us how to handle this among multiple parallel sections in the code except to place the entire code in a single OpenMP parallel section. Since the code will need to be spread across nodes anyway, we consider using only MPI to give the most benefit for development effort. There are overheads associated with using MPI to partition work within a single node, which we accept. Due to using only two GPUs, we currently decompose the domain only in the x-direction, as it contains the most cells in the tests cases herein.

Listing 17.1 C function to update cell means based on interface fluxes in the x-direction. STATE and FLUX are function macros. GS is the number of so called ghost cells which maintain boundary conditions

```
1  void update_x() {
2     int i, j, k;
3     for (i = GS; i < NXC+GS; i++)
4         for (j = GS; j < NZC+GS; j++)
5             for (k = 0; k < 4; k++)
6                 STATE(i,j,k) -= DT / DX * (FLUX(i+1-GS,j-GS,k) -
7                                            FLUX(i  -GS,j-GS,k));
8  }
```

Listing 17.2 CUDA kernel to update cell means based on interface fluxes in the x-direction that guarantees all work is done regardless of kernel launch specifications. STATE and FLUX are function macros. GS is the number of so called ghost cells which maintain boundary conditions

```
1   _ _global_ _ void update_x(fpprec *state, fpprec *flux, int numx) {
2       int i, j, k, ii, jj;
3       ii = blockIdx.x*blockDim.x+threadIdx.x;
4       jj = blockIdx.y*blockDim.y+threadIdx.y;
5       for (i = GS+ii; i < NXC+GS; i+=gridDim.x*blockDim.x)
6           for (j = GS+jj; j < NZC+GS; j+=gridDim.y*blockDim.y)
7               for (k = 0; k < 4; k++)
8                   STATE(i,j,k) -= DT / DX * (FLUX(i+1-GS,j-GS,k) -
9                                              FLUX(i  -GS,j-GS,k));
10  }
```

Listing 17.3 CUDA kernel to update cell means based on interface fluxes in the x-direction which allows efficient use of shared memory. STATE and FLUX are function macros. GS is the number of so called ghost cells which maintain boundary conditions

```
1   _ _global_ _ void update_x(fpprec *state, fpprec *flux, int numx) {
2       int i, j, k, ii, jj;
3       ii = blockIdx.x*blockDim.x+threadIdx.x;
4       jj = blockIdx.y*blockDim.y+threadIdx.y;
5       i = GS+ii;
6       j = GS+jj;
7       if (i >= NXC+GS || j >= NZC+GS) return;
8       for (k = 0; k < 4; k++)
9           STATE(i,j,k) -= DT / DX * (FLUX(i+1-GS,j-GS,k) -
10                                     FLUX(i  -GS,j-GS,k));
11  }
```

17.3.2 Run Time Comparison

The Modified code differs from the Original code in two places. First, reconstruction is changed to standard fifth-order polynomials and is included within the flux routine, computed on the fly. Second, a hyperdiffusion kernel was added to damp oscillations. Counting operations in the reconstruction and hyperdiffusion portions of the code

Table 17.1 Run times in seconds of the Original and Modified codes in simulating the convective thermal test case for 100 s for three CFL values, two problem sizes (640 × 320 cells and 1280 × 640 cells), and several computational setups

Max. CFL	640 × 320					1280 × 640				
	Double precision			*Single precision*		*Double precision*			*Single precision*	
	CPU	GPU	2×GPU	GPU	2×GPU	CPU	GPU	2×GPU	GPU	2×GPU
	Original					Original				
1	138.94	30.64	16.73	21.73	11.71	1114.67	240.31	124.90	164.55	84.82
2	101.33	20.19	10.97	13.28	7.20	819.32	157.18	81.15	100.39	51.94
3	88.90	16.97	9.09	10.67	5.75	721.83	131.54	67.18	80.54	41.15
	Modified					Modified				
1	102.06	16.91	9.27	9.54	5.46	861.48	128.08	66.56	68.75	36.03
2	84.60	12.73	7.04	7.01	4.03	708.07	97.75	50.44	50.59	26.28
3	81.27	11.97	6.53	6.63	3.73	676.92	91.01	46.82	48.86	25.48

The GPUs are Nvidia GTX 480 cards, and the CPU is an Intel Core i7 960 (quad-core 3.2 GHz)

only, the Modified formulation has 18 % fewer operations for CFL ≤ 1, 42 % more for $1 < \text{CFL} \leq 2$, and 96 % more for $2 < \text{CFL} \leq 3$. The run times will not follow these percentages because of compilation optimizations, memory fetching time, instruction level parallelism, and that the counts are only for reconstruction and hyperdiffusion portions of the code. Considering only the amount of computation, the Modified version should only run modestly faster for CFL ≤ 1, and for larger time steps, it should run significantly slower. We hypothesize that reducing the number of memory accesses will outweigh additional computation.

Table 17.1 shows run time results for the Original and Modified codes running a convective thermal test case for 100 model seconds over a variety of CFL numbers and computational setups. No file I/O is performed during these runs. The CPU code was parallelized with OpenMP using the 8 threads available on the Intel i7 960 processor with full optimizations and SSE enabled on Intel's icc version 11.1.073 compiler for a fair CPU-GPU comparison. There is a 1.07× to 1.36× speed-up even on the CPU associated with the modification (Original run time/Modified run time) with the larger speed-ups for lower CFL values. For GPUs, speed-up associated with the modification is 1.45× to 1.81× in double precision and 1.65× to 2.28× in single precision.

For the 640 × 320 cell problem size, the Original code gave speed-ups associated with going from CPU to GPU of 4.53× to 5.24× with the larger speed-ups for higher CFL values (all double precision). This follows a pattern we have observed in the past: the more the work, the greater the speed-up in going from CPU to GPU. For the modified version, the speed-ups in going from CPU to GPU were 6.04× to 6.79×. For the larger problem size, the Original code speed-ups were 4.64× to 5.49×, and the Modified code speed-ups were 6.72× to 7.44× (all double precision for 1 GPU).

In going from double precision to single precision, the associated speed-ups vary between 1.46× and 1.93× and without any obvious pattern for this small sample size. Speed-ups were, however, larger and less variable for the Modified code. Also,

the speed-up associated in going to from one GPU to two GPUs ranges from 1.74× to 1.96× largely depending upon problem size (i.e., computation to communication ratio). Speed-ups would probably be larger with an OpenMP implementation, but we chose MPI for the development effort and flexibility. Another factor preventing these speed-ups from reaching the optimal 2× is that when split, there is less work per GPU. Overall speed-ups in going from the CPU to single precision on one GPU ranged from 6.39× to 8.96× for the Original code and 10.70× to 14.00× for the Modified code. Please note that these results use very large problem sizes per GPU, and therefore, these form a sort of upper bound on what would be expected with these kernel implementations. In reality, especially for high-throughput climate codes, the problem size per GPU will be lower, meaning that speed-ups are expected to be lower because computation-to-communication ratio is lower.

17.3.3 GPU Profiling Results

We profiled the Original and Modified codes for CFL ≤ 1 using the Nvidia Compute Visual Profiler tool. The flux and reconstruction kernels together took up 92 % of the overall GPU run time for the Original code and the flux and hyperdiffusion kernels took up 89 % of run time for the Modified code. We will therefore restrict profiling discussion to these largest portions of the code. Both codes had a 60–65 % hit rate in the L2 cache when reading from global memory. What is alarming, however, is that the Modified code had a hit rate of 99.2 % in fast on-chip L1 cache for read requests while the Original code had a hit rate of only 57.4 %. This means that as is, there is less merit in using shared memory explicitly for the Modified code because nearly all values are coming from on-chip memory anyway. This does not address potential banking conflicts for the on-chip L1 accesses, however, but coding for shared memory also requires more development cost.

When blocks of threads are first mapped onto a SM during kernel execution, there are inevitably cold-read misses on the L1 cache as values are read in for the first time. Then, given the cache is not overflowed for local data reuse, data will hit the cache each time afterward. Paying attention to three things should help the L1 hit rate increase. First, if the volume of data read in by a kernel from main GPU DRAM fits into L1 cache in a given block, the hit rate will be high after the round of cold misses. Second, if successive threads in a warp reuse the same data, overall data requirements of the block are greatly reduced. Even if data does not fit into L1 cache across the block, threads reusing successive data increases the chance of an L1 cache hit due to temporal locality. Third, global synchronization (separate kernel launches) forces a round of cold misses, reducing the L1 hit rate.

In the Modified code, we specify a 16×16 block dimension for flux computations, and there is a halo of three cells past the boundary interfaces for CFL ≤ 1 in one dimension. Therefore, we need $\left(15 + 3 \times 2\right) \times 16 = 336$ (15 cells within the 16 interfaces and two 3-cell halo regions on either side) state variable vectors per CUDA thread block. With 4 variables per cell in double precision (8 bytes per float), and

laying 2 blocks on each SM, the Modified code requires 21504 bytes of state variable data which easily fits into the 48 kB of L1 cache (since we specify an L1 cache preference). Therefore, after the round of cold misses, each subsequent read request should hit L1 cache, resulting in the over 99 % L1 hit rate for read requests. The Modified flux kernel (which takes up over 80 % of the total runtime) ran at 25 % of the aggregate compute throughput in both single and double precision, neglecting kernel launch overheads. Given the 99 % L1 cache hit rate on global memory loads, we attribute this mainly to banking conflicts. Given that the code development effort is significantly reduced by using L1 cache rather than tedious and bug-prone shared memory optimizations, we consider this an acceptable throughput.

17.4 Conclusions

We have modified the finite volume non-hydrostatic atmospheric solver from NNS11 to achieve better efficiency on GPUs. Specifically, we removed the pre-computation of spatial reconstructions in favor of using cheaper reconstructions on the fly, thus reducing the volume of memory being accessed and eliminating a synchronization point, enabling better clustering of computations. In Sect. 17.3.2, we demonstrated a consistent improvement in run time with the Modified code over the Original code, by more than a factor of two in some cases. Given that the Modified code required more computation at CFL ≥ 2, the merit of improving memory handling is evident.

We began this study with a running list of constraints for GPU efficiency, and we ended with several more, specifically regarding the L1 per-SM cache. We were surprised by the extremely high hit rate on L1 cache for read requests in the Modified code, and we hypothesize that the reasons include the reduction in synchronization; the local thread-to-thread reuse of data within a block; and the low volume of data required for computing a flux, allowing it to fit entirely into the L1 cache. We are therefore led to believe that the development effort might be greatly simplified concerning memory optimizations if one abides by these constraints, allowing the L1 cache to do all of the work itself. This approach may not be applicable to every kernel, but when it is, the development effort will be minimal.

In the future, we wish to remove the need for hyperdiffusion for several reasons. First, it requires a synchronization. Second, for steady-state test cases, hyperdiffusion will usually destroy geophysical balances even when the coefficients are small, making it difficult to test the viability of the method in simplified frameworks. Finally, hyperdiffusion does not distinguish between smooth and non-smooth regions of the flow, damping all with the same coefficient of diffusion. WENO, which is adaptive in nature to the flow smoothness, is a better alternative. We propose, in the future, pre-computing the WENO reconstructions on a per-block basis to keep the memory access counts low.

References

Ahmad N, Linderman J (2007) Euler solutions using flux-based wave decomposition. Int J Numer Methods Fluids 54:47–72

Capdeville G (2008) A central weno scheme for solving hyperbolic conservation laws on non-uniform meshes. J Comput Phys 227:2977–3014

Cullen MJP, Davies T (1991) A conservative split-explicit integration scheme with fourth-order horizontal advection. Q J R Meteorol Soc 117:993–1002

Durran DR (1991) The third-order Adams-Bashforth method: an attractive alternative to leapfrog time differencing. Monthly Weather Rev 119:702–720

Evans KJ, Rouson DW, Salinger AG, Taylor MA, Weijer W, White IJB (2009). A scalable and adaptable solution framework within components of the community climate system model. In: ICCS 2009 proceedings of the 9th international conference on computational science

Gassmann A (2005) An improved two-time-level split-explicit integration scheme for non-hydrostatic compressible models. Meteorol Atmos Phys 88:23–38

Giraldo FX, Restelli M (2008) A study of spectral element and discontinuous Galerkin methods for the Navier-Stokes equations in nonhydrostatic mesoscale atmospheric modeling: equation sets and test cases. J Comput Phys 227:3849–3877

Khairoutdinov M, Randall D, DeMott C (2005) Simulations of the atmospheric general circulation using a cloud-resolving model as a superparameterization of physical processes. J Atmos Sci 62:2136–2154

Klemp JB, Skamarock WC, Dudhia J (2007) Conservative split-explicit time integration methods for the compressible nonhydrostatic equations. Monthly Weather Rev 135:2897–2913

Knoll DA, Keyes DE (2004) Jacobian-free Newton-Krylov methods: a survey of approaches and applications. J Comput Phys 193:357–397

Leveque RJ (2002) Finite volume methods for hyperbolic problems. Cambridge University Press, Cambridge

Lin S-J, Rood RB (1997) An explicit flux-form semi-Lagrangian shallow-water model on the sphere. Q J R Meteorol Soc 123:2477–2498

Nair RD, Choi H-W, Tufo HM (2009) Computational aspects of a scalable high-order discontinuous Galerkin atmospheric dynamical core. Comput Fluids 38:309–319

Norman MR, Nair RD, Semazzi FHM (2011) A low communication and large time step explicit finite-volume solver for non-hydrostatic atmospheric dynamics. J Comput Phys 230(4):1567–1584

Staniforth A, Cote J (1991) Semi-lagrangian integration schemes for atmospheric models: a review. Monthly Weather Rev 119(9):2206–2223

Taylor MA, Tribbia JJ, Iskandarani M (1997) The spectral element method for the shallow water equations on the sphere. J Comput Phys 130:92–108

Taylor MA, Edwards J, Thomas S, Nair R (2007) A mass and energy conserving spectral element atmospheric dynamical core on the cubed-sphere grid. J Phys Conf Ser 78:012074

Williamson DL (2007) The evolution of dynamical cores for global atmospheric models. J Meteorol Soc Jpn 85B:241–269

Chapter 18
Investigation of Solving 3D Navier–Stokes Equations with Hybrid Spectral Scheme Using GPU

Ying Xu, Lei Xu, D. D. Zhang and J. F. Yao

Abstract The approach of accelerating application with GPUs already delivers impressive computational performance compared to the traditional CPU. The hardware architecture of GPU is a significant departure from CPUs, hence the redesign and validation of the numerical algorithm are necessary. The spectral-finite-difference schemes usually used in the direction numerical simulation (DNS) for turbulent channel flows are studied here. In order to validate the numerical accuracy, the scalar diffusion equation is first solved with this scheme, and the results from GPU and CPU are validated with the analytical solution. The major computational kernels of the scheme are the fast Fourier transfer (FFT) and the linear equation solver, which are both implemented on GPU. The performance study of the scalar diffusion equation shows at least 20× speedup. For 3D Navier-Stokes equation, the performance on a single Nvidia S2050 card shows 25 times speedup.

18.1 Introduction

Modern graphic cards (GPUs) are very powerful processors, where the peak computational capabilities of the latest GPUs are much higher than the multi-core high-end CPUs. Three out of the top 5 supercomputers in the Nov. 2011s Top 500 List (2012) used GPUs as computational accelerators, such as Tianhe-1A, Nebula, and SUBAME 2.0. Besides GPUs' excellent raw computing power, GPUs also have very attractive metrics for performance per dollar and performance per watt.

Y. Xu (✉) · L. Xu · D. D. Zhang · J. F. Yao
Shanghai Supercomputing Center, 201203 Shanghai, China
e-mail: yxu@ssc.net.cn

L. Xu
e-mail: lxu@ssc.net.cn

D. D. Zhang
e-mail: ddzhang@ssc.net.cn

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 283
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_18,

GPUs and also the many-core CPUs use the shared instruction multiple data (SIMD) paradigm to allow up to hundreds of float point units to operate simultaneously. However not all algorithms can be implemented efficiently using SIMD paradigm, and with the trend of computing hardware, it is expected that scientific application will focus on migrating to parallel multi-core hardware. The programming model GPGPU (general purpose computing on GPU) has risen as a break-through for the large scale numerical computing, and the parallel computing architecture CUDA of NVIDIA (2008) and the parallel programming industry standard OpenCL (Khronos 2009) are both used for various scientific applications, such as physics, cosmology, molecular dynamics.

In this study, we investigate the possibility of accelerating the direct numerical simulations (DNS) for turbulent flows using GPUs. The turbulence DNS has benefited from the development of HPC systems over the past 30 years, but it still remains as a major challenge (Cant 2002). The current state of applying GPUs to computational fluid dynamics (CFD) problem is either simulations for the real time fluid effects where the speed and appearance weight over the numerical accuracy (Stam 1999; Harris et al. 2003; Liu et al. 2004; Crane et al. 2007), or 2D geometries simulations with simple numerical scheme which are not suited for engineering and scientific purposes (Zhao 2008; Li et al. 2003; Kuznik et al. 2010; Obrecht et al. 2010; Goodnight et al. 2003; Bolz et al. 2003). Here we review some previous works of CFD simulations on GPUs.

In the computer graphics where accuracy is not as important as the speed, the method of Stam (1999) are very popular for fluid simulations. Stam's method uses a semi-Lagrangian method which doesn't require very small time steps to solve the Navier-Stokes equation and also shows numerical stability with very large time-steps. This method suffers from too much "numerical dissipation" and hence is not accurate enough for engineering purposes. Many researchers (Harris et al. 2003; Liu et al. 2004; Crane et al. 2007) implemented this method on GPUs. The Lattice-Boltzmann method (LBM) falls in the category of embarrassingly parallel problems, and is simple to implement on both serial and parallel machines. This method also requires significant computational resources, and hence GPUs could greatly accelerate the LBM simulations (Zhao 2008; Li et al. 2003; Kuznik et al. 2010; Obrecht et al. 2010), with the raw computational capability.

The previous works on the acceleration of engineering and scientific CFD on GPUs are not as many as those in Stam's method and LBM. The early works (Goodnight et al. 2003; Bolz et al. 2003) are related to implementing linear operation kernel on GPUs and applying these kernels to solve conjugate gradient method, multi-grid method, and more complex problems like the Navier-Stokes equation. The most notable work of engineering importance is the work of Brandvik and Pullan (2008) who solved an Euler flow in 3D geometry. Elsen et al. (1982) also reported large calculation of flow over hypersonic vehicle on GPUs, where the steady solution of the compressible Euler equations is obtained. The measured speedups are at least 15× relative to the CPU implementation in Elsen's work.

In this study, we focus on using the spectral-finite-difference method to solve three-dimensional Navier-Stokes equations (Moin and Kim 1982; Cortese and

Balachandar 1994; Garg et al. 1997), where the method is often used to simulate the turbulent channel flows. In this method, the second order finite difference is used in the direction orthogonal to the walls, and the Fourier transform is used in the direction horizontal to the wall. The main computational kernels of this problem include the sparse linear equation solver and fast Fourier transform, and both kernels have been implemented on GPUs (NVIDIA 2010; Bell and Garland 2008).

Producing identical results in CPU and GPU implementations of an algorithm and evaluating the accuracy of numerical scheme on GPU are not a simple issue. Even as the exact same sequence of instructions is executed on CPU and GPU processors, it is quite possible for the results to be different (Elsen et al. 1982). To validate results from GPUs with the existing benchmark results, a certain number of grid points are required for 3D Navier-Stokes equations and the memory requirement of this problem often exceeds the amount of memory a single GPU could provide. In this study, we will discuss and implement the main computational kernel of the spectral-finite difference method for 3D Navier-Stokes equations. For the verification of the numerical accuracy, we solve a scalar diffusion problem and compare with the analytical solution to the PDE. This approach will provide us an opportunity to report the numerical errors on GPUs in a quantitative manner.

In Sect. 18.2 we describe the physical problems and the basic equation. The basic implementation of this problem is discussed in Sect. 18.3. The numerical accuracy and performance results are discussed in Sect. 18.4. The conclusion is presented in Sect. 18.5.

18.2 Basic Equations and Numerical Algorithms

The governing equations of motion for an incompressible fluid are shown as follows in a non-dimensional form:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{18.1a}$$

$$\frac{\partial u_i}{\partial t} + \frac{\partial \left(u_i u_j\right)}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i}{\partial x_j^2} \tag{18.1b}$$

where u_i is the instantaneous velocity, p is the pressure, and Re is the Reynolds number. All the primitive variables are non-dimensionalized by the half width of the channel and the steady state centerline velocity, which are the characteristic length and velocity respectively. The flow field is homogeneous along the horizontal direction (x and y), and Fourier transforming Eq. (18.1b):

$$\frac{\partial \widetilde{u}}{\partial t} + \iota k_x \widetilde{uu} + \iota k_y \widetilde{uv} + \frac{\partial \widetilde{uw}}{\partial z} = -\iota k_x \widetilde{p} - \frac{1}{Re}\kappa^2 \widetilde{u} + \frac{1}{Re}\frac{\partial^2 \widetilde{u}}{\partial z^2} \tag{18.2a}$$

$$\frac{\partial \widetilde{v}}{\partial t} + \iota k_x \widetilde{uv} + \iota k_y \widetilde{vv} + \frac{\partial \widetilde{vw}}{\partial z} = -\iota k_y \widetilde{p} - \frac{1}{Re}\kappa^2 \widetilde{v} + \frac{1}{Re}\frac{\partial^2 \widetilde{v}}{\partial z^2} \tag{18.2b}$$

$$\frac{\partial \widetilde{w}}{\partial t} + \iota k_x \widetilde{uw} + \iota k_y \widetilde{vw} + \frac{\partial \widetilde{ww}}{\partial z} = -\frac{\partial \widetilde{p}}{\partial z} - \frac{1}{Re}\kappa^2 \widetilde{w} + \frac{1}{Re}\frac{\partial^2 \widetilde{w}}{\partial z^2} \tag{18.2c}$$

Here $\widetilde{(\cdot)}$ denotes the primitive variables in Fourier space and $\kappa_x^2 + \kappa_x^2 = \kappa^2$. The second-order Crank-Nicolson schemes are employed for the spatial derivatives in the non-homogeneous direction (here z- direction). The aliasing error arising in the pseudo-spectral treatment of non-linear terms are removed using the 2/3 rule. The RK3 method is applied to the calculation of the convective term and the horizontal diffusion term. To ensure the flow field divergence free, the pressure Poisson equation is solved as follows:

$$\frac{\partial \widetilde{\Phi}}{\partial z^2} - \kappa^2 \widetilde{\Phi} = \iota k_x \widetilde{u}^* + \iota k_y \widetilde{v}^* + \frac{\partial \widetilde{w}^*}{\partial z} \tag{18.3}$$

where $\widetilde{u}^*, \widetilde{v}^*, \widetilde{w}^*$ is the velocity field solved from Eq. 18.2. The velocity after divergence correction is

$$\widetilde{u} = \widetilde{u}^* - \iota\, \kappa_x \widetilde{\Phi}, \tag{18.4a}$$

$$\widetilde{v} = \widetilde{v}^* - \iota\, \kappa_y \widetilde{\Phi}, \tag{18.4b}$$

$$\widetilde{w} = w^* - \frac{\partial \widetilde{\Phi}}{\partial z}. \tag{18.4c}$$

The pressure Poisson equation is solved with implicit Euler scheme and the incompressible constraint can be enforced at the new time step.

To solve the 3D Navier-Stokes equation, the memory requirement is huge and usually requires a distributed memory machine. To validate the numerical accuracy, a certain number of grid points must be maintained. For example, the DNS results of early works used $192 \times 160 \times 192$ grids Kim et al. (1987), and the largest simulation size of the isotropic turbulence is 14336^3 (http://www.nscc-tj.gov.cn/example/example_12.asp (2012/05)). NVIDIA's Fermi S2050 offers 2.6 GB memory for computation. To investigate the numerical accuracy of this scheme on GPU, a simpler case is studied first. We choose to solve a scalar diffusion equation using the spectral-finite-difference scheme. The analytical solution exists for this type of PDE with certain boundary conditions and initial conditions, which can be used to assess the numerical accuracy of GPU implementation. With the convective terms absorbed in the source terms, the 3D scalar diffusion problem can be discretized to a set of sparse linear equation systems which are similar to the discretized equations of Eqs. 18.2, 18.3.

The scalar diffusion equation is defined as follows

$$\frac{\partial \Phi}{\partial t} = -\Gamma \nabla^2 \Phi + g\,(x, y, z, t) \tag{18.5}$$

where Γ is the diffusion coefficient. Using fast Fourier transfer (FFT) on x and y directions Eq. 18.5 results in:

$$\frac{\partial \widetilde{\Phi}}{\partial t} = -\Gamma \kappa^2 \Phi + \Gamma \frac{\partial^2 \widetilde{\Phi}}{\partial z^2} + g\left(\kappa_x, \kappa_y, z, t\right) \tag{18.6}$$

Using the second central difference scheme and Crank-Nicolson second-order implicit method for Eq. 18.6 becomes:

$$\frac{\widetilde{\Phi}_k^{n+1} - \widetilde{\Phi}_k^n}{\Delta t} = \frac{\Gamma}{2}\left[\frac{\partial^2 \widetilde{\Phi}_k^{n+1}}{\partial z^2} + \frac{\partial^2 \widetilde{\Phi}_k^n}{\partial z^2}\right] - \frac{\Gamma}{2} k_2 \left(\widetilde{\Phi}_k^{n+1} + \widetilde{\Phi}_k^n\right) + \widetilde{g}\left(\kappa_x, \kappa_y, z, t\right) \tag{18.7}$$

where Φ^n is the value at nth time step. With periodic boundary condition imposed on all three directions, Eq. 18.7 can be reduced to a set of five-banded equations for each pair of $\left(\kappa_x, \kappa_y\right)$, where the linear equations are of size Nz (number of grid points along z direction. For the temporal integration, RK3 is used.

With certain boundary and initial conditions, the analytical solution exists for Eq. 18.6. In this study, the problem described below is chosen as the test problem, and the analytical solution to the PDF is used to verify the numerical accuracy on both GPU and CPU. The three-dimensional computational domain is set as a cubic box with each side as 2π. The source g(x, t) in Eq. 18.5 is set to be zero. The initial condition of the scalar field is set as

$$F(x) = \frac{1}{\left(2\pi\sigma^2\right)^{3/2}} \exp\left[-\frac{||x - x_c||^2}{2\sigma^2}\right], \tag{18.8}$$

where σ is a constant and x_c is set at the center of the domain. With the initial condition function prescribed in Eq. 18.8, the analytical solution can be derived from the general solution of Eq. 18.6 with respect to the prescribed initial condition:

$$F(x, t) = \frac{1}{\pi\left(4\Gamma t + 2\sigma^2\right)^{\frac{3}{2}}} \exp\left[-\frac{||x - x_c||^2}{4\Gamma t + 2\sigma^2}\right]. \tag{18.9}$$

18.3 Implementation of Numerical Solver on GPU

To map the algorithm on GPUs it is necessary to classify algorithm into computational kernels and the data dependence of the computational kernels. The major computational kernels of the spectral-finite-difference scheme are:

1. two dimensional fast Fourier transform (FFT),
2. sparse linear equation solver.

The five-banded sparse linear equations are often of size $\mathcal{O}\left(10^2 \sim 10^4\right)$ and diagonal dominant. In this study, two linear equation solvers are implemented: (i) Gauss-Seidel method (referred as GS method hereafter), and (ii) conjugate gradient (CG) method (referred as CG method). The GS method is inherently sequential, where each iteration result depends on the previous iteration, and hence it is not suitable to be parallelized using SIMD paradigm. The CG method is more adapted to the shared memory machine, where the data communication only occurs at the neighboring threads. LU decomposition as a general method to solve linear equation is not used here. This is because the equations solved here are sparse, the operation account of LU method is of order $\mathcal{O}\left(N_z^3\right)$ while the iteration methods, such as GS and CG methods, have the operation account of order $\mathcal{O}\left(N_z\right)$.

The spectral-finite-difference scheme has a special data parallelism, where the total of $N_x \times \left(N_y + 2\right)$ linear equation system is independent from each other for each $\left(\kappa_x, \kappa_y\right)$ wavenumber pair. The most efficient way to use GPU threads in parallel is to solve the linear equation for each $\left(\kappa_x, \kappa_y\right)$ pair at the same time. According to CUDA memory and programming paradigm, each GPU thread has its own local register and memory. GPU threads are grouped in to block, where the shared memory is available to all the threads in that block. The GPU thread blocks are organized into grids, where each thread can access the global memory on GPU. The coefficients of the sparse linear equation systems are generated on the local memory and register attached to each GPU thread with wavenumber module κ_2 as the input coefficient, and then Gauss-Seidel method is used to solve the linear equation system. This implementation method utilizes the data parallelism of the algorithm and the CUDA memory model and programming paradigm to a full extend. However the restriction is that the local data required by the kernel function need to be fitted in the local storage space on GPU threads (16 KB for NVIDIA S2050). It is found that the size of the linear equation system should be less than 256 on NVIDIA S2050 for the GS method. In this study, the Gauss-Seidel method is implemented on GPUs using the parallelism method discussed above, which is referred as "Implementation I" in Sect. 18.4.

We could also apply the parallelism discussed above to CG method. Due to the multiple auxiliary variables used in the CG method, the size of the linear equation system that can be solved on a single GPU thread is found to be much smaller than that of the GS method. From the performance profiling of the CG method on CPU, the sparse matrix vector multiplications take up most of the computational time. To map CG method to GPU, we choose to accelerate the performance by using CUSPARSE library from NVIDIA (2010). This implementation is referred as "Implementation II" in Sect. 18.4.

18.4 Numerical Accuracy and Performance Results

In this section, the validation of numerical accuracy for the scalar diffusion equation Eq. 18.5 is presented first. The performance study of 3D Navier-Stokes equation is analyzed in details. All the tests are executed on the quad-core AMD Barcelona

Table 18.1 The numerical accuracy and performance results for Eq. 18.6 on CPU

Problem size	FFT (s)	Linear equation solver (s)	Total time (s)	Maximum error
64^3	1.52×10^{-2}	3.85×10^{-2}	0.0534	2.408×10^{-4}
96^3	2.34×10^{-2}	0.131	0.154	1.079×10^{-4}
128^3	0.135	0.301	0.436	6.089×10^{-5}
160^3	0.276	0.545	0.821	3.902×10^{-5}
192^3	0.464	0.938	1.403	2.712×10^{-5}
240^3	0.950	1.699	2.649	1.737×10^{-6}

CPU at 1.9 GHz and the tests on CPU have been built with 64-bits PGI FORTRAN compiler 8.0-5 with basic optimization –O turned on. The GPU platform is NVIDIA Tesla S2050 GPU blade server, and the host CPUs are also quad-core AMD Barcelona CPU. CUDA version is 4.1.

We now compare the scalar diffusion test case on a single CPU core and one GPU card. Table 18.1 presents the results on CPU as the reference for numerical accuracy and performance. It is observed that maximum error $\| \cdot \|^{\infty}$ on CPU is around 10^{-6}–10^{-4} for six problem sizes when compared with the analytical solution Eq. 18.9. The numerical results are computed after 100 steps of iterations. For the largest grid size 240^3, the maximum error is around 10^{-6}, As the grid resolution increases, the numerical error also decreases correspondingly. However the error is still high compared to the analytical solution. This is because the Green function used in the general solution for Eq. 18.5 is actually for the infinite computation domain with zero boundary conditions. With the Fourier transforms imposed in x and y directions, the periodic boundary conditions are imposed to simulate the infinite computational domain. Since the boundary conditions of the analytical solution are different from the numerical solver imposed, the errors in Table 18.1 are considered to be in a reasonable range. If the L2 norm is used, the numerical error compared with the analytical solution is around 10^{-8}.

The total computational time in Table 18.1 is measured for each simulation time step, which includes Nz forward FFTs, Nz backward FFTs and the linear equation of size Nz solved Nx × (Ny + 2) times. A total of 100 steps are performed and the average is taken for the measured time. Since the iteration methods, GS and CG, are used here, even with the same convergence criteria, the numbers of iteration for these two methods might vary, and the total floating point operations cannot be counted exactly. In this study, we only compare the computational time measured for each time step.

In Sect. 18.3 two implementation methods for solving linear equation systems are discussed. One is to use Nx × (Ny + 2) GPU threads to solve the linear equation system in parallel ("Implementation I"). The other is to solve the equation with conjugate-gradient method using CUSPARSE library ("Implementation II"). The numerical error and the wall time for computational kernels are listed in Tables 18.2 and 18.3. Since CUFFT library is used for forward and backward FFT in the algorithm, the wall time for FFT in Tables 18.2 and 18.3 (second column) is almost the same.

Table 18.2 The numerical error and computational time for "Implementation I" on NVIDIA S2050

Problem size	FFT (s)	Linear equation solver (s)	Total time (s)	Speedup	Max error compared with analytical results	Max error compared with CPU results
64^3	3.23×10^{-3}	3.96×10^{-3}	7.20×10^{-3}	7.46	2.407×10^{-4}	3.797×10^{-14}
96^3	6.19×10^{-3}	7.85×10^{-3}	1.40×10^{-2}	16.68	1.079×10^{-4}	5.018×10^{-14}
128^3	6.42×10^{-3}	1.60×10^{-2}	2.25×10^{-3}	18.77	6.089×10^{-5}	8.127×10^{-14}
160^3	9.68×10^{-3}	3.07×10^{-2}	4.04×10^{-3}	17.77	3.902×10^{-5}	5.240×10^{-14}
192^3	1.60×10^{-2}	4.78×10^{-2}	6.38×10^{-3}	19.60	2.712×10^{-5}	7.127×10^{-14}
240^3	3.01×10^{-2}	9.66×10^{-2}	0 .127	17.59	1.736×10^{-6}	4.996×10^{-14}

Table 18.3 The numerical error and computational time for "Implementation II" on NVIDIA S2050

Problem size	FFT (s)	Linear equation solver (s)	Total time (s)	Speedup	Max error compared with analytical results
64^3	3.35×10^{-3}	4.06	4.07	1.33×10^{-2}	2.408×10^{-4}
96^3	7.09×10^{-3}	6.60	6.61	2.81×10^{-2}	1.053×10^{-4}
128^3	6.32×10^{-3}	10.55	10.55	4.13×10^{-2}	6.092×10^{-5}
160^3	1.02×10^{-3}	16.21	16.21	5.06×10^{-2}	3.915×10^{-5}
192^3	1.55×10^{-2}	22.15	22.16	6.32×10^{-2}	2.658×10^{-5}
240^3	2.94×10^{-2}	30.33	30.36	8.73×10^{-2}	2.011×10^{-6}

For "Implementation I", Gauss-Seidel method is executed in parallel on multiple GPU threads, and the measured time for linear equation solver is around 10^{-2} (as seen in Table 18.2 column 3), which takes up 33 % of the total computational time. For grid size 192^3, "Implementation I" for the scalar diffusion problem shows 19.6 times speedup when compared with CPU time. Even for grid size 96^3, the speedup is found to be 9.4 for "Implementation I".

The wall time for GPU "Implementation II" is larger than that from CPU wall time for all six problem size tested, as seen in Table 18.3. If splitting the total computational time into two parts, it is found the CUFFT library speeds up the computational time 2 to 5 times on GPU. The CG algorithm with CUSPARSE library takes up more than 95 % of the total wall time, and the GPU wall time is much larger compared to the CPU wall time. The reasons for no speedup with GPU Implementation II are: (i) the size of the linear equation system tested here is of order 10^2. With CUSPARSE library used, it is found that GPU Implementation II has better performance for larger problem size ($\sim 10^6$); (ii) GPU "Implementation II" solves the multiple linear equation system sequentially, and doesn't take the advantage of the parallelism embedded in the algorithm.

Table 18.4 The performance results for 3D Navier-Stokes equations Eq. 18.2, 18.3 (problem size $128 \times 128 \times 224$)

	FFT (s)	Linear equation solver (s)	Total time (s)	Speedup
GPU	0.129	0.101	0.231	25.8
CPU	3.91	2.05	5.96	1

The GPU implementation I use the same method to solve linear equations, and hence the maximum errors compared with CPU results are found to have 14 significant digits. Compared with the analytical solution, the maximum errors from GPU Implementation I and II are of order $10^{-6}-10^{-4}$, and shows slight difference.

After verifying the numerical accuracy and performance of the scalar diffusion equation on GPU, the 3D Navier-Stokes equation with the spectral-finite-difference scheme is implemented on a single GPU. The problem size chosen here is $128 \times 128 \times 224$, where the memory requirement is around 1GB total. To solve Eqs. 18.2 and 18.3 the "Implementation I" with the Gauss-Seidel method are employed and the performance results are listed in Table 18.4. The speedup for this problem is found to be around 25.8. However the problem size of Gauss-Seidel is limited to the local memory size on each thread, and in this case Nz is found to be less than 224. The calculation of the convective terms on the left-hand-side of Eq. 18.1 takes up around 40% of the computational time on CPU. The convective terms are formed in real space and then transformed to the phase space, where the expensive convolution on can be avoided.

It is also noted that for smaller problem size, all the data required in the computation could be stored on NVIDIA S2050 GPU, thus the data copy between host and GPU memory could be easily avoided. The synchronization of GPU threads is only required at the end of the time step. If the data copy between host and GPU memory, or data storage on the disk, is required, the speedup will be found to be much lower than that reported in this study.

18.5 Conclusions

In this study, we implemented the spectral-finite-difference scheme on GPU and solved the scalar diffusion equation and the three-dimensional Navier-Stokes equation. The scalar diffusion equation is studied here to validate the numerical accuracy of the scheme, since the analytical solution exists for specific boundary conditions. We also investigate the possibility of mapping 3D full Navier-Stokes equations with the GPGPU programming model.

For the scalar diffusion equation, two different types of implementation are studied. One is with Gauss-Seidel kernel executed on GPU threads in parallel, and the other method is to use the conjugate gradient method. It is observed these two implementations have numerical errors of order $10^{-6}-10^{-4}$ when compared with

the analytical solution and the error is around 10^{-14} compared with CPU results. The implementation with Gauss-Seidel method on GPU has speedup around 20 for six different problem sizes, while the implementation with the conjugate gradient method on GPU doesn't accelerate the performance. The performance regression for the CG method is because this implementation only accelerates the performance on GPU for large problem size. Although the implementation with Gauss-Seidel method on GPU accelerates the calculation 20 times, this way of mapping the computational kernels on GPU is not suitable for large size problems. As this method uses the memory local to each thread, the problem size solved should be less than 256. The set of linear equation systems is solved independently on GPU threads. This parallelization scheme fully utilizes the characteristics of the spectral-finite-difference algorithm, and the CUDA programming and memory model.

The three-dimensional Navier-Stokes equation with the spectral finite-difference scheme is solved on GPU. For relative smaller problem size $128 \times 128 \times 224$, the implementation on GPU achieves the speedup of 26 times under the condition that the data transfer between main memory and the GPU memory is minimized. This study demonstrates the use of GPUs for fluid dynamics simulations. It proposes an interesting way of mapping computational kernels onto GPUs and reveals the difficulties of extending to larger problem sizes.

Acknowledgments This work was supported by the National Science Foundation of China under Grant No. 10902063 and National Hi-tech Research and Development Program of China (863 Program) under Grant No. 2012AA01A308

References

Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical report NVR-2008-004, NVIDIA Corporation

Bolz J, Farmer I, Grinspun E, Schröer P (2003) Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: SIGGRAPH'03: ACM SIGGRAPH, ACM, New York, NY, USA, pp 917–924

Brandvik T, Pullan G (2008) Acceleration of a 3D Euler solver using commodity graphics hardware. In: 46th AIAA Aerospace sciences meeting and exhibit, Reno, Nevada, USA, AIAA-2008-607.

Cant S (2002) High-performance computing in computational fluid dynamics: progress and challenges. Phil Trans: Math Phys Eng Sci 360:1211–1225

Cortese TA, Balachandar S (1994) High performance spectral simulation of turbulent flows in massively parallel-machines with distributed memory. Technical report TAM Rep. 765, University of Illinois at Urbana-Champaign

Crane K, Llamas I, Tariq S (2007) In: Real-Time simulation and rendering of 3D fluids, 3rd edn. Addison-Wesley, New York, pp 633–677

Elsen E, LeGresley P, Darve E (2008) Large calculation of the flow over a hypersonic vehicle using a GPU. J Comput Phys 227:10148–10161

Garg R, Ferziger JH, Monismith SG (1997) Hybrid spectral finite difference simulations of stratified turbulent flows on distributed memory architectures. Int J Numer Meth Fluids 24:1129–1158

Goodnight N, Woolley C, Lewin G, Luebke D, Humphreys G (2003) A multigrid solver for boundary value problems using programmable graphics hardware. In: HWWS'03: proceedings of the ACM

SIGGRAPH/EUROGRAPHICS conference on graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland pp 102–111
Harris M, Baxter W, Scheuermann T, Lastra (2003) A Simulation of cloud dynamics on graphics hardware. In: HWWS'03: proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware, pp 92–101
Khronos (2009) OpenCLWorking Group: The OpenCL specification. Specification.
Kim J, Moin P, Moser RD (1987) Turbulence statistics in fully developed channel flow at low reynolds number. J Fluid Mech 177:133–166
Kuznik F, Obrecht C, Rusaouen G, Roux JJ (2010) LBM based flow simulation using GPU computing processor. Comput Math Appl 59:2380–2392
Li W, Wei X, Kaufman A (2003) Implementing Lattice Boltzmann computation on graphics hardware. Vis Comput 9:444–456
Liu Y, Liu X, Wu E (2004) Real-time 3d fluid simulation on gpu with complex obstacles. In: 12th Pacific conference on computer graphics and applications, Seoul, South Korea pp 247–256
Moin P, Kim J (1982) Numerical investigation of turbulent channel flow. J Fluid Mech 118:341–377
NVIDIA Corporation (2008) NVIDIA CUDA Compute Unified Device Architecture Programming Guide 2.0. (2008)
NVIDIA (2010) Cufft library. http://developer.nvidia.com/object/cuda-downloads.html (2010) CUDA Toolkit 3.2
Obrecht C, Kuznik F, Tourancheau B, Roux JJ (2010) A new approach to the lattice boltzmann method for graphics processing units. Comput Math Appl 61(12):3628–3638
Stam J (1999) Stable fluids. In: SIGGRAPH 99 conference proceedings, Annual conference series, pp 121–128
TOP 500 List. http://www.top500.org/lists/2011/06. Accessed April 2012
Zhao Y (2008) Lattice Boltzmann based pde solver on the GPU. Vis Comput 24:323–333

Chapter 19
Correlation of Reservoir and Earthquake by Multi Temporal-Spatial Scale Flow Driven Pore-Network Crack Model in Parallel CPU and GPU Platform

B. J. Zhu, C. Liu, Y. L. Shi and D. A. Yuen

Abstract Coulomb failure assumptions. Jaeger and Cook (Fundamentals of rock mechanics. Methuen, New York, 1969) is used to evaluate the earthquake trigger, and pore pressure (Biot, J Appl Phys 12:155, 1941; 26:182, 1955; 78:91, 1956; J Geophys Res 78:4924, 1973) parts reflect the effect of reservoir which closed to the earthquake slip. Fluid flow driven pore-network crack model (Zhu and Shi, Theor Appl Fract Mech 53:9, 2010) is use to study the reservoir and earthquake. Based on the parallel CPU computation and GPU visualization technology, the relationship between the water-drainage sluice process of the Zipingpu reservoir, stress triggers and shadows of 2008 Wenchuan M_s 8.0 earthquake and porosity variability of Longmenshan slip zone have been analyzed and the flow-solid coupled facture mechanism of Longmenshan coseismic fault slip is obtained.

Keywords Zipingpu reservoir · 2008 Wenchuan earthquake · Coulomb failure stress diffusion · Pore stress diffusion · Fluid flow driven pore-network crack model

B. J. Zhu (✉) · C. Liu · Y. L. Shi
Laboratory of Computational Geodynamics, College of Earth Science,
Graduate University of Chinese Academy of Sciences,
Beijing100049, People's Republic of China
e-mail: cynosureorion@gucas.ac.cn

C. Liu
Laboratoire De Geologie, Ecole Normale Supérieure,
24 Rue Lhomond, Paris CEDEX 575231, France

D. A. Yuen
Minnesota Supercomputer Institute, University of Minnesota,
Minneapolis, MN 55145, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 295
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_19,

19.1 Introduction

A number of factors may contribute to the generation or absence of post-impounding seismicity. Increased vertical stress due to the load of the reservoir and decreased effective stress due to increased pore pressure can modify the stress regime in the reservoir region. The combined effect of increased vertical load and increased pore pressure will have the greatest tendency to increase activity in regions where the maximum compressive stress is vertical (Simpson 1976). Gupta et al. (1972) studied the behavior of earthquakes associated with over a dozen artificial lakes and found that the tremors were initiated or their frequency increased considerably following the lake filling and that their epicenters were mostly located within a distance of 25 km from the lakes.

Zipingpu Key Water Control (2009) project is one of the most complex engineering projects in the world for its located on the most complex earthquake fault slips zone in the world (Maximum acceleration value of seismic oscillation is equal to 0.20 g). Zipingpu reservoir is located on the Longmenshan earthquake fault slip (below 2 km) and the distance between the reservoir and the 2008 Wenchuan M_s 8.0 earthquake initial source within 17 km (Fig. 19.1).

Longmenshan fault slip of 2008 Wenchuan Ms 8.0 earthquake is obtained by GPS & InSAR inversion technique. Shen et al. (2008) (Fig. 19.2), it composed with two slips and cross-wised Zipingpu reservoir zone. The relationship between the pore stress accumulation of Zipingpu reservoir and the trigging and propagation of the Longmenshan coseismic fault slip because very important for it direction effect the dynamic real-time security evaluation and monitor of Zipingpu key water control project.

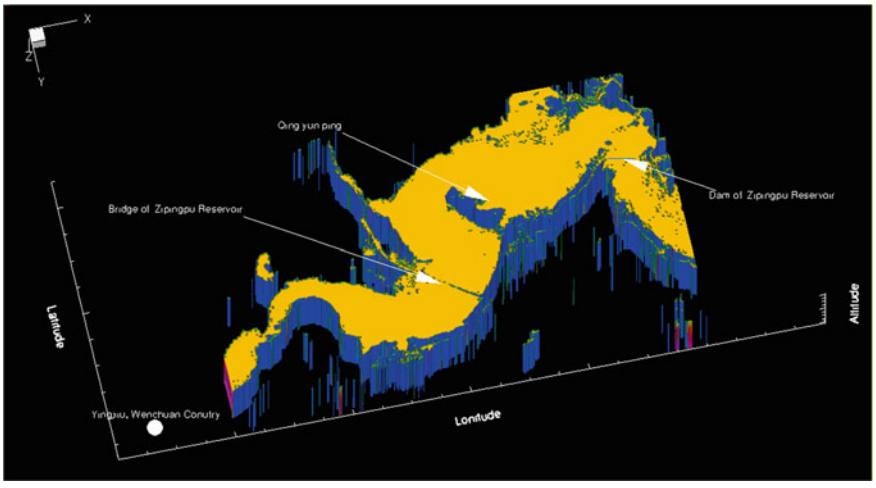


Fig. 19.1 Relatively position between Zipingpu reservoir and Earthquake source of Wenchuan M_s 8.0 earthquake (Zipingpu reservoir E103°30′18" to E103°34′48"; N31°00′36" to E31°03′00"]; Yingxiu[N30°59′58.56"; E103°29′21.12"])

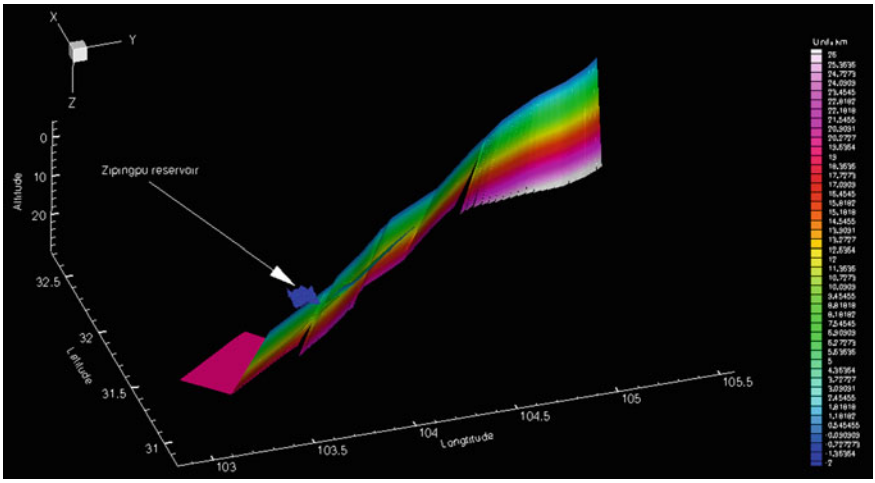


Fig. 19.2 Relatively position between Zipingpu reservoir and Longmenshan fault slip of (Longmenshan fault slip zone [E103.45° to E103.5767°; N30.975° to E31.105°])

Some researches have study 2D coulomb stress caused by reservoir and its effect on the Longmenshan fault (Ge et al. 2008). But the natural problem is rather complex than one scale 2D model, and little research about the 3D coulomb stress analysis under different scale has been done because of the current limitations both practical (computing time) and theoretical (3D flow driven pore-crack network theory (Zhu and Shi 2010), multiple scale fracture mechanics/physics theory (Sih 2006; Sih and Jones 2003; Sih and Tang 2004, 2006, 2008; Sih and Zuo 2000)) aspect.

In this paper, based on the previous work (Zhu and Shi 2010), the relationship between the pore stress accumulation on Zipingpu reservoir and the trigging and propagation mechanism of the Longmenshan coseismic fault slip on scale I and II [**Scale I: 30.976E_31.105E, 103.45N_103.577N; Scale II:30.7E_31.3E,103.05N_103.76N; Scale III:29E_33E,101N_105N; Scale IV: IN plate and EU plate**] have been studied (Fig. 19.3), and the correlation of Zipingpu reservoir and 2008 Wenchuan M_s 8.0 earthquake by fluid flow driven pore-network crack model had been studied.

19.2 Basic Equation

In the present paper, summation from 1 to 3 over repeated lowercase, and of 1–7 in uppercase, basic strain equation for strain porous elastic media can be defined as

$$\varepsilon_{iJ} = \frac{1}{2G}\left\{\sigma_{iJ} - \frac{v}{1+v}\delta_{iJ}\sigma_{kk} + \frac{3(v_u - v)}{B(1+v)(1+v_u)}\delta_{iJ}p\right\}$$

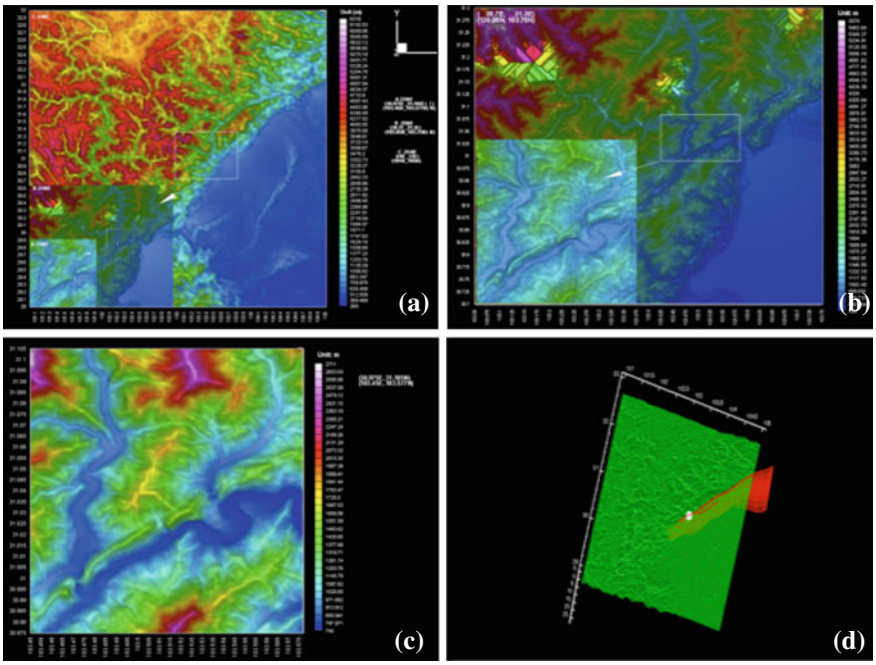


Fig. 19.3 Multiple Scale virtual model of Zipingpu reservoir/Longmenshan coseismic fault slip (**a**) Scale I; **b** Scale II; **c** Scale III; **d** Relatively position reservoir/slip)

$$m - m_0 = \left[\frac{3\rho_0(v_u - v)}{2GB(1+v)(1+v_u)}\right]\left(\sigma_{kk} + \frac{3p}{B}\right)$$
$$q_l = -\frac{\rho_0 \kappa}{\partial x_l}\frac{\partial p}{}$$

where σ_{iJ}, p, ε_{iJ} and m are represent as the total stress, pore pressure, total strain and fluid mass per unit volume of the medium. The parameters G, v, v_u, m_0, ρ_0, q_l, κ, B are represent as the elastic shear modulus (same for drained (p = constant) and undrained (m = constant) condition), drained condition Poisson's ratio, undrained condition Poisson's ratio, the fluid mass content in the unstressed state, mass density of the pore fluid, the mass flux rate per unit area, the permeability and constant which related to drained and undrained status, respectively.

The equations of motion for a homogeneous, linear elastic and isotropic medium can be defined as

$$(c_P^2 - c_S^2)u_{i,ij} + c_S^2 u_{j,ii} + \frac{f_j}{\rho} - \ddot{u}_j = 0$$
$$G_{ij}(P, Q, t) = P_{ij}(P, Q, t) + S_{ij}(P, Q, t) + PP_{ij}(P, Q, t) + SS_{ij}(P, Q, t) + PS_{ij}(P, Q, t) + SP_{ij}(P, Q, t)$$

where $G_{ij}(P, Q, t)$ denote the i component of the displacement at point P due to unite impulsive force at position Q acting j direction at time t.

19.3 Physical Model

As shown in Fig. 19.4, Zipingpu key water control project is located on the upstream of Minjiang river, the maximum reservoir storage capacity is $11 \times 10^9\,\mathrm{m}^3$, the adjustable reservoir storage capacity is $8 \times 10^9\,\mathrm{m}^3$, the normal impounded level is 877 m, the dam top altitude is 894 m and the dam bottom altitude is 728 m. The key water control project began March 3, 2001, stop flow time is November 1, 2002, storage time is December 1, 2004 and completed at December 1, 2006. The total pore stress accumulation time before Wenchuan Ms 8.0 earthquake (May 12, 2008) is 3–4 years. In our physical model, we use the 15000 time steps (10 ts/day) to describe the effect of pore stress of reservoir to the Longmenshan fault slip. From the GPS & InSAR inversion technology, the Longmenshan earthquake fault slip is divided into 673 parts.

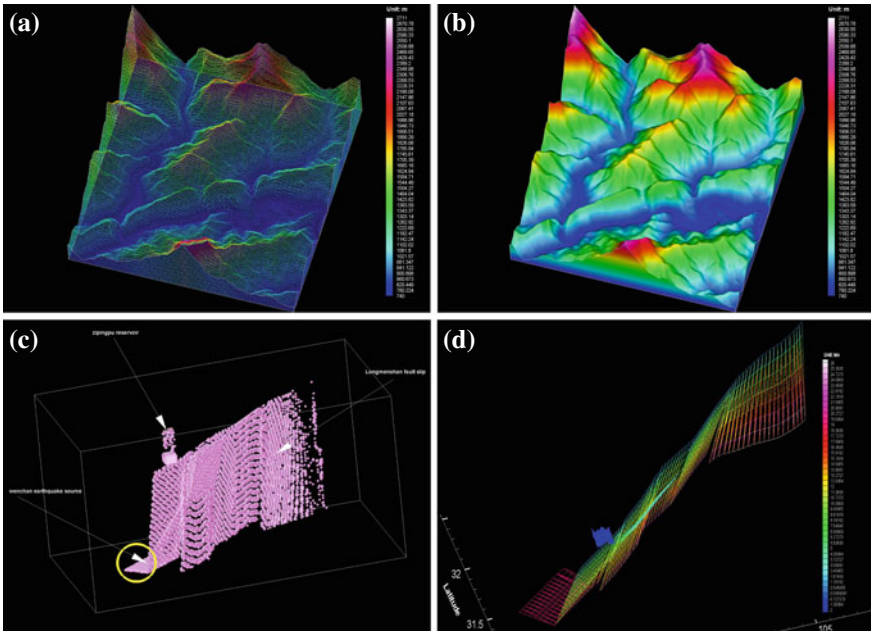


Fig. 19.4 Physical Model Zipingpu reservoir and Longmenshan coseismic fault slip **a** Mesh grid of Zipingpu reservoir; **b** Physical model of Zipingpu reservoir **c** Physical model of Longmenshan fault slip; **d** Detail description of Longmenshan fault slip (composed of 673 parts)

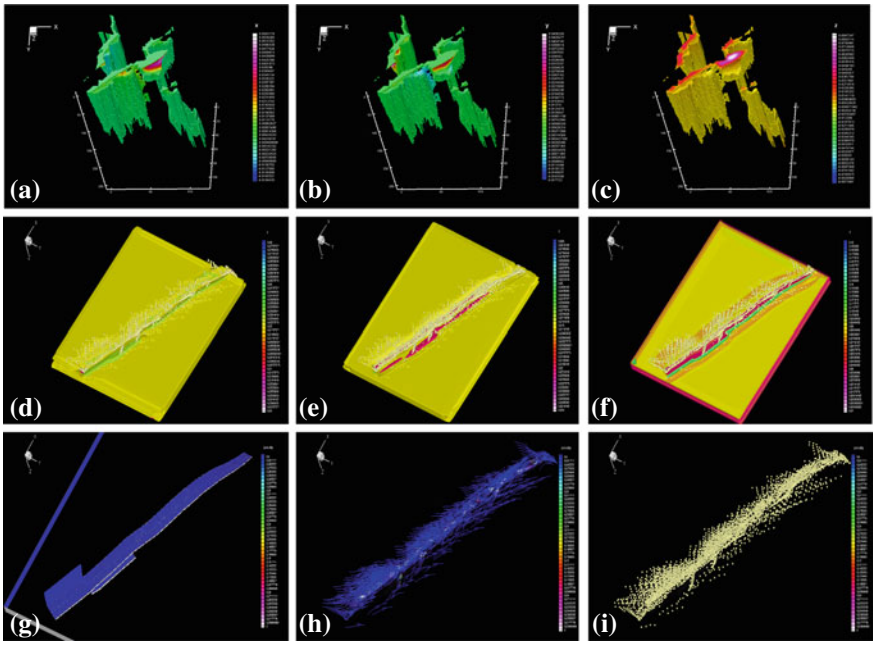


Fig. 19.5 Extended pore strain and stress on Zipingpu reservoir and Longmenshan coseismic fault slip on Scale I **a** Reservoir pore strain in x direction; **b** Reservoir pore strain in y direction; **c** Reservoir pore strain in z direction; **d** Fault slip pore strain in x direction **e** Fault slip pore strain in y direction; **f** Fault slip pore strain in z direction; **g** Fault slip pore stress; **h** Fault slip flow stream trace; **i** Fault slip flow marks

19.4 Numerical Process and Discussion

Figure 19.5 shows that the relationship between extended pore strain and stress on Zipingpu reservoir and Longmenshan coseismic fault slip on Scale I under 20000 ts. The pore stress accumulation value level is 0.3 Mp.

The relationship between extended pore strain and stress on Zipingpu reservoir and Longmenshan coseismic fault slip on Scale II under 20000 ts is shown in Fig. 19.6. In these scale, we can obtained that in the penetration process, if we defined the fault slip as a fluid-saturated elastic porous media, then the vadose energy (caused by pore pressure and can flow to the fault slip tip) is variable with the undrained or drained zone, more energy is released under drained zone than undrained zone; If the fault slip is a stable creep rupturing process, the criteria energy (strain energy function factors) must increase with the speed of faults spreading.

When penetrate reach a stable stage, the fluid flow pore-network crack function became domain, with the time scale increasing, the micro solid-fluid interface will became weak and blur, the macro phenomenon is the porosity become larger, the strain energy can be released to the faults process decreased with the drained spreading increasing.

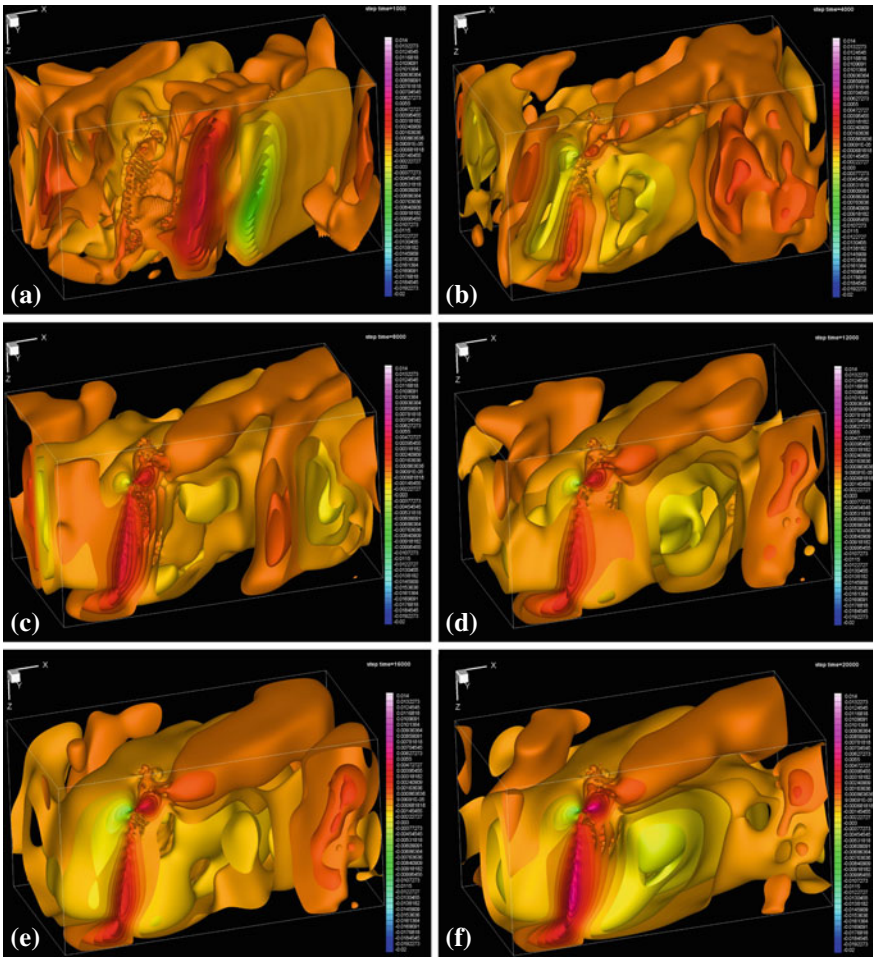


Fig. 19.6 Extended pore strain and stress on Zipingpu reservoir and Longmenshan coseismic fault slip on scale II

The reservoir loading and earthquake trigger relationship is depending on fault slip geometry and character, porosity variability of surrounding geological structure and time and size scale. To Zipingpu reservoir and 2008 Wenchuan earthquake case, porosity and time scale are the key factors.

19.5 Future Work

Because the problem of correlation of reservoir and earthquake is so complex that we can't give a general definite conclusion for all kind of cases by analyze this special case under little changed physical domain scales (Scale I and II) and time domain

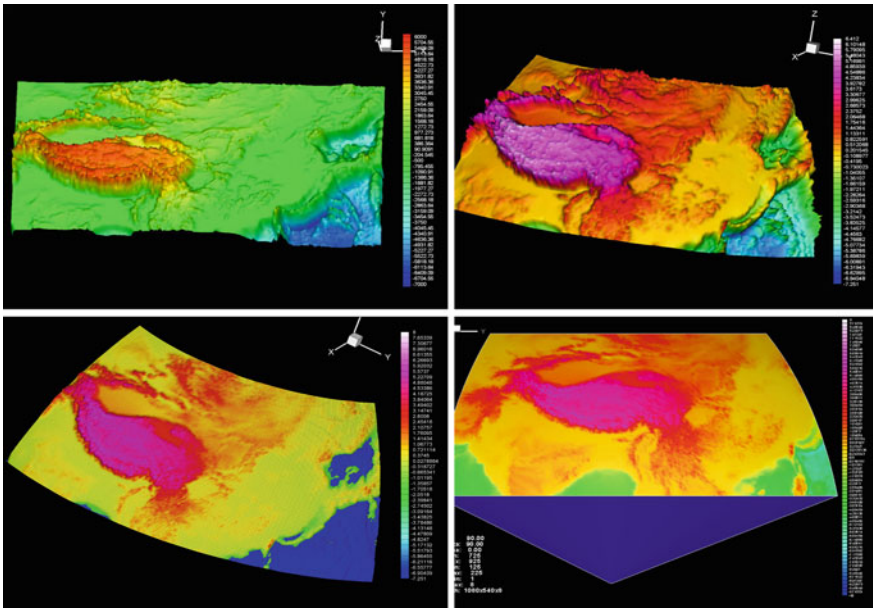


Fig. 19.7 Extended pore strain and stress on Zipingpu reservoir and Longmenshan coseismic fault slip on scale III

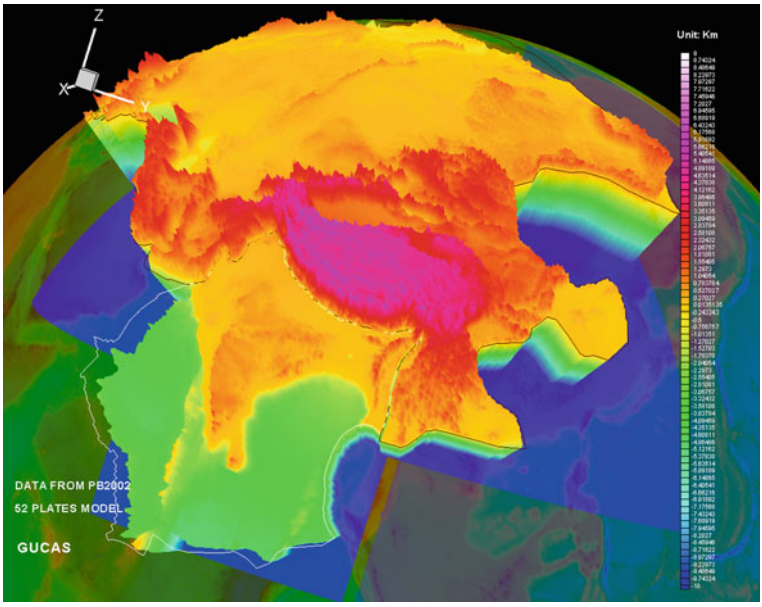


Fig. 19.8 Extended pore strain and stress on Zipingpu reservoir and Longmenshan coseismic fault slip on scale IV

scales (20000 ts). More analysis should be done on multiple physical domain scales and time domain scales. The future work will focus on two things.

Using extended coulomb stress to analyze the flow-solid coupled facture mechanism of Longmenshan coseismic fault slip under larger scale (Figs. 19.7, 19.8). This can provide a combined evaluation of the different effects that can influence reservoir/slip system, and can be further explored to compare the results with other results that have led to negative results (Ge et al. 2008).

References

Biot MA (1941) General theory of three-dimensional consolidation. J Appl Phys 12:155
Biot MA (1955) Theory of elasticity and consolidation for a porous anisotropic solid. J Appl Phys 26:182
Biot MA (1956) General solutions of the equations of elasticity and consolidation for a porous material. J Appl Phys 78:91
Biot MA (1973) Nonlinear and semilinear rheology of porous solids. J Geophys Res 78:4924
Gupta HK, Rastogi BK, Narain H (1972) Common features of reservoir-associated seismic activities. Bull Seismol Soc Am 62:481
Ge S, Liu M, Lu N, Godt JW, Luo G (2008) Did the Zipingpu Reservoir trigger the 2008 Wenchuan earthquake? Geophys Res Lett 36:1
Jaeger JC, Cook NGW (1969) Fundamentals of rock mechanics. Methuen, New York, p 513
Seismic safety evaluation of composite report of Zipingpu Key Water Control Project on Minjiang river in Sichuan provience. Earthquake Prediction Research Institute of the China Seismological Bureau Earthquake Disaster Prevention Center of P.R. China 2009
Sih GC (2006) Multiscale evaluation of microstructural worthiness based on the physical-analytical matching (PAM) approach. Theor Appl Fract Mech 46:243
Simpson DW (1976) Seismicity changes associated with reservoir loading. Eng Geol 10:123
Sih GC, Jones R (2003) Crack size and speed interaction characteristics at micro-, meso- and macro-scale. Theor Appl Fract Mech 39:127
Sih GC, Tang XS (2004) Dual scaling damage model associated with weak singularity for macroscopic crack possessing a micro/mesoscopic notch tip. Theor Appl Fract Mech 42:1
Sih GC, Tang XS (2006) Simultaneous occurrence of double micro/macro stress singularities for multiscale crack model. Theor Appl Fract Mech 46:87
Sih GC, Tang XS (2008) Micro/macro-crack growth due to creep-fatigue dependency on time-temperature material behavior. Theor Appl Fract Mech 50:9
Sih GC, Zuo JZ (2000) Multiscale behavior of crack initiation and growth in piezoelectric ceramics. Theor Appl Fract Mech 34:123
Shen Z-K, Sun J, Zhang P, Wan Y, Wang M, Bürgmann R, Zeng Y, Gan W, Liao H, Wang Q (2009) Slip maxima at fault junctions and rupturing of barriers during the 2008 Wenchuan earthquake. Nat Geosci 2(718):724
Zhu B, Shi YL (2010) Three-dimensional flow driven pore-crack networks in porous composites: boltzmann Lattice method and hybrid hypersingular integrals. Theor Appl Fract Mech 53:9

Chapter 20
A Full GPU Simulation of Evolving Fracture Networks in a Heterogeneous Poro-Elasto-Plastic Medium with Effective-Stress-Dependent Permeability

Boris Galvan and Stephen Miller

Abstract The wide range of timescales and underlying physics associated with simulating poro-elasto-plastic media present significant computational challenges. GPU technology is particularly advantageous to overcome these problems because even though the physics are the same, computational times are orders of magnitude faster. Poro-elasticity could be implemented in GPU, however GPU implementation of plastic stresses pose problems because branching is introduced into the program and thus introduces efficiency penalties. In general any element by element evaluation to deal with branching in GPU is very inefficient. In this paper, we describe fracture evolution in a poro-elasto-plastic medium and use a switch-on/switch-off function to avoid branching, allowing efficient computation of plasticity in GPU. We benchmark for the elasto-plastic part by investigating the angles of developed shear bands, and benchmark the non-linear diffusion part of the code using the method of manufactured solutions. Model results are presented for fluid pressure propagation through an elasto-plastic matrix subjected to compression, and another for extension. The results demonstrate how fluid flow is restricted in the compression case because of the load-induced low permeability, while fluid flow is encouraged in the extensional case because of the extension-induced high permeability. Code performance is excellent in GPU, and we are able to runs months of simulation using time steps of a few seconds within a few hours. With this new algorithm, many problems of couple fluid flow and the mechanical response can be efficiently simulated at very high resolution.

20.1 Introduction

Triggering of earthquakes by high pressure fluids is well documented in enhanced geothermal systems (Häring et al. 2008; Shapiro and Dinske 2009; Audin et al. 2002) and natural environments (Miller et al. 2004; Bols and Nur 2002; Ohtake 1974).

B. Galvan (✉) · S. Miller
Department of Geodynamics and Geophysics, Bonn University,
Bonn, Germany

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 305
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_20,

Injection of over-pressurized fluids into fault zones reduces the frictional resistance, thus lowering of the shear stress necessary to failure (Terzaghi 1923; Nur 1971). Documented cases of fluid-triggered or fluid-assisted earthquake sequences include the Mw = 6.3 1997 Colfiorito (Miller et al. 2004) and Mw = 6.3 L'Aquila (Terakawa et al. 2010) earthquake sequences in Italy, and the 2004 Mw = 6.8 Niigata earthquake in Japan (Sibson 2007). Observations of direct fluid generated by earthquake slip have been reported for the 1995 Mw = 7.2 Kobe (Japan) earthquake (Famin et al. 2008) where large volumes of CO_2 were produced from temperature-induced decarbonation.

Although fluids and faulting have long been known to be an important part of the earthquake process, modeling the spatio-temporal evolution of such systems is computationally challenging primarily through the dynamical property of intrinsic permeability. Namely, permeability can change by orders of magnitude over short timescales because of the switch to high permeability at the onset of slip (Miller and Nur 2000). Here we take a modeling approach that combines poro-elasto-plastic model of Rozhko et. al. (2007) with a non-linear diffusion model (Rice 1992; Miller et al. 2004), where the non-linearity arises through an effective-stress dependence of the permeability. The solid deformation is modeled using the FLAC (Fast Lagrangian Analysis of Continua) algorithm with density scaling (Cundall 1982), which is coupled to the non-linear diffusion model using an explicit finite difference algorithm with adaptive time-stepping.

In general, simulations over time scales of months (relevant for modeling fluid-driven aftershock sequences) takes many hours to days of computation time. Reducing the numerical resolution is the typical strategy to reduce the computational time, but in our case this would mean introducing unrealistically large intrinsic length scales for the fractures. More importantly, natural fracture networks occur over a wide range of size scales, from centimeters to kilometers, so reduced resolution is not an affordable sacrifice. The advantage of Graphics Processor Unit (GPU) technology is that it allows much faster computations due to its inherent parallel architecture, allowing much shorter computational times while also increasing numerical resolution. GPUs are particularly powerful for solving governing equations that can be formulated into explicit finite difference algorithms, like for example our full GPU poro-elasto-plastic model with adaptive time stepping discussed below.

20.2 Materials and Methods

20.2.1 Physics Equations

Different studies (Rice 1992; Miller et al. 2004) show that diffusion fluid pore pressure in the crust can be modeled using a nonlinear equation with permeability being an exponential function of stresses (Zhang et al. 1999; David et al. 1994) of the form

$$\frac{\partial P_f}{\partial t} = \frac{1}{\phi(\beta_f + \beta_\phi)} \left[\nabla \frac{\kappa_o \cdot \exp^{\left(-\frac{\bar{\sigma_n}}{\sigma^*}\right)}}{\eta} \nabla P_f + \dot{\Gamma}(P_f, T) \right] \tag{20.1}$$

where $\bar{\sigma_n}$ is the effective normal stress given by

$$\bar{\sigma_n} = \frac{\sigma_1 + \sigma_3 - 2(P_f + \varrho_f g z)}{2} + \frac{\sigma_1 - \sigma_3}{2} \cdot cos(2\theta) \tag{20.2}$$

and P_f is the fluid overpressure, κ_o is the permeability at zero normal stress, σ^* is a constant related to the degree of fracturing of the rock, ρ_f is the fluid density, η is the viscosity, ϕ is the porosity, β_f is the fluid compressibility, β_ϕ is the pore compressibility and $\dot{\Gamma}(P, T)$ is the source term.

The elastodynamic equations in their velocity-stress form describe the elastic response of the rock skeleton

$$\frac{\partial V_x}{\partial t} = \frac{1}{\rho} \left(\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} \right) \tag{20.3}$$

$$\frac{\partial V_y}{\partial t} = \frac{1}{\rho} \left(\frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{xy}}{\partial x} \right) + \rho g \tag{20.4}$$

$$\frac{\partial \sigma_{xx}}{\partial t} = (\lambda + 2\mu) \frac{\partial V_x}{\partial x} + \lambda \frac{\partial V_y}{\partial y} \tag{20.5}$$

$$\frac{\partial \sigma_{yy}}{\partial t} = \lambda \frac{\partial V_x}{\partial x} + (\lambda + 2\mu) \frac{\partial V_y}{\partial y} \tag{20.6}$$

$$\frac{\partial \tau_{xy}}{\partial t} = \mu \left(\frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} \right) \tag{20.7}$$

with μ and λ are the Lamé constants, ρ is the density, v_x and v_z is the velocity vector and σ_{xx}, σ_{zz}, τ_{xz} is the stress tensor. In saturated porous rock, where the pores form a connected network, deformation is controlled by the Terzaghi effective stress

$$\sigma_{ij}^{eff} = \sigma_{ij} - P\delta_{ij}. \tag{20.8}$$

Plastic deformation of rocks is modeled using Mohr-Coulomb and Griffith criteria

$$F_{tension} = \tau - \sigma_m - \sigma_t \tag{20.9}$$

$$F_{shear} = \tau - \sigma_m \cdot sin(\varphi) - C \cdot cos(\varphi) \tag{20.10}$$

$$F = max(F_{tension}, F_{shear})) \tag{20.11}$$

where F is the yield function, φ is the internal frictional angle, τ is the stress deviator, σ_m is the mean stress, σ_t is the tensile strength of the rock. The plastic strain rates are given by

$$\dot{\epsilon}_{ij}^{pl} = 0 \text{ for } F < 0 \text{ or } F = 0 \text{ and } \dot{F} < 0 \tag{20.12}$$

$$\dot{\epsilon}_{ij}^{pl} = \lambda \frac{\partial q}{\partial \sigma_{ij}} \text{ for } F = 0 \text{ and } \dot{F} = 0.) \tag{20.13}$$

We use non-associative plastic flow rules (Vermeer and Borst 1984)

$$q_{tension} = \tau - \sigma_m \tag{20.14}$$

$$q_{shear} = \tau - \sigma_m \cdot sin(\psi). \tag{20.15}$$

In this report the dilatancy angle is $\psi = 0$. From linear theory of poroelasticity the full strain tensor is given by

$$\dot{\epsilon}_{ij} = \dot{\epsilon}_{ij}^{pe} + \dot{\epsilon}_{ij}^{pl} \tag{20.16}$$

where $\dot{\epsilon}_{ij}^{el}$ is the poroelastic strain tensor. The poroelastic stress tensor is given by

$$\sigma_{ij} = 2G\epsilon_{ij}^{pe} + 2G\epsilon_{kk}^{pe} \frac{\nu}{1-2\nu}\delta_{ij} + \alpha P_f \delta_{ij} \tag{20.17}$$

where α is the Biot-Willis constant, G is the shear modulus and ν is Poisson's ratio (Jaeger et al. 2007; Detournay and Cheng 1993).

20.2.2 GPU Implementation

Since its introduction in 1999, Graphics Processor Units (GPU) have been successfully applied to accelerate non-graphical computations due to it highly parallel architecture. GPU implementations have been reported, to name a few, in fluid dynamics (Griebel and Zaspel 2010; Zaspel and Griebel 2011), medical sciences (Nageswarana et al. 2009; Sorensen and Mosegaard 2006), geophysics (Michéa and Komatitsch 2010; Lastra et al. 2009), quantum chemistry (Vogt et al. 2008), molecular dynamics (Yang et al. 2007) and biology (Stivala et al. 2010). The CUDA programming language, develop by NVIDIA and based in a C-like programming model, facilitates GPU usage in physical modeling (?).

GPU programs written with CUDA will run part of the code in the CPU and parts in the GPU. The CUDA functions that run in the GPU, called kernels, admit only a limited number of parameters. To avoid reaching this limit, the initial data matrices (stresses, velocities, rheological properties, hydraulic properties, etc.) of size $nx \times nz$, are grouped together into large one-dimensional vectors of size *number of matrices* $\cdot nz \cdot nx$ to be passed to GPU. To access correctly different data segments within the GPU matrices, we use the index expression:

$$index = x + z * nx + (position\ of\ the\ matrix\ in\ the\ large\ vector - 1) * nz * nx \tag{20.18}$$

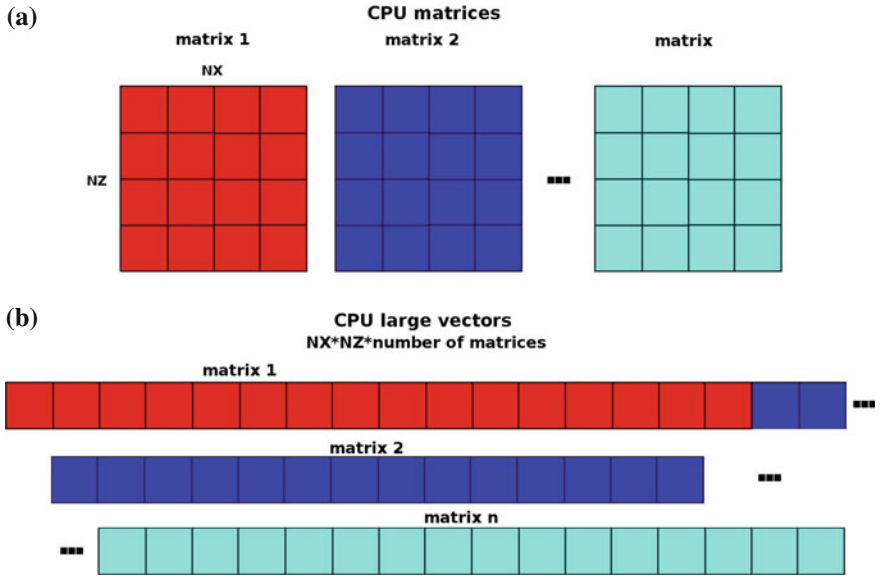


Fig. 20.1 Initial data matrices (**a**) of size $nx \times nz$, are grouped together in large one dimensional vectors (**b**) of size *number of matrices* $\cdot nz \cdot nx$ to be passed to GPU

where $x = 0$ to $nx - 1$ and $z = 0$ to $nz - 1$.
Figure 20.1 shows and sketch of this procedure.

We divide the problem in four main steps:

- solution of the nonlinear diffusion equation (20.1),
- computation of the effective stresses (20.8),
- solution of the velocities equations (20.3, 20.4) and
- computation of the total stresses (20.5–20.7).
- evaluation of the yield function (20.9–20.11) and computation of plastic stresses using (20.12–20.13).

GPU Nonlinear Diffusion Implementation

We use a first order in time, fourth order in space finite difference scheme to solve the nonlinear diffusion equation. Boundary conditions are zero flux boundary conditions at the left, right and bottom edges and Dirichlet boundary conditions at the top, $P_f = 0$. We compute the solution of the nonlinear diffusion equation using shared memory for the center of the domain and global memory for the boundary conditions. Two kernels perform the computation of the nonlinear diffusion: *non_lin_diff* and *fluid_diff_write*. First, nonlinear permeability is computed using Eq. (20.2) and the new permeability is written to global and shared memories to be used in the Eq. (20.17). The inner part of the fluid pressure solution is computed using shared

memory and the boundary conditions using global memory. The new fluid pressure profile solution is written to a new position on the large GPU global memory matrices. We call this vector P_{fnew}. If we try to write the result directly to the initial position, let us call it P_{finit}, errors appears due to the fact that GPU tries to read and write the same memory address at the same time. For that reason, a second kernel fluid_diff_write writes the solution back to the initial row vector P_{finit}. A pseudocode of the kernels *non_lin_diff* and *fluid_diff_write* is presented in Algorithms 19.1 and 19.2.

Algorithm 20.1: *non_lin_diff*

Copy $P_{fold}, k_{fo}, \theta, \eta, \sigma^*, \phi, \sigma_{xx}$ and σ_{zz} on shared memory
Use shared memory variables to compute k_f and save in global memory
Save new value of k_f in shared memory
Use shared memory variables to compute P_{fnew} for the center of the domain and save in global memory
Compute P_{fnew} at the boundaries using global memory

Algorithm 20.2: *fluid_diff_write*

Copy P_{fnew} in global memory to P_{fold} in global memory

At the end of every iteration, porosity, σ^*, bulk modulus and Poisson's ratio are updated to be used in the next time step. We assume drained values of the rheological properties (bulk modulus, Poisson ratio) if $P_f =$ hydrostatic, and undrained values for $P_f =$ maximum overpressure. For intermediate values of P_f, we use a linear function to update rheological properties. The same procedure is used to update porosity and σ^*: porosity is maximum and σ^* is minimum if P_f is maximum and porosity is minimum and σ^* maximum if P_f equal to hydrostatic.

GPU Elasto-Plasticity Implementation

Solutions of elasto-dynamic equations using standard staggered grids introduce instabilities when the domain contains heterogeneities, e.g. cracks, density or rheology changes. To overcome this we use a staggered grid scheme with centered cells. Stresses were located in the cell centers and velocities on its corners. In the program, stresses have a size $(nx+1)*(nz+1)$ and velocities $nx*nz$. However, all matrices are rearranged in the GPU to have size $(nx+1)*(nz+1)$ by filling extra positions with zeros. Boundary conditions for stresses are zero slip at the right, left and bottom edges and free surface boundary condition at the top. Velocity boundary conditions are $v_x = V$ and $v_z = 0$ at the right edge, $v_x = -V$ and $v_z = 0$ at the left edge and $v_z = 0$ at the bottom.

The kernel *effective_stresses* computes Eq. (20.8). As in the case of the diffusion, the central part of the effective stress matrices is computed using shared memory and boundary conditions using global memory. Algorithm 19.3 present the pseudocode of this kernel. Velocities are computed in the kernel *velocity_computation* using the

Algorithm 20.3: *effective_stresses*

Copy P_{fnew}, σ_{zz} and σ_{xx} on shared memory
Perform Eq. (20.3) using shared memory variables and save results on global memory
Apply boundary conditions using global memory

same procedure. Algorithm 19.4 shows the pseudocode for this kernel.

Algorithm 20.4: *velocity_computation*

Copy effective stresses σ_{zz}^{eff}, σ_{xx}^{eff}, σ_{xz}^{eff}, velocities v_x, v_z and displacements U_x, U_z on shared memory
Compute Eq. (20.8) for the central part of the effective stresses matrices using shared memory variables and save results on global memory
Apply boundary conditions using global memory

Elastic and plastic stresses are computed in the kernel *elasto_plasticity_computation*. Plastic stresses are applied in the specific points where conditions (20.12) are fulfilled. This step introduces program branching, which is a major problem for the GPU implementation. CPU can easily perform an element by element search and testing using the if or else logical controllers. Although GPU can compile these logical controllers, program branching introduces a performance penalty due to its inherent parallel architecture. In NVIDIA GPUs, threads within a block are grouped in 32 elements called warps. Within a warp, all threads perform the same instruction at the same time. If warp divergence occurs some threads will take one branch and others the other branch. The first threads to finish the computation must wait until all threads from all different branches finish their computations, thus significantly decreasing the overall performance. In general, GPU will be much slower than CPU when performing element by element operations.
There are different techniques to deal with branching in GPU (Harris and Buck 2005), and we developed a new and easy technique, called switch functions, to handle GPU branching. The details of this implementation will be presented in future communications.

20.3 Results

20.3.1 Nonlinear Diffusion Benching

The nonlinear diffusion algorithm was benched using the *Method of Manufactured Solutions* (MMS) (Salari and Knupp 2000). The MMS has been applied to different problems of computational fluid dynamics (Bond et al. 2004; Roy et al. 2004; Shunn and Ham 2007). In this method, an artificial solution G is proposed. This solution does not need to be physically meaningful but it must be smooth enough to be differentiable within the domain at the higher order of the differential equation. The function G must not be a trivial solution of the differential equation and it must be complicated enough to test the accuracy of the numerical solution. Using symbolic calculus software, we can apply the differential operator to this manufactured solution to compare to our numerical solution.
In our case we choose the following functions:

$$P_f = \exp\left(-\left(\frac{(x-a)^2}{b} + \frac{(z-a)^2}{b}\right)\right) \tag{20.19}$$

$$k_{fo} = \left(0.1 - 1^{-2} \cdot z\right) \cdot \exp\left(-\frac{0.5\left(\sigma_{xx} + \sigma_{zz} - 2 \cdot P_f\right) + 0.5(\sigma_{xx} - \sigma_{zz})cos(2 \cdot \theta)}{\sigma^*}\right) \tag{20.20}$$

with $a = 5, b = 25, \sigma_{xx} = \sigma_{zz} = 10 \cdot z, \sigma^* = 20$ and $\theta = 60°$ and the domain of the function is $\Omega = x \times z$ for $x = [0, 10]$ and $z = [0, 10]$. These two functions are smooth and their derivatives are continuous over this domain. We use the MATLAB symbolic calculus toolbox to compute the derivatives. We evaluate the equation

$$\nabla \frac{\kappa_o \cdot \exp\left(-\frac{\bar{\sigma_n}}{\sigma^*}\right)}{\eta} = \text{RHS} \tag{20.21}$$

with $\eta = 1$, using the MATLAB symbolic calculus toolbox a CPU implementation and the GPU implementation. We measured the global error between the analytical, MATLAB symbolic solution, and numerical RHS using the L_2 norm. For all computational experiments grid points in X direction where equaled to grid points in Z, i.e. $n = nx = nz$. We vary the grid points number per axis n from 100 to 2000. Figure 20.2 shows the results. Both implementations, CPU and GPU, converge to the analytical solution in the same way when the numerical resolution is increased.

20.3.2 Elasto-Plastic Benching

We validate our elasto-plastic model by checking the formation of localized shear zones, shear bands, with the correct angle. There are three different theories that

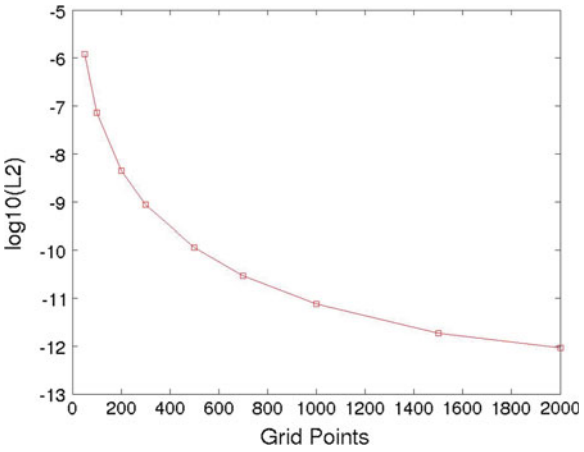


Fig. 20.2 $log_{10}(L_2)$ versus grid points per axis. Numerical resolution is grid points per axis squared, i.e. $nx = nz$. L_2 error norm decrease when the number of grid points is increased. GPU results correspond to CPU ones

describe the orientation of the shear bands; Coulomb, Roscoe and Arthur (Arthur et al. 1977; Bardet 1990). For a material with frictional angle φ and dilation angle ψ, shear bands form with dip angles θ between Rosco-Coulomb range, $45° - \frac{\varphi}{2} < \theta < 45° - \frac{\psi}{2}$ (Kaus 2010; Poliakov et al. 1994).

We perform extension and compaction experiments. The domain is a 100 km^2 crust square with free surface boundary condition at the top, zero tangential velocity at the bottom and Dirichlet boundary conditions for the X direction velocity at the sides accounting for extension or compression. To improve shear band formation, we use a high v_x velocity at the sides of 1 m/year. We varied internal frictional angles, $\varphi = 20°$, $30°$ and $40°$.

Figures 20.3 and 20.4 show the second invariant of the strain tensor for compressional and extensional cases, respectively. Compressional experiments show dip angle values very close to Coulomb angles. Kaus (2010) and Popov and Sobolev (2008) reported similar results for elasto-visco-plastic rheology materials and Hansen et. al. reported dip angles near to Coulomb values for elasto-plastic rheology using a mesh-free finite elements method (Hansen 2003). For the extensional cases, the upper part of the domain shows effects due to boundary conditions where tensile failure seems to play a principal role. The reason for the long tensile fractures is the fast extensional velocity. In the bottom part clear shear bands are present at angles that lie between Roscoe and Coulomb angles for all cases.

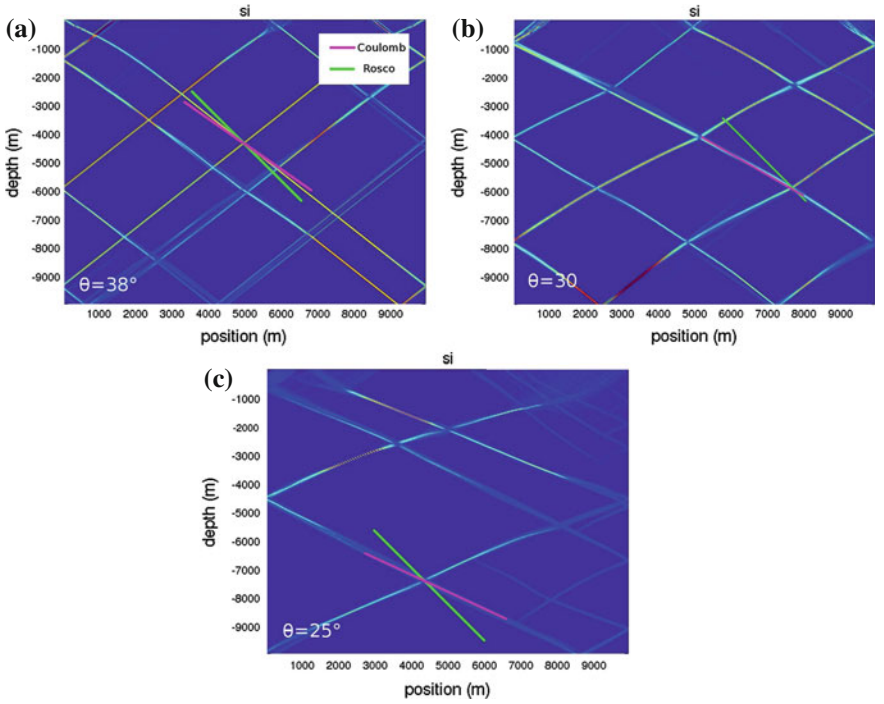


Fig. 20.3 Second invariant of strain tensor (si) showing shear bands formation and dip angles θ for elastoplastic media under compression with internal frictional angles (**a**) $\varphi = 20°$, (**b**) $\varphi = 30°$ and (**c**) $\varphi = 40°$. For all numerical experiments dilation angle was set equal to zero, $\psi = 0$. *Pink lines* are the Coulomb angles and *green lines* are Rosco angles. In all cases θ is very close to the Coulomb value. Numerical resolution: 300×300

20.3.3 Poro-Elasto-Plastic Modeling

We consider a section of crust of 10×10 km under compression or extension with an over-pressurized region at 6 km depth, with model parameters listed in Table 20.1. The fluid overpressure is 100 MPa. First, we apply a fast compression/extension to create a system of fractures and avoid any artificial introduction of fractures using, for example, stochastic methods. Fractures are created in response to the regional stress field. The frictional angles are normally random distributed around 30°.

During fracture network formation, cohesion of failed points is set to zero. If the point fails in tensile mode the permeability is set to two orders of magnitude higher than the overpressure intrinsic permeability. In case of shear fracturing, permeability increases one order of magnitude from the overpressure layer permeability. Following this fast compression or extension time, we set the edges velocities to zero and we let the system relax until steady state is achieved. During this relaxation time, energy is released by fracturing and deformation, thus allowing further growth of the

Table 20.1 Model parameters

Parameter	Hydrostatic pressure layer	Overpressurized layer
Cohesion (MPa)	20	20
Poisson ratio	0.27	0.3
Bulk modulus (GPa)	35	41
Porosity ϕ	0.01	0.10
Intrinsic permeability κ_o (m^2/s)	10^{-17}	10^{-16}
σ^* (MPa)	33	35
Pore compressibility β_ϕ (Pa^{-1})	10^{-8}	10^{-8}
Fluid compressibility β_f (Pa^{-1})	10^{-10}	10^{-10}

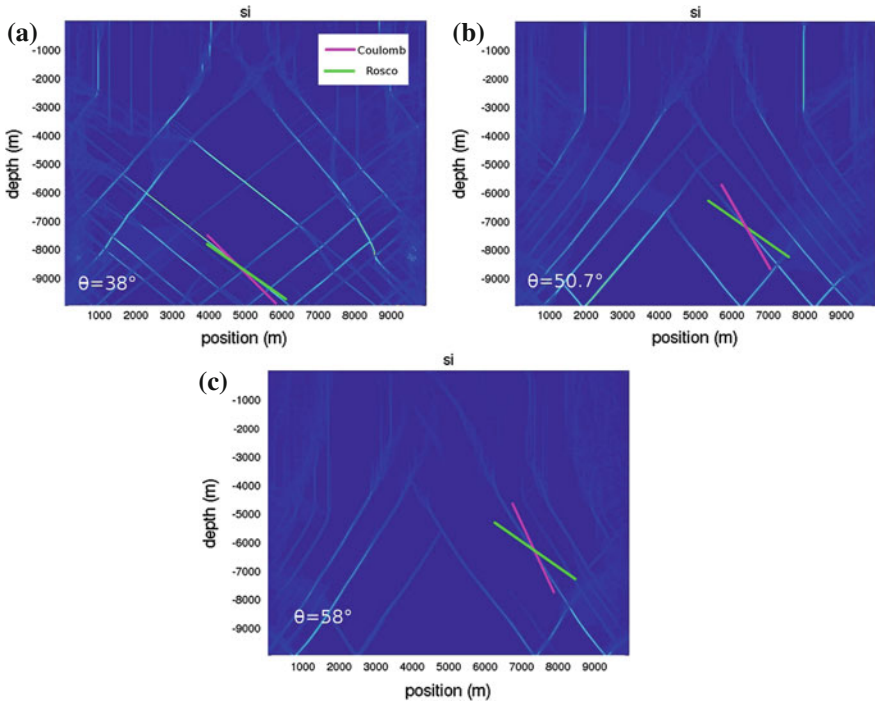


Fig. 20.4 Shear band formation and dip angles θ for elastoplastic media under extension cases. Dilation angle $\psi = 0$ and frictional angles (**a**) $\varphi = 20°$, (**b**) $\varphi = 30°$ and (**c**) $\varphi = 40°$. For all cases, dip angle θ lays between Roscoe (*green line*) and Coulomb (*pink line*) angles. In the upper part of the domain tensile fracturing is appreciated. Numerical resolution: 300×300

network, and relaxing much of the localized stressed points. When stress relaxation is complete, cohesion of the failed points is set to a value lower than the background cohesion to simulate healing of the fractures. In our numerical experiments we set the cohesion of the cracks after relaxation to a fourth of the background cohesion. Time is set to zero and diffusion is initiated. Figures 20.5 and 20.6 show the permeability,

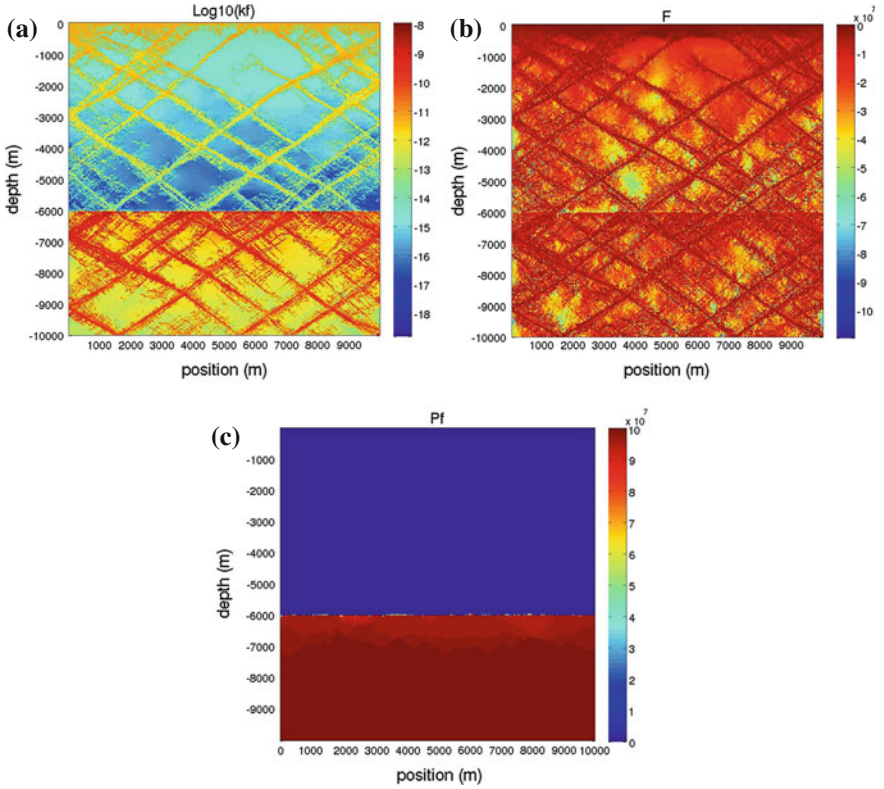


Fig. 20.5 Permeability (**a**), yield function (**b**) and fluid pressure profile (**c**) at time = 180 days after the diffusion release for the compressional case. Beside shear bands, non-localized tensile fractures can be appreciated. Maximum values of the fractures permeability is around $1e^{-8}\,\mathrm{m/s^2}$ in the overpressurized layer. However, permeability drops in the hydrostatic layer locking the fluid pressure. Numerical resolution: 300×300

yield function and fluid pressure at time = 180 days after the onset of diffusion for compressional and extensional cases respectively. In the compressional case, fluids are locked in the over-pressured layer because of the reduced permeability under high normal stress. In the extensional case fluid pressure propagates through the previously created fractures because of reduced normal (thus increased permeability). An extended discussion of this results and its geodynamic implications will be discussed in future communications.

20.4 Conclusions

We developed a poro-elasto-plastic model coupled to non-linear diffusion and implemented in GPU. The model is completely explicit so its computation is very efficient using the GPU architecture. GPU-based modeling offers significant

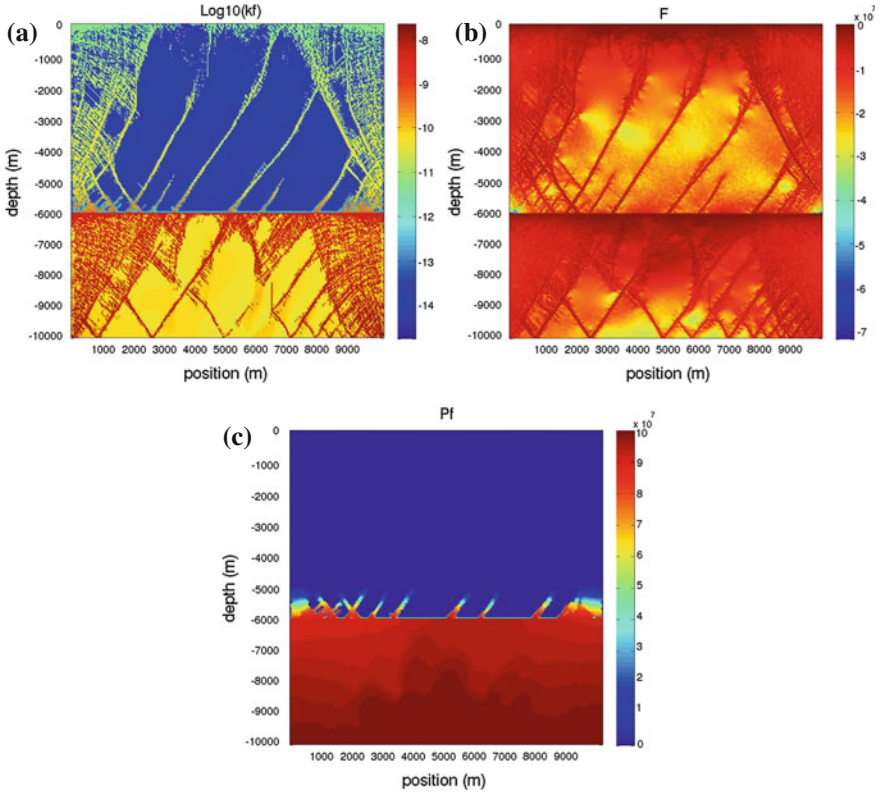


Fig. 20.6 Permeability (**a**) and yield function (**b**) and fluid pressure profile (**c**) at time = 180 days after the diffusion release for the extensional case. Beside shear bands, non-localized tensile fractures can be appreciated. Extension enhance fluid pressure migration through the fractures on the hydrostatic pressurized layer. Numerical resolution: 300×300

advantages and allows fast, high resolution simulations of the underlying process. Nonlinear diffusion equations and elasto-plastic equations have been implemented with von-Neumann and Dirichlet boundary conditions. The introduction of switch functions allows implementing plasticity in GPU and, in general, to handle program branching in a GPU efficient way. We presented examples of bench marks that are in accordance with previous studies. The results of the dip angles of shear bands of our model agrees with other numerical models. The Method of Manufactured Solutions (MMS) was used to bench the nonlinear diffusion equation giving accordance between CPU and GPU double precision implementations. The case of the full poro-elasto-plastic system under extension and compression were presented. This model can be used in different problems in geodynamics, e.g. effect of earthquakes on permeability and fluid migration, hydrofracturing in engineering applications such as geothermal reservoirs or oil and gas reservoir simulations. To the best of our knowledge, this is the first implementation of poro-elasticity or plastic rheology in GPU.

Due to complexity of our system its correct implementation in GPU architecture opens by itself a new area of application for GPU based modeling. Developments of a 3D implementation of our model using GPU clusters will be presented in the future.

Acknowledgments We thank the German Research Foundation, Deutsche Forschungsgemeinschaft (DFG) for the financial support through the project no. MI 1237/2-1.

References

Arthur JRF, Dunstan T, Al-Ani QAJ, Assadi A (1977) Plastic deformation and failure of granular media. Geotechnique 27:53–74

Audin L, Avouac J, Flouzat M, Plantet J (1091) Fluid-driven seismicity in a stable tectonic context: the Remiremont fault zone, Vosges. France. Geophys Res Lett 29:2002

Bardet JP (1990) A comprehensive review of strain localization in elastoplastic soils. Comput Geotech 10:163–188

Bols WJ, Nur A (2002) Aftershocks and pore fluid diffusion following the 1992 Landers earthquake. J Geophys Res 107(B12), 2366:17–1, 17–9

Bond RB, Knupp PM, Ober CC (2004) A manufactured solution for verifying CFD boundary conditions. In: Proceeding of 34th AIAA fluid dynamics conference and exhibit, Portland, Oregon, pp AIAA. 2004–2629, June 2004

Cundall PA (1982) Adaptive density scaling for time-explicit calculations. In: Proceeding of 4th international conference on numerical methods in geomechanics. Edmonton, Canada, vol 1:23–26

David C, Wong TF, Zhu W, Zhang J (1994) Laboratory measurement of compaction-induced permeability change in porous rocks: implications for the generation and maintenance of pore pressure excess in the crust. Pure and Appl Geophys 143(1–3):425–456

Detournay E, Cheng AHD (1993) Fundamentals of poroelasticity. In: Hudson JA (ed) Comprehensive rock engineering, vol 2. Pergamon, New York, pp 113–171

Famin V, Nakashima S, Boullier A, Fujimoto K, Hirono T (2008) Earthquakes produce carbon dioxide in crustal faults. J Struct Geol 265:487–497

Griebel M, Zaspel P (2010) A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations. Comput Sci Res Dev 25:65–73

Hansen DL (2003) A meshless formulation for geodynamic modelling. J Geophys Res 108(B11):2549

Häring MO, Schanz U, Ladner F, Dyer BC (2008) Characterisation of the Basel 1 enhanced geothermal system. Geothermics 37:469–495

Harris M, Buck I (2005) GPU flow control idioms. In: Pharr M (ed) GPU gems 2. Addison-Wesley, Reading, pp 547–555

Jaeger J, Cook NG, Zimmerman R (2007) Fundamentals of rock mechanics, 4th edn. Blackwell Publishing, Malden

Kaus BJP (2010) Factors that control the angle of shear bands in geodynamic numerical models of brittle deformation. Tectonophysics 484(1–4):36–47

Lastra M, Mantas JM, Urena C, Castro MJ, García-Rodríguez JA (2009) Simulation of shallow-water systems using graphics processing units. Math Comput Simul 182:598–618

Michéa D, Komatitsch D (2010) Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophys J Int 182:389–402

Miller SA, Nur A (2000) Permeability as a toggle switch in fluid-controlled crustal processes. Earth Plan Sci Lett 183:133–146

Miller SA, Collettini C, Chiaraluce L, Cocco M, Barchi M, Kaus BJP (2004) Aftershocks driven by a highpressure CO_2 source at depth. Nature 427:724–727

Nageswarana JM, Dutt N, Krichmar JL, Nicolau A, Veidenbaum AV (2009) A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Netw 22:791–800

Nur A (1971) Effects of stress on velocity anisotropy in rocks with cracks. J Geophys Res 76(8):2022–2034

Ohtake M (1974) Seismic activity induced by water injection at Matsushiro, Japan. J Phys Earth 22:163–176

Poliakov ANB, Herrmann HJ, Podladchikov YY (1994) Fractal plastic shear bands. Fractals 2:567–581

Popov AA, Sobolev SV (2008) SLIM3D: a tool for three-dimensional thermomechanical modeling of lithospheric deformation with elasto-visco-plastic rheology. Phys Earth Plan Inter 171:55–75

Rice JR (1992) Fault stress states, pore pressure distributions, and the weakness of the San Andreas fault. In: Evans B, Wong T-F (eds) Fault mechanics and transport properties in rocks. Academic Press, London, pp 475–503

Roy CJ, Nelson CC, Smith TM, Ober CC (2004) Verification of Euler/Navier-Stokes codes using the method of manufactured solutions. Int J Num Methods Fluids 44:599–620

Rozhko AY, Podladchikov YY, Renard F (2007) Failure patterns caused by localized rise in pore-fluid overpressure and effective strength of rocks. Geophys Res Lett 34:L22304

Salari K, Knupp P (2000) Code verification by the method of manufactured solutions. Technical Report, SAND2000-1444, Sandia National Laboratories, June 2000

Shapiro SA, Dinske C (2009) Fluid-induced seismicity: pressure diffusion and hydraulic fracturing. Geophys Prospect 57:301–310

Shunn L (2007) Ham F (2007) Method of manufactured solutions applied to variable-density flow solvers. Technical report, Center for Turbulence Research Annual Research Briefs

Sibson RH (2007) An episode of fault-valve behavior during compressional inversion? The 2004 MJ 6.9 Mid-Niigata Prefecture, Japan, earthquake. Earth Plan Sci Lett 257:188–199

Sorensen TS, Mosegaard J (2006) Haptic feedback for the GPU-based surgical simulator. Med Meets Virtual Real 14:523–528

Stivala AD, Stuckey PJ, Wirth AI (2010) Fast and accurate protein substructure searching with simulated annealing and GPUs. BMC Bioinformatics 11:446

Terakawa T, Zoporowski A, Galvan B, Miller SA (2010) High-pressure fluid at hypocentral depths in the L'Aquila region inferred from earthquake focal mechanisms. Geology 38(11):995–998

Terzaghi K (1923) Die berechnung des durchlassigkeitsziffer des tones aus dem verlauf der hydrodynamischen spannungserscheinungen. Sitz Akad Wiss Wien Abt IIa 132:125–138

Vermeer PA, Borst R (1984) Non-associative plasticity for soils, concrete and rock. Technical report, Heron

Vogt L, Olivares-Amaya R, Kermes S, Shao Y, Amador-Bedolla C, Aspuru-Guzik A (2008) Accelerating resolution-of-the-identity second-order Møller-Plesset quantum chemistry calculations with graphical processing units. J Phys Chem 112:2049–2057

Yang J, Wang Y, Chen Y (2007) GPU accelerated molecular dynamics simulation of thermal conductivities. J Comput Phys 221:799–804

Zaspel P, Griebel M (2011). Solving incompressible two-phase flows on massively parallel multi-GPU clusters. Comput Fluids (Submitted:INS, Preprint No.1113)

Zhang S, Tullis TE, Scruggs VJ (1999) Permeability anisotropy and pressure dependency of permeability in experimentally sheared gouge materials. Earth Plan Sci Lett 21:795–806

Chapter 21
GPU Implementation of Multigrid Solver for Stokes Equation with Strongly Variable Viscosity

Liang Zheng, Taras Gerya, Matthew Knepley, David A. Yuen, Huai Zhang and Yaolin Shi

Abstract Solving Stokes flow problem is commonplace for numerical modeling of geodynamic processes, because the lithosphere and mantle can be always regarded as incompressible flow for long geological time scales. For Stokes flow, the Reynold Number is effectively zero so that one can ignore the advective transport of momentum equation thus resulting in the slowly creeping flow. Because of the ill-conditioned matrix due to the saddle points problem that coupling mass and momentum partial differential equations together, it is still extremely to efficiently solve this elliptic PDE system, especially with the strongly variable coefficients due to rheological structure of the earth. However, since NVIDIA issued the CUDA programming framework in 2007, scientists can use commodity CPU-GPU system to do such geodynamic simulation efficiently with the advantage of CPU and GPU respectively. In this paper, we try to implement a GPU solver for Stokes Equations with variable viscosity based on CUDA using geometric multigrid methods on the staggered grids. For 2D version, we used a mixture of Jacobi and Gauss-Seidel iteration with conservative finite difference as the smoother. For 3D version, we called the GPU smoother

L. Zheng · H. Zhang · Y. Shi(✉)
Key Laboratory of Computational Geodynamics,
Chinese Academy of Sciences and College of Earth Science,
University of Chinese Academy of Sciences, Beijing, China
e-mail: shiyl@ucas.ac.cn

T. Gerya
Institute of Geophysics, ETH-Zurich,
Zurich, Switzerland

M. Knepley
Computational Institute, University of Chicago,
Chicago, IL, USA

L. Zheng · D. A. Yuen
Minnesota Supercomputing Institute, University of Minnesota,
Minneapolis, MN, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 321
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_21,

which is rewritten with the Red-Black Gauss-Seidel updating method to avoid the problem of disordered threads with Matlab 2010b.

21.1 Introduction to Stokes Equations and GPU Applications in Geophysics and Geodynamics

21.1.1 Stokes Flow in Geodynamics

The solid earth deforms slowly, and behaves as viscous fluid over geological time. So geoscientists can study the earth using fluid dynamic methods with the basic principles of mass, momentum and energy conservation. The continuity equation which describes the mass conservation is:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{21.1}$$

The momentum conservation equation connects forces and deformation, which is derived from Newton's second law, can be described under Eulerian form:

$$\frac{\partial \sigma_{ij}}{\partial x_j} + \rho g_i = \rho \left(\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) \tag{21.2}$$

where the right hand side of Eq. (21.2) is the inertia force. From the hydrostatic stress state we can define the deviatoric stress $\sigma_{ij}^{'}$ as

$$\sigma_{ij}^{'} = \sigma_{ij} + P\delta_{ij}, \tag{21.3}$$

where δ_{ij} is Kronecker delta. Then we can obtain the Navier-Stokes equation from Eqs. (21.2) and (21.3) which describes the conservation of momentum for the fluid with gravity:

$$\frac{\partial \sigma_{ij}^{'}}{\partial x_j} - \frac{\partial P}{\partial x_i} + \rho g_i = \rho \left(\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) \tag{21.4}$$

Consider the Renolds number

$$\mathbf{Re} = \frac{\rho u l}{\mu} = \frac{u l}{\nu} \tag{21.5}$$

where μ is the dynamic viscosity and ν is the kinematic viscosity that $\nu = \frac{\mu}{\rho}$. u is the mean fluid velocity while l is characteristic length. In fact, the $\frac{\rho u^2}{l}$ represents inertia force while $\frac{\mu u}{l^2}$ represents viscous force. So the Renolds number **Re** is the ratio of inertia force and viscous force. When the viscous force is very large compared with

inertia force we can ignore the inertia force so that $\mathbf{Re} \ll 1$ or just simplifies to $\mathbf{Re} = 0$, that means we can rewrite the Eq. (21.4) as:

$$\frac{\partial \sigma_{ij}^{'}}{\partial x_j} - \frac{\partial P}{\partial x_i} + \rho g_i = 0 \tag{21.6}$$

Equation (21.6) is the Stokes equation which describes the Stokes flow problem coupled with mass conservation equation who has a lot of applications in geodynamic processes.

Solving stokes problem is very important in Geodynamics because we need to study the viscous fluid behavior of the solid earth, which appears to be folded on long geological time scales. We can take mantle convection for example. The process of mantle convection is believed to be the main driven force of the movement of lithospheric plates. Although the mantle indeed can be regarded as elastic material on short time, the movement of the plates which we can observe need long time to accumulate. For long time scales (such as million years), the mantle's dynamic properties behave like viscous creeping fluid. Some other geodynamic processes can also be described as Stokes flow problems including the lava flow, rock deformation, subduction zone and so on (Turcotte and Schubert 2012; Alik and Tackley 2010; Gerya 2010; Trompert and Hansen 1996).

21.1.2 Stokes Equations with Strongly Variable Viscosity

In most situations, the fluid-like solid earth is incompressible so the viscous constitutive relationship can be simplified to:

$$\sigma_{ij}^{'} = 2\mu\, \dot{\varepsilon_{ij}} \tag{21.7}$$

where $\dot{\varepsilon_{ij}}$ is the strain rate that can be defined as:

$$\dot{\varepsilon_{ij}} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{21.8}$$

According to Eqs. (21.6–21.8) we can rewrite Eq. (21.6) coupled with the mass conservation Eq. (21.1) as:

$$\begin{cases} \nabla \cdot (\mu(\nabla \mathbf{u} + (\nabla \mathbf{u})^{\mathrm{T}})) + \mathbf{f} = \nabla p \\ \nabla \cdot \mathbf{u} = 0 \end{cases} \tag{21.9}$$

From experimental data, the effective viscosity of rocks is as following: $\mu_{eff} \propto exp\left(\frac{E_a + V_a P}{nRT}\right)$ where E_a is activation energy, V_a is activation volume, R is gas constant, P is pressure and T is temperature. That means the effective

viscosity may vary many orders of magnitudes even with small changes of environmental properties like temperature or pressure. These strongly variable viscosity phenomenons are commonplace in geodynamic processes. For example the contrast of viscosity of surface conditions of the earth and upper mantle reaches $\frac{\mu_1}{\mu_0} \sim 10^{39}$ for the huge difference of temperature (Gerya 2010; Deubelbeiss and Kaus 2008).

21.1.3 GPU Applications in Geophysics and Geodynamics

Recent years the computing capacity of GPU (Graphics Processing Units) increased rapidly. With the many-core structure, the energy consume and speedup rate of single GPU are both better than traditional CPU based PC cluster for parallel computing. Since NVIDIA issued CUDA (Compute Unified Device Architecture) framework in 2007, programming on GPGPU (General-Purpose computations on Graphics Processing Units) became easy going for scientific computation. So scientists including geophysicists and geodynamicists start turning to commodate CPU-GPU hybrid platform for solving large scale problems.

Normally there are three classes of partial differential equations which are studied by geophysicists and mathematicians. Parabolic equations and hyperbolic equations are first been devised on GPU, such as heat diffusion equations and seismic wave propagation. Taking seismic wave propagation problem for example Walsha et al. (2009) and Komatitsch et al. (2010) implemented the high order spectral-finite-element codes separately and Michéa (2010) worked out the finite difference codes. In addition, some other computational methods in geophysics have been translated into CUDA for GPU computing, among which the Lattice-Boltzmann code appeared earlier (Walsha et al. 2009; Tolke and Krafczyk 2008; Tolke et al. 2010). What we want to solve here is elliptic equation which also has many applications in geodynamics.

21.2 Strategies for Solving Stokes Equations

21.2.1 Methods Used on CPU

A lot of numerical methods have been applied to solve Stokes equations on CPU before the birth of GPU, including finite difference, finite volume and finite element method. Coupled with the mass equation, the Stokes equation becomes the saddle problem that some special iterative methods are there for solving it such as the multigrid method and preconditioned Krylov subspace methods, etc. Multigrid method can speed up the iteration because of the fast convergence for the longer wavelength residuals. Generally speaking, the multigrid method can be used as a

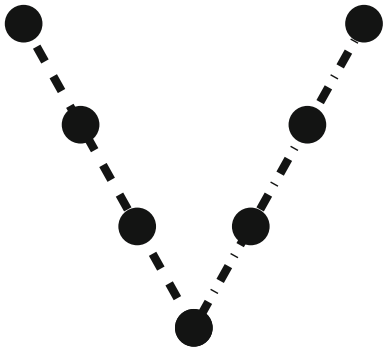


Fig. 21.1 V-cycle mulgrid

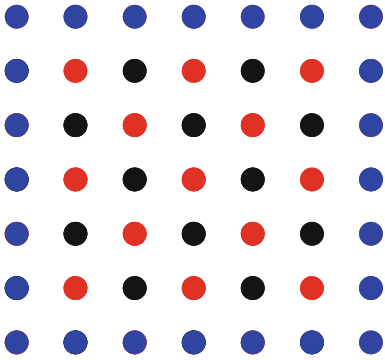


Fig. 21.2 Red-Black updating method

preconditioner as krylov subspace method which we want to implement later (May and Moresi 2008; Gerya 2010; LeVeque 2010).

21.2.2 Strategies on GPU

On GPU we used geometric multigrid (GMG) coupled with Red-Black updating method to solve the Stokes equations. Geometric multigrid method is suitable for finite difference method, because we don't need to build the coefficient matrix explicitly. The V-cycle GMG is showing as Fig. 21.1 which has two part: 1.'–' represents the restriction; 2.'-.' represents the prolongation. Using Red-Black Gauss-Seidel (RBGS) iteration technology (Wallin et al. 2006) for the smoother can avoid the disordered threads when executing the GPU kernels. Figure 21.2 shows the RBGS technology which can be divided into two parts: 1. Set the boundary condition and ghost points around the nodal points (blue points); 2. Update the red and black points in different kernels (red and blue points).

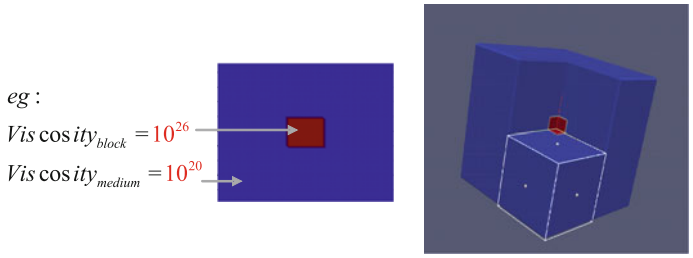


Fig. 21.3 Testing model

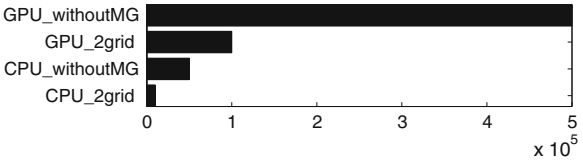


Fig. 21.4 Comparison of iterations on CPU and GPU

21.3 Implementation of the GPU Solver

We set a testing model showing as Fig. 21.3 to check the solver's ability of dealing with variable viscosity, in which the viscosity has a contrast of 10^6. All the boundary nodes are set with free slip velocity so the force of this model is only gravity. 2D and 3D codes are implemented on GPU to compute the result of the testing model.

21.3.1 2D Version

The 2D version codes were first translated into CUDA from Taras Gerya's matlab codes (Gerya 2010) for experiment to test the GPU's ability and compare the compute capacity of different GPU platform. Conservative finite difference methods and staggered stencil are used. We have not noticed the disordered threads for the 2D version so the smoother's performance is actually between the Jacobi and Gauss-Seidel iteration. Restriction and prolongation processes are also implemented on GPU. To check if the multigrid is effective, we compare the codes without multigrid and the 2-level grids under 128*128 resolution showing as Fig. 21.4 that we can find GPU codes need much more iterations than CPU as we mentioned before the smoother's performance on GPU is much worse than CPU for the problem of disordered threads. However, in 3D version we improved the smoother using Red-Black updating method to avoid this problem. Even that using 2-grid can still reduce the iterations on GPU as the same as CPU. We also compared the running time on different GPUs including Tesla 1060c (GT200) and GTX 480 (Fermi) architecture with the same cycles (10^5 iterations) showing as Table 21.1.

Table 21.1 Comparison of different platforms with different precision

Platform	Single precision (s)	Double precision (s)
GPU(Tesla 1060C)	303	619
GPU(GTX 480)	160	245

21.3.2 3D Version

The real earth and real geodynamic problems always need the 3D description, so that we also implemented the 3D version codes. By now 3D version is very primitive that we only rewrote the smoother codes by CUDA and called it by Matlab 2010b (released on Oct. 2010) which is a good numerical test bed that supports CUDA programming by calling the PTX (Parallel Thread Execution) instructions compiled by NVCC. We used 6-level grids and each levels iteration numbers are: 5, 10, 20, 40, 80, 160. From 2D to 3D the programming will be much more complex especially with the staggered grid because of the different index systems for different variables. So we are giving the detailed discrete finite difference scheme here.We rewrite the Eq. (21.6) in 3D condition as:

$$\begin{cases} \dfrac{\partial \sigma'_{xx}}{\partial x} + \dfrac{\partial \sigma'_{xy}}{\partial y} + \dfrac{\partial \sigma'_{xz}}{\partial z} - \dfrac{\partial P}{\partial x} + \rho g_x = 0 \\ \dfrac{\partial \sigma'_{yx}}{\partial x} + \dfrac{\partial \sigma'_{yy}}{\partial y} + \dfrac{\partial \sigma'_{yz}}{\partial z} - \dfrac{\partial P}{\partial y} + \rho g_y = 0 \\ \dfrac{\partial \sigma'_{zx}}{\partial x} + \dfrac{\partial \sigma'_{zy}}{\partial y} + \dfrac{\partial \sigma'_{zz}}{\partial z} - \dfrac{\partial P}{\partial z} + \rho g_z = 0 \end{cases} \tag{21.10}$$

Combined with rheology constitutive Eq. (21.8) and mass conservation Eq. (21.1) we can obtain:

$$\begin{cases} \dfrac{\partial \left(2\mu \dfrac{\partial u_x}{\partial x}\right)}{\partial x} + \dfrac{\partial \left(\mu \left(\dfrac{\partial u_x}{\partial y} + \dfrac{\partial u_y}{\partial x}\right)\right)}{\partial y} + \dfrac{\partial \left(\mu \left(\dfrac{\partial u_x}{\partial z} + \dfrac{\partial u_z}{\partial x}\right)\right)}{\partial z} - \dfrac{\partial P}{\partial x} + \rho g_x = 0 \\ \dfrac{\partial \left(\mu \left(\dfrac{\partial u_x}{\partial y} + \dfrac{\partial u_y}{\partial x}\right)\right)}{\partial x} + \dfrac{\partial \left(2\mu \dfrac{\partial u_y}{\partial y}\right)}{\partial y} + \dfrac{\partial \left(\mu \left(\dfrac{\partial u_y}{\partial z} + \dfrac{\partial u_z}{\partial y}\right)\right)}{\partial z} - \dfrac{\partial P}{\partial y} + \rho g_y = 0 \\ \dfrac{\partial \left(\mu \left(\dfrac{\partial u_x}{\partial z} + \dfrac{\partial u_z}{\partial x}\right)\right)}{\partial x} + \dfrac{\partial \left(\mu \left(\dfrac{\partial u_z}{\partial y} + \dfrac{\partial u_y}{\partial z}\right)\right)}{\partial y} + \dfrac{\partial \left(2\mu \dfrac{\partial u_x}{\partial x}\right)}{\partial z} - \dfrac{\partial P}{\partial z} + \rho g_z = 0 \\ \dfrac{\partial u_x}{\partial x} + \dfrac{\partial u_y}{\partial y} + \dfrac{\partial u_z}{\partial z} = 0 \end{cases} \tag{21.11}$$

In the staggered grid one-nodal viscosity should be set in different indexes to satisfy the different systems of variables. Taking x-stokes equation for example we can get the staggered discrete equation as:

$$\begin{aligned}
&\left(4\mu_{\mathrm{n}(i-1,j,l-1)}\frac{u_{x(i,j+1,l)}-u_{x(i,j,l)}}{\Delta x_{j+1/2}\left(\Delta x_{j-1/2}+\Delta x_{j+1/2}\right)}-4\mu_{\mathrm{n}(i-1,j-1,l-1)}\frac{u_{x(i,j,l)}-u_{x(i,j-1,l)}}{\Delta x_{j-1/2}\left(\Delta x_{j-1/2}+\Delta x_{j+1/2}\right)}\right)\\
&+\left(\begin{array}{l}2\mu_{xy(i,j,l-1)}\left(\dfrac{u_{x(i+1,j,l)}-u_{x(i,j,l)}}{\Delta y_{i-1/2}\left(\Delta y_{i-1/2}+\Delta y_{i+1/2}\right)}+\dfrac{u_{y(i,j+1,l)}-u_{y(i,j,l)}}{\Delta y_{i-1/2}\left(\Delta x_{j-1/2}+\Delta x_{j+1/2}\right)}\right)\\-2\mu_{xy(i-1,j,l-1)}\left(\dfrac{u_{x(i,j,l)}-u_{x(i-1,j,l)}}{\Delta y_{i-1/2}\left(\Delta y_{i-3/2}+\Delta y_{i-1/2}\right)}+\dfrac{u_{y(i-1,j+1,l)}-u_{y(i-1,j,l)}}{\Delta y_{i-1/2}\left(\Delta x_{j-1/2}+\Delta x_{j+1/2}\right)}\right)\end{array}\right)\\
&+\left(\begin{array}{l}2\mu_{xz(i-1,j,l)}\left(\dfrac{u_{x(i,j,l+1)}-u_{x(i,j,l)}}{\Delta z_{l-1/2}\left(\Delta z_{l-1/2}+\Delta z_{l+1/2}\right)}+\dfrac{u_{z(i,j+1,l)}-u_{z(i,j,l)}}{\Delta z_{l-1/2}\left(\Delta x_{j-1/2}+\Delta x_{j+1/2}\right)}\right)\\-2\mu_{xz(i-1,j,l-1)}\left(\dfrac{u_{x(i,j,l)}-u_{x(i,j,l-1)}}{\Delta z_{l-1/2}\left(\Delta z_{l-3/2}+\Delta z_{l-1/2}\right)}+\dfrac{u_{z(i,j+1,l-1)}-u_{z(i,j,l-1)}}{\Delta z_{l-1/2}\left(\Delta x_{j-1/2}+\Delta x_{j+1/2}\right)}\right)\end{array}\right)\\
&-2\frac{P_{i-1,j,l-1}-P_{i-1,j-1,l-1}}{\Delta x_{j-1/2}+\Delta x_{j+1/2}}+\frac{1}{4}\left(\rho_{i-1,j,l-1}+\rho_{i,j,l-1}+\rho_{i-1,j,l}+\rho_{i,j,l}\right)g_x=0
\end{aligned}\tag{21.12}$$

The discrete equation for mass conservation is :

$$\begin{aligned}
&\frac{u_{x(i+1,j+1,l+1)}-u_{x(i+1,j,l+1)}}{\Delta x_{j+1/2}}+\frac{u_{y(i+1,j+1,l+1)}-u_{y(i,j+1,l+1)}}{\Delta y_{i+1/2}}\\
&+\frac{u_{z(i+1,j+1,l+1)}-u_{z(i+1,j+1,l)}}{\Delta z_{l+1/2}}=0
\end{aligned}\tag{21.13}$$

In 3D version codes we apply two order coloring updating method to accelerate the smoother using the RBGS (Red-Black Gauss-Seidel) iteration mentioned before. In order to keep the same style of Taras' matlab codes (to reuse the discrete finite difference format) and avoid the index problem, we define a series of macros as following:

```
#define vx(i,j,k) vx[(i-1)+(j-1)*(ynum+1)+(k-1)*(xnum)*(ynum+1)]
#define RX(i,j,k) RX[(i-1)+(j-1)*(ynum+1)+(k-1)*(xnum)*(ynum+1)]
#define vy(i,j,k) vy[(i-1)+(j-1)*(ynum)+(k-1)*(xnum+1)*(ynum)]
#define RY(i,j,k) RY[(i-1)+(j-1)*(ynum)+(k-1)*(xnum+1)*(ynum)]
#define vz(i,j,k) vz[(i-1)+(j-1)*(ynum+1)+(k-1)*(xnum+1)*(ynum+1)]
#define RZ(i,j,k) RZ[(i-1)+(j-1)*(ynum+1)+(k-1)*(xnum+1)*(ynum+1)]
#define pr(i,j,k) pr[(i-1)+(j-1)*(ynum-1)+(k-1)*(xnum-1)*(ynum-1)]
#define RC(i,j,k) RC[(i-1)+(j-1)*(ynum-1)+(k-1)*(xnum-1)*(ynum-1)]
```

Reducing the logic operations in the CUDA kernel is the main strategy, because NVIDIA's GPU architecture at present has one logic unit in one SM (Streaming Multiprocessor) which means if you use the 'if' in incorrect way (not for the GPU's) the logic operations may run 16 times (half wrap). To avoid this situation we implemented the codes by dividing the smoother function into different parts in different CUDA kernels for 3D version. This way has another benefit that we can use as fewer variables as possible in one kernel that most variables can be put on the registers not the local memory. Another thing we should notice is the index of array is ordered by column-major in matlab and starts from 1, so we need to use the definition of macro as we have mentioned before and modify the index in the kernels.Taking x-direction velocity in red points for example, the codes are showed as following:

```
#include "Index.h"

__global__ void rb_vx_r( ... ) {
  int i = blockIdx.x;
  int j = blockIdx.y;
  int k = threadIdx.x;

  i+=2;j+=2;k+=2;
  //+2 means start from 1 and skip the boundary points
  if((i+j+k)%2!=0) return;
  //decide if it's the red nodes

  double resxcur,kfxcur;

  resxcur = RX(i,j,k)+(pr(i-1,j,k-1)-pr(i-1,j-1,k-1))/xstp;
  resxcur = resxcur-(xkf2*(etan(i-1,j,k-1)*(vx(i,j+1,k)-vx(i,j,k))-etan(i-1,j
      -1,k-1)*(vx(i,j,k)-vx(i,j-1,k))));
  resxcur = resxcur-(etaxy(i,j,k-1)*(ykf*(vx(i+1,j,k)-vx(i,j,k))+xykf*(vy(i,j
      +1,k)-vy(i,j,k)))-etaxy(i-1,j,k-1)*(ykf*(vx(i,j,k)-vx(i-1,j,k))+xykf*(vy
      (i-1,j+1,k)-vy(i-1,j,k))));
  resxcur = resxcur-(etaxz(i-1,j,k)*(zkf*(vx(i,j,k+1)-vx(i,j,k))+xzkf*(vz(i,j
      +1,k)-vz(i,j,k)))-etaxz(i-1,j,k-1)*(zkf*(vx(i,j,k)-vx(i,j,k-1))+xzkf*(vz
      (i,j+1,k-1)-vz(i,j,k-1))));
  kfxcur = -xkf2*(etan(i-1,j,k-1)+etan(i-1,j-1,k-1))-ykf*(etaxy(i,j,k-1)+etaxy(
      i-1,j,k-1))-zkf*(etaxz(i-1,j,k)+etaxz(i-1,j,k-1));
  vx(i,j,k) = vx(i,j,k)+resxcur/kfxcur*krelaxs;
}
```

Figures 21.5, 21.6, 21.7, 21.8 show the result of the 3D codes. (Y direction points to the inner earth.) Table 21.2 is the comparison of time and iterations under different platforms (CPU:Xeon 5520 GPU:Tesla 1060C) using double precision. It shows using Red-Black iteration technology does not need much more iterations than traditional GS (Gauss-Seidel) method. In fact with higher resolution the iteration times are less that the GS method. Compared with Table 21.3, we found that the smoother itself gets a better speedup than the whole process which means the sequential part takes too much time.

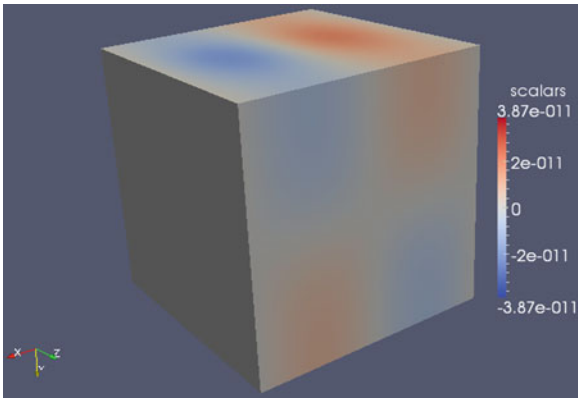


Fig. 21.5 Velocity-X unit: m/s

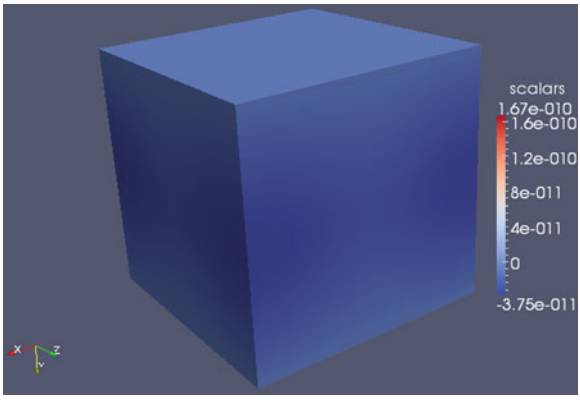


Fig. 21.6 Velocity-Y unit: m/s

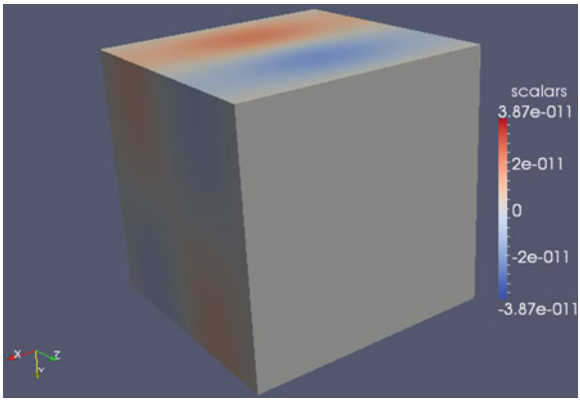


Fig. 21.7 Velocity-Z unit: m/s

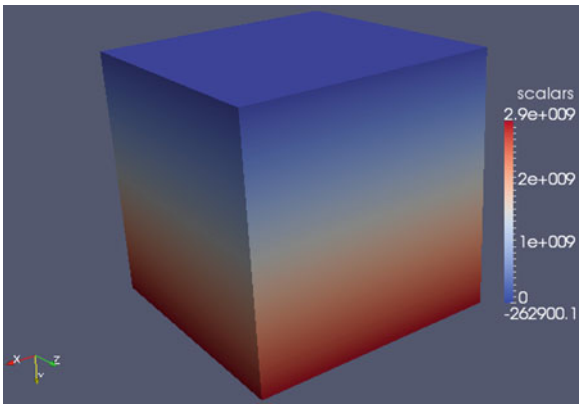


Fig. 21.8 Pressure unit: Pa

Table 21.2 Comparison of time and iterations on different platforms

Platform	Model: 32*32*32	Model: 64*64*64
CPU iterations	85	141
GPU iterations	148	106
CPU time (min)	10	96
GPU time (min)	8	31
Speedup	1.2×	3.1×

Table 21.3 Comparison of speedup on different models

Model	CPU Time(s)	GPU time(s)	Speedup
32*32*32	1.2	0.3	4.0×
64*64*64	10.5	2.2	4.8×
128*128*128	101.6	21.4	4.7×
256*256*256	787.3	202.8	3.9×

21.4 Conclusion and Future

We have finished the preliminary implementation of GPU based multigrid solver for Stokes equation with strongly variable viscosity. It has increased the performance of the Matlab codes on CPU. But it's only a start and we still need to apply some technologies to improve the codes. Next we are considering deploying it as a preconditioner using in PETSc (URL: http://www.mcs.anl.gov/petsc/petsc-as/index.html) and trying to run it with MPI on GPU cluster.

Acknowledgments This research was funded by Deep Exploration Program in China (SinoProbe 07), NSF CMG program, National High Technology Research and Development program in China (2010AA012402) and NSFC 41274103.

References

Alik I-Z, Tackley PJ (2010) Computationalmethods for geodynamics. Cambridge University Press, Cambridge

Deubelbeiss Y, Kaus BJP (2008) Comparison of Eulerian and Lagrangian numerical techniques for the Stokes equations in the presence of strongly varying viscosity. Phys Earth Planet Inter 171:92–111

Gerya TV (2010) Introduction to numerical geodynamic modelling. Cambridge University Press, Cambridge

Komatitsch D, Erlebacher G, Göddeke D, Michéa D (2010) High-order finite element seismic wave propagation modeling with MPI on a large GPU cluster. J Comput Phys 229:7692–7714

LeVeque RJ (2007) Finite difference methods for ordinary and partial differential equations steady state and time dependent problems. Society for Industrial and Applied Mathematics (SIAM), Philadelphia

May DA, Moresi L (2008) Preconditioned iterative methods for Stokes flow problems arising in computational geodynamics. Phys Earth Planet Inter 171:33–47

Michéa D, Komatitsch D (2010) Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophys J Int 182:389–402

Tolke J, Baldwin C et al (2010) Computer simulations of fluid flow in sediment: from images to permeability. Lead Edge 29:68–74

Tolke J, Krafczyk M (2008) TeraFLOP computing on a desktop PC with GPUs for 3D CFD. Int J Comput Fluid Dynamics 7:443–456

Trompert RA, Hansen U (1996) The application of a finite volume multigrid method to three-dimensional flow problems in highly viscous fluid with a variable viscosity. Geophys Astrophys Fluid Dyn 83:261–291

Turcotte DL, Schubert G (2002) Geodynamics, 2nd edn. Cambridge University Press, New York

Walsha SDC, Saara MO, Bailey P, Lilja DJ (2009) Accelerating geoscience and engineering system simulations on graphics hardware. Comput Geosci 35:2353–2364
Wallin D, Lof H, Hagersten E, Holmgren S (2006) Multigrid and GaussSeidel smoothers Revisited: Parallelization on chip multiprocessors. ICS06 June 2830, Cairns, Queensland, Australia.

Chapter 22
High Rayleigh Number Mantle Convection on GPU

David A. Sanchez, Christopher Gonzalez, David A. Yuen, Grady B. Wright and Gregory A. Barnett

Abstract We implemented two- and three-dimensional Rayleigh–Benard convection on Nvidia GPUs by utilizing a 2nd-order finite difference method. By exploiting the massive parallelism of GPU using both CUDA for C and optimized CUBLAS routines, we have on a single Fermi GPU run simulations of Rayleigh number up to 6×10^{10} (on a mesh of 2000×4000 uniform grid points) in two dimensions and up to 10^7 (on a mesh of $450 \times 450 \times 225$ uniform grid points) for three dimensions. On Nvidia Tesla C2070 GPUs, these implementations enjoy single-precision performance of 535 GFLOP/s and 100 GFLOP/s respectively, and double-precision performance of 230 GFLOP/s and 70 GFLOP/s respectively.

22.1 Introduction

While a host of fluid mechanical models, such as incompressible Navier–Stokes (Cohen and Molemaker 2009; Thibault and Senocak 2009) have already been approached and solved on GPU, there is a certain need for clear and meaningful problems that are straightforward to implement on this new architecture. By virtue of their simplicity, such examples are well-suited to tutorial. In order to explore the application of CUDA programming techniques in general, and CUBLAS in particular, we investigate such a problem—the simulation of two and three dimensional

D. A. Sanchez (✉) · D. A. Yuen
Minnesota Supercomputing Institute, University of Minnesota, Minneapolis, MN 55455, USA
e-mail: sanchda@gmail.com

C. Gonzalez · D. A. Yuen
Department of Geology, University of Minnesota, Minneapolis, MN 55455, USA

G. B. Wright
Department of Mathematics, Boise State University, Boise, ID 83725, USA

G. A. Barnett
Applied Mathematics Department, University of Colorado Boulder, Boulder, CO 55455, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 335
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_22,

Rayleigh–Benard thermal convection, in the context of mantle convection, using a second-order finite difference method. Naturally, this class of examples is constantly increasing. For example, Cohen has since released the incompressible Navier–Stokes code under his OpenCurrent project (http://code.google.com/p/opencurrent/wiki/OpenCurrent).

In this model we take the limit of the Prandtl number to infinity and in so doing reduce the number of parameters to a solitary quantity: the Rayleigh number (Ra). Although we have to assure our grid mesh is sufficient to resolve the increasing convection velocities associated with a higher Ra, it is this number alone that governs the physics of our model.

22.2 Model

We model a constant-density, constant-viscosity fluid of infinite Prandtl number. We also assume the Boussinesq approximation (Guyon et al. 2001).

22.2.1 Two Dimensions

In two dimensions, coordinates are given by x and z (pointing vertically downward). This yields the following equations, which describe conservation of momentum, energy, and mass:

$$\frac{\partial \mathrm{T}}{\partial t} = \nabla^2 \mathrm{T} - \overline{v} \cdot \nabla \mathrm{T} \tag{1a}$$

$$\nabla^2 \omega = \mathrm{Ra} \frac{\partial \mathrm{T}}{\partial x} \tag{2a}$$

$$\nabla^2 \psi = -\omega \tag{3a}$$

where Ra is the Rayleigh number, T is non-dimensional temperature, $\overline{V}$ is velocity (its x-, z-components are u and v), and the vorticity ω is defined as $\nabla \times \overline{V}$ (which we note is a vector pointing out of the plane). The streamfunction ψ is given by:

$$u = -\frac{\partial \psi}{\partial z}, \quad \mathrm{v} = \frac{\partial \psi}{\partial x}. \tag{4a}$$

22.2.2 Three Dimensions

In three dimensions, our coordinates are x, y, and z. Our physical assumptions yield the energy equation:

$$\frac{\partial \mathrm{T}}{\partial t} = \nabla^2 \mathrm{T} - \bar{v} \cdot \nabla \mathrm{T}, \tag{1b}$$

and the 3-D momentum equation:

$$\nabla^2 \Omega = \mathrm{Ra}\ \mathrm{T} \tag{2b}$$

$$\nabla^2 \psi = \Omega. \tag{3b}$$

Ω is not vorticity as it was in two-dimensions, but rather is a scalar function analogous to the vorticity. ψ is the scalar function associated with the poloidal vector of the Helmholtz decomposition of an arbitrary three-dimensional vector. Time t is non-dimensionalized according to the time it takes for heat to diffuse along the height of the box. ψ also satisfies the following relationship with the components of $\overline{V}$:

$$u = \frac{\partial^2 \Psi}{\partial x \partial z}, v = \frac{\partial^2 \Psi}{\partial y \partial z}, \mathrm{w} = -\left(\frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2}\right). \tag{4b}$$

As for the boundary conditions, we wish to have the no-slip condition (no outward-pointing velocity component at the face) on all but the top and bottom faces. On the top face, we wish to have constant temperature 0. On the bottom face, we wish to have constant temperature of 1 (Larsen et al. 1997).

22.3 Numerical Solution

We wish to approximate the solution of this system by discretizing space using a second order finite difference scheme and time by a third-order Runge–Kutta method (RK3). We apply this same method to both the 2-D and 3-D cases, resulting in very similar numerical solutions. We describe the 3-D solution in detail and mention where and how the 2-D solution differs in several places (Fig. 22.1).

At every timestep k, g (defined below) is called with T^{k} as its input and the result is named q_1. Next, we compute $q_2 = g(\mathrm{T} + \frac{\Delta \mathrm{t}}{3} q_1)$ and then $q_3 = g(\mathrm{T} + \frac{2}{3}\frac{\Delta \mathrm{t}}{3} q_2)$. The final step defines the temperature matrix for the next timestep $\mathrm{T}^{\mathrm{k}+1}$ as $\mathrm{T}^{\mathrm{k}} + \frac{\Delta \mathrm{t}}{4}(q_1 + 3q_2)$. These steps correspond to the phases of RK3.

Adopting the subscript notation for partial derivatives, g is given by:

$$g(\mathrm{T}) = \mathrm{T}_{XX} + \mathrm{T}_{YY} + \mathrm{T}_{ZZ} - \Psi_{ZX} .^{*} \mathrm{T}_X - \Psi_{ZY} .^{*} \mathrm{T}_Y + (\Psi_{XX} + \Psi_{ZZ}) .^{*} \mathrm{T}_Z,$$

where ψ is the same scalar field from (3b) and (4b), found by applying the appropriate method—in this case, application of the proper DCT (described in more detail below)—to the xy- sheets (by "sheet" we mean the xy-, xz- and yz- subspaces of the uniformly discretized space the model inhabits) of T. The operation A.*B denotes elementwise multiplication of A and B. Our finite-difference method allows us to phrase

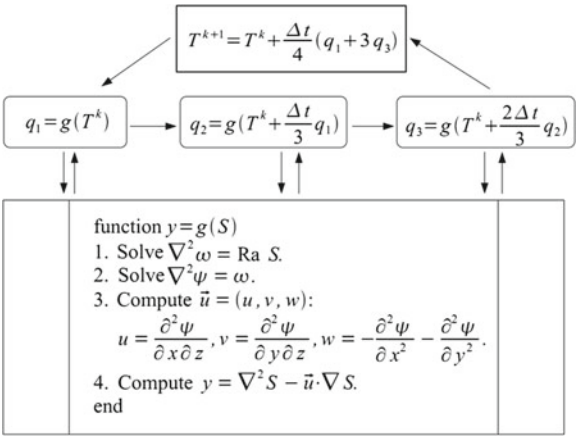


Fig. 22.1 3-D numerical solution flowchart for an arbitrary timestep k with temperature T^k and timestep width Δt

Table 22.1 Peak performance on a single Tesla C2070 in GFLOP/s

	2-D	3-D
Single precision	535	100

these first- and second-derivatives as sheetwise scalar multiplications followed by sheet-sheet additions.

In the 2-D case, g takes on the form:

$$g(\mathrm{T}) = \mathrm{T}_{XX} + \mathrm{T}_{ZZ} - \psi_Z \cdot^* \mathrm{T}_X - \psi_Y \cdot^* \mathrm{T}_Z.$$

Analogously to the 3-D case ψ is the matrix T with the Hockney method (described below) applied to it and the $\cdot^*$ operator is representative of elementwise multiplication, as before.

In the 3-D scenario we have a pair of coupled Poisson equations which can be solved generically with a discrete Fourier transform (DFT). This is often implemented by utilizing a fast Fourier transform (FFT). We employ a direct method based on a DFT (Van Loan 1992). This method involves performing two matrix–matrix multiplies per Poisson equation. A few other methods were also attempted and investigated. We summarize these methods in the table below (method A was the fastest) Table 22.1.

While remaining almost computationally identical to the 3-D method, the 2-D method employed a slightly different approach for solving the pair of coupled Poisson equations. We solved these equations using the Hockney method—a technique which predates FFT (Hockney 1965). This method utilizes DSTs, which we implement as matrix–matrix multiplications (Fig. 22.2).

Examining the definitions for g, we can see the dominating computational aspect is matrix–matrix multiplication (specifically, in determining Ψ and ψ). While this may be cause for some initial celebration—after all, matrix–matrix multiplication is considered to be fast on GPU—it may be the case that there is a lower-FLOP/s algorithm which advances through some equivalent measure of non-dimensional

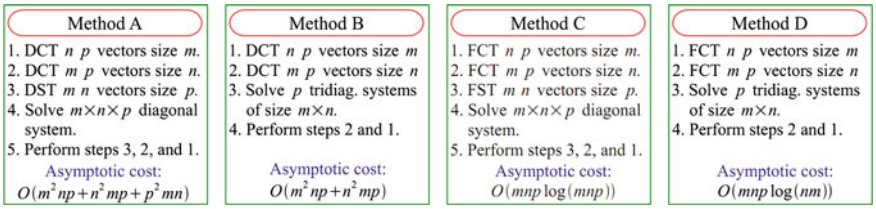


Fig. 22.2 Algorithmic comparison of various solutions for our coupled Poisson equations for the 3-D case. In addition to the aforementioned DCT, discrete sine transforms (DST), Fourier cosine transforms (FCT), and Fourier sine transforms (FST) were used

time more rapidly. This would be especially true if we were to implement the same problem on CPU and should be considered whenever a new attempt is made on different hardware architecture.

22.4 Implementation

22.4.1 CUBLAS

We make a few notes on CUBLAS conventions in order to clarify later decisions. First and foremost, CUBLAS inherits the legacy of BLAS (Basic Linear Algebra System), a library of dense linear algebra routines originally crafted for Fortran. CUBLAS uses column-major order for arrays and begins its arrays with 1 (contrasting with CUDA's adherence to C-like, 0-indexed row-major order). Accordingly, there is some risk of creating confusion when transitioning into and out of CUBLAS routines in one's code.

This can become especially troublesome when utilizing GEMM (Generalized Matrix-Matrix Multiply) routines, as arrays defined outside of CUBLAS will be interpreted as their transpose. Fortunately, many CUBLAS BLAS-3 (the BLAS level associated with matrix–matrix operations) routines can perform their tasks on the transpose as well as normal matrices. Since we can also rephrase the operation $(AB)^T$ as $B^T A^T$ (where the superscripted T denotes matrix transposition of rows with columns) for matrices A and B, it is also possible to retain a degree of row-major thinking in CUBLAS' column-major world.

To make this explicit, we provide an example from our code. Consider two matrices, an $m \times m$ matrix A and an $m \times n$ B. We have initialized these matrices according to typical C convention (row-major order), but now we would like to compute the matrix–matrix product AB from CUBLAS. The generic GEMM convention is to perform $C \longleftarrow \alpha AB + \beta C$, so regardless of our data's orientation in memory, we know is equal to one and β is equal to zero. Although GEMM accepts arguments telling it to perform the operation on A^T and B^T, it will always write C in column-major order. However, if we compute $B^T A^T$ instead of AB, the array put in C will be $(AB)^T$. This is ideal because converting arrays between row- and column-major

orders is the same as performing a matrix transpose. It is worthwhile to note that C cannot occupy the same memory location as A or B, otherwise the values in these latter matrices will be overwritten during the computation, thoroughly corrupting the entire operation. While a more complete discussion of GEMM inputs and conventions is beyond the scope of this article, we reproduce below our code for performing the above operation on the double-precision arrays A and B described above. Below is some code making use of this observation to perform a DST.

```
/* Solving Poisson equations on GPU. Single precision, two dimensions.
** Author: David A. Sanchez, 2011
** Description: This code segment illustrates how various components of
** our code fit together to solve a set of two coupled Poisson equations.
** Please refer to the article for a more complete exposition of what
** these equations are and what they do. Also refer to Hockney 1965 for
** technical details of the solution method (exact reference in article).
**
** Mainly, we use four matrix--matrix multiplications and an element-wise
** multiplication to perform the solution. The two components of the
** matrix--matrix multiplications are always one matrix whose contents we
** don't know a priori (upon which we perform the DST) and another matrix,
** dsr or dsc which performs the transform on the rows or columns,
** respectively.
*/

/* We define some arrays on the device-side in order to compute the
** elements for dsr and dsc:
** dsc: a discrete sine transform to the columns, when applied as a
** matrix--matrix multiplication to an array.
** dsr: a discrete sine transform of the rows.
** lambda, mu: intermediate arrays for defining ei
** ei: an array we perform an elementwise multiply with.
**
** Since T, temperature, has its top and bottom rows fixed for all time,
** it is not necessary for us to compute them. Accordingly, while the
** entire grid has dimensions of M and N, T as an array has dimensions of
** (M-2)xN.
**
**========================================================================
                            HOST-SIDE COMPONENT
**========================================================================*/
// 2-D arrays
float* h_dsc = (float*)malloc((M-2)*(M-2)*sizeof(float));
float* h_dsr = (float*)malloc((N-2)*(N-2)*sizeof(float));
float* h_ei = (float*)malloc((M-2)*(N-2)*sizeof(float));

// Vectors
float* h_lambda = (float*)malloc((M-2)*sizeof(float));
float* h_mu = (float*)malloc((N-2)*sizeof(float));

// Define a discrete sine transform of the columns, as in Hockney 1965
for(int i = 0; i < M-2; i++) {
    for(int j = 0; j < M-2; j++) {
       h_dsc[(M-2)*i+j]=sqrt(2.0/(M-1.0))*sin((i+1.0)*(j+1.0)*PI/(M-1.0));
    }
}
```

```
// Define a discrete sine transform of the rows, also as in Hockney 1965
for(int i = 0; i < N-2; i++) {
    for(int j = 0; j < N-2; j++) {
        h_dsr[(N-2)*i+j]=sqrt(2.0/(N-1.0))*sin((i+1.0)*(j+1.0)*PI/(N-1.0));
    }
}

// Initialize lambda and mu, which are used to compute ei.
for(int i = 0; i < M-2; i++) {
    h_lambda[i] = 2.0*cos((i + 1.0)*PI/(M - 1.0)) - 2.0;
}
for(int i = 0; i < N-2; i++) {
    h_mu[i] = 2.0*cos((i + 1.0)*PI/(N - 1.0)) - 2.0;
}

/* Compute ei from lambda and mu.
** The elements of ei are saved in reciprocal form on the last step.
** Normally, these elements would participate in an elementwise division
** between matrices, but since we perform this operation three times per
** timestep it pays to perform the division one extra time now so we can
** perform multiplications instead in the future.
*/

for(int i = 0; i < M-2; i++) {
    for(int j = 0; j < N-2; j++) {
        h_ei[(N-2)*i + j] = h_lambda[i] + h_mu[j];
        h_ei[(N-2)*i + j] = (h_ei[(N-2)*i + j])*(h_ei[(N-2)*i + j]);
        h_ei[(N-2)*i + j] = 1/(h_ei[(N-2)*i + j]);
    }
}

/* Now we allocate device-side room for the arrays omega, psi, dsc, dsr,
** and ei. These arrays all have the same shapes as their host-side
** counterparts.
** Note that d_omega was not defined in any way on the device-side.
*/
float* d_omega;
custat = cublasAlloc((M-2)*(N-2), sizeof(float), (void**)&d_omega);
if(custat != CUBLAS_STATUS_SUCCESS) printf("Could not allocate memory\n");

float* d_dsc;
custat = cublasAlloc((M-2)*(M-2), sizeof(float), (void**)&d_dsc);
if(custat != CUBLAS_STATUS_SUCCESS) printf("Could not allocate memory\n");

float* d_dsr;
custat = cublasAlloc((N-2)*(N-2), sizeof(float), (void**)&d_dsr);
if(custat != CUBLAS_STATUS_SUCCESS) printf("Could not allocate memory\n");

float* d_ei;
custat = cublasAlloc((N-2)*(M-2), sizeof(float), (void**)&d_ei);
if(custat != CUBLAS_STATUS_SUCCESS) printf("Could not allocate memory\n");

// Copy the completed data over.
cublasSetVector((M-2)*(M-2), sizeof(float), h_dsc, 1, d_dsc, 1);
cublasSetVector((N-2)*(N-2), sizeof(float), h_dsr, 1, d_dsr, 1);
cublasSetVector((M-2)*(N-2), sizeof(float), h_ei, 1, d_ei, 1);
```

```
/*============================================================================
                            GPU-SIDE COMPONENT
**==========================================================================*/

/* Although these manipulations are all being called from the CPU, the
** actual operations are being done on GPU. This is a syntactic
** convenience due to CUBLAS.
**
** In this section, we take a device-side temperature array T and subject
** it to some manipulations which will transform it into omega, the
** vorticity array.
*/

// Define omega to be the interior columns of DxT. T is currently stored
// in f so its value isn't overwritten throughout the various RK3 phases.
// DxOmega is a special routine which applies Dx only to the interior
// columns of the first argument, saving the result to the second argument
// (which is assumed to be the correct shape, as is omega).
DxOmega(f, DxT);

/* Perform some matrix multiplications. cublas assumes everything is in
** column-major, so while we want to perform:
**
** omega = dsc*omega
** omega = omega*dsr
** omega = omega.*ei
** omega = dsc*omega
** omega = omega*dsr
**
** we observe that Transpose(A*B) = Transpose(B)*Transpose(A) to perform
** these same manipulations while preserving row-major storage.
** Omega has dimensions (M-2)x(N-2), but cublas thinks this is (N-2)x(M-2)
** dsc and dsr are square, so cublas agrees with us on the dimensions,
** thought it may still disagree about the contents.
** Perform Transpose(omega) = Transpose(omega)*Transpose(dsc):
**
** ld below means "leading dimension," which is the row or column size,
** depending on whether row- or column- major order is chosen
**
**                        _M-2_
**                       |     |
**              M-2      | dsc |
**                       |_____| ld = M-2
**
**         _M-2_              _M-2_
**        |     |            |     |
**        |     |            |     |
**        |     |            | new |
**     N-2|omega|         N-2|omega|
**        |     |            |     |
**        |ld=N-2|           |ld=N-2|
**        |______|           |______|
** The call for cublasSgemm is ('n', 'n', m, n, k, alpha, A, lda, B, ldb,
** beta, C, ldc), where A is m-by-k, B is k-by-n, and C is m-by-n.
** Since A = C, k = n, so B is k-by-k. B = dsc, so k=n=M-2 and m = N-2.
*/
```

```
/* We save into a universal temperature-shaped buffer, Tbuff, even if it
** is not quite the right shape for our result--that's fine. We can't
** write directly into omega, otherwise it will be overwitten during the
** operation, leading to the corruption of its values.
*/
cublasSgemm('n','n',N-2, M-2, M-2,1.0,omega,N-2,dsc,M-2, 0.0,Tbuff,N-2);
// Now put Tbuff into omega.
cublasScopy((N-2)*(M-2), Tbuff, 1, omega, 1);

// Perform Tranpose(dsr)*Transpose(omega), store in omega.
//                            _M-2_
//                           |     |
//                           |     |
//                           |     |
//                       N-2|omega|
//                           |     |
//                           |ld=N-2
//                           |_____|
//
//          __N-2__             _M-2_
//         |       |           |     |
//         |       |           |     |
//         |       |           | new |
//      N-2| dsr   |       N-2|omega|
//         |       |           |     |
//         | ld=N-2|           |ld=N-2
//         |_______|           |_____|
// since A is square, M = k = N-2, so n must be M-2.
cublasSgemm('n','n',N-2, M-2, N-2, 1.0, dsr, N-2,omega,N-2,0.0,Tbuff,N-2);
cublasScopy((N-2)*(M-2), Tbuff, 1, omega, 1);

/* Now perform elementwise matrix multiply, storing the result in omega.
** This is an example of a GPU kernel call. grid and block were defined
** previously to the appropriate values. ElemMultOmega doesn't call into
** a generic library, we wrote it ourselves.
*/
ElemMultOmega<<<grid, block>>>(omega, ei);

// This is the exact same operation as the Transpose(omega)*Transpose(dsc)
// computation we performed previously.
cublasSgemm('n','n',N-2, M-2, M-2, 1.0, omega, N-2,dsc,M-2,0.0,Tbuff,N-2);
cublasScopy((N-2)*(M-2), Tbuff, 1, omega, 1);
// This is the exact same operation as the Transpose(dsr)*Transpose(omega)
// computation we performed previously.
cublasSgemm('n','n',N-2, M-2, N-2, 1.0, dsr, N-2,omega,N-2,0.0,Tbuff,N-2);
cublasScopy((N-2)*(M-2), Tbuff, 1, omega, 1);
/* To finish the algorithm, omega is scaled by a constant. This is part of
** the method. DX and RA are \#define'd to the mesh spacing and Ra number
** respectively.
*/
cublasSscal((N-2)*(M-2), (DX*DX*DX*DX)*(RA), omega, 1);
```

However, not all indexing conventions can be waved away. For these cases, it is possible to simplify code by creating C macros that will convert one type of index to the other. Employment of such macros improves code readability with no loss of

performance. For example, in a 2-D array with leading dimension *ld*, the following macro will rewrite conventional C-language indices into Fortran-style indices:

```
#define C2F (i,j,ld) ((((j)-1)*(ld))+((i)-1))
```

However, since one-dimensional arrays (vectors) are stored the same way in both row- and column-major ordering, our solution was to rephrase as many matrix operations as possible into vector operations. This has the additional advantage of exposing the position of data in linear memory, allowing us to keep coalescing in mind. Although such a choice also carries the risk of obscuring the underlying mathematics, we consider this a risk worth taking.

22.4.2 Routines

Our first step was to determine which operations should be done on GPU and which should be moved to (or shared by) CPU. We could have adopted a hybrid computing approach and allowed the CPU to participate in the main computation loop, but we opted to limit the CPU activities to memory initialization and file I/O. This allowed us to reliably benchmark GPU performance alone and simplify host/device memory transfer for later, multi-device development.

Because of our finite-difference scheme, we have a class of operations that look like the accumulation of adjacent xy-, xz-, or yz-sheets of the temperature array. As mentioned, we would like to treat this as a class of vector operations in order to expose the arrangement of matrix elements in linear memory. Clearly, this will have to be done differently for all three orientations of 2-D arrays within the grid.

For example, the second-order finite-difference approximation to the first derivative in x is defined for each x-slice X_i (yz-sheet) in the interior of the matrix as $(\mathrm{X}_i)_x = \mathrm{X}_{i+1} - X_{i-1}$. The obvious way of trying to implement this in code is to simply iterate through the interior X_i, performing this accumulation as we go. Due to the way these slices are arranged in linear memory, each X_i is going to look like some vector with a large stride (separation between its elements). In hardware, if the 3-D index (i, j, k) is on an array of dimensions n, m, and p, then the corresponding 1D index is of the form $((kn + i)m + j)$.

Since GPU is best at trying to access contiguous arrays in memory, a better way of performing this operation is by realizing the accumulation on an element-by-element basis, instead of a sheet-by-sheet one. Specifically, since we wish to iterate through the X_i of the array, a practical alternative is to iterate through the $\mathrm{Z'}_i$ (where each $\mathrm{Z'}_i$ is a Z_i missing its first and last rows, because this derivative is only defined on the interior x-slices), moving elements within the plane instead of moving perpendicular to it. This method enjoys much greater efficiency. For an input matrix f and output y with x-y-z dimensions given by N, M, and P respectively, we implement this operation as follows:

```
  // First second-order derivative with respect to x.
  // X\_i = X\_i+1 -- X\_i-1.
  void Dx(double* f, double* y) \{
    for(int k = 0; k $<$ P-2; k++) \{
        // Notice that these vectors have a stride of 1,
which is our
        // best-case scenario!
        cublasDcopy((M-2)*N, f+k*M*N+2*M, 1, y+k*M*N+M, 1);
        cublasDaxpy((M-2)*N, -1.0, f+k*M*N, 1, y+k*M*N+M, 1);
  }
  // Scale everything by 1/(2*DX), for mesh-spacing DX, to
  finish the // derivative.
  cublasDscal(M*N*(P-2),REC2-DX,y,1);
  return;
}
```

22.5 Performance

While our analysis and visualization used data from double-precision runs exclusively, we also timed single-precision code for both two and three dimensional cases. Best-case performance was achieved for all cases on aspect 1:1 or 1:1:1 grids, and tabulated below. These figures were obtained by running the code on a single Tesla C2070. We note that the 2-D single-precision performance approaches the GPU's maximum matrix–matrix multiply capability.

If a plot is made of the 2-D performance on a square aspect ratio, with data points taken when a side length is a multiple of 4, a staggered pattern develops. This appears for a number of reasons. Many threads remain unused during the matrix–matrix multiply if the blocks they inhabit are asked to compute fewer than sixteen elements at the end of a vector. Additionally, memory transactions are optimal only when a multiple of the pipeline width is fetched or stored, also introducing a new degree of memory latency to these edge cases. While one may think a suitable alternative is to have these unused threads compute the elements of the next vector, it turns out that the latency of global memory makes this an unattractive option. These innocuous details seem trivial, but added together they can account for fluctuations on the order of 50 GFLOP/s in the breadth of only a few array elements. For this reason, it may be economical to perform a slightly larger problem than intended, particularly if it forces the various computational elements to align into the proper multiples of sixteen.

On the other hand, 3-D single-precision performance has a different characteristic. Despite its similar computational nature, its performance is markedly lower than that of the 2-D code. Generically one may attribute this to sloppy coding—for a wide range of problems the 3-D case is considered more computationally efficient than 2-D, since proper utilization of fast (or otherwise cached) memory allows computational saturation to improve far beyond what is possible in the 2-D case. Unfortunately,

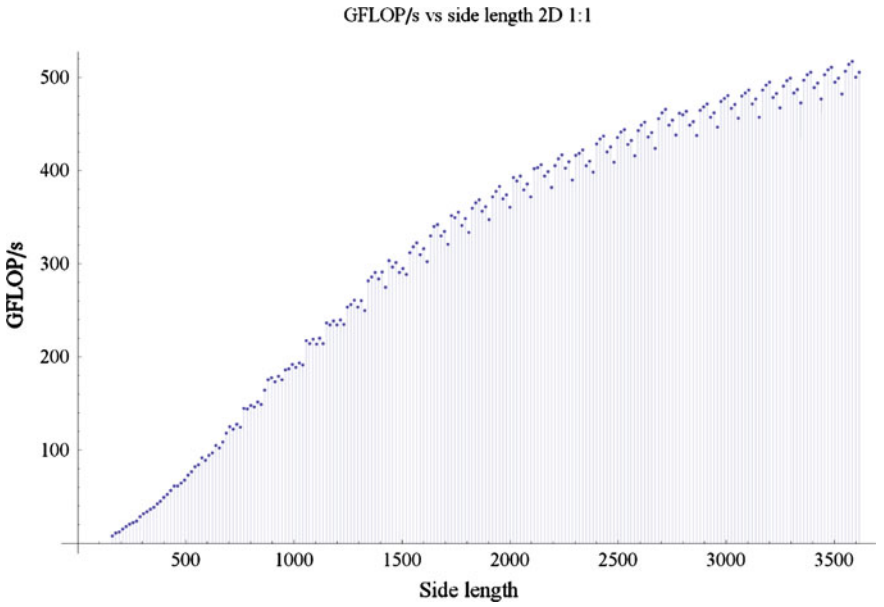


Fig. 22.3 2-D single-precision performance on a *square grid*, where length of side is a multiple of 4

this observation is not universal. Since the 3-D problem looks like the 2-D problem stacked up on itself, there is not enough memory available to make the size of each matrix–matrix multiply very large. While we can consider matrices with dimensions on the order of 3500×3500 in the 2-D case, we are reduced to dimensions of 450×450 for 3-D. This is insufficient for properly saturating the GPU, forcing us to live with suboptimal performance (Fig. 22.3).

22.5.1 Accelerated GPU Code

Previously, we had implemented this code on GPU by accelerating it from Matlab with AccelerEyes Jacket (Barnett et al. 2009). In this context, the 3-D CPU (Intel Core i7 960, running at 2.93 GHz) code could simulate $400 \times 400 \times 200$ grid points in slightly less than 44 s per timestep, while the Jacked-accelerated code on a Fermi GPU could perform the same computation in about 10 s. On the other hand, the 3-D CUDA code on a Fermi GPU can simulate these $400 \times 400 \times 200$ grid points in about 5.18 s per timestep. Another benefit in hand-crafting the code was a greatly reduced memory signature, allowing us to utilize the same GPU hardware for even larger runs. Matlab and AccelerEyes have since improved their GPU offering, which may necessitate a renewed investigation at a later time (Fig. 22.4).

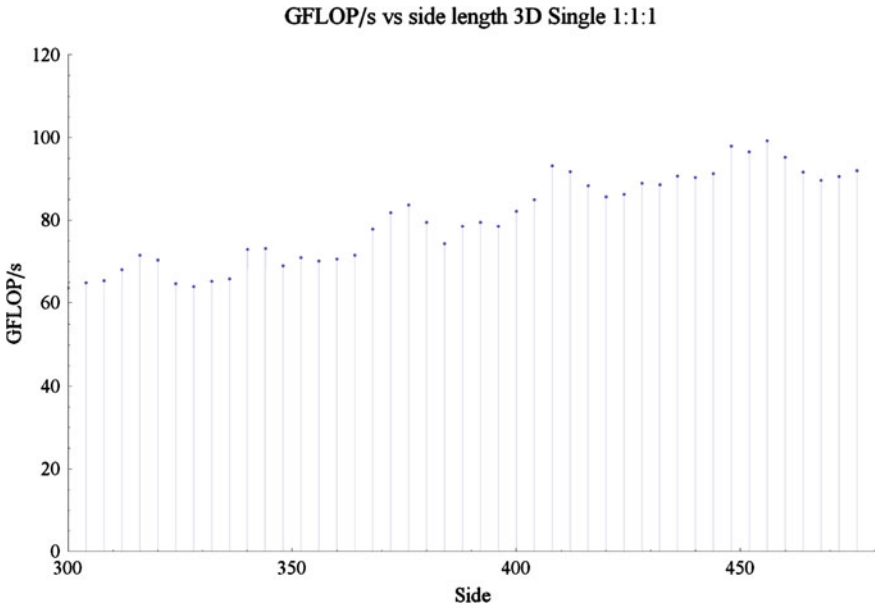


Fig. 22.4 3-D single-precision performance on a *cubic grid*

22.5.2 Application

The major application of our model was to simulate flow in a primitive Earth's mantle. The mantle today is approximately 2,850 km thick and varies greatly both chemically and physically with depth. Flow in the mantle is a solid-state flow influenced by subductive processes from plate tectonics and chemically distinct regions. Heat that drives convection in the mantle is generated by the radioactive decay of unstable isotopes and as well as heat transfer from the core-mantle boundary.

The primitive mantle differed greatly compared to the present day mantle in several respects. Generally, the primitive mantle is believed to have been much less chemically mature with higher overall convective velocities (Solomatov 2007). Additionally, Earth was undergoing massive bombardment by large bodied planetesimals leftover from the formation of the solar system during its accretion period (Melosh and Tonks 1993). This period of Earth's history is called the accretion period and was an extremely hostile environment. The impacts generated enough energy that a significant portion of the Earth's mantle became molten (Melosh and Tonks 1993). The impacts, together with heat being generated from radioactive decay and gravitational coalescing of the inner Earth, resulted in a mantle with an enormous amount of thermal energy.

This excess thermal energy influenced the overall mantle viscosity. The viscosity was much lower than the contemporary motion and as a result fast amounts of magma began pond at the surface of the planet. This pooling of magma earned the phrase

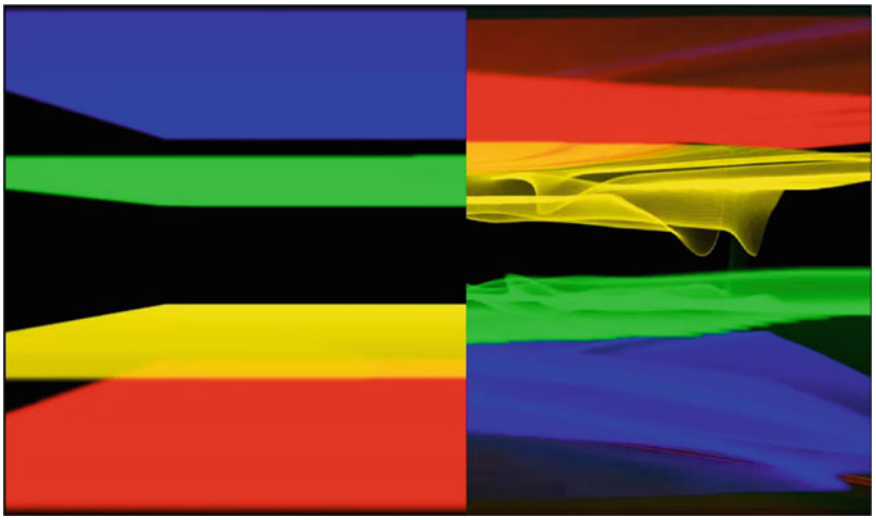


Fig. 22.5 Boss transition of the temperature field in 3-D, using isosurfaces at Ra $= 10^7$. When temperature is normalized to the interval between zero and one, *red* displays a temperature of 0.9, *yellow* a temperature of 0.75, *green* a temperature of 0.25, and *blue* a temperature of 0.1. The left half of the figure is the visualization at initial condition, while the right half is a time shortly thereafter

"magma ocean" to describe the situation (Melosh and Tonks 1993). Our model aimed to investigate this style of turbulent convection and its characteristics right after a planet's formation, but just prior to any crystallization that occurred shortly after.

The behavior of our model is governed by the single dimensionless quantity known as the Rayleigh number (Ra) (Breuer and Hansen 2009). It has been suggested that convective motion begins at Ra $\sim 10^3$. To compare, the present mantle is thought to have Ra on the order of 10^5–10^7 (Karato 2003). On the other hand, computations indicate magma oceans have Ra around 10^{28}–10^{29} (Solomatov 2007). It is interesting to note a convective feature of our model known as the Boss transition. This transition occurs as a result of our initial conditions (which is a linear temperature profile with its cold side near the top), which the model reacts to by inverting the temperature column to facilitate more rapid transport of heat.

Our model utilized Ra as high as 6×10^{10}. By doing so, we were looking for two characteristics that commonly frequent thermal turbulent convection—hard turbulence and flow reversals (Fig. 22.5).

Hard turbulence is a convective motion in which the entire system flows in an organized fashion (Solomatov 2007). This turbulent characteristic of whole-mantle convection has been thought to arise under various conditions. One particular view advanced by numerical studies suggests that global circulation is the result of convective rolls in which the wave number increases with higher Rayleigh numbers (Breuer and Hansen 2009).

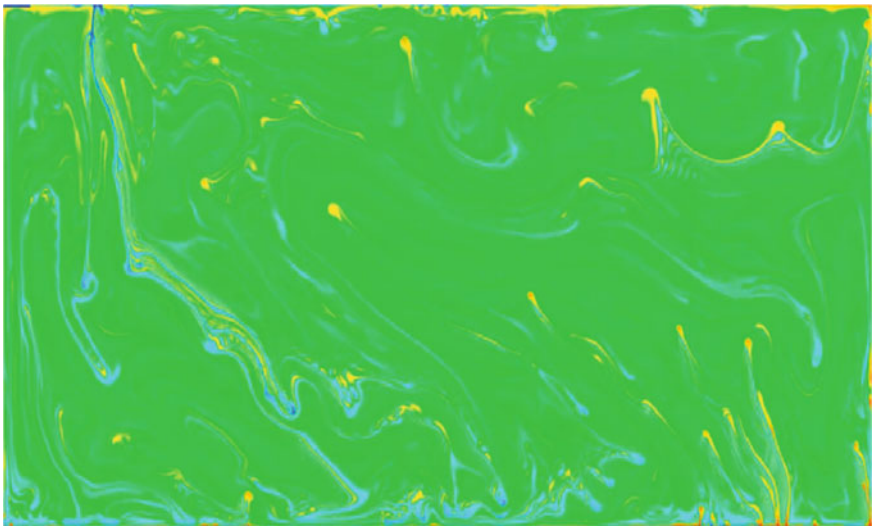


Fig. 22.6 Global circulation at Ra $= 6 \times 10^{10}$. Visualization is a nonlinear color map of the temperature field, where *blue* represents *cold*, *red* and *yellow* represent hot, and *green* represents a temperature close to average

Flow reversals are concerned with organization of a different kind. Specifically, flow taking place in the convective system is only temporarily stable. As such, the orientation and the direction of the flow constantly shifts and changes (Breuer and Hansen 2009). When these changes occur, a change in the overall structure of the flow takes place (Fig. 22.6). The flow will begin to stagnate and if the conditions are right, the orientation of the flow will reverse 180°. This reversal is only temporary and the overall global circulation will remain approximately constant once the flow reverses again to its original orientation (Breuer and Hansen 2009).

There are several mechanical characterizations of flow that result in reversals; however the phenomenon of the flow reversal itself and the reorganization of the flow are poorly understood in the realm of Rayleigh–Benard convection. Regardless of this poor understanding relating to flow reversals in both the present day mantle and a primitive mantle, empirical evidence suggests that flow reversals may be at the heart of a rapid shifting in the tectonic plates (Ghiaz and Jarvis 2007). A perfect case of this is seen in the rapid shifting of the Hawaiian-Emperor sea mount chain.

Another important convective feature of the model is a scalar quantity known as the Nusselt number (Nu). Nu is a dimensionless number which is defined as the ratio between convective and conductive heat transfer. Because faster convection transfers more heat, it is particularly useful for describing the intensity of the convection at a given point in the system's evolution (Fig. 22.7). For example, a run with Ra $= 3\times10^{10}$ suddenly being perturbed mid-run to Ra $= 6\times10^{10}$ results in a corresponding increase in Nu. A following figure captures this information in histogram form. This number was also useful for us as a diagnostic measure of our model's numerical

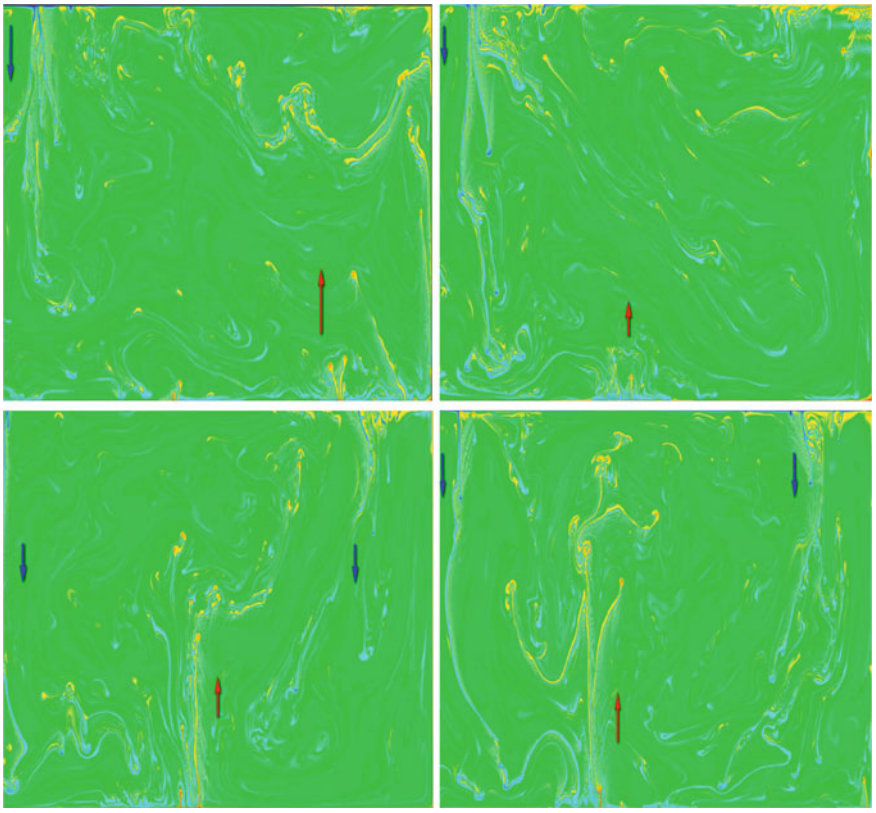


Fig. 22.7 The evolution of a flow-reversal, counter-clockwise. Ra $= 6 \times 10^{10}$, temperature field

correctness. Specifically, we found agreement with published results (Zhong 2005) on a wide range of cases in 3-D (Fig. 22.8).

22.6 Conclusion

We enjoyed a practical benefit from implementing our code on GPU using CUDA and CUBLAS. Achieving a maximum of 535 GFLOP/s in single-precision for the 2-D code on a Fermi GPU, we utilized over half of the theoretical performance of the hardware. In addition, use of standard libraries makes the software very modular, allowing us to easily use systems such as MPI to scale our code to multiple GPUs in the future. In particular, we are working towards implementing this 3-D code on the full 360-GPU body of the NSF's new Keeneland GPU cluster (Fig. 22.9).

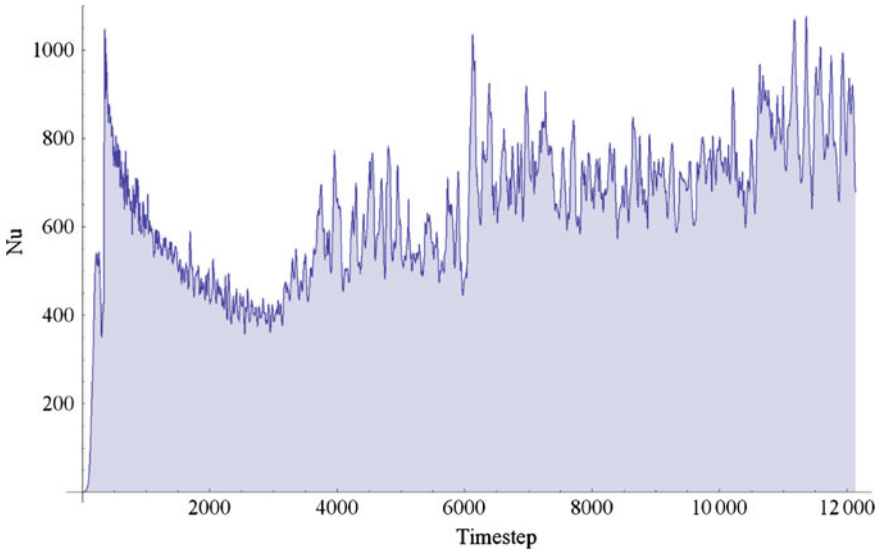


Fig. 22.8 Nu over time. Note that at 300,000, Ra doubles from 3×10^{10} to 6×10^{10}

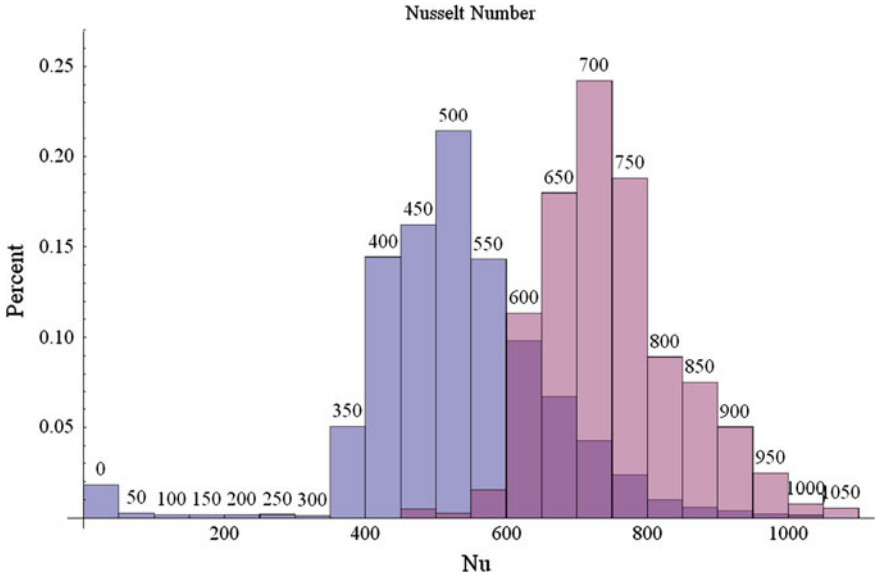


Fig. 22.9 Nu histogram corresponding to the data in (Fig. 22.3). Ra 3×10^{10} is in *blue* and 6×10^{10} is in *purple*

Acknowledgments We thank Matt Knepley for stimulating discussions on GPU. This research has been supported by NSF CMG grant.

References

Breuer M, Hansen U (2009) Turbulent convection in the zero Reynolds number limit. EPL 86(2):3024–3027

Cohen JM, Molemaker MJ (2009) A fast double precision CFD code using CUDA. Proceedings of parallel CFD 2009

Ghiaz S, Jarvis G (2007) Mantle flow reversals in cylindrical earth models, phys Earth Planet. Interiors 165(3–4):194–207

Guyon E, Hulin JP, Petit L, Mitescu CD (2001) Physical hydrodynamics. Oxford University Press, Oxford, p 505

Hockney RW (1965) A fast direct solution of Poisson's equation using Fourier analysis. J ACM 12(1):95–113

Karato S (2003) The dynamic structure of the deep earth: an interdisciplinary approach. Princeton University Press, Chichester

Larsen TB, Yuen DA, Moser JM, Fornberg B (1997) A high-order finite-difference method applied to large Rayleigh number mantle convection. Geophys. and Astrophys. Fluid Dyn 84:53–83

Solomatov VS (2007) Magma oceans and primordial mantle differentiation. In: Schubert G (ed) Treatise on Geophysics, vol 9. Elsevier, Oxford, pp 91–120

Thibault JC, Senocak I (2009) CUDA implementation of a Navier-Stokes Solver on multi-GPU desktop platforms for incompressible flows. 47th AIAA Aerospace Sciences Meeting, paper no: AIAA-2009-758, 2009

Tonks WB, Melosh HJ (1993) Magma ocean formation due to giant impacts. J Geophys Res 98:5319–5333

Van Loan C (1992) computational frameworks for the fast fourier transform. SIAM Publications, Philadelphia

Barnett GA, Wright GB, Yuen DA (2009) GPU implementation for three-dimensional mantle convection at high Rayleigh number. AGU Fall Meeting. San Francisco, CA, December 14–18, 2009. Contributed. ID DI31A-1602

Zhong S (2005) Dynamics of thermal plumes in three-dimensional isoviscous thermal convection. Geophys J Int 162:289–300

Chapter 23
High-Order Discontinuous Galerkin Methods by GPU Metaprogramming

Andreas Klöckner, Timothy Warburton and Jan S. Hesthaven

Abstract Discontinuous Galerkin (DG) methods for the numerical solution of partial differential equations have enjoyed considerable success because they are both flexible and robust: They allow arbitrary unstructured geometries and easy control of accuracy without compromising simulation stability. In a recent publication, we have shown that DG methods also adapt readily to execution on modern, massively parallel graphics processors (GPUs). A number of qualities of the method contribute to this suitability, reaching from locality of reference, through regularity of access patterns, to high arithmetic intensity. In this article, we illuminate a few of the more practical aspects of bringing DG onto a GPU, including the use of a Python-based metaprogramming infrastructure that was created specifically to support DG, but has found many uses across all disciplines of computational science.

23.1 Introduction

Discontinuous Galerkin methods (Reed and Hill 1973; Lesaint and Raviart 1974; Cockburn et al. 1990; Hesthaven and Warburton 2007) are, at first glance, a rather curious combination of ideas from Finite-Volume and Spectral Element methods. Up close, they are very much high-order methods by design. But instead of perpetuating

A. Klöckner (✉)
Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA
e-mail: kloeckner@cims.nyu.edu

T. Warburton
Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005, USA
e-mail: timwar@caam.rice.edu

J. S. Hesthaven
Division of Applied Mathematics, Brown University, Providence, RI 02912, USA
e-mail: jan.hesthaven@brown.edu

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 353
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_23,

the order increase like conventional global methods, at a certain level of detail, they switch over to a decomposition into computational elements and couple these elements using Finite-Volume-like surface Riemann solvers. This hybrid, dual-layer design allows DG to combine advantages from both of its ancestors. But it adds a third advantage: By adding a movable boundary between its two halves, it gives implementers an added degree of flexibility when bringing it onto computing hardware.

Using graphics processors for computational tasks is by no means a new idea. In fact, even in the days of marginally programmable fixed-function hardware, some (especially particle-based) methods obtained large performance gains from running on early GPUs (e.g., Li et al. 2003). In the domain of solvers for partial differential equations, Finite-Difference Time-Domain (FDTD) methods are a natural fit to graphics processors and obtained high performance with relative ease (e.g., Krakiwsky et al. 2004). Finite Element solvers were also brought onto GPUs relatively early on (e.g., Göddeke et al. 2005), but often failed to reach the same impressive speed gains observed for the simpler FD methods. In the last few years, high-level abstractions such as Brook and Brook for GPUs (Buck et al. 2004) have enabled more and more complex computations on streaming hardware. Building on this work, Barth and Knight (2005) already predicted promising performance for two-dimensional DG on a simulation of the Stanford Merrimac streaming architecture (Dally et al. 2003). Nowadays, compute abstractions are becoming less encumbered by their graphics heritage (Lindholm et al. 2008; Nvidia Corporation 2009). This has helped bring algorithms of ever higher complexity onto the GPU. Taking advantage of these advances, our paper (Klöckner et al. 2009) presented, to the best of our knowledge, the first implementation of a discontinuous Galerkin method on a single real-world consumer graphics processor. Now, a few years after the publication of the original paper, interest in GPUs and their use for solving partial differential equations continues unabated. A few implementers have followed in our footsteps and brought their versions of DG onto GPUs.

Let us briefly place this text within the sequence of articles on GPU-DG we have authored. The first one (Klöckner et al. 2009) is rather technical and introduces all the tricks and details needed to make the method go fast. The second one (Klöckner et al. 2011a) serves as an introduction to be read by a larger, somewhat non-technical audience. This latest one addresses some of the software challenges involved in achieving fast execution of GPU-based discontinuous Galerkin methods.

The article is structured as follows: In Sect. 23.2, we review the details of the discontinuous Galerkin method and its implementation in general, followed by a discussion of considerations required by its implementation on GPUs specifically in Sect. 23.3. Responding to the challenges of this section, we introduce the motivation and implementation details of our Python-based infrastructure for run-time code generation (RTCG) in Sect. 23.4. We then discuss the specifics of RTCG in the context of DG in Section and confirm the success of the method through experimental results in Sect. 23.5. In closing and summing up what was achieved, we outline avenues for future work in Sect. 23.6.

23.2 The Discontinuous Galerkin Method

By their design and origins, DG methods are particularly suited to approximating the solution of a hyperbolic system of conservation laws

$$u_t + \nabla \cdot F(u) = 0. \tag{23.1}$$

Initial boundary value problems for PDEs that can be cast in the form (23.1) as well as slight generalizations thereof, include Maxwell's equations, Euler's equations of gas dynamics, the Navier-Stokes equations, equations arising from Lattice-Boltzmann models, the equations of magnetohydrodynamics, or the shallow-water equations. In summary, a wide variety of physical phenomena in the time domain can be modeled using this type of equation.

Equation (23.1) is to be solved on a domain $\Omega = \biguplus_{k=1}^{K} \mathsf{D}_k \subset \mathbb{R}^d$ consisting of disjoint, face-conforming tetrahedra D_k with boundary conditions

$$u|_{\Gamma_i}(x,t) = g_i(u(x,t),x,t), \quad i = 1,\ldots,b,$$

at inflow boundaries $\biguplus \Gamma_i \subseteq \partial\Omega$. As stated, we will assume the flux function F to be linear. We find a weak form of (23.1) on each element D_k:

$$\begin{aligned} 0 &= \int_{\mathsf{D}_k} u_t\varphi + [\nabla \cdot F(u)]\varphi \,\mathrm{d}x \\ &= \int_{\mathsf{D}_k} u_t\varphi - F(u)\cdot\nabla\varphi \,\mathrm{d}x + \int_{\partial\mathsf{D}_k} (\hat{n}\cdot F)^*\varphi \,\mathrm{d}S_x, \end{aligned}$$

where φ is a test function, and $(\hat{n}\cdot F)^*$ is a suitably chosen numerical flux in the unit normal direction $\hat{n}$. Following Hesthaven and Warburton (2007), we find a 'strong'-DG form of this system as

$$0 = \int_{\mathsf{D}_k} u_t\varphi + [\nabla \cdot F(u)]\varphi \,\mathrm{d}x - \int_{\partial\mathsf{D}_k} [\hat{n}\cdot F - (\hat{n}\cdot F)^*]\varphi \,\mathrm{d}S_x. \tag{23.2}$$

We seek to find a numerical vector solution $u^k := u_N|_{\mathsf{D}_k}$ from the space $P_N^n(\mathsf{D}_k)$ of local polynomials of maximum total degree N on each element. We choose the scalar test function $\varphi \in P_N(\mathsf{D}_k)$ from the same space and represent both by expansion in a basis of $N_p := \dim P_N(\mathsf{D}_k)$ Lagrange polynomials l_i with respect to a set of interpolation nodes (Warburton 2006). We define the mass, stiffness, differentiation, and face mass matrices

$$M_{ij}^k := \int_{\mathsf{D}_k} l_i l_j \,\mathrm{d}x, \tag{23.3a}$$

$$S_{ij}^{k,\partial v} := \int_{\mathsf{D}_k} l_i \partial_{x_v} l_j \,\mathrm{d}x, \tag{23.3b}$$

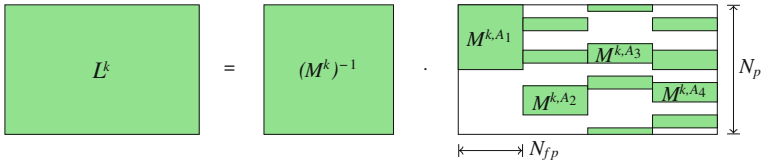


Fig. 23.1 Construction of the lifting matrix L^k

$$D^{k,\partial v} := (M^k)^{-1} S^{k,\partial v}, \tag{23.3c}$$

$$M_{ij}^{k,A} := \int_{A \subset \partial \mathsf{D}_k} l_i l_j \, \mathrm{d}S_x. \tag{23.3d}$$

Using these matrices, we rewrite (23.2) as

$$\begin{aligned} 0 &= M^k \partial_t u^k + \sum_v S^{k,\partial_v}[F(u^k)] - \sum_{F \subset \partial \mathsf{D}_k} M^{k,A}[\hat{n} \cdot F - (\hat{n} \cdot F)^*], \\ \partial_t u^k &= -\sum_v D^{k,\partial_v}[F(u^k)] + L^k[\hat{n} \cdot F - (\hat{n} \cdot F)^*]|_{A \subset \partial \mathsf{D}_k}. \end{aligned} \tag{23.4}$$

The matrix L^k used in (23.4) deserves a little more explanation. It acts on vectors of the shape $[u^k|_{A_1}, \ldots, u^k|_{A_4}]^T$, where $u^k|_{A_i}$ is the vector of facial degrees of freedom on face i. For these vectors, L^k combines the effect of applying each face's mass matrix, embedding the resulting facial values back into a volume vector, and applying the inverse volume mass matrix. Since it "lifts" facial contributions to volume contributions, it is called the *lifting matrix*. Its construction is shown in Fig. 23.1.

It deserves explicit mention at this point that the left multiplication by the inverse of the mass matrix that yields the explicit semidiscrete scheme (23.4) is an element-wise operation and therefore feasible without global communication. This strongly distinguishes DG from other finite element methods. It enables the use of explicit (e.g., Runge-Kutta) time stepping and greatly simplifies parallel implementation efforts such as this one.

23.2.1 Implementing DG

DG decomposes very naturally into four stages, as visualized in Fig. 23.2. This clean decomposition of tasks stems from the fact that the discrete DG operator (23.4) has two additive terms, one involving an element volume integral, the other an element surface integral. The surface integral term then decomposes further into a 'gather' stage that computes the term

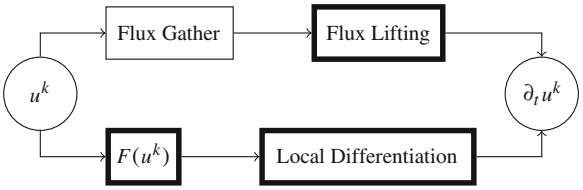


Fig. 23.2 Decomposition of a DG operator into subtasks. Element-local operations are highlighted with a *bold outline*

$$[\hat{n} \cdot F(u_N^-) - (\hat{n} \cdot F)^*(u_N^-, u_N^+)]|_{A \subset \partial \mathsf{D}_k}, \tag{23.5}$$

and a subsequent lifting stage. The notation u_N^- indicates the value of u_N on the face A of element D_k, u_N^+ the value of u_N on the face opposite to A.

As is apparent from the use of a Lagrange basis, we employ a *nodal* version of DG, in which the stored degrees of freedom ("*DOF*s") represent the values of u_N at a set of interpolation nodes. This representation allows us to find the facial values used in (23.5) by picking the facial nodes from the volume field. (This contrasts with a *modal* implementation in which DOFs represent expansion coefficients in a non-Lagrange basis. Finding the facial information to compute (23.5) requires a different approach in these schemes.)

Observe that most of DG's stages are *element-local* in the sense that they do not use information from neighboring elements. Moreover, these local operations are often efficiently represented by a dense matrix-vector multiplication on each element.

It is worth noting that since simplicial elements only require affine transformations Ψ_k from reference to global element, the global matrices can easily be expressed in terms of reference matrices that are the same for each element, combined with scaling or linear combination, for example

$$M_{ij}^k = \underbrace{\left|\det \frac{\mathrm{d}\Psi_k}{\mathrm{d}r}\right|}_{J_k :=} \underbrace{\int_{\mathsf{I}} l_i l_j \,\mathrm{d}x}_{M_{ij} :=}, \tag{23.6a}$$

$$S_{ij}^{k,\partial v} = J_k \sum_{\mu} \frac{\partial \Psi_v}{\partial r_\mu} \underbrace{\int_{\mathsf{I}} l_i \partial_{r_\mu} l_j \,\mathrm{d}x}_{s_{ij}^{\partial \mu} :=}, \tag{23.6b}$$

where $\mathsf{I} = \Psi_k^{-1}(\mathsf{D}_k)$ is a reference element. We define the remaining reference matrices D, M^A, and L in an analogous fashion.

23.3 GPU-DG: Motivation and Challenges

As we begin our study of bringing DG methods onto GPU-like architectures, we should first establish what we intend to achieve in doing so. Our main motivation is a gain in performance available from a desk-side workstations. We believe that the amount of computing power easily available to an engineer often determines the amount of computing power used in a given engineering challenge. Remote resources such as big clusters provide large amounts of power quite readily, but their use also implies a complexity burden in management, cost, and access. Nonetheless, good performance on clusters and large machines is clearly a secondary goal. Next, we would like to be able to apply the technology under discussion to a wide range of partial differential equations. While DG methods are designed for and best suited to hyperbolic PDEs, there is no conceptual restriction to this type of PDE—and our GPU-DG technology is not restricted in this way, either. A tertiary goal of ours is to make the technology not just worthwhile on a desk-side workstation, but also simple enough to apply that an engineer can easily manage his or her own computations. While this is partially a software design issue beyond the scope of this article, some prerequisites at the GPU computation level must be met to accommodate the desired ease of use. In particular, no knowledge of GPU computing is necessary to manage a computation.

The discontinuous Galerkin method further allows considerable user choice at the level of the reference discretization. It is not practical to support *all* such choices, and thus we introduce the following (fairly non-restrictive) stipulations:

- We will specialize to straight-sided simplices, as we perceive the required volume mesh generation machinery to be the most mature for this type of element. The restriction to straight-sidedness is comparatively easy to lift (Warburton 2010).
- We will optimize for (but not specialize to) the three-dimensional case, i.e. tetrahedral elements, as it bears both the most relevance to application problems and the greatest computational complexity.
- We will further optimize for "medium" order ($N = 3, \ldots, 5$) polynomial spaces, as those maintain the DG time step restriction ($\Delta t \sim \Delta x / N^2$, see Hesthaven and Warburton (2007)) at a reasonable level.

Next, we consider what possible obstacles our effort to bring DG to the GPU may face. Perhaps the first challenge that comes to mind is that of data movement. As in any matrix-product-type workload, there is much data reuse in DG, but matrices grow rapidly as N increases. In terms of data reuse, that is good—however it does compete with the limited size of on-chip memories that are needed to realize the possible reuse. Further, while on-chip memories are growing at a moderate pace and management of these memories becomes more automatic, it should be noted that even CPU-based matrix-matrix multiplies benefit from explicit management of their L1 caches (Bilmes et al. 1997; Whaley et al. 2001) in matrix-based workloads. Strongly interwoven with the challenge of having to manage on-chip memories is that of accommodating hardware granularities, such as memory sizes, memory bus widths,

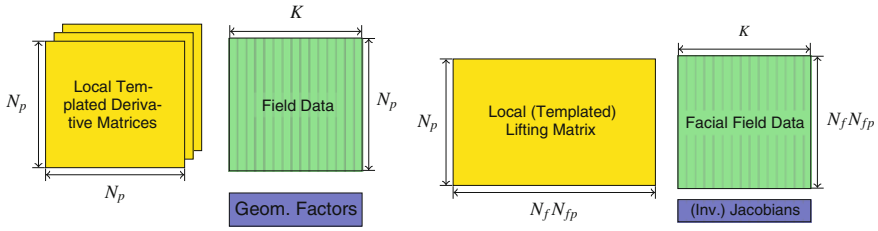


Fig. 23.3 Workload size characterization for element-local linear operators. *Left* element-local differentiation. *Right* lifting from flux values along element faces into volume data. In each case, matrix sizes are given in terms of the quantities of Sect. 23.2

SIMD widths, workgroup sizes, and so on. In addition, discontinuous Galerkin methods have their own preferred granularities, such as the number of degrees of freedom in each element, the number of dimensions, or the number of degrees of freedom on an element's face. Unfortunately, while machine-related granularities have a tendency to be powers of two, the exact opposite is true of those related to the numerical method. Another challenge posed by GPU computing is the necessity to map the computation onto a two-level, grid-based parallel execution structure, with the first level corresponding to parallelization across cores, and the second to parallelization across SIMD lanes within a core. While a coarse-grain structure may often be immediate, various finer details of this choice require careful tuning.

We will now discuss how these challenges are met by our approach, through a few representative examples, beginning with the question of data movement for element-local linear operators such as lifting and elementwise differentiation. Figure 23.3 illustrates the type and size of data that these procedures operate on. The figure also makes it obvious that while the method primarily relies on matrix-vector products, it is profitable to view the field vectors in aggregate as a matrix, thereby giving rise to a matrix-matrix computation, albeit with very off-balance matrix dimensions. An obvious first approach would be to use vendor-supplied BLAS matrix libraries for such a task, however it turns out that these are often tuned for large, square matrices and rarely deal well with the matrix sizes occurring in DG. One is therefore left to build a home-grown algorithm. Given that, depending on the local polynomial order N, only a limited amount of this data can fit onto the chip, the implementer is faced with a decision of which data to store locally and which data to stream onto the chip. In particular, one might consider the following alternatives:

- Store the matrices, stream the vector data. This seems like an obvious choice—however the matrices are often too large, and vector data is much more easily partitioned.
- Store part of a matrix. This complicates the access logic, but can often profitably be done—especially by using row-wise partitions.
- Store only field data. If streaming of the matrix is achieved through a cached data path, this can also be an attractive option.

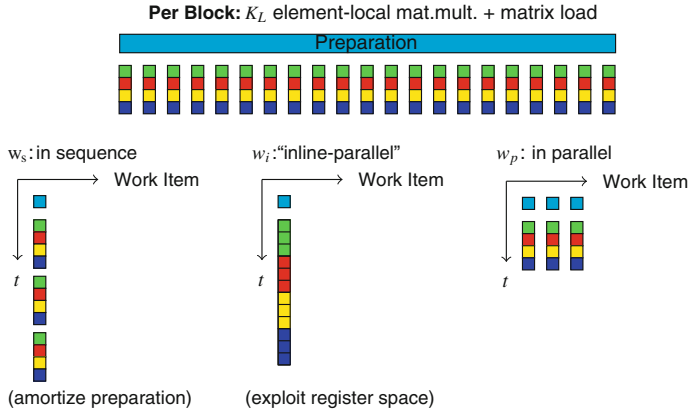


Fig. 23.4 Choices for the amount of work done by a workgroup in an element-local (differentiation, lift) operation

- Store parts of the matrix and the field vector. While this could, in theory, provide the best balance of data reuse, we were unable to turn this approach into competitive code.

This is a choice that an implementer needs to make, however we have found no universally valid heuristic that might provide guidance on which alternative to choose, especially given that the optimality of each option strongly depends on the hardware being used.

For the same workload of elementwise local differentiation and lifting, there is also the question of which work decomposition to use, where the work decomposition is given (in vendor-neutral OpenCL terminology) by the number of workgroups and their sizes. Each of these quantities can further be decomposed into a three-component vector. Order in this three-component vector matters, as it determines which work items execute memory accesses at the same time, and which branches may require serialization.

Abstractly, the workload under consideration consists of an (optional) preparatory step that preloads matrix data into on-chip memory, followed by dot products for each matrix row and all the columns (field vectors). The most immediate choice would be to have each workgroup deal with one such matrix-vector product, leading to a one-to-one mapping between workgroups and DG elements. While this is certainly straightforward, it has a number of drawbacks. For the polynomial orders N targeted in this work, these workgroup sizes are unable to fill the (32- or 64-) wide SIMD architectures exhibited by today's GPUs—at least not efficiently, and not without leaving unused 'gaps' in the SIMD vector. Further, one also typically adds padding to conform to a device's memory alignment, and this choice leads to a maximum number of gaps in the data, thereby wasting a considerable amount of (typically precious) GPU memory. In addition to that, if one workgroup only performs one matrix-vector product, any preparation steps would be poorly amortized.

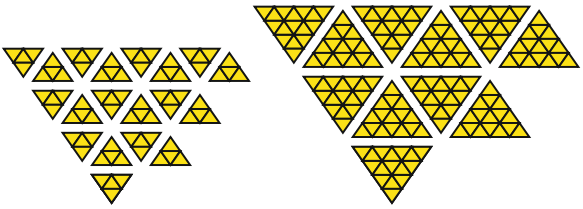


Fig. 23.5 Multiple granularities for inter-element flux computation. Obviously, larger blocks lead to more data reuse as fewer face pairs are split

Irrespective of more advanced blocking options (as described in Klöckner et al. 2009), there are three basic, orthogonal (i.e. arbitrarily combinable) possibilities of remedying this situation, outlined in Fig. 23.4. Two of these choices are entirely obvious, the third slightly less so. The obvious choices include letting a workgroup do more things *in sequence* and *in parallel*. The former of these leads to better amortization of preparation steps, while the latter does that, and in addition increases utilization of parallel processing resources. The third choice, called *in-line parallel* in Fig. 23.4, occupies a middle-ground between the two by accomplishing a number of dot products along with each other within a single work item. This exploits the fact that, in order for the matrix to be operated on, its components must be resident within the GPU's register file—but once they are there, it is economical to use them not just once, but multiple times. All of these strategies are specific forms of *work item coarsening*. How many elements are worked on in each of these fashions is captured by the numbers w_s, w_i and w_p.

Obviously, regardless of the choices for these numbers, the same amount of work is begin done—it is just the partitioning that differs. Nonetheless, in Sect. 23.5, we will observe fairly significant performance differences between such partitionings.

We have just seen that a question of granularities arises even in a simple situation like that of the element-local operations. There is an even more important concern of this nature in the only inter-element communication operation within DG, the computation of surface fluxes. Since the computation of each surface flux refers to data from two opposite element faces, there is definite savings potential if data for a number of such faces is brought onto the chip at the same time and reused (Fig. 23.5). Obviously, this leads to a decrease in the amount of parallelism available, but for large enough problems (which are the main driver for the application of GPU technology), this becomes a non-issue. The amount of parallelism is however limited by two sets of data that need to be fit onto the chip, namely the metadata indicating which faces with what geometry data need to be processed, and the output buffer used to write vectors of face data that can then be processed in the lifting stage of the computation. Both of these could theoretically be accomplished in streaming mode without on-chip storage, however we have found that buffering them improves performance measurably. Once a granularity has been found that suitably balances these factors with data reuse, the computational mesh needs to be partitioned in a way that maximizes the number of interior faces in each partition. Fortunately,

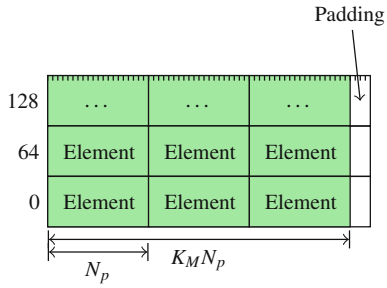


Fig. 23.6 Element storage in "microblock" format as described in the text. An small, integer number of elements is followed by enough padding to satisfy device alignment requirements. Other computation granularities are specified as integer numbers of microblocks

we have found that performance is somewhat insensitive to the absolute quality of this partition, and a simple greedy algorithm, as outlined in Klöckner et al. (2009), suffices.

Overall, we have seen a few examples of computations requiring that the implementer select a granularity entirely unrelated to the computation itself. Each of these granularities is bound to want to manifest itself somehow in the in-memory data storage format, likely through coalescing/alignment concerns. On the other hand, it is *not* likely that a single data storage format can satisfy *all* restrictions of *all* parts of the computation. A compromise therefore needs to be made. In calling the granularities of each of the computations "*blocks*" (related to the Nvidia term for workgroups), we arrived at the idea of an intermediate granularity consisting of an integer number of elements and just big enough to satisfy the device's basic alignment preference, but not necessarily conforming to any particular computation. This would then be called a "*microblock*" (illustrated in Fig. 23.6), and we would demand that all the actual computation granularities be integer multiples of a microblock. A similar technique was independently discovered in Filipovič and Fousek (2010). Seemingly, this just introduces yet another semi-arbitrary number to be chosen before the computation can begin, but nonetheless its introduction does some good by relieving the tension over the data storage format between different parts of the computation.

As we conclude our overview of a few of the challenges of bringing discontinuous Galerkin methods onto the GPU, we observe that there is a common theme uniting many of them—the answers are strongly hardware-dependent. This has a number of important consequences:

- The questions themselves are difficult to answer. Modern processor hardware tends to be very complicated, with many clock domains, bandwidth figures, possibilities for resource contention, and so on.
- Published information on hardware provides insufficient heuristics to make well-founded decisions on any of these.
- Even if a good answer to these questions existed, then it would not necessarily have any lasting value. Software tends to have a much longer shelf life than hardware, as new hardware revisions with programmer-visible changes to microarchitecture (in both the GPU and CPU markets) appear at a rate of about one every two years. Some things (such as the OpenCL programming model) are expected to be durable

for at least some time, but the fine features determining tuning decisions such as those outlined above are subject to frequent change.

One obvious solution to this tuning dilemma stems from the realization that computer cycles are cheap—and that it is thus reasonable to let a computer help as much as possible in solving these challenges. If that means letting the machine try out a large number of possible combinations of parameter settings, that is acceptable—computer time is less expensive than human time, and this trend will almost certainly continue. Furthermore, this shifts two aspects of GPU programming in the right direction. First, it shifts the programmer's role from caring about tuning results to coming up with tuning ideas—and letting the computer determine to what extent those are effective. Second, it decreases the amount of detailed hardware knowledge necessary to come up with a high-performance program. Arguably, both of these represent steps in the right direction. In the next section, we will present ideas on the concrete implementation of *automated tuning*.

23.4 Run-Time Code Generation

The capability to do automated tuning, i.e. to do an automated benchmark of a large number of variants of a program turns out to be a special case of a much more general facility—that of *Run-Time Code Generation* ("RTCG").

This phrase has two parts, 'code generation' and 'run-time'. In itself, the *generation* of source code is merely a text processing task of which most modern programming languages are more than capable. What is being discussed here is thus not the actual generation of the code (which is just a piece of ASCII text), but rather the ability to compile and run this code in-process, at *run-time*. Figure 23.7 shows the basic operating principle of RTCG.

GPU programming environments such Nvidia's CUDA "runtime" interface make this difficult because they insist that all code be compiled ahead of time and into one final binary. In such a setting, all possible tuning variants must already be precompiled into the application binary. This restriction can of course be worked around using dynamic linking and/or shell scripting, but none of these lead to particularly elegant or robust solutions.

We aim to demonstrate in this article that *scripting languages* make a very hospitable environment for run-time code generation, especially when using interfaces such as OpenCL or the Nvidia CUDA "driver" interface which facilitate RTCG more easily. Scripting languages usually have no need for a user- or developer-visible compilation step, and thus everything they do is, by definition, done at run time.

Even beyond what was discussed so far, there are many good reasons to ask for the ability to do run-time code generation:

- *Automated Tuning*, as discussed. (This is also done, although in a variety of ways accommodating ahead-of-time compilation, by packages such as ATLAS (Whaley et al. 2001), FFTW (Frigo and Johnson 2005), or PHiPAC (Bilmes et al. 1997)).

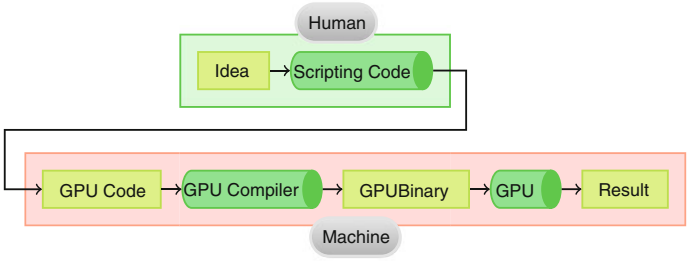


Fig. 23.7 Operating principle of GPU code generation

- The ability to vary *data types* at run time. This might include the ability to run in double or single precision, with complex- or real-valued data, or even more complicated variants, such as interval arithmetic. Templates (in C++) partially address this need in the ahead-of-time-compiled world, however, they also incur the overhead of having to compile each possible variant into the executed binary, as there is no possibility for compilation at run time.
- From the perspective of a library writer, another attractive possibility opened up by RTCG is the possibility to *specialize code for a user-given problem*. While many rather complicated systems involving C++ metaprogramming strive to achieve this goal, they cannot match the simplicity (and performance) of textually pasting a fragment of purpose-specific C code into an overall code framework. Also note that this benefit is really not specific to library writers at all. At some level, every programmer strives to write code that is general and covers a wide variety of use cases. RTCG opens up a very simple and high-performance avenue towards this goal.
- Lastly, it should be observed that *constants faster than variables*. This can be easily understood from the standpoint of *register pressure*—where space in the register file is just one of many resources that are scarce in a GPU, and less contention means that some trade-off does not need to be made, which usually results in higher performance. Another specific example of this is *loop unrolling*. Loops with unknown trip counts necessarily come with fixed overhead in the form of end-of-loop tests and branching instructions, in addition to loop-related state being kept in the register file. If the loop trip count is known at run-time, then this overhead is easily done away with.

All of these arguments in favor of RTCG rest on a simple fact: *More information is available to a code generator and compiler at run time than at any time before that*. And unsurprisingly, the more information is available to the code generator and the compiler, the better the code it is able to generate. Also observe that in this picture, the code generator and the compiler start to merge together conceptually, and the representation in which they exchange data (often a variant of C, for now) moves towards being an implementation detail. This is a good thing, as it makes it expedient for

programmers to develop representations that best serve their application. Interfaces like CorePy (Mueller et al. 2007) and LLVM (Lattner and Adve 2004) demonstrate that C is not the only possible intermediate representation.

As we discuss the advantages of RTCG, we should likely also mention the (in our opinion few and minor) disadvantages. First, RTCG obviously adds more moving parts (such as a compiler and a just-in-time execution environment) to a program, which introduces more possible sources of issues. Second, as generated code must be compiled, there is often a noticeable delay before a piece of code is first executed. However, caching and parallel compilation are effective remedies for this.

Despite these perceived drawbacks, the creators of the OpenCL specification seem to agree with our point of view and have made RTCG a standard part of the OpenCL interface—which, in our opinion, is one of the most interesting contributions OpenCL makes to the high-performance computing arena. When OpenCL is compared to CUDA, one drawback that is often cited is OpenCL's lack of support for C++ templates. This is a moot point, in our opinion, as RTCG is strictly more powerful than C++ templates.

Next, we would like to continue to argue that RTCG is most effective when practiced from a scripting language. Scripting languages are in many ways polar opposites to GPUs. GPUs are highly parallel, subject to hardware subtleties, and designed for maximum throughput. On the other hand, scripting languages (such as Python (van Rossum et al. 1994)) favor ease of use over computational speed, are largely hardware-agnostic, and do not generally emphasize parallelism. We have created two packages, PyOpenCL and PyCUDA (Klöckner et al. 2009), that join GPUs and scripting languages in one programming environment.

Before we move on, however, let us comment on a practicality: In today's GPU programming environments (OpenCL, CUDA), all the host computer is required to do is submit work to the compute device at a certain rate, typically around 1000 Hz. As long as the scripting-based host program can maintain this rate, there is no loss in performance.

PyOpenCL and PyCUDA can be used in a large number of roles, for example as a prototyping and exploration tool, to help with optimization, as a bridge to the GPU for existing legacy codes (in Fortran, C, or other languages), or, perhaps most excitingly, to support an unconventional *hybrid way of writing high-performance codes*, in which a high-level controller generates and supervises the execution of low-level (but high-performance) computation tasks to be carried out on varied CPU or GPU-based computational infrastructure.

Scripting languages already excel at text processing and are routinely used for this task at extreme scales, as exemplified by their use in the generation of HTML pages. This already makes them a good choice for the textual part of code generation. A number of further points contribute to making the programming environment created by joining GPUs and scripting greater than just the sum of its two parts. First, scripting languages lend themselves to very clean programming interfaces, with seamless (but invisible) error reporting and automatic resource management. In addition, scripting languages are very suited to creating abstractions, and Python especially follows a "batteries included" approach that puts many of these abstractions directly within

a user's reach. PyOpenCL and PyCUDA strive to make ideal use of these characteristics. They are fully documented, and also come with "batteries included"—for instance, users do not have to reinvent vectors, arrays, reductions or prefix sums. Both packages also cache compiler output, to support RTCG and retain the development "feel" of a scripting language.

The Python programming language (van Rossum et al. 1994) is well-suited for such packages for a number of reasons:

- The existence of a mature array abstraction (`numpy` (Oliphant 2006)) facilitating (in-process) transport and manipulation of bulk numerical data.
- The large ecosystem of software that has sprung up around `numpy`.
- Its main-stream syntax and language-features, which make the language easy to learn while not impeding more advanced use.

Other languages may certainly be just as suitable.

In concluding this argument for GPUs, scripting and RTCG, let us remark that the packages introduced here are distributed under the liberal MIT license and are available at the URLs http://mathema.tician.de/software/pyopencl (or http://mathema.tician.de/software/pycuda). A mailing list, a wiki, and a number of contributed computational add-on packages are available. Both packages are routinely used on Windows, OS X, and Linux.

23.5 Results: RTCG for Discontinuous Galerkin

Having introduced run-time code generation as a way of addressing the challenges encountered in Sect. 23.3, we will refocus on some of the specific benefits that RTCG brings in the context of a discontinuous Galerkin solver and discuss a few of the achieved results. For definiteness, we will be discussing results obtained using the solver "hedge", which was built to explore and develop the ideas in this article.

The first example emphasizes the impact of RTCG on the ability to write maintainable software with reasonable user interfaces. In particular, we will demonstrate the user interface that our DG solver code uses to specify numerical flux terms—the terms $(n \cdot F^*)$ in (23.2). Figure 23.8 shows three representations of a (partial) numerical flux for the Maxwell equation. First, Fig. 23.8a shows the mathematical notation as one might find in a scientific article. Next, Fig. 23.8b shows the Python code that a user might need to write to capture the numerical flux expression of Fig. 23.8a in our solver. The final part of Fig. 23.8c shows a fraction of the generated code. What this seeks to demonstrate is that a high-performance, low-level, scalar C-language representation can easily be generated from a high-level, vectorial statement in a scripting language. It is obvious that the code in Fig. 23.8b is much easier to check for correctness than the resulting C code. Nonetheless, even textbooks such as (Hesthaven and Warburton 2007) contain code like that of Fig. 23.8c for demonstration purposes. By using RTCG, in many situations it becomes a rather easy proposition to enable the user to write maintainable, transparent code, and still

(a)

$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} \left[\hat{n} \times (\llbracket H \rrbracket - \alpha \hat{n} \times \llbracket E \rrbracket) \right]$$

(b)

```
flux = 1/2*cross(normal, h.int-h.ext
       -alpha*cross(normal, e.int-e.ext))
```

(c)

```
a_flux += (
    ((( val_a_field5 - val_b_field5 )* fpair->normal[2]
      -( val_a_field4 - val_b_field4 )* fpair->normal[0])
     + val_a_field0 - val_b_field0 )* fpair->normal[0]
    -(((val_a_field4 - val_b_field4 ) *fpair->normal[1]
       -( val_a_field1 - val_b_field1 )*fpair->normal[2])
      + val_a_field3 - val_b_field3 ) * fpair->normal[1]
    )*value_type(0.5);
```

Fig. 23.8 Three representations of a (partial) numerical flux for the Maxwell equations. (**a**) Shows the mathematical specification as first given in Mohammadian et al. (1991). (**b**) Shows the Python code used to instruct the solver. (**c**) Shows a fraction (about one sixth) of the C code ultimately generated by the solver to implement the flux in (**a**)

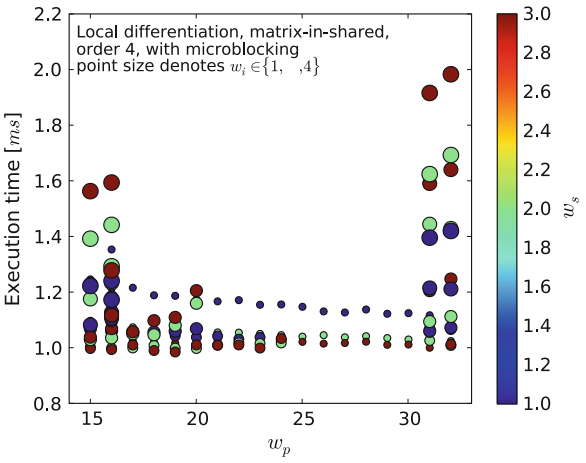


Fig. 23.9 Sample tuning study for local differentiation on fourth-order elements with microblocking enabled, showing time spent for a constant amount of work depending on the values w_s, w_p and w_i introduced in Sect. 23.3

obtain all the performance of a program that would have previously required a rather large amount of manual labor and checking.

Further, our solver obviously makes extensive use of automated tuning. Figure 23.9 shows results from a particular tuning run attempting to optimize the parameters w_s, w_p and w_i introduced in Sect. 23.3 for element-local differentiation. The vertical axis of the plot shows timing information, and each of the dots in the plot represents a particular timing run. The same amount of numerical work was done for each of the dots, yet surprisingly, the final performance varied by more than a factor of 2

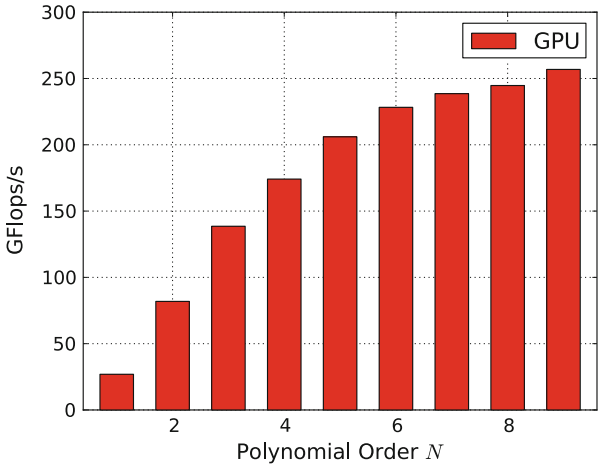


Fig. 23.10 Floating point performance in GFlops/s achieved by our auto-tuning solver on a large 3D Maxwell problem in single precision on an Nvidia GTX 280. Performance is calculated by measuring wall time from one time step to the next and dividing the number of flops performed (including timestepping) by this value

depending on parameter choice. In addition, there is little observable regularity in the graph, which seems to limit the amount of success that any given heuristic might have in predicting this behavior. There is almost no other option besides automated tuning to find an at least somewhat optimal combination within this parameter space. Also note that any performance gain in this part of the operator has a rather large impact on the performance of the method as a whole—element-local differentiation is the asymptotically most work-intensive part of a DG operator.

In addition to this application of automated tuning in the determination of a parallel work decomposition, our solver also applies this technique in finding memory layouts and flux gather granularities. Further, by virtue of code generation, it naturally benefits from being able to "hard-code" certain variable values such as matrix sizes, polynomial degrees, or loop trip counts.

In the following, we will present a number of overall performance results for our solver on an Nvidia GTX 280, to confirm that a high-performance solver can be written using the techniques described. Unless otherwise specified, all performance numbers are based on the wall clock time from the beginning of one time step to the beginning of the next, including RK4 timestepping. Timings were averaged over a run of 100 (CPU) or several hundred (GPU) time steps to minimize the influence of timing transients. Timings were observed to be consistent across runs, even when using automated tuning.

Figure 23.10 shows overall performance expressed in billions of floating point operations per second (GFlops/s), measured by counting flops over a time step and dividing by the duration of that same time step. This is a reasonable (and reproducible) measurement, because unlike for many other numerical methods, the number of flops

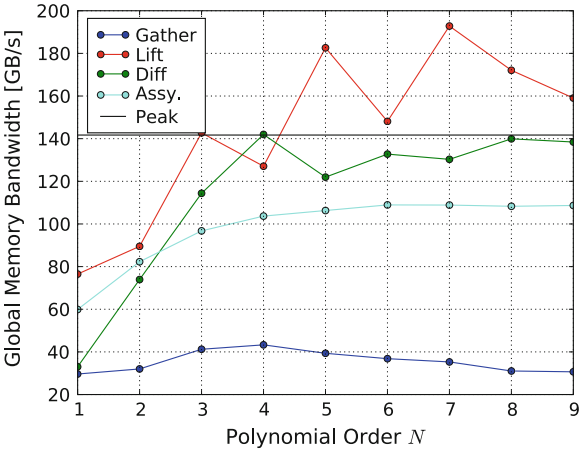


Fig. 23.11 Memory bandwidths in GB/s achieved by each part of the DG operator on an Nvidia GTX 280. The peak memory bandwidth published by the manufacturer is 141.7 GB/s. Values exceeding peak bandwidth are believed to be due to the presence of a texture cache

required for simplicial DG is relatively uniquely determined. Note that because the measurement corresponds to an average, individual components of the method (such as element-local differentiation/lift) achieve significantly higher flop rates. Since the elementwise dense linear operators asymptotically (as $N \to \infty$) determine the run time, it may be reasonable to relate the measured performance to that achieved by dense matrix-matrix multiplies on this architecture. The best results achieved on an SGEMM workload on large, square matrices hover around 350 GFlops/s. It is therefore remarkable that our method achieves 250 GFlops/s on much less benignly shaped matrices, also taking into account that the method does much more varied work than simple matrix-matrix multiplies.

We would also like to comment on the progression of performance results as we vary N in Fig. 23.10, and in particular the rapid rise in performance from $N = 1$ to $N = 3$. We mentioned earlier that optimal results for $N = 3, \ldots, 5$ were an explicit goal of this work. Many of the finer tuning points of past sections (such as microblocking and work item coarsening) become rather unnecessary at $N \geq 6$ (because matrix sizes have grown significantly, and therefore enough work is available within each element). Compared to a simpler code (such as the one described in Klöckner et al. 2011a), it is precisely these optimizations that lead to large gains at $N = 3, \ldots, 5$.

It is interesting to correlate the achieved floating point bandwidth from Fig. 23.10 with the bandwidth reached for transfers between the processing cores and global memory, shown in Fig. 23.11. We have obtained these numbers by counting the number of bytes fetched from global memory either directly or through a texture unit in each component of the method. The published theoretical peak memory bandwidth of the card on which this experiment was performed is 141.7 GB/s, shown as a black horizontal line. Perhaps the most striking feature here is that the calculated

memory bandwidth sometimes transcends this theoretical peak. We attribute this phenomenon to the presence of various levels of texture cache. Its occurrence is especially pronounced in the case of flux lifting, and it should perhaps be sobering that the other parts of the DG operator do not manage the same feat. In any case, flux lifting uses the fields-on-chip strategy, and therefore fetches and re-fetches the rather small matrix L, making large amounts of data reuse a plausible proposition. Aside from this surprising behavior of flux lifting, it is both interesting and encouraging to see how close to peak the memory bandwidth for element-local differentiation gets. As a converse to the above, this makes it likely that the operation does not get much use out of the texture cache in most situations. It does imply, however, that rather impressive work was done by Nvidia's hardware designers: The theoretical peak global memory bandwidth can very nearly be attained in real-world computations. Next, the fact that the flux-gather part of the operator achieves rather low memory throughput is not too surprising—the access pattern is (and, for a general grid, has to be) rather scattered, decreasing the achievable bandwidth. Lastly, operator assembly, which computes linear combination of vectors, consists mainly of global memory fetches and stores. It seems likely that ancillary operations such as index calculations, loop overhead and bounds checks drive this component's shortfall from peak memory bandwidth.

It is worth noting that one would not initially expect a matrix-matrix workload like DG to be memory-bound, at least at high polynomial degrees N. After all, such workloads do offer large amounts of arithmetic intensity to keep floating point units busy. On the other hand, it is worth keeping in mind that there is simply *so much* floating point power available on GPU-like chips that it is quite unlikely that a code like DG might get to the point of actually being limited by it. As such, it is reasonable, in our view, to expect that for the foreseeable future, the limiting factor for most DG-like algorithms will in fact remain memory bandwidth, as evidenced by Fig. 23.11.

Another issue that frequently draws questions is that of the support of double precision within GPU-like devices. Marketing pressure in this area has led GPU manufactures to increase the ratios of the number of double precision (DP) units to the number of single precision (SP) units. Current high-end offerings hover between a factor of 1/2 and 1/4, where this feature is often used to differentiate between 'consumer-grade' and 'professional-grade' hardware. We would like to remark that in bandwidth-bound applications, there is no reason to expect a DP code to go any faster than half as fast as an equivalent SP code, for the simple reason that DP numbers are exactly twice as big as SP numbers, and therefore require twice as much memory bandwidth. In addition, DP requires twice as much on-chip memory to obtain equivalent levels of data reuse—an amount that simply might not be available. With respect to DG, we observe that at low N (e.g., $N = 1, 2$), the ratio (DP GFlops/s)/(SP GFlops/s) is about a factor 1/2, as the algorithm is completely bandwidth bound. As N increases, it approaches the above-mentioned ratio of (available DP units)/(available SP units), which further substantiates the conjecture made above that the code is "underway" to being compute-bound.

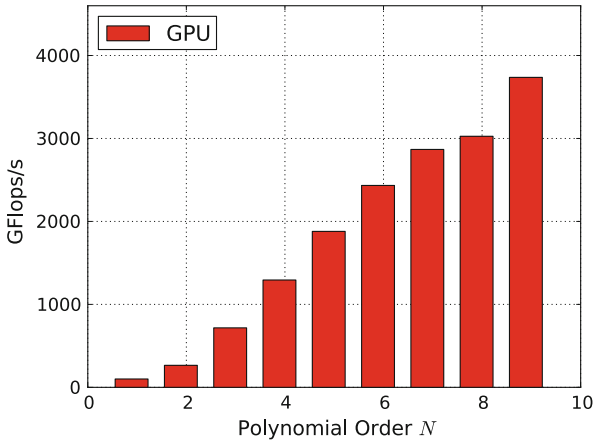


Fig. 23.12 Floating point performance in GFlops/s achieved by our auto-tuning solver on a very large 3D Maxwell problem on 16 Nvidia T10 GPUs (parts of an Nvidia S1070 compute server) in single precision. Performance is calculated by measuring wall time from one time step to the next and dividing the number of flops performed (including timestepping) by this value

Lastly, we would like to comment on Fig. 23.12, which illustrates the performance of our solver in GFlops/s at various polynomial orders N on a cluster of 16 Nvidia Tesla T10 GPUs. Two features of this plot are immediately noteworthy. First, computational performance approaches 25 % of the overall machine peak at $N = 9$ with nearly four teraflops/s. It is remarkable that such performance is achievable on a cluster that costs a small fraction of the large machines whose hallmark such performance was previously. Second, it is also obvious where the distributed-memory inter-node communication (via MPI in this case) is taking its toll, as one may, in principle, directly compare the shape of Fig. 23.10 with that of Fig. 23.12. It is obvious that there is a much steeper performance dropoff in the parallel run as N decreases than there is in the sequential performance data. This is owed to the fact that high orders are significantly heavier on element-local volume work (which scales as N^3), than on communication-heavy work dominated by degrees of freedom on faces (which scales as N^2). Thus, as there is more communication work compared to local compute work, the method incurs larger communication overhead. This is (in our opinion) quite expected, and it should be noted that even at $N = 5$, our code nearly achieves a still very respectable 2 teraflops/s on this cluster. This also contains an important message about the parallelization of DG, which holds true at both the distributed-memory and the shared-memory scale: High polynomial orders N, along with all their other benefits, also much improve the parallelizability of the method.

23.6 Conclusions

In this article, we have shown that high-order DG methods can reach double-digit percentages of published theoretical peak performance values for the hardware under consideration. This speed increase translates directly into an increase of the size of the problem that can be treated using these methods. A single compute device can now do work that previously required a roomful of computing hardware. Alternatively, a cluster of machines equipped with these cards can run simulations that were previously outside the reach of all but the largest supercomputers. This lets the size and complexity of simulations that researchers can afford on a given hardware budget jump significantly.

We find that GPU-DG is far more economical to run at medium to large scales than CPU-DG. In our opinion, this is due to the fact that the computational structure of the method, with its two levels of "element" and "individual degree of freedom", is very well-suited to the GPU a priori—better even than finite-difference methods, which are often cited as a "GPU poster child". Through the use of the auto-tuning technology described in this article along with a number of further tricks discussed in detail in Klöckner et al. (2009), we have shown that rather good performance and machine utilization can be achieved by GPUs in DG-like workloads.

In addition to highlighting our work on GPU-DG, this article also serves to introduce the reader to the idea that scripting languages and GPUs are a good combination. Beyond the core benefit of enabling run-time code generation, they also facilitate a clear separation of the code into 'administrative' and 'computational' parts. Such a separation contributes to code clarity and helps make code more maintainable.

As we continue to explore the benefits of GPUs for DG and DG-like workloads, we will be focusing on areas such as adaptivity in both space and time, nonlinear equations, and the use of curvilinear geometries, as well as much larger scaling of GPU-DG. Initial work on these matters can be found in the articles (Klöckner et al. 2011b; Burstedde et al. 2010; Warburton 2010).

We believe that GPU-DG will have a bright future, with many more applications benefiting from the ease with which large-scale time-domain simulations can be performed using DG, and we hope that our work has helped and will help application scientists use DG computations in their role as part of the 'third pillar of science'.

Acknowledgments AK's research was partially funded by AFOSR under contract number FA9550-07-1-0422, through the AFOSR/NSSEFF Program Award FA9550-10-1-0180 and also under contract DEFG0288ER25053 by the Department of Energy. TW acknowledges the support of AFOSR under grant number FA9550-05-1-0473 and of the National Science Foundation under grant number DMS 0810187. JSH was partially supported by AFOSR, NSF, and DOE. The opinions expressed are the views of the authors. They do not necessarily reflect the official position of the funding agencies.

References

Barth T, Knight T (2005) A streaming language implementation of the discontinuous Galerkin method. Technical report 20050184165. NASA Ames Research Center

Bilmes J, Asanovic K, Chin C, Demmel J (1997) Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: Proceedings of the 11th international conference on supercomputing. ACM, New York, pp 340–347

Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P (2004) Brook for GPUs: stream computing on graphics hardware. In: International conference on computer graphics and interactive techniques. ACM, New York, pp 777–786

Burstedde C, Ghattas O, Gurnis M, Isaac T, Stadler G, Warburton T, Wilcox L (2010) Extreme-scale amr. In: International conference for high performance computing, networking, storage and analysis (SC), pp 1–12, Nov 2010. doi:10.1109/SC.2010.25

Cockburn B, Hou S, Shu C-W (1990) The runge-kutta local projection discontinuous galerkin finite element method for conservation laws IV: the multidimensional case. Math Comput 54(190):545–581. doi:10.2307/2008501

Dally WJ, Hanrahan P, Erez M, Knight TJ, Labonté F, Ahn JH, Jayasena N, Kapasi UJ, Das A, Gummaraju J (2003) Merrimac: supercomputing with streams. In: Proceedings of the ACM/IEEE SC2003 conference (SC'03), vol 1

Filipovič J, Fousek J (2010) Medium-grained functions mapping using modern GPUs. In: Proceedings of the symposium on application accelerators in high performance computing (SAAHPC'11), Knoxville, TN

Frigo M, Johnson SG (2005) The design and implementation of FFTW3. Proc IEEE 93(2):216–231. doi:10.1109/JPROC.2004.840301. Special issue on "Program Generation, Optimization, and Platform Adaptation"

Göddeke D, Strzodka R, Turek S (2005) Accelerating double precision FEM simulations with GPUs. In: Proceedings of ASIM

Hesthaven JS, Warburton T (2007) Nodal discontinuous galerkin methods: algorithms, analysis, and applications. 1st edn, Springer. ISBN 0387720650

Klöckner A, Pinto N, Lee Y, Catanzaro B, Ivanov P, Fasih A (2012) PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. Parallel Comput 38(3):157–174. doi:10.1016/j.parco.2011.09.001

Klöckner A, Warburton T, Bridge J, Hesthaven J (2009) Nodal discontinuous galerkin methods on graphics processors. J Comp Phys 228:7863–7882. doi:10.1016/j.jcp2009.06.041

Klöckner A, Warburton T, Hesthaven J (2011a) Solving wave equations on unstructured geometries. In: Hwu W-m (ed) GPU computing gems, Jade Edn. Morgan Kaufmann Publishers, Waltham

Klöckner A, Warburton T, Hesthaven JS (2011b) Viscous shock capturing in a time-explicit discontinuous galerkin method. Math Model Nat Phenom 6:57–83. doi:10.1051/mmnp/20116303

Krakiwsky S, Turner L, Okoniewski M (2004) Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU). In: IEEE MTT-S international microwave symposium digest, vol 2, pp 1033–1036, ISBN 0149-645X. doi:10.1109/MWSYM.2004.1339160

Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis and transformation. In: IEEE/ACM international symposium on code generation and optimization, 0:75. doi:10.1109/CGO.2004.1281665

Lesaint P, Raviart P (1974) On a finite element method for solving the neutron transport equation. Mathematical aspects of finite elements in partial, differential equations. Academic Press, New York, pp 89–123

Li W, Wei X, Kaufman A (2003) Implementing lattice boltzmann computation on graphics hardware. Vis Comput 19:444–456

Lindholm E, Nickolls J, Oberman S, Montrym J (2008) Nvidia tesla: a unified graphics and computing architecture. IEEE Micro 28:39–55. doi:10.1109/MM.2008.31

Mohammadian AH, Shankar V, Hall WF (1991) Computation of electromagnetic scattering and radiation using a time-domain finite-volume discretization procedure. Comput Phys Commun 68(1–3):175–196. doi:10.1016/0010-4655(91)90199-U

Mueller C, Martin B, Lumsdaine A (2007) CorePy: high-productivity Cell/BE programming. In: Procceedings of the first STI/Georgia tech workshop on software and applications for the Cell/BE processor, Georgia

Nvidia corporation (2009) NVIDIA CUDA 2.2 compute unified device architecture programming guide. Nvidia corporation, Santa Clara, USA, April 2009

Oliphant T (2006) Guide to NumPy. Trelgol Publishing, Spanish Fork

Reed WH, Hill TR (1973) Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Scientific Laboratory, Los Alamos

van Rossum G et al (1994) The python programming language. http://python.org

Warburton T (2006) An explicit construction of interpolation nodes on the simplex. J Eng Math 56:247–262. doi:10.1007/s10665-006-9086-6

Warburton T (2010) A low storage curvilinear discontinuous galerkin time-domain method for electromagnetics. In: IEEE international symposium on electromagnetic theory (EMTS) (URSI 2010), pp 996–999

Whaley RC, Petitet A, Dongarra JJ (2001) Automated empirical optimizations of software and the ATLAS project. Par Comp 27:3–35. doi:10.1016/S0167-8191(00)00087-9

Chapter 24
Accelerating Large-Scale Simulation of Seismic Wave Propagation by Multi-GPUs and Three-Dimensional Domain Decomposition

Taro Okamoto, Hiroshi Takenaka, Takeshi Nakamura and Takayuki Aoki

Abstract We adopted the GPU (graphics processing unit) to accelerate the large-scale finite-difference simulation of seismic wave propagation. We describe the main part of our implementation: the memory optimization, the three-dimensional domain decomposition, and overlapping communication and computation. With our GPU program, we achieved a very high single-precision performance of about 61 TFlops by using 1,200 GPUs and 1.5 TB of total memory, and a scalability nearly proportional to the number of GPUs on TSUBAME–2.0, the recently installed GPU supercomputer in Tokyo Institute of Technology, Japan. In a realistic application by using 400 GPUs, only a wall clock time of 2,068 s (including the times for the overhead of snapshot output) was required for a complex structure model with more than 13 billion unit cells and 20,000 time steps. We therefore conclude that GPU computing for large-scale simulation of seismic wave propagation is a promising approach.

T. Okamoto(✉)
Department of Earth and Planetary Sciences,
Tokyo Institute of Technology, Tokyo, Japan
e-mail: okamoto.t.ad@m.titech.ac.jp

H. Takenaka
Department of Earth and Planetary Sciences,
Kyushu University, Fukuoka, Japan

T. Nakamura
Earthquake and Tsunami Research Project for Disaster Prevention,
Japan Agency for Marine-Earth Science and Technology,
Yokosuka, Japan

T. Aoki
Global Scientific Information and Computing Center,
Tokyo Institute of Technology,
Tokyo, Japan

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 375
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_24,

24.1 Introduction

The seismic wave signals that propagate through the planets' interior are essential data in geoscience: they are used to probe the Earth's and other planets' interiors, to study earthquake sources, and to evaluate strong ground motions due to earthquakes. Simulating the seismic wave propagation in complex media is necessary for interpreting those signals as the irregular topography, irregular internal discontinuities and heterogeneity in the Earth and planets all affect the propagation of seismic waves. For the large-scale simulations efficient numerical methods are required as more than one billion of grid points are required in real applications e.g., Olsen et al. (2008); Furumura (2009).

GPU (Graphics Processing Unit) is a highly parallel processor that executes the arithmetic instructions on more-than-one-hundred processing units (Fig. 24.1). It delivers extremely high computing performance at a reduced power and cost compared to conventional CPUs (Central Processing Units): the performance of recent GPUs exceeds 1 TFlops in single-precision arithmetic. Because of its many-core architecture, large acceleration can be expected in the case of *compute intensive* applications.

The simulation of continuum mechanics such as the computational fluid dynamics and the simulation of seismic wave propagation is *memory intensive* applications which is characterized by large amount of data transfer and small amount of computation. Nevertheless, these problems can still benefit from GPU because of GPU's high memory bandwidth (e.g., Aoki (2009)). Thus, several approaches to adopt GPUs to the simulation of seismic wave propagation have been proposed recently (e.g., Abdelkhalek et al. (2009); Aoi et al. (2009); Komatitsch et al. (2009, 2010); Michéa and Komatitsch (2010); Micikevicius (2009); Okamoto et al. (2009, 2010a,b)).

We present here our approach to adopt GPUs to the large-scale simulation of seismic wave propagation based on the finite-difference method (FDM). After brief descriptions on the finite-difference method (Sect. 24.2) and on the GPU system (Sect. 24.3), we describe the main part of our implementation: the memory

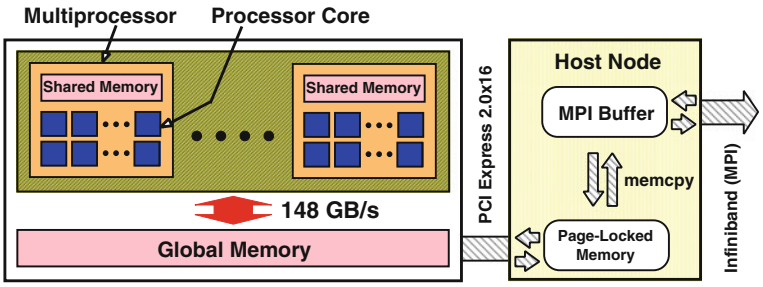


Fig. 24.1 A simplified diagram of a GPU (M2050) used in this study (modified from Okamoto et al. 2010b). A single GPU has 14 processors called *multiprocessors*. One multiprocessor has 32 *processor cores* so that 448 processor cores are integrated in a GPU. The size of the *global memory* is 3 GB (giga-byte)

optimization (Sect. 24.4.1), the three-dimensional domain decomposition (Sect. 24.4.2), and overlapping communication and computation (Sect. 24.4.3). Then in Sect. 24.5 we discuss the block size optimization and the scalability, and show an example of the simulation for realistically complex structure model.

24.2 Finite-Difference Method

We apply the time-domain, staggered-grid, three dimensional (3D) finite-difference scheme (e.g., Graves (1996); see Fig. 24.2). This FDM scheme is one of the standard methods in seismology because of its good numerical dispersion behavior and of its stability in treating highly complex media (e.g., Virieux (1986); Levander (1988)).

In this scheme the basic equations for isotropic and elastic material split in two systems for particle velocity (Eq. 24.1) with force term f_i and stress (Eq. 24.2), respectively: we use components of particle velocity (v_i: three components) and stress (τ_{ij}: six components) as the field variables. Three material parameters (density ρ and Lamé coefficients, λ and μ, with μ the rigidity) are also required so that twelve variables are assigned to a unit cell. That is, we apply a heterogeneous formulation to treat complex media.

$$\rho \frac{\partial v_i}{\partial t} = \frac{\partial \tau_{xi}}{\partial x} + \frac{\partial \tau_{yi}}{\partial y} + \frac{\partial \tau_{zi}}{\partial z} + f_i \tag{24.1}$$

$$\frac{\partial \tau_{ij}}{\partial t} = \lambda\, \delta_{ij} \left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \right) + \mu \left(\frac{\partial v_j}{\partial x_i} + \frac{\partial v_i}{\partial x_j} \right) \quad (i, j = x, y, z) \tag{24.2}$$

(In Eq. 24.2, x_i and x_j also denote one of x, y, and z, respectively.)

The precision of finite-difference operator is fourth-order in space and second-order in time. We apply the absorbing boundary condition (Cerjan et al. 1985) near

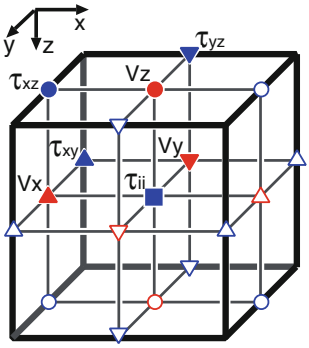


Fig. 24.2 Schematic illustration of the staggered grid (from Okamoto et al. 2010b). A single unit cell is shown. *Open symbols* denote the variables that belong to the neighboring cells

Table 24.1 An outline of TSUBAME–2.0 supercomputer (GPU cluster) used in this study

GPU	GPUs/node	Total GPUs	Host CPU	Cores/node	PCIe	Infiniband/node
NVIDIA M2050	3	4224	Xeon X5670 ×2	12	2.0×16	40 Gbps×2

"PCIe" is an abbreviation of PCI Express. The bandwidth of PCIe 2.0×16 is 8 GB/s at maximum

the side and the bottom boundaries, and the A1 absorbing condition of Clayton and Engquist (1977) at the bottom. A periodic condition is imposed on the side boundaries.

Note that a staggered-grid scheme similar to that employed here has been widely used in the electromagnetic simulations (e.g., Yee (1966)). Therefore, GPU implementations described in this paper would be also applicable to the electromagnetic problems.

24.3 GPU System

We use TSUBAME–2.0 supercomputer in Global Scientific Information and Computing Center, Tokyo Institute of Technology (Table 24.1). TSUBAME–2.0 has been operational since November 2010, and is ranked as world fourth fastest supercomputer in the November 2010 TOP-500 list (www.top500.org).

The data for computation, such as the velocity and stress field at each finite-difference grid point, are stored in the global memory (Fig. 24.1) because we apply the "full GPU" computing. The bandwidth of the global memory, 148 GB/s, is much faster than those of the conventional CPUs (e.g., it is 32 GB/s at maximum per a processor socket in the case of Intel Xeon X5670). This is the reason that GPU is effective for the memory-intensive problems.

However, 400 to 600 clock cycles of memory latency still occurs in transferring the data between the global memory and the multiprocessors. Thus reducing the amount of data transfer from and to the global memory is the key in improving the performance.

Therefore, we use the fast (but small) memories in the multiprocessors, the *registers* and the *shared memory*, as software managed cache memories. This is typically done by copying the data in a small *block* of FDM domain stored in the global memory to the shared memory and registers, and subsequently by reusing the data stored there (e.g., Aoki (2009); see Fig. 24.3). Note the difference that the shared memory is accessible from all the processing units (*threads*) assigned to a block, while the registers are local to each processing unit (see NVIDIA CUDA C Programming Guide (NVIDIA Corporation 2010) for detail). We exploit this difference in our program as described below.

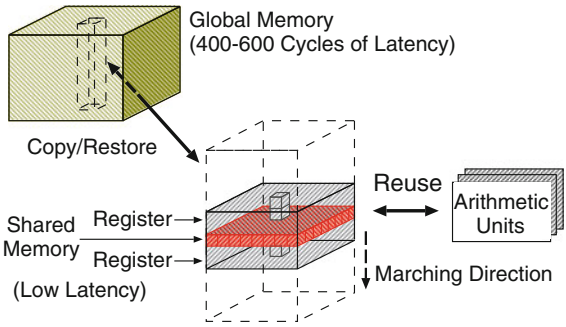


Fig. 24.3 Schematic illustration of a small block made of the shared memory and registers. The ghost zones attached to the shared memory are omitted

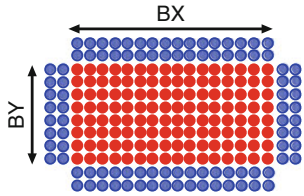


Fig. 24.4 Example of two-dimensional block assigned to the shared memory. Each circle corresponds to a unit cell. A case of 16×8 is shown (*red circles*). For fourth order finite-difference, two points of ghost zones (*blue*) are also required to be assigned in the shared memory

24.4 GPU Implementations

24.4.1 Memory Optimization

In the seismic wave problem the number of variables in a unit cell is large, while the size of the shared memory is small (16 or 48 kB in the case of M2050). Thus, assigning three-dimensional block(s) in the shared memory of the GPU is not practical because only block(s) of very small size can be assigned.

Therefore, we store the variables belonging to the unit cells on a chosen level into the two-dimensional (2D) blocks assigned in the shared memory (Fig. 24.3 and 24.4). Then we assign a single thread to each unit cell: the thread computes the Eqs 24.1 and 24.2 only for the assigned cell. To the variables not on the plane, only "vertical" finite-differences are operated so that the variables are not required to be shared among threads. Thus we store them into the registers. As the computation loop proceeds to the next level, the data in the register are moved to the shared memory and vice versa: for this movement no data access to the global memory occur (Fig. 24.5). The use of the shared memory and registers as described has been applied by Abdelkhalek et al. (2009) and Micikevicius (2009) for acoustic case, and by Michéa and Komatitsch (2010) and Okamoto et al. (2010a,b) for elastic case.

Also, we define the material parameters only at the center of the unit cell. The material parameters at the grid points for particle velocities and shear stresses are computed by using the values defined at the center of the unit cells at every time steps

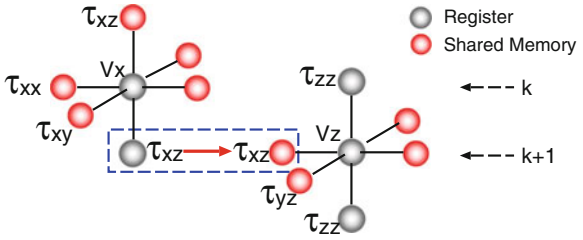


Fig. 24.5 Example of the use of the shared memory and registers for the integration of components of velocity (from Okamoto et al. 2010b). Values on *red points* are stored in the shared memory and those on *black point* in the registers. In the computations for k-th level the stress component τ_{xz} is stored in the register. For the next level the value in the register is moved to the shared memory to reduce the data transfer from the global memory

to reduce the access to the global memory. In computing the material parameters, we apply a method proposed by Takenaka et al. (2009) (see Sect. 24.5.3 for detail).

24.4.2 3D Domain Decomposition

Decomposing the FDM domain is necessary for large-scale GPU computing because the size of the global memory of GPU is not large (e.g., 3 GB for M2050). We here divide the FDM domain into *subdomains*, and allocate computation for a single subdomain to a single GPU by using MPI library (i.e., a MPI only parallel programming model).

In the parallel computing we need to exchange the data in the *ghost zones* (Fig. 24.6) between the neighboring subdomains. We adopt the *three-dimensional* (3D) domain decomposition (Fig. 24.6): the domain can be extended in all three Cartesian directions, and the communication time decreases with the increasing number of the subdomain (for a fixed total domain). The latter is because the communication time is proportional to the surface area which decreases with the size of the subdomain. On the other hand, with *one-dimensional* (1D) domain decomposition (e.g., Micikevicius (2009)), the domain can be extended only along one direction, and the communication time does not decrease with the size (or the thickness) of the subdomain because of fixed size of the ghost zones (Fig. 24.6). Therefore the scalability is limited in 1D decomposition (Ogawa et al. 2010).

It is possible to extend the memory array of the internal grid points to cover the ghost zones on the side faces of the subdomain (Fig. 24.7a). However, we found that it took quite a long time to separately send the resultant non-contiguous data from GPU to the host node by repeated calls of memory transfer function. (Note that direct communication between GPUs is not available. The data must be sent from GPU to the host node in order that the data be exchanged with the other nodes (Fig. 24.1)). Thus, we prepare contiguous memory buffers for ghost zones (Fig. 24.7b) so that

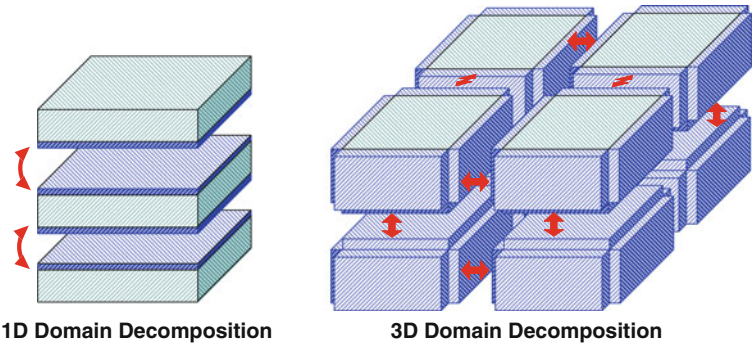


Fig. 24.6 Schematic illustration of 1D and 3D decomposition (from Okamoto et al. 2010b). *Blue region* indicates the ghost grids

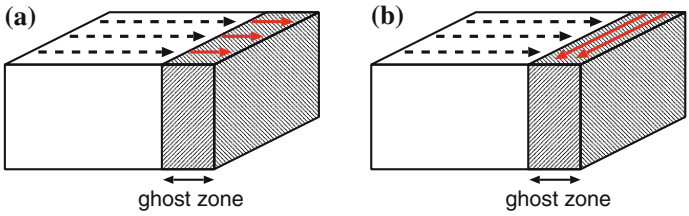


Fig. 24.7 Schematic illustration of ghost zone of a subdomain. *Arrows* denote the memory alignment. **a** Non-contiguous memory alignment in the ghost zone (*red arrows*). The memory array of the internal grid is extended to cover the ghost grid. **b** Contiguous memory alignment. A memory buffer separate from that of the internal domain is used

we are able to copy the data from GPU to the host node by a single call of memory transfer function.

24.4.3 Overlapping

We overlap the communication and the computation to reduce the total processing time (see e.g., Abdelkhalek et al. (2009); Aoki (2010); Ogawa et al. (2010) for methods for overlapping). The overlapped procedures are indicated in the flowchart of our program (Fig. 24.8). We use separate GPU kernels for time-integrations of the velocity and of the stress, respectively (Eqs. 24.1 and 24.2). We further prepare different kernels, one for inner blocks and the other for outermost side blocks, respectively, in both integrations. With these kernels, we first compute for the outermost side blocks that contain the ghost zones. Second we start the computation for the inner blocks and the communication simultaneously (Fig. 24.9).

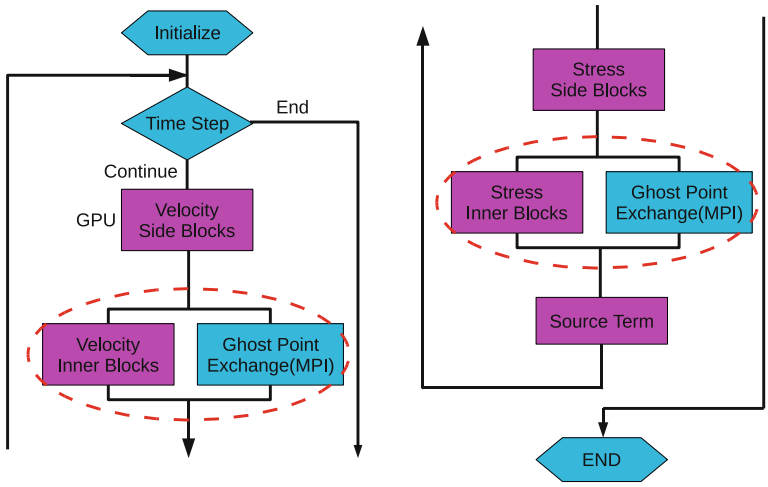


Fig. 24.8 Simplified flowchart of the GPU program. *Blue items* denote the procedures processed by host CPUs. *Pink items* denote the GPU kernels that processed by GPUs. Items circled by *red broken lines* are the overlapped procedures

Figure 24.9 shows the relevant functions used for overlapping computation and communication: CUDA asynchronous memory copy function (`cudaMemcpyAsync`) and non-blocking MPI functions (`MPI_Isend`, `MPI_Irecv`). For `cudaMemcpyAsync` we need the page-locked host memory that can be allocated by a CUDA function `cudaMallocHost`. Note that the copying procedure between the GPU and the host memory must be also treated as one of the communication procedures, and be processed concurrently with the computation for inner blocks.

In the present program the data in the page-locked memory is copied to a (usual) memory buffer ("memcpy" in Figs. 24.1 and 24.9) and the memory buffer is then used for MPI. In the future work we will test the recently released technology (called "GPUdirect") that can eliminate the "memcpy" procedures.

24.5 Examples of Simulation

As mentioned previously, we use NVIDIA CUDA C (NVIDIA Corporation 2010) and the MPI library for our multi-GPU program. Single precision arithmetic is used in all the results presented in this paper both for GPU and CPU. In Sects. 24.5.1 and 24.5.2 we use a structure model of homogeneous media with flat free surface.

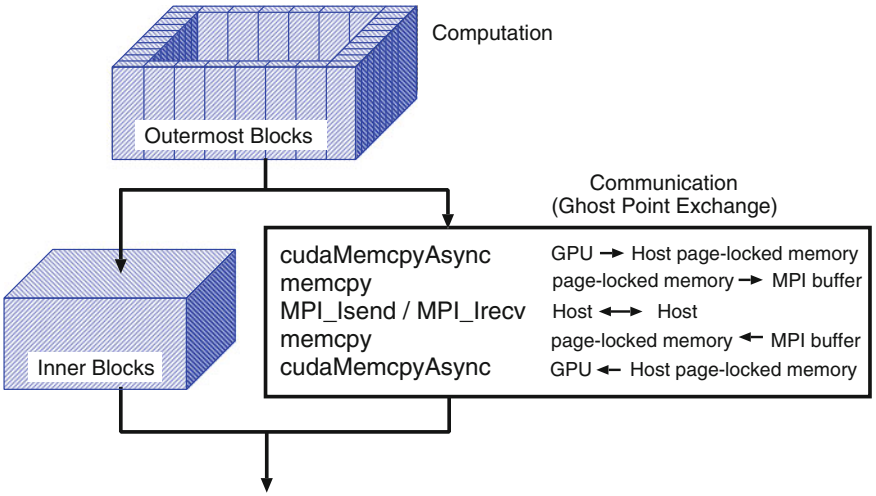


Fig. 24.9 Diagram showing the relevant functions used for overlapping communication and computation

24.5.1 Block Size Optimization

The block size described in Sect. 24.4.1 has large effect on the performance. The edge lengths of around 8 to 64 are usually used because the threads that are assigned to each unit cell are executed simultaneously as a group of 32 threads (a *warp*). We here determine the optimum block size experimentally.

Table 24.2 shows the performance of the multi-GPU program for different block sizes executed on a single-GPU. For Flops (floating point number operations per second) we count the number of floating point operations in the source code including those for the computation of the material parameters, and divide the number of operations by the time required for the time-step loop. Since we align the memory in *x-major* order, better performance is observed for longer edge length in x-direction (BX). The best performance is obtained for a block size of 64×4. We however use the block size of 32×8 in the examples presented later in this paper because parallel computing with the latter (32×8) block size provides better performance in multi-GPU case. (Note that the larger inner blocks are better in hiding the communication, and the inner blocks for block size of 32×8 are larger than those for block size of 64×4.)

As a comparison we show the performance of a FDM program executed on the host CPU in Table 24.3. Note that this example for CPU does not involve MPI inter-node communications as the program is parallelized with OpenMP and executed on a single node. For these programs, the performance of a single GPU is roughly three-fold faster than that of a single node (12 cores).

Table 24.2 Single-GPU performance in GFlops

	BX				
BY	4	8	16	32	64
4	16.4	28.5	52.3	73.2	73.9
8	—	26.7	49.2	73.3	—
16	—	—	47.2	—	—

BX and BY denote the edge lengths of the block (Fig. 24.4). A FDM domain of 320 × 320 × 320 is used. In the case of the block size of 32 × 8, the computation time is 24.7 s for 300 time steps (i.e., 2.5 ms per 10^6 unit cells per a time step). Note that we imposed a condition of BY≤BX. Also, the size of the shared memory (16 kB for nvcc option "`sm_13`") limits the available block size

Table 24.3 Performance of CPU program in GFlops

Domain Size	1 core	12 cores
320 × 320 × 320	4.8	24.2
320 × 320 × 3840	4.2	25.0

On CPUs we parallelized a Fortran source code by using OpenMP with 1D domain decomposition, compiled it by PGI Fortran compiler with `-fastsse` option, and executed the program on a single host node with 12 cores in total

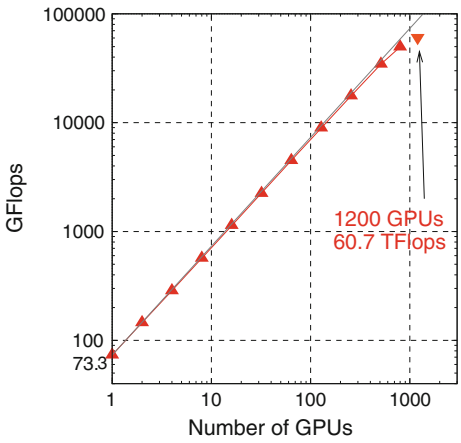


Fig. 24.10 Weak scaling curve of Multi-GPU case on TSUBAME–2.0. The subdomain size was fixed to 320 × 320 × 320. The total number of subdomains is equal to the number of used GPUs. The experiments were performed with 2 GPUs per a node, except in the case of 1200 GPUs that was performed with 3 GPUs per a node

24.5.2 Scalability

In Fig. 24.10 we show the *weak scaling* curve of the performance of our multi-GPU program. The performance of our multi-GPU program scales well with the number of GPUs. A very high performance of about 50 TFlops is achieved in the case of 800 GPUs (400 nodes), and even higher performance of about 61 TFlops is achieved in the case of 1200 GPUs (400 nodes). These values are about 85 % and about 69 %, respectively, of those for complete scalability (i.e., values proportional to the number of GPUs). In the case of 1200 GPUs the performance is slightly degraded because we used 3 GPUs per a node only in this case: increased communication among

Table 24.4 Material parameters of the assumed model used in Sect. 24.5.3

Layer	V_P (km/s)	V_S (km/s)	ρ (g/cm^3)
Ocean	1.5	0.0	1.00
Sediment	3.0	1.5	2.25
Upper Crust	5.5	3.2	2.65

V_P and V_S denote P- and S-wave velocities, respectively

the nodes decreased the total performance. Improving the communication procedure and/or modifying the parallelization model would be necessary for our GPU program in the future study to improve the performance in similar situations with many GPUs on a node.

24.5.3 Application

Recent studies have been revealing the effects of land topography and oceanic layer on the seismic ground motions (e.g., Nakamura et al (2009)). Thus the effects of the land-ocean topography need be incorporated in the simulations and be studied further for better understanding of the ground motions. As mentioned previously in Sect. 24.4.1, we have recently proposed an approach to model structures with both the land and ocean topography in 3D seismic modeling with the finite-difference method (Takenaka et al., 2009). The approach unifies the implementation for irregular free-surface (i.e., the land topography) proposed by Ohminato and Chouet (1997), and that for irregular water-solid interface (e.g., ocean bottom) proposed by Okamoto and Takenaka (2005) and subsequently extended to 3D case by Nakamura et al. (2011). The water-solid interface scheme satisfies the physically required boundary conditions with an accuracy of $O(h)$ where h being the space increment, provided that (1) the boundary (i.e. zero rigidity) is placed through the shear stress grid points and (2) the standard second-order centered equation with averaged density is applied to the normal velocity component at the grid points on the boundary. The second condition imposes second-order operators instead of the fourth-order ones *only* near the interface. The same conditions must be imposed on the free-surface. In the method of Takenaka et al. (2009) the material parameters at the center of the cell are first defined. Second, in order that the above conditions for material parameters be automatically satisfied, the arithmetic average values of the densities of two adjoining cells are used for effective values on the faces of the cell, and the harmonic average values of the rigidities of four adjoining cells are used for effective value on the edges.

We here show an example of seismic wave propagation through a simplified model with land-ocean topography for Kanto area, Japan. Figure 24.11 shows several snapshots of the wave propagation computed by using 400 GPUs. Different modes of wave propagation are clearly observed in land and ocean area: in oceanic area large

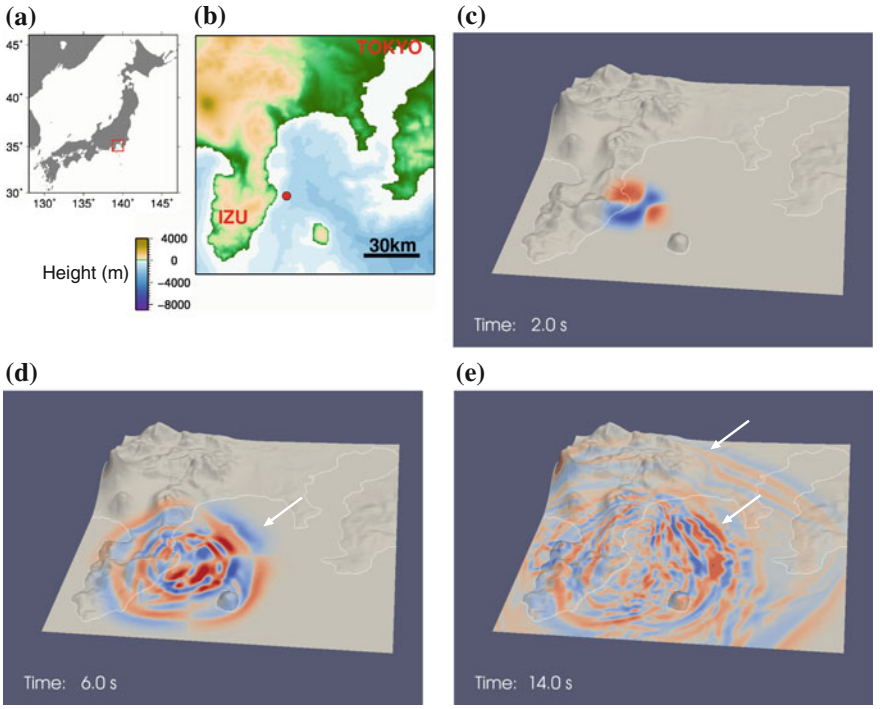


Fig. 24.11 Snapshots of the wave propagation through the Kanto area, Japan. The wavefield recorded on the land-ocean surface are shown. **a** Japanese islands. *Red rectangle* denote the study area. **b** Land-ocean topography of the study area. *Red solid circle* denote the epicenter. **c** Snapshot at 2 s. Vertical component of the particle velocity is plotted with color scale. *Blue* and *red color* denote *upward* and *downward motions*, respectively. **d** Snapshot at 6 s. *White arrow* indicates the wave front of *P*-wave. **e** Snapshot at 14 s. *White arrows* indicate the wave trains of surface waves and scattered waves, respectively. For this simulation we used source parameters of a real earthquake by using the location, depth and moment tensor of the Global CMT solution (www.globalcmt.org). The selected event occurred on April 20, 2006, (Mw 5.6). The centroid location is (34.92N, 139.20E) and centroid depth is 12.0 km. We used three layer model structure (Table 24.4). In assuming the model, the ETOPO1 model (Amante and Eakins 2009) and a part of the structure model developed by The Headquarters for Earthquake Research Promotion (2009) are used for topography and internal structure, respectively. The number of unit cells is 3200 × 3200 × 1280 (i.e., 13.1 billion of unit cells). The size of the subdomain is 320 × 320 × 320 and 400 GPUs (2 GPUs/node) are used in parallel. The grid spacing is 50 m and the time step is 0.002 s. The source term (i.e., moment tensor) is applied using the stress components (Okamoto 1994; Olsen 1995). A wall clock time of 2068 s, including the times for the overhead of snapshot output, was required for a 20000 time steps of integrations with single precision arithmetic. The total performance is 23.5 TFlops

amplitude surface waves develop because of the ocean and sedimentary layer with low seismic wave velocities, and in land area scattering effect due to topography is observed. Such GPU-accelerated simulations will enable us to incorporate fine scale topography and internal structures to study and predict the mode of wave propagation in realistically complex media (eg., Okamoto et al., (2012)).

24.6 Conclusion

We adopted the GPU to accelerate the large-scale finite-difference simulation of seismic wave propagation. We described the main part of our implementation: the memory optimization, the three-dimensional domain decomposition, and overlapping communication and computation. With our GPU program, we achieved a very high single-precision performance of about 61 TFlops by using 1200 GPUs and 1.5 TB of total memory on TSUBAME–2.0, the GPU supercomputer in Tokyo Institute of Technology. Near ideal scalability was achieved by our GPU program: that is, the weak scaling was nearly proportional to the number of GPUs. We also showed that practical simulation of wave propagation in realistically complex media is possible by using our multi-GPU program. We therefore conclude that GPU computing for large-scale simulation of seismic wave propagation is a promising approach.

Acknowledgments The author (TO) is grateful to the organizing committee of the *GPU Solutions to Multiscale Problems in Science and Engineering*, held in Harbin, China in July 2010, for giving the opportunity to present a talk. We are grateful to Tsugunobu Nagai for supporting this research.

References

Abdelkhalek R, Calandra H, Coulaud O, Roman J, Latu G (2009) Fast seismic modeling and reverse time migration on a GPU cluster, International conference on high performance computing simulation, pp 36–43

Amante C, Eakins BW (2009) ETOPO1 1 Arc-minute global relief model: procedures, data sources and analysis, NOAA Technical Memorandum NESDIS NGDC-24, pp 19

Aoi S, Nishizawa N, Aoki T (2009) 3-D wave propagation simulation using GPGPU, programme and abstracts, Seism. Soc. Japan, 2009 Fall Meeting, abstract A12–09

Aoki T (2009) Full-GPU CFD applications. IPSJ Mag 50(2):107–115

Aoki T (2010) Multi-GPU scalabilities for mesh-based HPC applications, SIAM conference on parallel processing for scientific computing (PP10), Seattle

Cerjan C, Kosloff D, Kosloff R, Reshef M (1985) A nonreflecting boundary conditions for discrete acoustic and elastic wave equations. Geophysics 50(4): 705–708

Clayton R, Engquist B (1977) Absorbing boundary conditions for acoustic and elastic wave equations. Bull Seism Soc Am 67(6): 1529–1540

Furumura T (2009) Large-scale simulation of seismic wave propagation in 3D heterogeneous structure using the finite-difference method. J Seism Soc Japan (Zisin) 61:S83–S92

Graves RW (1996) Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences. Bull Seism Soc Am 86(4):1091–1106

Komatitsch D, Michéa D, Erlebacher G (2009) Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. J Parallel Distrib Comput 69(5):451–460

Komatitsch D, Erlebacher G, Göddek D, Michéa D (2010) High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. J Comp Phys 229(20):7692–714

Levander AR (1988) Fourth-order finite-difference P-SV seismograms. Geophysics 53(11): 1425–1436

Michéa D, Komatitsch D (2010) Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophys J Int doi:10.1111/j.1365-246X.2010.04616.x

Micikevicius P (2009) 3D finite-difference computation on GPUs using CUDA, in GPGPU-2: Proceedings of the 2nd workshop on general purpose processing on graphics processing units, Washington DC, USA, pp 79–84

Nakamura T, Takenaka H, Okamoto T, Kaneda Y (2009) Finite-difference simulation of strong motion from a sub-oceanic earthquake: modeling effects of land and ocean-bottom topographies. American Geophysical Union, Fall Meeting, pp S43B–1981

Nakamura T, Takenaka H, Okamoto T, Kaneda Y (2011) A study of the finite difference solution for 3D seismic wavefields near a fluid-solid interface. J Seism Soc Japan (Zisin), 2nd Series, 63(3):189–196

NVIDIA corporation, NVIDIA CUDA C programming guide, Version 3.2, (2010). http://www.nvidia.com/

Ogawa S, Aoki T, Yamanaka A (2010) Multi-GPU scalability of phasefield simulation for phase transition – 5 TFlop/s performance on 40 GPUs. Trans. IPS Japan, Adv Comput Syst 3(2):67–75

Ohminato T, Chouet BA (1997) A free-surface boundary condition for including 3D topography in the finite-difference method. Bull Seism Soc Am 87(2):494–515

Okamoto T (1994) Moment tensor in finite-difference calculation, Programme and Abstracts, Seism Soc Japan 1994 Fall Meeting, abstract C-06

Okamoto T, Takenaka H (2005) Fluid-solid boundary implementation in the velocity-stress finite-difference method. J Seism Soc Japan (Zisin), 2nd Series, 57(3):355–364

Okamoto T, Takenaka H, Nakamura T (2009) Computation of seismic wave propagation with GPGPU, programme and abstracts. Seism Soc Japan, 2009 Fall Meeting, abstract P3–22

Okamoto T, Takenaka H, Nakamura T (2010a) Simulation of seismic wave propagation by GPU, Symposium on Advanced Computing Systems and Infrastructures, pp 141–142

Okamoto T, Takenaka H, Nakamura T, Aoki T (2010b) Accelerating large-scale simulation of seismic wave propagation by multi-GPUs and three-dimensional domain decomposition. Earth Planets Space 62(12):939–942

Okamoto T, Takenaka H, Nakamura T, Aoki T (2012) Large-scale simulation of seismic-wave propagation of the 2011 Tohoku-Oki M9 earthquake. Proceedings of the International Symposium on Engineering Lessons Learned from the 2011 Great East Japan Earthquake, 349–360

Olsen KB, Archuleta R, Matarese JR (1995) Three-dimensional simulation of a magnitude 7.75 earthquake on the San Andreas fault. Science 270:1628–1632

Olsen KB, Day SM, Minster JB, Cui Y, Chourasia A, Okaya D, Maechling P, Jordan T (2008) TeraShake2: Spontaneous rupture simulations of Mw 7.7 earthquakes on the Southern San Andreas Fault. Bull Seism Soc Am 98(3):1162–1185

Takenaka H, Nakamura T, Okamoto T, Kaneda Y (2009) A unified approach implementing land and ocean-bottom topographies in the staggered-grid finite-difference method for seismic wave modeling, Proceedings of the 9th SEGJ international symposium, CD-ROM Paper No. 37

The Headquarters for Earthquake Research Promotion, Long-period ground motion maps (experimental version) in 2009. http://www.jishin.go.jp
Virieux J (1986) P-SV wave propagation in heterogeneous media: velocity-stress finite-difference method. Geophysics 51(4):889–901
Yee K (1966) Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. IEEE Trans Antennas Propag 14(3):302–307

Chapter 25
Support Operator Rupture Dynamics on GPU

Shenyi Song, Yichen Zhou, Tingxing Dong and David A. Yuen

Abstract The method of Support Operator (SOM) is a numerical method to simulate seismic wave propagation by solving the three dimension viscoelastic equations. Its implementation, the Support Operator Rupture Dynamics (SORD) has been proved to be highly scalable in large-scale multi-processors calculations. This paper discusses accelerating SORD using on GPU using NVIDIA CUDA C. Compared to its original version on CPU, we have acrhieved a maximum 12.8X speed-up.

Keywords Support operator · CUDA · Seismic wave propagation

S. Song (✉) · Y. Zhou · T. Dong · D. A. Yuen
Minnesota Supercomputing Institute, University of Minnesota,
Minneapolis 55455, USA

S. Song · T. Dong
Graduate School of China Academy of Science,
Beijing 100190, China

S. Song · T. Dong
Computer Network Information Center, China Academy of Science,
Beijing 100190, China

Y. Zhou
Department of Computer Science, University of Minnesota,
Minneapolis 55455, USA

D. A. Yuen
Department of Geology and Geophysics, University of Minnesota,
Minneapolis 55455, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 391
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_25,

25.1 Introduction

25.1.1 The Method of Support Operator

The method of Support Operator is a generalized finite difference method introduced by Samarskii et al. (1981) and Shashkov (1996). SOM is a general scheme for discretizing the differential form of partial differential equations. The Support Operator Rupture Dynamics (SORD), an application of this method in the simulation of earthquake rupture dynamics is developed by Ely et al. (2008). This application uses single-precision floating-point Support Operator Method. SORD can be used to investigate idealized wave propagation and rupture dynamics problems and to simulate potential future earthquakes with realistic fault and basin models. One example is the simulation of Mw 7.6 earthquake scenarios on the southern San Andreas fault (Ely et al. 2010).

25.1.2 Solving Partial Differential Equations on GPU

A variety of applications require solving partial differential equations (PDE), such as Laplace equation in image denoising, Poisson equation in image editing and mesh editing and Navier-Stocks equations in fluid simulation, etc. Numerical simulation of the PDEs usually requires high-intensity computation and large consumption of computational resources (Zhao 2008).

As a multiple SIMD processing unit, GPU has inherent parallelism, which is suited for explicit and lattice-based computations. Solid-earth geophysics remains one of the last bastions to have resisted the use of GPUs, especially in geodynamics.

GPU programming on NVIDIA graphics cards has become significantly easier with the introduction at the end of 2006 of the CUDA C programming language, which is relatively easy to learn because its syntax is similar to C (NVIDIA 2009).

25.2 Support Operator Rupture Dynamics

25.2.1 Theoretical Formulation

The governing equations of wave propagating in 3D, isotropic viscoelastic medium are

$$g_{ij} = \partial_j(u_i + \gamma v_i), \tag{25.1}$$

$$\sigma_{ij} = \lambda\delta_{ij}g_{kk} + \mu(g_{ij} + g_{ji}), \tag{25.2}$$

$$a_i = (1/\rho)\partial_j\sigma_{ij}, \tag{25.3}$$

$$\dot{v}_i = a_i, \tag{25.4}$$

$$\dot{u}_i = v_i. \tag{25.5}$$

where σ is the stress tensor, **u** and **v** are displacement and velocity vectors, ρ is density, λ and μ are elastic moduli, and γ is viscosity.

25.2.2 Numerical Method

Finite Difference Method (FDM) is widely used in modeling three dimensional seismic wave propagation and rupture dynamics problems (Michea and Komatitsch 2010). We apply the Support Operator Method (SOM). Many simple FDMs are special cases of SOM. The approach constructs discrete analogs of continuum derivative operators that satisfy important integral identities, such as the adjoint relation between gradient and divergence. SOM brings to an FDM-type formulation the FEM advantage that energy is conserved in the semi-discrete equations.

The scheme is explicit in time, and discretized on a hexahedral, logically rectangular mesh. On the mesh we define the space of nodal function H^N consisting of the hexahedra vertices, and the space of cell H^C consisting of the hexahedra volumes. If we do a difference to a variable in H^N, we can get a variable in H^C, and if we do a difference to a variable in H^C, we can get a variable in H^N. So we define two discrete difference operators (Ely et al 2009).

$$D_i : H^N \to H^C \text{ and } \mathsf{D}_i : H^C \to H^N \tag{25.6}$$

On the nodes we have (ρ, γ, β, **u**,**v**,**a**) $\in H^N$, and on the cells we have (λ, μ, y, **σ**, g) $\in H^C$. Using the two operators we can obtain a variable in H^N through variables in H^C and vice versa. In this case D_i is called the natural operator and D_i is called the support operator. As for time, we adopt a centered difference in second-order accuracy. So the discretized difference equations are:

$$g_{ij} = D_j\left(u_i^n + \gamma v_i^{n-1/2}\right), \tag{25.7}$$

$$\sigma_{ij} = \Lambda\delta_{ij}g_{kk} + M\left(g_{ij} + g_{ji}\right), \tag{25.8}$$

$$a_i = R\mathsf{D}_j\sigma_{ij} - \mathsf{Q}_k y Q_k\left(u_i^n + \beta v_i^{n-1/2}\right), \tag{25.9}$$

$$v_i^{n+1/2} = v_i^{n-1/2} + \Delta t a_i, \tag{25.10}$$

$$u_i^{n+1} = u_i^n + \Delta t_i^{n+1/2}. \tag{25.11}$$

The material variable incorporate the cell volumes V^C and the node volumes V^N :

$$\Lambda = \lambda / V^C \tag{25.12}$$

$$M = \mu / V^C \tag{25.13}$$

$$R = 1 / \rho V^N \tag{25.14}$$

Viscous as well as stiffness hourglass control may be used, for which we define the viscosity β, and stiffness

$$y = \mu(\lambda + \mu) / [6(\lambda + 2\mu)] \tag{25.15}$$

The form we choose for hourglass stiffness y is based on the approximate analysis of Kosloff (1978). Instabilities in the numerical method due to non-uniform stress modes are corrected for by hourglass operators:

$$Q_k : H^N \to H^C \quad \text{and} \quad \mathsf{Q}_k : H^C \to H^N \tag{25.16}$$

25.3 Implement on GPU Using CUDA

25.3.1 Algorithm: Thread Strategy

As a finite differential method on structured grid, SORD scales well on GPU. We develop the GPU code based on David Wang's (McQuinn and Wang 2009) work. Use CUDA instead of Fortran to imply the program on NVIDIA GPU device. The program performs as Fig. 25.1.

C++ class is used for the data store in CPU. Ely's Fortran code uses some multi-dimensional arrays. We use class of C++ to implement multidimensional arrays.

We define a class floatnd include some basic operations such as add, multiply and copy, and some classes float3d, float4d and float5d inherited from floatnd, these classes have operations such as idx, set and get. The data can be stored in 1D array, and operated by the operations in class.

```
class floatnd
{
protected:
    int   m_count;
    float *m_data;
    int   m_size;
public:
    inline void Fill(float val);
    inline void Copy(floatnd *other);
    inline void Add(floatnd *other);
    inline void Multiply(floatnd *other);
    floatnd(){}
};

class float3d : public floatnd
{
 protected:
    int   m_dimx;
    int   m_dimy;
    int   m_dimz;
 public:
    float3d(int dimx, int dimy, int dimz);
    inline int Idx(int j, int k, int l)
    {
        return j + k * m_dimx + l * m_dimx * m_dimy;
    }
    inline float Get(int j, int k, int l)
    {
        int idx = Idx(j, k, l);
        return m_data[idx];
    }
    inline void Set(int j, int k, int l, float r)
    {
        int idx = Idx(j, k, l);
        m_data[idx] = r;
    }
};
```

Data is stored as 1D array in Memory and in GPU. So we can use 1D blocks and 1D threads of CUDA. For the data on GPU, we define another function to operate.

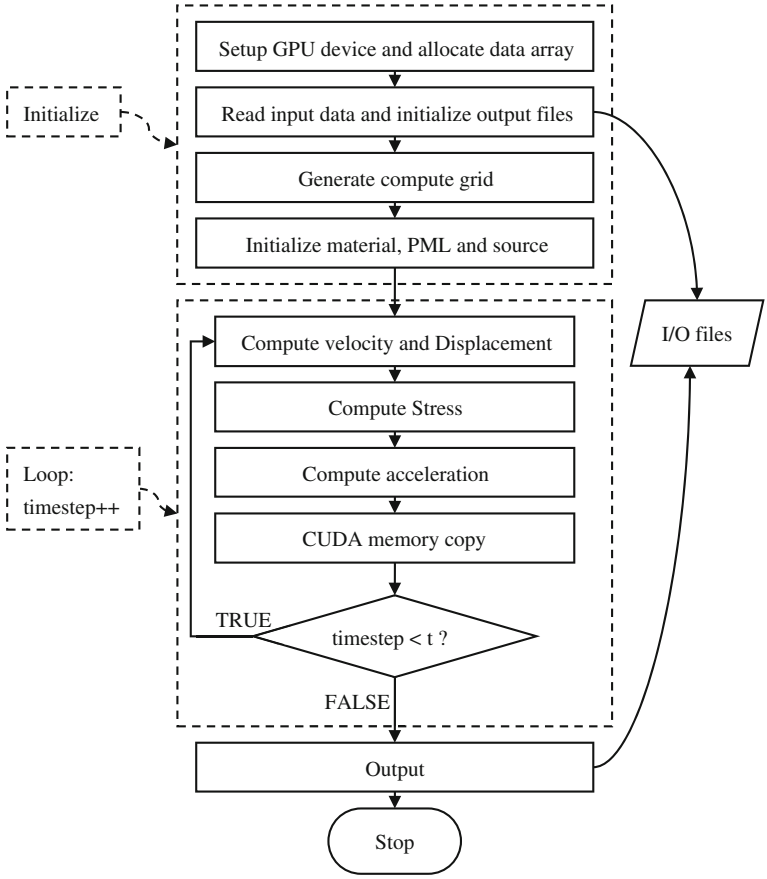


Fig. 25.1 Program flowchart of SORD on GPU

```
void cuda_Mult(float *this, float *other, int cnt)
{
    int cnt_b = (cnt <= MAX_THREAD) ? cnt : MAX_THREAD;
    int cnt_g = (cnt + MAX_THREAD - 1) / MAX_THREAD;
    dim3 geo_g(cnt_g);
    dim3 geo_b(cnt_b);
    kernel_Mult <<< geo_g, geo_b >>> (this, other, cnt, 0);
}

__global__ void kernel_Mult(float *this, float *other, int cnt, int
offset)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < cnt) this[offset + idx] *= other[idx];
}
```

25.3.2 Data Structure: Memory Strategy

We implement 3 kinds of kernels:

1. Directly access global memory without shared memory.
   Store all of the data in global memory and register, but accessing data in global memory will take about 500 cycles.
2. Through shared memory.
   First copy data from global memory into shared memory. The shared memory has very short access latency. But there is very little shared memory in one streaming multiprocessor, and the copying from global memory to shared memory will also cost about 500 cycles.
3. Directly access global memory and use texture memory.
   The texture memory space is read only and resides in device memory, but it is cached, so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache.

We compared the performance of the kernel with directly accessing global memory and the one using shared memory.

In this case, the shared memory is not very useful. In the seven cases of Table 25.1, the kernel using shared memory did not have obvious advantages. That's because the shared memory is too small, and the data copied to shared memory is takes too much times. So we use the kernels directly accessing global memory.

Reading device memory through texture fetching gives some benefits that can make it an advantageous alternative to reading device memory from global memory. In the SORD on CUDA, we have a five dimension vector to be stored, and after first calculated, it is read-only. So it can be stored in texture memory. Table 25.2 shows the speedup by texture memory.

Table 25.1 The performance of the kernel with directly accessing global memory and the one using shared memory

Data zone	Global memory (ms)	Shared memory (ms)	Speedup
$61 \times 61 \times 61$	78.3	120	0.652
$73 \times 73 \times 73$	135	152	0.889
$85 \times 85 \times 85$	213	205	1.04
$97 \times 97 \times 97$	315	290	1.09
$109 \times 109 \times 109$	433	372	1.16
$121 \times 121 \times 121$	601	733	0.820
$133 \times 133 \times 133$	800	854	0.937

Table 25.2 Speedup by texture memory

Data zone	Accessing global memory without texture memory (ms)	Accessing global memory with texture memory (ms)	Speedup
$101 \times 101 \times 161$	100	97.8	1.02
$131 \times 131 \times 161$	181	160	1.13

25.3.3 PML Absorbing Boundary Condition on GPU

PML sets up an absorbing layer where waves are exponentially attenuated and the reflection coefficient at the layer interface is nearly zero for all angles of incidence. The modified formulation is where $d(x_j)$ is the damping profile, and x_j is the distance measured from the node to cell location to the PML interface along the x, y or z direction. For the interior of the model, not in the PML, $d(x_j) = 0$, and the equations reduce to the elastic wave equations.

$$\dot{g}_{ij} + d(x_j) g_{ij} = \partial_j v_i \tag{25.17}$$

$$\sigma_{ij} = \lambda \delta_{ij} \sum_k g_{kk} + \mu (g_{ij} + g_{ji}) \tag{25.18}$$

$$\dot{p}_{ij} + d(x_j) p_{ij} = \partial_j \sigma_{ij} \tag{25.19}$$

$$a_i = (1/\rho) \sum_j \dot{p}_{ij} \tag{25.20}$$

$$\dot{v}_i = a_i \tag{25.21}$$

$$\dot{u}_i = v_i \tag{25.22}$$

We define a new structure to store PML information and use this structure to compute the data in PML region. In interior and PML region, we use different GPU codes.

25.4 Layered Model Test

To test SORD on GPU, we reproduce the double-couple point source test LOH.1 of Day and Bradley (2001). The model diagrammed in Fig. 25.1, consists of an underlying layer of 7 km and a layer of 1 km over the underlying layer. In the surface layer $V_x = 2,000\,\mathrm{m/s}$, $V_p = 4,000\,\mathrm{m/s}$, and density $\rho = 2,600\,\mathrm{kg/m^3}$, and in the underlying layer $V_x = 3,464\,\mathrm{m/s}$, $V_p = 6,000\,\mathrm{m/s}$, and density $\rho = 2,700\,\mathrm{kg/m^3}$. A double-coupled point source is located at 2 km depth lasts 0.1 s to produce seismic waves which then propagate across the whole space.

We do the calculations with a rectangular mesh of node spacing $\Delta x = 50\,\mathrm{m}$, and the grid size is $161 \times 161 \times 161$. The point source is at (0, 0, 41). This case is run for 1000 steps with a time step of $\Delta t = 0.004$.

Table 25.3 Compare the performances of CPU and GPU (time for one step)

Data zone	Data size ($\times 10^3$)	CPU performance (ms)	GPU performance (ms)	Speedup
$81 \times 81 \times 161$	1056	1530	120	12.8
$101 \times 101 \times 161$	1642	2141	171	12.5
$121 \times 121 \times 161$	2357	3141	255	12.3
$141 \times 141 \times 161$	3201	4209	343	12.3
$161 \times 161 \times 161$	4173	4482	409	11.0
$181 \times 181 \times 161$	5275	5681	584	9.73
$201 \times 201 \times 161$	6505	6514	606	10.7
$221 \times 221 \times 161$	7863	7333	718	10.2
$241 \times 241 \times 161$	9351	7906	845	9.36

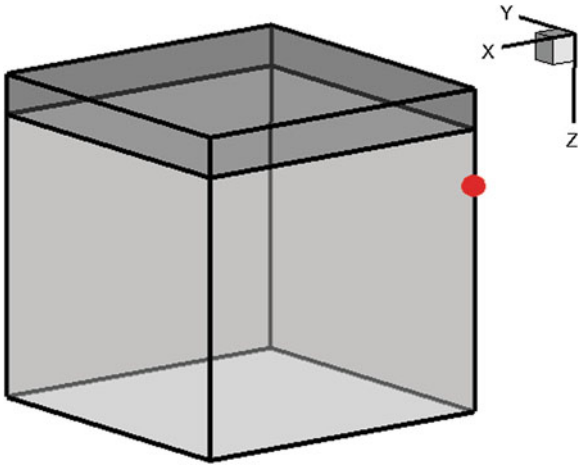


Fig. 25.2 Perspective view of the layer. The layer is 1 km thick the source is located on 2 km depth

The Table 25.3 lists the performance of the test. The baseline, single-thread CPU performance is measured on an Intel XEON E5410 system running at 2.33 GHz with 8 GB main memory. The CPU code is serial written by FORTRAN. The CUDA C code is based on a Tesla C1060 with 240 stream processors running at 1.296 GHz and 4 GB global memory.

Figure 25.2 shows the simulation of seismic wave propagation by GPU. We use CPU and GPU to compute different grid sizes. The CPU time of one step is shown in Table 25.3.

Figure 25.3 shows the performance of CPU and GPU, and the speedup data is in Fig. 25.4. The speedup ranges from 9.36 to 12.8 X. The maximum speedup is 12.8 X on the data size of $81 \times 81 \times 161$. We find when the data size increases the execution time varies almost linearly, that is, scaling is very good (Fig. 25.5).

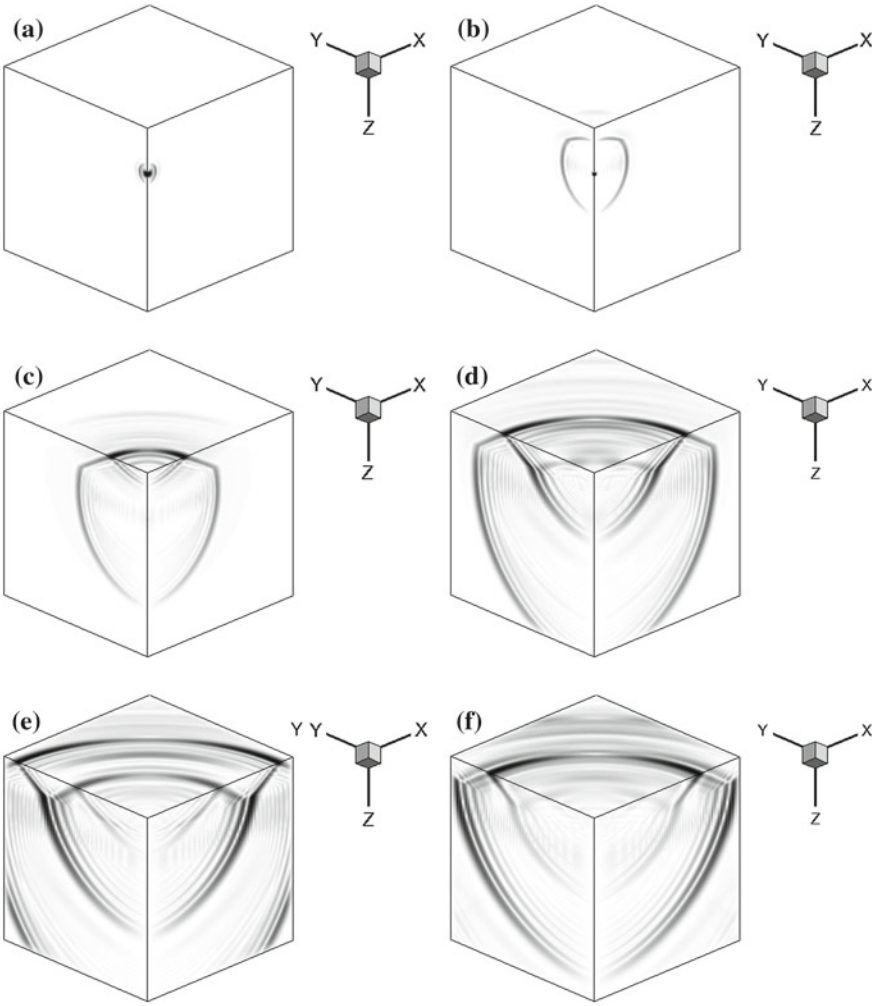


Fig. 25.3 Seismic wave propagate in the layers. Simulated ground velocity in different time. **a** t = 0.2 s, **b** t = 0.6 s, **c** t = 1.2 s, **d** t = 2 s, **e** t = 2.66 s, **f** t = 3.2 s

25.5 Conclusion and Future Work

We present a method of Support Operator on GPU using CUDA. We optimize the code using many threads to hide latency, and use texture memory to accelerate. Compared to CPU, GPU has over 9 X speedups.

Memory access bandwidth is small, need a high efficiency data structure. The global memory of single GPU is not enough for further large scale data size. We need use many GPUs to parallelize the code with MPI. We have tested a hybrid

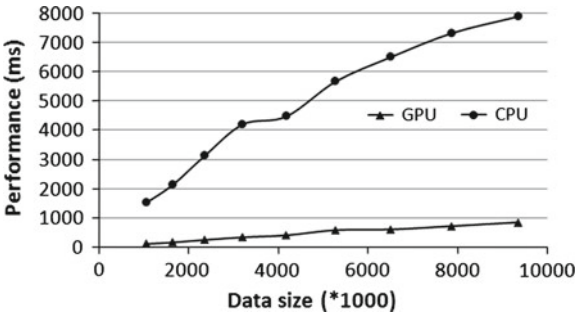


Fig. 25.4 Performance of CPU and GPU. When the data size increases, the time spent by GPU increases, too. Scaling is almost linear

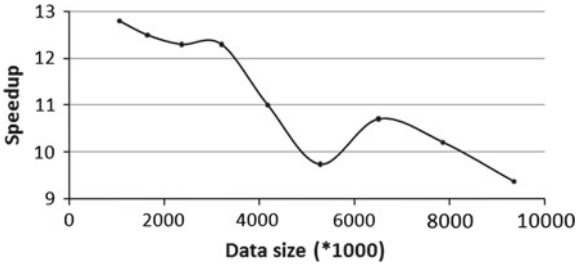


Fig. 25.5 The speedup of GPU compared to CPU

programming strategy of FORTRAN+CUDA, that is, MPI FORTRAN is used for inter-CPU communication and CUDA kernels are called on each GPU. The code is tested on both Tesla cards in single precision. 4 Tesla C1060 runs 11 times faster than 4 Intel Xeon E5410 cores when the mesh size is $501 \times 501 \times 501$. But host and device (GPU and CPU) data transfers usually hurt performance, because the bandwidth of PCI-E bus is smaller than the GPU bandwidth.

Appendix A: Hourglass Control from Node to Cell

This is the CUDA C code of hourglass operator Q_k. We use a modified form of the hourglass control scheme described by Flanagan & Belytschko. Flanagan and Belytschko (1981) and more recently by Day et al. Day et al (2005). and Ma & Liu. Ma and Liu (2006). As Equations (16), the code is used for transforming the node variables to cell variables.

$$(Q_1F)_{000} = F_{000} + F_{011} - F_{101} - F_{110} + F_{111} + F_{100} - F_{010} - F_{001} \quad (25.23)$$

$$(Q_2F)_{000} = F_{000} + F_{101} - F_{110} - F_{011} + F_{111} + F_{101} - F_{001} - F_{100} \quad (25.24)$$

$$(Q_3F)_{000} = F_{000} + F_{110} - F_{011} - F_{101} + F_{111} + F_{001} - F_{100} - F_{010} \quad (25.25)$$

$$(Q_4F)_{000} = F_{000} + F_{011} + F_{101} + F_{110} - F_{111} - F_{100} - F_{010} - F_{001} \quad (25.26)$$

```
#define UNIT unsigned int
#define FLOAT double
__global__ void kernel_hourglassnc(FLOAT *df_output, FLOAT *f, UINT iq,
UINT i, UINT j_start,UINT k_start,UINT l_start, UINT j_end,UINT
k_end,UINT l_end,UINT dimx, UINT dimy,UINT dimz, UINT off_1d_f, UINT
off_2d_f, UINT off_3d_f)
{
   FLOAT val = 0.0;
// Calc thread id
   UINT th_id = (threadIdx.y) +(blockIdx.x  * dimx) + (blockIdx.y  *
dimx * dimy);
          if(( threadIdx.y >= j_start && threadIdx.y < j_end )
          && ( blockIdx.x  >= k_start && blockIdx.x  < k_end )
          && ( blockIdx.y  >= l_start && blockIdx.y  < l_end ))
          {
// switch statement is unfolded - no branch - due to compile time template
           switch (iq)
           {
           case 0:
            val =
            f[th_id + (                          (off_3d_f * i))] +
            f[th_id + (1 + off_1d_f + off_2d_f + (off_3d_f * i))] +
            f[th_id + (    off_1d_f + off_2d_f + (off_3d_f * i))] +
            f[th_id + (1 +                       (off_3d_f * i))] -
            f[th_id + (1 +             off_2d_f + (off_3d_f * i))] -
            f[th_id + (    off_1d_f +            (off_3d_f * i))] -
            f[th_id + (1 + off_1d_f +            (off_3d_f * i))] -
            f[th_id + (               off_2d_f + (off_3d_f * i))];
             break;
           case 1:
            val =
            f[th_id + (                          (off_3d_f * i))] +
            f[th_id + (1 + off_1d_f + off_2d_f + (off_3d_f * i))] -
            f[th_id + (    off_1d_f + off_2d_f + (off_3d_f * i))] -
            f[th_id + (1 +                       (off_3d_f * i))] +
            f[th_id + (1 +             off_2d_f + (off_3d_f * i))] +
            f[th_id + (       off_1d_f +         (off_3d_f * i))] -
            f[th_id + (1 +  off_1d_f +           (off_3d_f * i))] -
            f[th_id + (               off_2d_f + (off_3d_f * i))];
           break;
```

```
            case 2:
              val =
              f[th_id + (                          (off_3d_f * i))] +
              f[th_id + (1 + off_1d_f + off_2d_f + (off_3d_f * i))] -
              f[th_id + (    off_1d_f + off_2d_f + (off_3d_f * i))] -
              f[th_id + (1 +                       (off_3d_f * i))] -
              f[th_id + (1 +            off_2d_f + (off_3d_f * i))] -
              f[th_id + (    off_1d_f +            (off_3d_f * i))] +
              f[th_id + (1 +  off_1d_f +           (off_3d_f * i))] +
              f[th_id + (               off_2d_f + (off_3d_f * i))];
            break;
            case 3:
               val =
              f[th_id + (                          (off_3d_f * i))] -
              f[th_id + (1 + off_1d_f + off_2d_f + (off_3d_f * i))] +
              f[th_id + (    off_1d_f + off_2d_f + (off_3d_f * i))] -
              f[th_id + (1 +                       (off_3d_f * i))] +
              f[th_id + (1 +            off_2d_f + (off_3d_f * i))] -
              f[th_id + (    off_1d_f +            (off_3d_f * i))] +
              f[th_id + (1 + off_1d_f +            (off_3d_f * i))] -
              f[th_id + (               off_2d_f + (off_3d_f * i))];
              break;
            }
            df_output[th_id] = val;
          }
      else
      {
       return;
      }
}
```

References

Day SM, Dalguer LA, Lapusta N, Liu Y (2005) Comparison of finite difference and boundary integral solutions to three-dimensional spontaneous rupture. J Geophys Res Solid Earth 110:23

Ely GP, Day SM, Minster JB (2008) A support-operator method for viscoelastic wave modelling in 3-D heterogeneous media. Geophys J Int 172:331–344

Ely GP, Day SM, Minster JB (2009) A support-operator method for 3-D rupture dynamics. Geophys J Int 177:1140–1150

Ely GP, Day SM, Minster JB (2010) Dynamic rupture models for the Southern San Andreas fault. Bull Seismol Soc Am 100:131–150

Flanagan DP, Belytschko T (1981) A uniform strain hexahedron and quadrilateral with orthogonal hourglas control. Int J Numer Methods Eng 17:679–706

Kosloff D, Frazier GA (1978) Treatment of hourglass patterns in low order finite-element codes. Int J Numer Anal Methods Geomech 2:57–72

Ma S, Liu PC (2006) Modeling of the perfectly matched layer absorbing boundaries and intrinsic attenuation in explicit finite-element methods. Bull Seismol Soc Am 96:1779–1794

McQuinn E, Wang D (2009). Nife: a GPU port of the support operator for rupture dynamics (SORD). San Diego: CSE 260

Michea D, Komatitsch D (2010) Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophys J Int 182:389–402

NVIDIA (2009) NVIDIA CUDA programming guide version 2.3. NVIDIA Corporation, Santa Clara

Samarskii AA, Tishkin VF, Favorskii AP, Shashkov MY (1981) Operational finite-difference schemes. Differ Equ 17:854–862

Shashkov MY (1996) Conservative finite-difference methods on general grids. CRC Press, Boca Raton

Zhao Y (2008) Lattice Boltzmann based PDE solver on the GPU. Vis Comput 24:323–333

Part VII
Algorithms and Solvers

Chapter 26
A Geometric Multigrid Solver on GPU Clusters

Harald Koestler, Daniel Ritter and Christian Feichtinger

Abstract Recently, more and more GPU HPC clusters are installed and thus there is a need to adapt existing software design concepts to multi-GPU environments. We have developed a modular and easily extensible software framework called WaLBerla that covers a wide range of applications ranging from particulate flows over free surface flows to nano fluids coupled with temperature simulations. In this article we report on our experiences to extend WaLBerla in order to support geometric multigrid algorithms for the numerical solution of partial differential equations (PDEs) on multi-GPU clusters. We discuss the object-oriented software and performance engineering concepts necessary to integrate efficient compute kernels into our WaLBerla framework and show that a large fraction of the high computational performance offered by current heterogeneous HPC clusters can be sustained for geometric multigrid algorithms.

Keywords MPI parallelization · GPGPU · CUDA · Multigrid solver

26.1 Introduction

The multi-disciplinary field of computational science and engineering (CSE) deals with large scale computer simulations and optimization of mathematical models. CSE is used successfully, e.g. by aerospace, automotive, and processing industries, as well as in medical technology. In order to obtain physically meaningful results, many of these simulation tasks can only be done on HPC clusters due to the high memory and compute power requirements. Therefore, software development in CSE is dominated by the need for efficient and scalable codes on current compute clusters.

H. Koestler (✉) · D. Ritter · C. Feichtinger
System Simulation Group, University of Erlangen-Nuremberg,Erlangen, Germany
e-mail: harald.koestler@informatik.uni-erlangen.de
http://www10.informatik.uni-erlangen.de

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 407
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_26,

Graphics processing units (GPUs) typically offer hundreds of specialized compute units operating on dedicated memory. In this way they reach outstanding compute and memory performance and are more and more used for compute-intensive applications, often called general purpose programming on graphics processing units (GPGPU). GPUs are best suitable for massively-data parallel algorithms, inadequate problems, that e.g. require a high degree of synchronization or provide only limited parallelism, are left to the host CPU. Recently, GPUs are more and more used to build up heterogeneous multi-GPU HPC clusters. In the current Top 500 list[1] of the fastest machines world-wide there are three of these clusters amongst the Top 5.

In order to achieve good performance on these GPU clusters, software development has to adapt to the new needs of the massively parallel hardware. As a starting point, GPU vendors offer proprietary environments for GPGPU. NVIDIA, e.g., provides the possibility to write single-source programs that execute kernels written in a subset of C and C++ on their Compute Unified Device Architecture (CUDA) (NVIDIA 2010). Since we are exclusively working on NVIDIA GPUs in this article we have done our implementations in CUDA. An alternative would have been to use the Open Compute Language (OpenCL).[2] Within OpenCL one can write code that runs in principle on many different hardware platforms, but to achieve good performance the implementation has to be adapted to the specific features of the hardware. Both CUDA and OpenCL are low-level languages. To make code development more efficient, one either has to provide wrappers for high-level languages like e.g. OpenMP (Ohshima et al. 2010) and PyCUDA (Klöckner et al. 2009) or easy to use frameworks.

Our contributions in this article are that we first discuss the concepts necessary to integrate efficient GPU compute kernels into our software framework called WaLBerla and second that we show scaling results for an exemplary multigrid solver on multi-GPU clusters.

Various other implementations of different multigrid algorithms on GPU exist (e.g. Bolz et al. 2003; Goddeke et al. 2008; Haase et al. 2010), and multigrid is also incorporated in software packages like OpenCurrent (Cohen 2011) or PETSc (Balay et al. 2009).

The paper is organized as follows: In Sect. 26.2 we briefly describe the multigrid algorithm and its parallelization on GPUs. Section 26.3 summarizes the MPI-parallel WaLBerla framework that easily enables us to extend our code to multi-GPUs, and in Sect. 26.4 we present performance results for different CPU and GPU platforms before concluding the paper in Sect. 26.5.

[1] http://www.top500.org

[2] see http://www.khronos.org/opencl/

26.2 Parallel Multigrid

26.2.1 Multigrid Algorithm

Multigrid is not a single algorithm, but a general approach to solve problems by using several levels or resolutions (Brandt 1977; Hackbusch 1985). We restrict ourselves to *geometric multigrid* (*MG*) in this article that identifies each level with a (structured) grid.

Typically, multigrid is used as an iterative solver for large linear systems of equations that have a certain structure, e.g. that arise from the discretization of PDEs and lead to sparse and symmetric positive definite system matrices. The main advantage of multigrid solvers compared to other solvers like Conjugate Gradients (CG) is that multigrid can reach an asymptotically optimal complexity of $\mathcal{O}(N)$, where N is the number of unknowns in the system. For good introductions and a comprehensive overview on multigrid methods, we, e.g., refer to Briggs et al. (2000) and Trottenberg et al. (2001), for details on efficient multigrid implementations see Douglas et al. (2000), Hülsemann et al. (2005), Stürmer et al. (2008) and Köstler (2008).

We assume that we want to solve the PDE

$$-\nabla c \nabla \boldsymbol{u} + \alpha u = f \quad \text{in } \Omega \tag{26.1a}$$

$$\langle \nabla \boldsymbol{u}, \boldsymbol{n} \rangle = 0 \quad \text{on } \partial\Omega \tag{26.1b}$$

with $\alpha > 0$, smoothly varying or constant coefficients $c : \mathbb{R}^3 \to \mathbb{R}^+$, solution $\boldsymbol{u} : \mathbb{R}^3 \to \mathbb{R}$, right hand side (RHS) $f : \mathbb{R}^3 \to \mathbb{R}$, and (natural) Neumann boundary conditions on a rectangular domain $\Omega \subset \mathbb{R}^3$. Equation 26.1 is discretized by finite volumes on a structured grid. This results in a linear system

$$A^h u^h = f^h, \quad \sum_{j \in \Omega^h} a_{ij}^h u_j^h = f_i^h, \quad i \in \Omega^h \tag{26.2}$$

with system matrix $A^h \in \mathbb{R}^{N \times N}$, unknown vector $u^h \in \mathbb{R}^N$ and right hand side (RHS) vector $f^h \in \mathbb{R}^N$ on a discrete grid Ω^h with mesh size h.

In order to solve the above linear system, we note that during the iteration the algebraic error $e^h = u_*^h - u^h$ is defined to be the difference between the exact solution u_*^h of Eq. 26.2 and the approximate solution u^h. With the residual equation $r^h = f^h - A^h u^h$ we obtain there so-called error equation

$$A^h e^h = r^h. \tag{26.3}$$

The multigrid idea is now based on two principles:

Smoothing Property: Classical iterative solvers like red-black Gauß-Seidel (RBGS) are able to smooth the error after very few steps. That means the high frequency

components of the error are removed well by these methods. But they have little effect on the low frequency components. Therefore, the convergence rate of classical iterative methods is good in the first few steps and decreases considerably afterwards. *Coarse Grid Principle*: A smooth function on a fine grid can be approximated satisfactorily on a grid with less discretization points, whereas oscillating functions would disappear. Furthermore, a procedure on a coarse grid is less expensive than on a fine grid. The idea is now to approximate the low frequency error components on a coarse grid.

Multigrid now combines these two principles into one iterative solver. The smoother reduces the high frequency error components first, and then the low frequency error components are approximated on coarser grids, interpolated back to the finer grids and eliminated there. In other words on the finest grid Eq. 26.1 is first solved approximately by a few smoothing steps and then an approximation to the error equation is computed on the coarser grids. This leads to recursive algorithms which traverse between fine and coarse grids in a grid hierarchy. Two successive grid levels Ω^h and Ω^H typically have fine mesh size h and coarse mesh size $H = 2h$.

One multigrid iteration, here the so-called *V-cycle*, is summarized in algorithm 1. Note that in general the operator A^h has to be computed on each grid level. This is either done by rediscretization of the PDE or by Galerkin coarsening, where $A^H = RA^hP$.

Algorithm 1 Recursive V-cycle: $u_h^{(k+1)} = V_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$

if coarsest level **then**
  solve $A^h u^h = f^h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}(u_h^{(k)}, A^h, f^h)$ {presmoothing}
  $r^h = f^h - A^h \bar{u}_h^{(k)}$ {compute residual}
  $r^H = Rr^h$ {restrict residual}
  $e^H = V_H(0, A^H, r^H, \nu_1, \nu_2)$ {recursion}
  $e^h = Pe^H$ {interpolate error}
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ {coarse grid correction}
  $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}(\tilde{u}_h^{(k)}, A^h, f^h)$ {postsmoothing}
end if

In our cell-based multigrid solver we use the following components:

- A RBGS smoother $\mathcal{S}_h^{\nu_1}, \mathcal{S}_h^{\nu_2}$ with ν_1 pre- and ν_2 postsmoothing steps.
- The restriction operator R from fine to coarse grid is simple averaging over the neighboring cells.
- We apply a nearest neighbor interpolation operator P for the error.
- The coarse grid problem is solved by a sufficient number of RBGS steps.
- The discretization of the Laplacian was done via the standard 7-point stencil (case $c = 1$ in Eq. 26.1).
- For varying c we apply Galerkin coarsening with a variable 7-point stencil on each grid level.

26.2.2 GPU Implementation

To implement the multigrid algorithm on GPU we have to parallelize it and write kernels for smoothing, computation of the residual, restriction, and interpolation together with coarse grid correction. In the following, we choose the RBGS kernel as an example and discuss it in more detail. Algorithm 25.2 shows the source code of the CUDA kernel. It is called from Algorithm 25.3.

The kernel can handle arbitrary variable seven-point stencils. The `GET3D_tex` and `GET3D_ST_tex` functions are macros that provide access to the solution resp. stencil field that is stored in global or texture GPU memory. Due to the splitting in red and black points within the RBGS to enable parallelization, only every second solution value is written back, whereas the whole solution vector is processed. Note that the outer if statement to check if the point is not a boundary point can be dropped on the new Fermi GPUs since they are much less sensitive to data alignment than older GPUs.

Algorithm 2 Red-black Gauss-Seidel smoother kernel in CUDA.

```
__global__ void kr_RBGS(double * stencil, double* solution, double* rhs,
                        const Uint xSize, const Uint ySize,
                        const Uint zSize, const Uint red_black)
{

  unsigned int x = threadIdx.x;
  unsigned int y = blockIdx.x;
  unsigned int z = blockIdx.y;

  if ((x > 0) && (y > 0) && (z > 0) &&
      (x < xSize-1) && (y < ySize-1) && (z < zSize-1) )
  {

    double fak = 1./GET3D_ST_tex(tex_stencil,x,y,z,C);
    double v = 0.0;
    v += GET3D_tex(tex_solution,x,y+1,z)*GET3D_ST_tex(tex_stencil,x,y,z,N);
    v += GET3D_tex(tex_solution,x,y-1,z)*GET3D_ST_tex(tex_stencil,x,y,z,S);
    v += GET3D_tex(tex_solution,x-1,y,z)*GET3D_ST_tex(tex_stencil,x,y,z,W);
    v += GET3D_tex(tex_solution,x+1,y,z)*GET3D_ST_tex(tex_stencil,x,y,z,E);
    v += GET3D_tex(tex_solution,x,y,z+1)*GET3D_ST_tex(tex_stencil,x,y,z,T);
    v += GET3D_tex(tex_solution,x,y,z-1)*GET3D_ST_tex(tex_stencil,x,y,z,B);

    double new_val = (GET3D(rhs,x,y,z)- v)/fak;

    if (((x+y+z)&1) == red_black)
      GET3D(solution,x,y,z) = new_val;
  }
}
```

The distributed memory parallelization is simply done by decomposing the finest grid into several smaller sub-grids and introducing a layer of ghost cells between them. Now the sub-grids can be distributed to different MPI processes and only the ghost cells have to be communicated to neighboring sub-grids. In case of multi-GPU processing, the function calling this kernel (shown in Algorithm 25.3) has to handle the ghost cells. The solution values at the borders of the sub-grids have to be initialized with the values that were already communicated via MPI. This transfer

from the MPI buffer in main memory to the memory of the GPU is the only part of communication that is not done implicitly by the WaLBerla framework. After that, the stencil, solution and right-hand side values are put to texture arrays to have a more efficient read access to them later. This is not necessary for the new NVIDIA Fermi GPUs, since they offer a built-in cache for the global GPU memory. Now, the actual red and the black sweep are done. After the sweeps, the Neumann boundary conditions are set by copying the border values to a ghost layer. Finally, the values to be communicated are transferred again to the MPI buffers, before the mapping of the texture objects is released.

Algorithm 3 Cuda kernel wrapper

```
void RBGSSweepVarGPU(double * st, double* sol,
                         double* rhs, double** buffers,
                         const Uint xSize, const Uint ySize,
                         const Uint zSize, const Uint * blockID,
                         const Uint * numBlocks)
{

  dim3 dimblock(xSize,1);
  dim3 dimgrid(ySize,zSize);

  //COPY MPI_BUFFER -> GPU
  SetBuffers(sol,buffers,xSize,ySize,zSize,blockID, numBlocks);

  LoadTextureFrom1DArray(st, xSize*ySize*zSize*StenCellSize, tex_stencil);
  LoadTextureFrom1DArray(sol, xSize*ySize*zSize, tex_solution);
  LoadTextureFrom1DArray(rhs, xSize*ySize*zSize, tex_rhs);

  kr_RBGS<<<dimgrid, dimblock>>>(st, sol, rhs, xSize, ySize, zSize,0.);
  kr_RBGS<<<dimgrid, dimblock>>>(st, sol, rhs, xSize, ySize, zSize,1.);

  Treatboundary(sol,xSize, ySize, zSize, blockID, numBlocks);

  //COPY GPU -> MPI_BUFFER
  CopyBuffers(sol,buffers,xSize,ySize,zSize,blockID, numBlocks);

  UnloadTextureFrom1DArray(tex_stencil);
  UnloadTextureFrom1DArray(tex_solution);
  UnloadTextureFrom1DArray(tex_rhs);

}
```

26.3 WaLBerla

WaLBerla is a massively parallel multi-physics software framework developed for HPC applications on block-structured domains (Feichtinger et al. 2010). It has been successfully used in many simulation tasks ranging from free surface flows (Donath et al. 2009) to particulate flows (Götz et al. 2010) and fluctuating lattice Boltzmann (Dünweg et al. 2007) for nano fluids.

The main design goals of the WaLBerla framework are to provide excellent application performance across a wide range of computing platforms and the easy integration of new algorithms. The current version WaLBerla 2.0 is capable of running

heterogeneous simulations on CPUs and GPUs with static load balancing (Feichtinger et al. 2010).

26.3.1 Patch, Block, and Sweep Concept

A fundamental design concept of WaLBerla is to rely on block-structured grids, what we call our *Patch* and *Block* data structure. We restrict ourselves to block-structured grids in order to support efficient massively parallel simulations.

In our case a Patch denotes a cuboid describing a region in the simulation that is discretized with the same resolution. This Patch is further subdivided into a Cartesian grid of Blocks consisting of cells. The actual simulation data is located on these cells. In parallel one or more Blocks can be assigned to each process in order to support load balancing strategies. Furthermore, we may specify for each Block, on which hardware it is executed. Of course, this requires also to be able to choose different implementations that run on a certain Block, what is realized by our functionality management.

The functionality management in WaLBerla 2.0 controls the program flow. It allows to select different functionality (e.g. kernels, communication functions) for different granularities, e.g. for the whole simulation, for individual processes, and for individual Blocks.

When the simulation runs, all tasks are broken down into several basic steps, so-called *Sweeps*. A Sweep consists of two parts: a communication step fulfilling the boundary conditions for parallel simulations by nearest neighbor communication and a communication independent work step traversing the process-local Blocks and performing operations on all cells. The work step usually consists of a kernel call, which is realized for instance by a function object or a function pointer. As for each work step there may exist a list of possible (hardware dependent) kernels, the executed kernel is selected by our functionality management.

26.3.2 MPI Parallelization

The parallelization of WaLBerla can be broken down into three steps:

1. a data extraction step,
2. a MPI communication step, and
3. a data insertion step.

During the data extraction step, the data that has to be communicated is copied from the simulation data structures of the corresponding Blocks. Therefore, we distinguish between process-local communication for Blocks lying on the same and MPI communication for those on different processes.

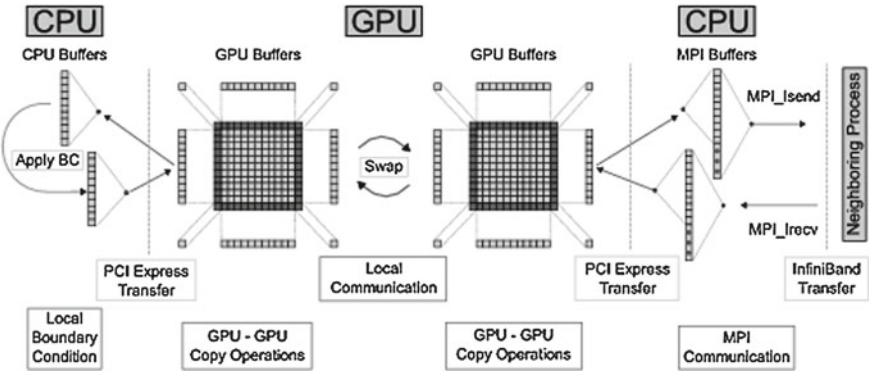


Fig. 26.1 Communication concept within WaLBerla (Feichtinger et al. 2010)

Local communication directly copies from the sending Block to the receiving Block, whereas for the MPI communication the data has first to be copied into buffers. For each process to which data has to be sent, one buffer is allocated. Thus, all messages from Blocks on the same process to another process are serialized.

To extract the data to be communicated from the simulation data, extraction function objects are used that are again selected via the functionality management. The data insertion step is similar to the data extraction, besides that we traverse the block messages in the communication buffers instead of the Blocks.

26.3.3 Multi-GPU Implementation

For parallel simulations on GPUs, the boundary data of the GPU has first to be copied by a PCIe transfer to the CPU and then be communicated via MPI routines. Therefore, we need buffers on GPU and CPU in order to achieve fast PCIe transfers. In addition, on-GPU copy kernels are added to fill these buffers. The whole communication concept is depicted in Fig. 26.1.

The only difference between parallel CPU and GPU implementation is that we need to adapt the extraction and insertion functions. For the local communication they simply swap the GPU buffers, whereas for the MPI communication the function *cudaMemcpy* is used to copy the data directly from the GPU buffers into the MPI buffers and vice versa.

To support heterogeneous simulations on GPUs and CPUs, we execute different kernels on CPU and GPU and also define a common interface for the communication buffers, so that an abstraction from the hardware is possible. Additionally, the work load of the CPU and the GPU processes can be balanced e.g. by allocating several Blocks on each GPU and only one on each CPU-only process.

Table 26.1 Technical hardware specifications

	Xeon 5550	Tesla M1060	Tesla C2070
Processor frequency	2.66 GHz	1.3 GHz	1.15 GHz
Memory frequency	1.3 GHz	800 MHz	1.5 GHz
Memory size	12 GB	4 GB	6 GB
# Streaming units/cores	4	240	448
Floating-point performance (SP)	85.1 GFLOP/s	933 GFLOP/s	1030 GFLOP/s
Floating-point performance (DP)	42.6 GFLOP/s	78 GFLOP/s	515 GFLOP/s
Memory bandwidth	32 GB/s	102 GB/s	144 GB/s

26.4 Performance Results

The main focus within this article is put on parallel efficiency of our multigrid implementation on multi-GPU clusters. As a baseline we also evaluate the single GPU runtime and identify the performance bottlenecks.

26.4.1 Platforms for Tests

The numerical tests were performed on three different platforms provided by our local computing center,[3] an Intel Core i7 CPU (Xeon 5550), an NVIDIA G80 (Tesla M1060) and an NVIDIA Fermi (Tesla C2070) platform. The detailed hardware specifications are depicted in Table 26.1.

The cluster for our test is the *TinyGPU* cluster of the RRZE. It consists of 8 dual-socket nodes, hosting two Xeon 5500 processors and two Tesla M1060 boards. The nodes are connected via Infiniband. Additionally, we had access to one of those nodes with two Tesla C2070 GPUs instead of the M1060.

All numerical tests run with double floating-point precision.

26.4.2 Scaling Experiments

In the following, we show how the runtime code behaves with increasing problem size on one or several compute nodes. Baseline is the performance on one node, i.e. two Teslas resp. two Xeons.

We distinguish two types of experiments: *Weak scaling* relates to experiments were the problem size is increased linearly with the number of involved devices, whereas the term *strong scaling* implies that we have a constant global problem size and vary only the number of processes. Assuming a perfect parallelization, we

[3] http://www.rrze.de

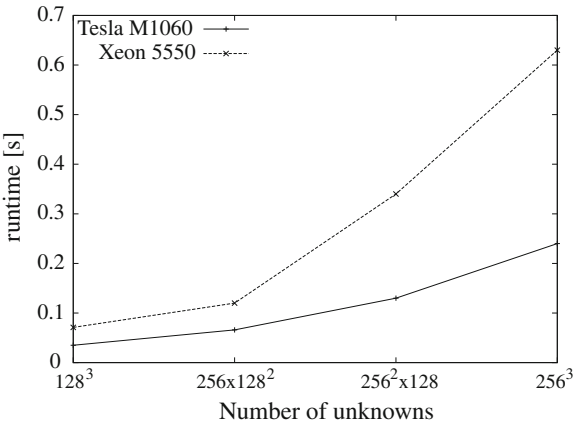


Fig. 26.2 Single node (i.e. two Teslas resp. two Xeons) performance of one multigrid V(2,2)-cycle

expect the runtime to be constant in weak scaling experiments, while we expect the runtime to be reciprocally proportional to the number of parallel processes in strong scaling experiments. We measure the runtime of one V(2,2)-cycle (i.e. a V-cycle with 2 RBGS iterations for pre- and postsmoothing each) on four grid levels with parameters from Sect. 26.2. On the coarsest grid 50 RBGS steps are performed to obtain a solution.

Single-node performance Here, the problem sizes vary between $128^3 = 2{,}097{,}152$ and $256^3 = 16{,}777{,}216$ unknowns and the runtimes for one V(2,2)-cycle on Xeon 5550 and Tesla M1060 machines are depicted in Fig. 26.2.

Since the performance of the multigrid algorithm is memory-bandwidth bounded and there is roughly a factor of three in theoretical peak memory bandwidth between CPU and GPU we expect the same factor in the runtime of both platforms. Indeed, the GPU shows a speedup factor of about three for larger problem sizes. For smaller problems this factor shrinks down due to several reasons: first, the GPU overhead e.g. for CUDA kernel calls becomes visible and there is not enough work to be done in parallel, especially on the coarse grids. Furthermore, the CPU can profit from its big caches.

Weak scaling Figure 26.3 shows the weak scaling behavior of the code for problem size 256^3. On the Xeon—having eight physical cores per node on two sockets—tests are performed with four and eight MPI instances per node. We did not pin the MPI processes to fixed cores and thus the runtimes slightly vary between 0.6 and 0.9 seconds for one V(2,2)-cycle. In average the results do not differ much for four and eight MPI instances per node. In case of the Tesla we have only two MPI instances per node and the runtime is quite stable. However, it increases by approximately 20 % for eight nodes compared to the baseline performance. This slightly worse scaling factor on GPU is mainly due to the effect of additional intra-node memory transfers of ghost layers between CPU and GPU.

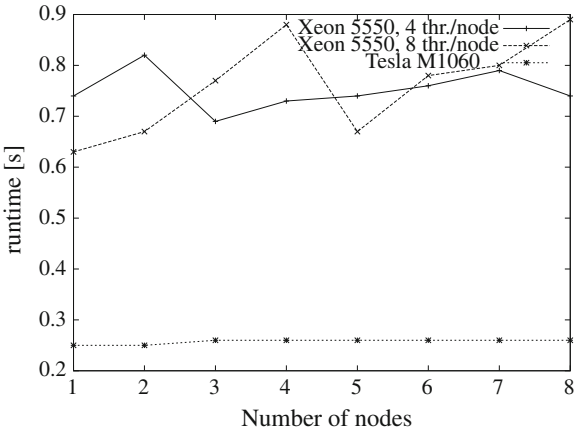


Fig. 26.3 Weak scaling behavior per computer node of one multigrid V(2, 2)-cycle

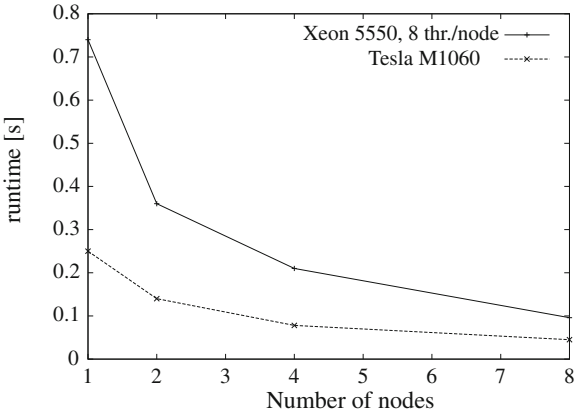


Fig. 26.4 Strong scaling behavior of one multigrid V(2, 2)-cycle with 256^3 unknowns

Table 26.2 Speedup factors for strong scaling experiment in Fig. 26.4

#Nodes	2	4	8
Tesla M1060	1.79	3.21	5.56
Xeon 5550	2.06	3.52	7.71

Strong scaling Next, we scale the number of involved processing units, but leave the total problem size, i.e. the number of unknowns, constant. Figure 26.4 shows the runtimes on the Xeon and the Tesla for $2 \cdot 256^3$ unknowns are shown for one to eight nodes of the cluster and Table 26.2 the corresponding relative speedup factors.

For the Xeon tests the parallel efficiency is relatively high: The speedup factor on 8 nodes of 7.71 stays only slightly below the ideal one, which is linear with the

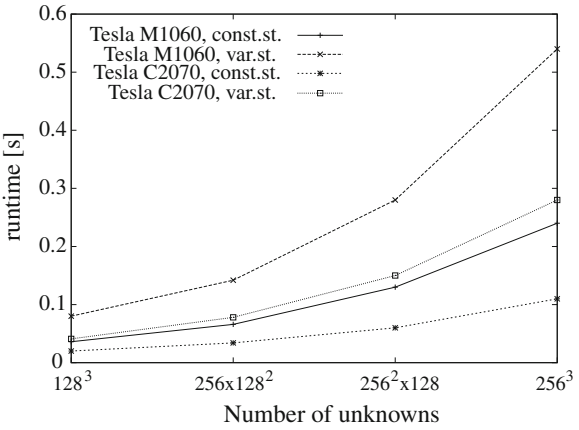


Fig. 26.5 Performance comparison of constant and variable 7-point stencil within one multigrid V(2,2)-cycle

number of nodes and thus 8. For two nodes even super-linear scaling is reached with the CPU code, this could be caused for example by cache effects.

The speedup on the Tesla is just 5.56, which is a result of different factors: on the one hand the problems for small size mentioned when discussing the single-node performance and on the other hand the communication overhead addressed within the weak scaling experiments.

26.4.3 CUDA Compute Capability 1.3 Versus 2.0

Next we evaluate the performance of a newer NVIDIA Fermi Tesla C2070 graphics board that implements CUDA compute capability 2.0. Its technical specifications are listed in Table 26.1 and the difference to the previous generation (CUDA compute capability 1.3) is, beneath a higher bandwidth, a tremendous increase in double-precision floating-point performance and real caches. Furthermore, it is easier to program: Alignment requirements are weakened and C++ support is enabled.

We now test also variable 7-point stencils, i.e. c from Eq. 26.1 is no more constant within the domain Ω. In this case we have in contrast to the constant coefficient stencil to store additionally the stencil at each grid point. We then use the RBGS smoother as sketched in Algorithm 25.2. The runtimes for constant and variable stencils are shown in Fig. 26.5 on both GPU platforms. With the constant stencil the C2070 is 2.2 times faster than the M1060, whereas for the variable stencil the speedup factor is 1.9. The performance gain cannot be a single effect of the increased memory bandwidth. In addition to that we also profit from the new features of the Fermi, especially from the cache and the weaker alignment requirements.

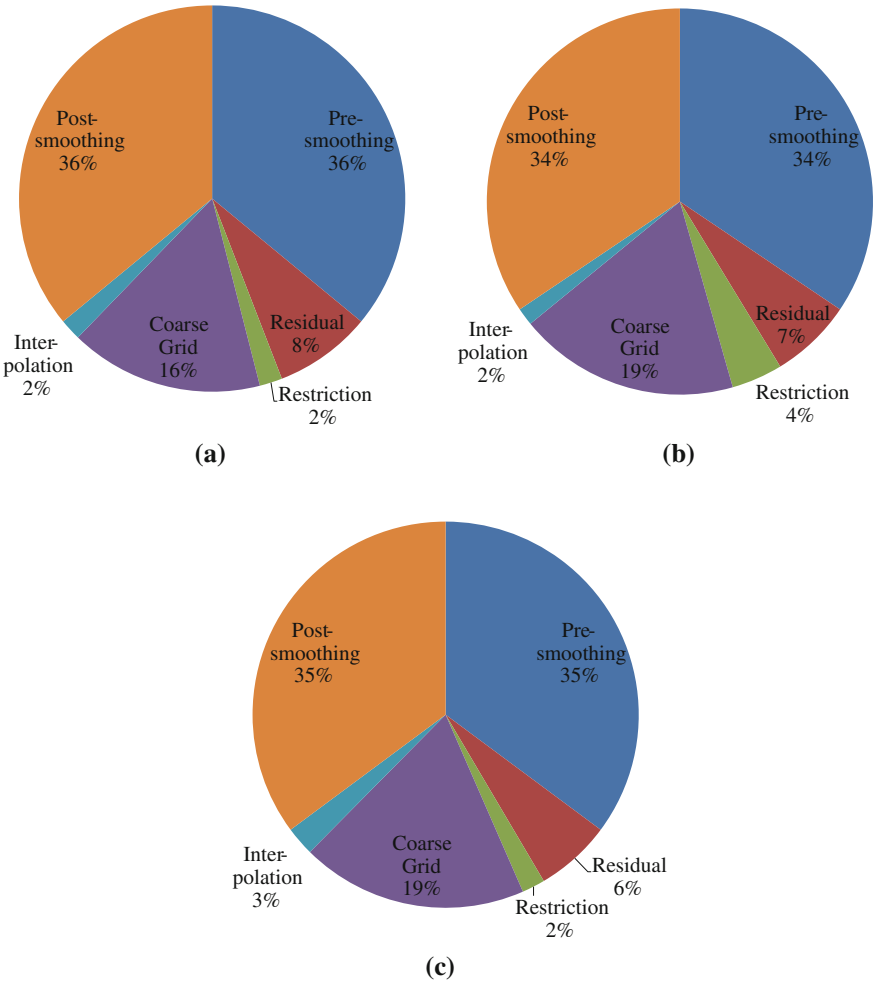


Fig. 26.6 Runtime percentage for different components on **a** one and **b** eight Tesla M1060 and **c** one Tesla C2070 (problem size 256^3, constant stencil)

26.4.4 Runtime of Components

In this section we try to give more insight in the runtime behavior of the different parts of the multigrid solver in order to identify the performance bottlenecks.

Therefore, Fig. 26.6 shows the portion of runtime spent in different components of a V(2, 2)-cycle on one resp. eight Tesla M1060 and one Tesla C2070. Since the overall performance of the multigrid solver is bounded by the memory bandwidth it is not astonishing that in each case smoothing on the finest grid takes around 70 % of the runtime. The problem size shrinks by a factor of 8 for each grid level, thus

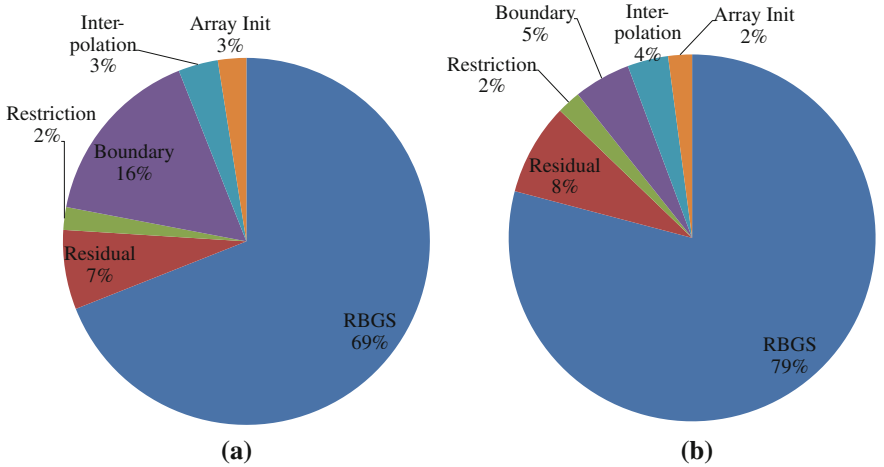


Fig. 26.7 Runtime percentage of different CUDA kernels on one Tesla C2070 for problem size **a** 128^3 and **b** 256^3 (constant stencil)

one expects the coarse grid (this includes all the previous components on all three coarser grids plus solving the problem on the coarsest grid with 50 RBGS iterations) to require about 1/8 of the runtime. This lies a little bit higher especially for the Tesla C2070, because the smaller sizes are not very efficient on the GPU as seen before. In the multi-GPU case the cost for the communication between the single processes is an important issue. In contrast to the calculations that are simply distributed, the communication has to be done extra when switching from sequential to parallel code. Therefore, it has to be ensured that this extra work does only consume little time and does not dominate the whole runtime. Within the multigrid algorithm the communication is part of pre- and postsmoothing, where the solution ghost layer has to be communicated to the neighboring processes, and of the restriction, where the residual ghost layer has to be distributed. In Fig. 26.6 the communication effects are an increase of the restriction runtime including the residual ghost layer transfer and the increase of the portion spent on the coarser grids, because of the worse ratio between local points and ghost layer points there and the bigger influence of latencies in case of smaller transfers. Altogether, runtime distribution in the one-GPU and multi-GPU case looks quite similar. To explain this, note that the difference in runtimes on one and eight Tesla M1060 is 37 ms or 13 % of the total runtime (for 256^3 unknowns per GPU and a constant stencil). This means, at least for that number of GPUs, the communication does not have a dominating effect on the runtime.

To provide another point of view we also measured the performance of our CUDA kernels for the same setting as above on one Tesla C2070 with the profiler *computeprof* provided by NVIDIA. The results are summarized in Fig. 26.7. In contrast to the previous measurements these do not contain the overhead from thread creation and cleanup. Additionally, they show more details since some of the components in Fig. 26.6 call more than one CUDA kernel. In total, we observe that the RBGS kernel

dominates the calculation. For 256^3 unknowns it requires almost 80 % of the runtime and for 128^3 unknowns still almost 70 %. For the small problem size the boundary treatment, which is included in the smoothing in Fig. 26.6, plays quite a role with 15 % of the total runtime.

26.5 Conclusions and Future Work

We have implemented a geometric multigrid solver for Eq. 26.1 on GPU and integrated it into the WaLBerla framework. We observed that the runtimes decrease on the current NVIDIA Fermi architecture due to its new hardware features like an incorporated memory cache. The speedup factor between CPU and GPU implementations of our multigrid algorithm corresponds roughly to the ratio of their memory bandwidths in the single-node case. Our experiments on a small compute cluster showed good scaling behavior for CPUs and slightly worse for GPUs.

Next steps would be a performance optimization of our code and a comparison of CUDA and OpenCL. One obvious improvement would be to use an optimized data layout, by splitting the red and black grid points into two separate fields. Finally, we currently investigate further applications for the parallel multigrid solver within the WaLBerla framework.

References

Balay S, Buschelman K, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2009) PETSc web page. http://www.mcs.anl.gov/petsc

Bolz J, Farmer I, Grinspun E, Schröder P (2003) Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: ACM SIGGRAPH 2003 papers, pp 917–924

Brandt A (1977) Multi-level adaptive solutions to boundary-value problems. Math. Comput. 31(138):333–390

Briggs W, Henson V, McCormick S (2000) A multigrid tutorial, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia

Cohen J (2011) OpenCurrent. NVIDIA research. http://code.google.com/p/opencurrent/

Donath S, Feichtinger C, Pohl T, Götz J, Rüde U, (2009) Localized parallel algorithm for bubble coalescence in free surface lattice-Boltzmann method. In: Sips H, Epema D, Lin H-X (eds) Euro-Par, (2009) Lecture notes in computer science, vol 5704. Springer, Berlin, pp 735–746

Douglas C, Hu J, Kowarschik M, Rüde U, Weiß C (2000) Cache optimization for structured and unstructured grid multigrid. Elect Trans Numer Anal 10:21–40

Dünweg B, Schiller U, Ladd AJC (Sep 2007) Statistical Mechanics of the Fluctuating Lattice Boltzmann Equation. Phys. Rev. E 76(3):036704

Feichtinger C, Donath S, Köstler H, Götz J, Rüde U (2010) WaLBerla: HPC software design for computational engineering simulations. J Comput Sci (submitted)

Feichtinger C, Habich J, Köstler H, Hager G, Rüde U, Wellein G (2010) A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters. J Parallel Comput. Arxiv, preprint arXiv:1007.1388 (submitted)

Goddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Wobker H, Becker C, Turek S (2008) Using GPUs to improve multigrid solver performance on a cluster. Int J Comput Sci Eng 4(1):36–55

Götz J, Iglberger K, Feichtinger C, Donath S, Rüde U (2010) Coupling multibody dynamics and computational fluid dynamics on 8192 processor cores. Parallel Comput 36(2–3):142–151

Haase G, Liebmann M, Douglas C, Plank G (2010) A parallel algebraic multigrid solver on graphics processing units. In: Zhang W et al (eds) High performance computing and applications. Springer, Berlin, pp 38–47

Hackbusch W (1985) Multi-grid methods and applications. Springer, Berlin

Hülsemann F, Kowarschik M, Mohr M, Rüde U (2005) Parallel geometric multigrid. In: Bruaset A, Tveito A (eds) Numerical solution of partial differential equations on parallel computers. Lecture notes in computational science and engineering, vol 51. Springer, Berlin, pp 165–208

Klöckner A, Pinto N, Lee Y, Catanzaro B, Ivanov P, Fasih A (2009) PyCUDA: GPU run-time code generation for high-performance computing. Arxiv preprint arXiv 911. http://mathema.tician.de/software/pycuda

Köstler H (2008) A multigrid framework for variational approaches in medical image processing and computer vision. Verlag Dr, Hut, München

NVIDIA Cuda Programming Guide 3.2 (2010). http://developer.nvidia.com/object/cuda_3_2_downloads.html

Ohshima S, Hirasawa S, Honda H (2010) OMPCUDA: OpenMP execution framework for CUDA based on omni OpenMP compiler. In: Beyond loop level parallelism in OpenMP: accelerators, tasking and more, pp 161–173

Stürmer M, Wellein G, Hager G, Köstler H, Rüde U, (2008) Challenges and potentials of emerging multicore architectures. In: Wagner S, Steinmetz M, Bode A, Brehm M (eds) High performance computing in science and engineering. Garching/Munich, (2007) LRZ. KONWIHR. Springer, Berlin, pp 551–566

Trottenberg U, Oosterlee C, Schüller A (2001) Multigrid. Academic Press, San Diego

Chapter 27
Accelerating 2-Dimensional CFD on Multi-GPU Supercomputer

Sen Li, Xinliang Li, Long Wang, Zhonghua Lu and Xuebin Chi

Abstract In this paper, we describe the domain decomposing strategy of finite-difference to implement and optimize GPU codes in solving 2-D N-S equations. To satisfy GPU architecture, our algorithms emphasize on the decomposition strategy and the maximum of exploiting the GPU memory hierarchy so that high rate of speedup can be expected. Tests on two CFD cases, respectively being cavity flow and aerofoil RAE 2822, are used. For cavity flow, we ran our simulation both on CUDA and OpenCL platform and witnessed 30–60x speedup. In aerofoil, we used 6–60 GPU devices and get speedup of 5–29 times depending on the grid size and number of devices used.

Keywords CUDA · OpenCL · Cavity flow · Aerofoil · CFD

27.1 Introduction

In recent years, Graphic Processing Units (GPUs) are becoming a compelling solution in the high performance computing field. Driven by many cores inside its architecture, GPU outperforms the performance of traditional CPU in terms of computational power with a very attractive cost/performance ratio. With the release of Compute Unified Device Architecture (CUDA) by Nvidia and OpenCL by Khronos, GPU has opened an era for general purpose computation. Greatly increased by programmability and specially designed for scientific computing, GPUs are now employed in a wide variety of fields such as physics, bioinformatics, finance, astronomy and numerical calculation. In this paper, we focused on the use of GPU to accelerate two cases in computational fluid dynamics area.

S. Li (✉) · X. Li · L. Wang · Z. Lu · X. Chi
Supercomputing center, Computer Network Information Center,
Chinese Academy of Sciences, Beijing, China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 423
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_27,

Traditionally, the numerical simulation of CFD has been studied over a long period of time. But results are often limited to high demand of computing power each simulation required. Traditional approaches to solve CFD were based on CPU and thus always appeared to be time consuming and only calculated a small grid size. To satisfy performance requirements, strong devices are required. GPUs, with extraordinary computing power, are exactly the one to meet that need.

In fact, efforts to exploit the GPU for higher performance in CFD have already been underway. For example, Dennis (2009) studied the issues in accelerating a well-known CFD code, Overflow, on the GPU; T. Brandvik and Pullan (2007) accelerated a 2-dimensional Euler flow solver using commodity graphics hardware. But all these studies have two defects. First, with the innovation of new GPU technology, the performance test on the latest GPU platform is not included. Typically, OpenCL is one technology being ignored. Second, solutions are based on single GPU. Though speedups are gained, they are not scalable to large problem size.

Based on previous work, we port cavity flow problem on OpenCL platform and RAE 2822 codes on multi-GPUs clusters to further verify the GPU performance in CFD area. Normally, a single GPU hardware contains hundreds of processing units and utilizes single instruction multiple thread (SIMT) mode. Threads are dispatched to different processing units and execute at the same time. In hybrid paradigm, by containing both message passing and GPU, we expect a tremendous increment in performance. Our codes are written in C and run on Nvidia C1060 and ATI 9270 GPU clusters with MPI for data exchange.

27.2 The Governing Equations and Finite-Difference Discretization

27.2.1 The Cavity Flow Equation

The governing equations of the cavity flow are the incompressible Navier-Stokes equations as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{27.1}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{1}{\mathrm{Re}}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{27.2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \frac{1}{\mathrm{Re}}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{27.3}$$

where ρ, u, v, p are density, x, y-direction velocity components and pressure respectively. Re represents the Reynolds number based on the velocity of the up-boundary and the cavity's width. Since the flow is laminar, no turbulence model is used. The 3rd order Quick method is used to solve the equations numerically.

27.2.2 The Flow Over a RAE2822 Aerofoil

The governing equations are compressible aerofoil RAE2822 N-S equations as:

$$\frac{\partial}{\partial t}U + \frac{\partial}{\partial x}f_1 + \frac{\partial}{\partial y}f_2 = \frac{\partial}{\partial x}V_1 + \frac{\partial}{\partial y}V_2 \tag{27.4}$$

where U is the solution vector, V_1 and V_2 are viscous fluxes, f_1 and f_2 are the Cartesian components of the convective fluxes that

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, f_1 = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E+p) \end{bmatrix}, f_2 = \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ v(E+p) \end{bmatrix} \tag{27.5}$$

In above equations, ρ represents the fluid density, u and v are the x and y components of fluid velocity, E the total energy, and p is the pressure which can be calculated from the equation of state of a perfect gas.

To simulate the flow over RAE2822 aerofoil, we use a two-layer algebraic 0-equation Baldwin-Lomax turbulence model, which gives the eddy viscosity, μ_t, as a function of the local boundary layer velocity profile. The Baldwin-Lomax model is suitable for high-speed flows with thin attached boundary layers, and aerofoil is one of such typical application. In this paper, we focus on the GPU solution not the formulae themselves, thus the detail of BL turbulence model equations, inner region of the Prandtl-Van Driest formula and outer region can refer to B.S. Baldwin and H. Lomax (1978).

In spatial point of view, we concerned the numerical method that replaces the continuous problem by the Partial Differential Equations (PDEs) by a finite set of discrete values. In practice, we used finite difference approach to regards these values as point values defined at grid points. A five orders upwind scheme is used to deal with those PDEs with equation as:

$$f'_j = (-2f_{j-3} + 15f_{j-2} - 60f_{j-1} + 20f_j + 30f_{j+1} - 3f_{j+2})/60h \tag{27.6}$$

where f'_j is the grid point we need to calculate.

A distinguishing feature of upwind method tells that the discretization of the equations on a mesh is performed according to the direction of propagation of information on that mesh. So salient features of the physical phenomena modeled by the equations are incorporated into the discretization schemes and we can use flux vector splitting method to identify upwind directions. In our experiment, by decomposing the coefficient matrix into a positive component and a negative component, we employed Steger-Warming splitting approach as seen in Eleuterio F. Toro (1999). On the discretization of time step, we used three-order Runge-Kutta approach that let $\frac{\partial U}{\partial t} = f$, then:

$$
\begin{aligned}
U^{(1)} &= \alpha_1 U^n + \beta_1 f^n \Delta t \\
U^{(2)} &= \alpha_2 U^n + \beta_2 (U^{(1)} + f^{(1)} \Delta t) \\
U^{(3)} &= \alpha_3 U^n + \beta_3 (U^{(2)} + f^{(2)} \Delta t) \\
U^{n+1} &= U^{(3)}
\end{aligned}
\tag{27.7}
$$

where

$$\alpha_1 = 1, \beta_1 = 1, \alpha_2 = 3/4, \beta_2 = 1/4, \alpha_3 = 1/3, \beta_3 = 2/3$$

27.3 Implementation on GPUs

27.3.1 Brief Introduction of GPU Architecture

The prevailing GPU tools available for scientific world are CUDA and OpenCL, with the latter less popular since it is just released last year and regarded as API instead of real hardware architecture. Regardless of software being used, most GPU devices include a unified shader pipeline, allowing each ALU on the chip to be marshaled by a program intending to perform general-purpose computations. CUDA uses thread hierarchy to describe the problem yet to be solved. The entire space are divided into several blocks with each block contains several threads as illustrated by Fig. 27.1. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processor in the system. Inside this model, threadIdx is a three component vector so that each thread can be identified. All the threads within a block are expected to reside on the same multi-processor core and must share the limited memory resources of that core. On the memory hierarchy, CUDA threads can access data from multiple memory space during their execution. Each thread has

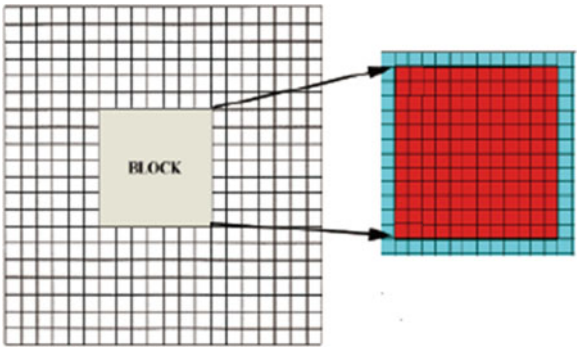


Fig. 27.1 Element-to-thread mapping, block & ghost cell

private local memory. Each block has shared memory visible to all threads in the block and all threads can access the same global memory, which has a high latency compared to shared memory and constant memory.

Open Computing Language (OpenCL) released last year is the first truly open and royalty-free programming standard for general-purpose computations on heterogeneous systems. Though it has special framework of runtime, logically its programming model, i.e. the basic idea of solving problems and the memory hierarchy is very similar to CUDA. They both use domain decomposition strategy of blocks and threads defined by different names. Due to the above reason, in this paper we mainly describe how to solve Cavity flow and aerofoil simulations using CUDA. The corresponding strategy for OpenCL can be easily derived.

To run both CUDA and OpenCL, the code, program involves running code on two different platforms concurrently: a host part with one of more CPUs to control the program flow, and one or more GPU devices for intensive computing task. When calling a kernel function, the runtime system takes care of any complexity introduced by the parameters needed to get from the host to the device.

27.3.2 GPU Solution for CFD Using Finite-Difference

As described in the second section, we used finite-difference method to deal with both cavity flow and aerofoil RAE 2822 problems. To solve NS Eqs. (27.1), (27.1)–(27.5) by (27.6), traditional CPU needs to calculate the problem domain stencils one by one. Two-dimension needs a two layer loop, which three-dimension needs a three-layer loop. For simplicity, we cite NS equation with 5-point stencil as an example to describe how CFD solver is implemented on GPU. The procedure can be illustrated in the four steps below:

1. Decomposing domain and Mapping thread. For structured grid applications like the two above tests, a natural way is to split the domain into smaller parts which can fit into the thread block.Thread computes the updated variables of one element at time t + 1 using the information at time t within each of the smaller parts in a straightforward way. For example, the classical aerofoil domain is 369*65. As we can use 16*16 threads inside a thread block, there are $369/16+1 = 24$ blocks in the x axis, and $65/16 + 1 = 5$ blocks in the y axis. All the blocks except the last one on each axis are responsible for computing each stencil's information in the problem domain. The last one block on each axis only uses part of threads for computing with the rest padding with trivial information. Note that currently the maximum number of achievable threads per block is 512 on Tesla architecture regardless of the dimension, we have to carefully set the number of threads in the block in order to achieve efficiency.
2. Memory Access manipulation. The algorithm dictates the complexity of the stencil. Consider a straightforward 2D 5-point stencil (like central difference, upwind can also implemented in this way) as shown in Fig. 27.2. To update the element

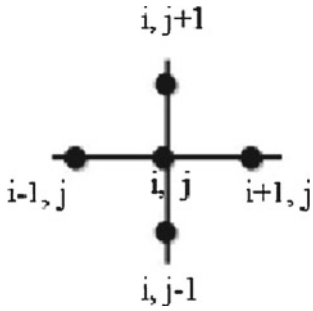


Fig. 27.2 5-point stencil

(i, j), thread (i, j) needs to read its four neighbors. That means that data of each point will be loaded at least four times from global memory to GPU register. It procedure appeared to be too expensive. In GPU memory hierarchy, shared memory and register is much faster than global memory. So an efficient way is to copy data within one block from DRAM into shared memory, which serves as cache where threads read directly. The use of shared memory can reduce global memory access redundancy and maximize bandwidth.

3. Shared Memory Padding. Because threads on the block boundary has only three neighbors in the same thread block, Shared memory has to be padded with ghost cells before computing.
4. Execution. Each thread is identified with a unique ID in the whole grid, and the kernel executed across these parallel threads in the gird. A synchronization instruction is usually required after loading data form DRAM into Shared Memory. Host functions are in charge of controlling the programming flow, indentifying when to compute inviscous and viscous terms, when to advance a time step. The kernel function running on the GPU is responsible for computing the finite-difference algorithm.

27.3.3 Optimization of Single GPU Codes

GPU codes must be optimized after it was developed. In these cases, we optimized our code based on the considerations below:

1. Detecting hotspots. Use timer to time each function on the kernel. Try to optimize the most time-consuming functions first if possible according to other optimization principles. Several tools can be employed in this step, like Intel VTune and clock timer programmed by our own.
2. Reduce divergence. GPU runs in SIMT mode. The stream multiprocessor's SIMT multi-threaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. 32 consecutive threads are grouped into a warp following the same instructions. To minimize divergence in one warp,

size of the thread block is better to be arranged as a multiple of 32. However, when it comes to add number domain like 369*65 in the simulation of aerofoil, we arrange extra threads to cover the entire domain, say a grid of 384*80 threads with16*16 threads per block (consequently, the total number of blocks is 120 in this example).

3. Minimize non-coalescing memory by using texture memory. When half warp-size consecutive threads access global memory, it coalesced into a single transaction if the addresses fall in the same block and meet alignment criteria. Coalescing memory greatly improves bandwidth and boost performance significantly over separate requests. However, when accessing ghost cell it is very hard to achieve memory coalescing because the address does not meet alignment requirements. For simple stencil like the second order central finite difference scheme, this penalty is minor because there is only one layer of ghost cell in a side. But for high order of accuracy numerical methods with complex stencil, such as 6th order central scheme in the aerofoil simulation where computing one point needs 6 neighbors resulting a high ghost and inner cell ratio of 12:16. This will lead to high memory non-coalescing. In this case, we use texture when it is related to sophisticated stencil differencing. Texture is a read-only space of DRAM and cached, thus can minimize the penalty of ghost cell.
4. Minimize data transfer between GPU and CPU across PCI-Express bus. Because bandwidth of PCI-E is merely 8GB/s which lag far behind bandwidth between GPU' DRAM and processor. Therefore if possible, try full implementation of GPU to ensure data transfer across GPU and CPU only occurs at the start and the end.

27.3.4 Acceleration of CFD on Multi-GPUs

Currently GPU has limit physical memory space. For example, the wildly used Tesla C1060 has 4 gigabyte of physical memory. All the memory is used for program running and storing data. To small CFD problem such as cavity flow, a single GPU can deal with at most 5120*5120 grid size of domain. As the problem becomes more complicated with increasing number of variables, the 4 GB memory will be not enough to use. The same problem occurs if we intent to enlarge our grid solution. To deal with it, multi-GPUs strategy must be developed.

From hardware aspect, in most circumstance, GPU cluster are built in this way: every computing node is equipped with one or more GPU cards, all the GPU on a single node are scheduled by the central CPU and all computing nodes are connected by network. To maximize the computing power of this system, we employed a MPI + CUDA strategy to solve CFD problem in the following 3steps:

1. Allocating GPUs to processes. The pseudo code is shown in Appendix A. The programmer and MPI environment must take responsibility for deciding how many processes are invoked on each computing node. The number of processes

on each node should be equal to the number of GPUs on this node. Otherwise, at least on process cannot manipulate GPU computing, thus an error should be raised. In real case, software of resource management and job scheduling system can be employed (In our experiment, Torque and Maui are used). To assign each GPU to process, we first sort the processes with MPI hostnames and then spilt them into different sub-common world according to hostname, so that processes on the same node must belong to the same sub-common world. After that, we assigned the GPUs on the same node to each process on this specific node so that each GPU can be labeled with order.

2. Decomposing process domain and GPU domain. It looks the same as we discussed in the first step when only single GPU is used. But here, the entire domain is first divide into process-computing domain, and each process domain is then divided into GPU-computing domain. As each process is in charge of one GPU card, the total number of GPU cards used in program decides the split number. In this step, as we know that GPU only outperforms CPU where there is large bulk of data to be preceded, so we must control the process number to let GPU fully used to achieve high efficiency.
3. Reducing communication between CPU to CPU and GPU to CPU. As in Fig. 27.2 when we deal with a central difference, the central point needs to read its four neighbors. In single GPU solution, all the data are on the GPU memory and we won't need any communication. But in Multi-GPU solution, the neighbors needed by the elements on the boundary of each process are computed on another process. Thus we must copy data from GPU to CPU and communicate between surrounding process and copy back the newly received data to GPU for further computing. To reduce communication cost, only the elements needed should be copy out and do communication. An alternative way to reduce to cost is to do asynchronous computing by first compute the elements to be communicated and then compute the other elements while taking communication at the same time. But to do this, we must carefully design data structure to match GPU memory friendly.

27.4 Experiment Results

To test our algorithm of solving CFD using GPUs, we implemented GPU code for both cavity flow and aerofoil RAE 2822 simulations. The detail description of the problems we solved and numerical methods used are listed in Table 27.1.

We ran cavity flow problem on both CUDA and OpenCL GPU platform. From hardware aspect, CPU platforms are Intel Xeon E5450 at 3.00 Hz on Lenovo 7000 and AMD Operton 2220 at 2.8 GHz with Intel Fortran and C compiler; the GPU platforms are Tesla C1060 with NVCC complier and ATI 9270.

Figure 27.3 shows the streamlines for the cavity flow (test 1), and Fig. 27.4 shows the plot of the pressure coefficients on the aerofoil (test 2). It shows that both CPU's

Table 27.1 Descriptions of numerical test

Descriptions	Equations	Numerical methods	Grid solution
2D cavity flow	Incompressible N-S	3rd order quick method; laminar	512*512-5120*5120
2D flow over a RAE 2822 aerofoil	Compressible N-S	5th order upwind scheme for inviscousterm; 6th order central scheme for viscous term; 3rd Runge-Kutta; B-L model; 1st for boundary 2d for sub-boundary	369*65-7380*1300

This table shows the numerical methods and equations used in the experiments

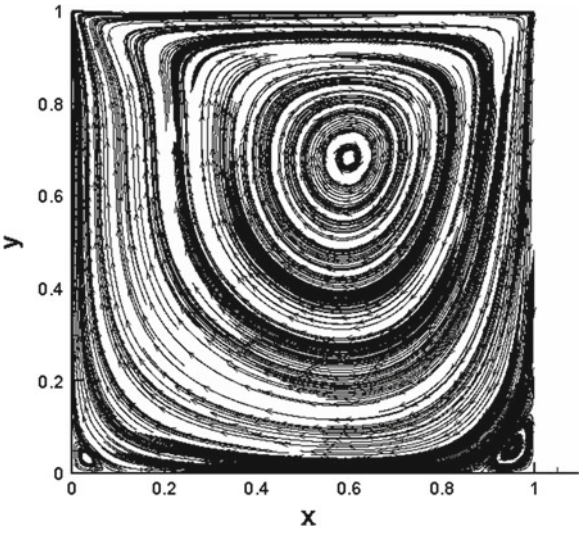


Fig. 27.3 Streamlines for the cavity flow with Re = 100

result and GPU's agreed well with the experimental data, though there is still a little difference between CPU's result and GPU's one.

From the speedup aspect, the time of program for cavity flow is shown in Table 27.2. According to Amdahl's law, the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. When the grid size is small, a lot of time is spent on the sequential part and CPU-GPU data transformation thus the speedup is not that significant. As the size increases, the program is limited to the heavy burden of computing. GPU is so good at computing, and accelerated the program for 30–60 times. We also

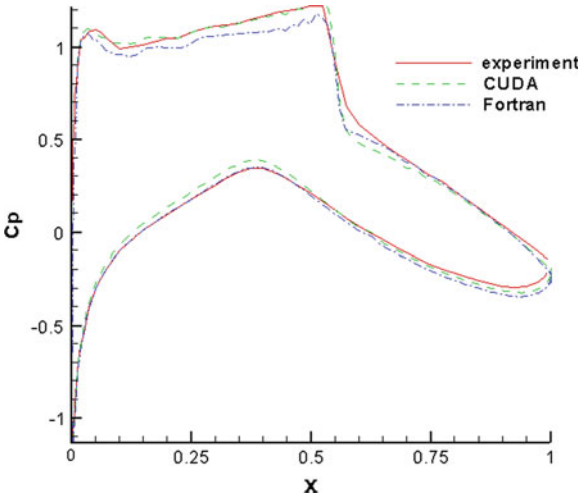


Fig. 27.4 Plot of the pressure coefficients on the aerofoil

Table 27.2 OpenCL simulation speedup for cavity flow problem

Grid solution	Serial	Openmp(4 cores)	OpenCL(Nvidia)	OpenCL(ATI)	CUDA
512 * 512	13.9844	9.406	2.975	7.588	1.34
1024 * 1024	70.83	52.2614	9.213	24.785	4.81
2048 * 2048	281.4307	217.9449	32.651	91.246	17.54
4096 * 4096	3823.8038	1515.912	124.742	XX	66.91
5120 * 5120	6843.9784	2378.9607	196.625	XX	108.3499

This table shows the time in second of cavity flow simulation on both CUDA and OpenCL platform, with OpenCL ran both on Nvidia card and ATI card. The simulation stopped at time step = 1000. we see that GPU speedup increases as the grid solution scale increases

witness a 2–3 times faster of CUDA code compared with OpenCL code, since the latter is a specific GPU device and the former is a general GPU API.

Similar result can be drawn in the aerofoil case. To test GPU performance on aerofoil RAE 2822, we both developed single-GPU and multi-GPU solutions. From hardware aspect, we totally used 30 GPU computing nodes with each node equipped with 2 Tesla C1060 4 GB memory. As we see in Table 27.3, when the grid solution is small, CPU outperforms GPU. This certainly makes sense since the scale is so small that the time of GPU solution was dragged down by large fraction of non-computing functions. As the grid size became larger, GPU gradually ran faster than CPUs and reached about 29x speedup. In our experiment, the fact that our 60 cores GPU can not beat 6cores may due to the heavy amount of swapping data among processors that costs a lot of time.

Table 27.3 MPI + GPU simulation speedup for aerofoil problem

Grid solution	CPU(6 cores)	GPU(6 cores)	Speedup	CPU(60 cores)	GPU(60 cores)	Speedup
369*65	9.22	30.92	0.30	2.82	19.44	0.14
1845*325	381.64	59.52	6.41	24.64	40.77	0.60
3690*650	1723.37	110.66	15.57	136.19	55.11	2.47
5535*975	3739.71	196.07	19.07	354.82	88.37	4.02
7380*1300	6568.57	225.38	29.14	643.69	114.66	5.6

This table shows the time in second of simulation of aerofoil running on CPU and GPU platforms at time step = 1000. We see as the the grid solution increases, GPU outperforms the CPU from 0.30 to 29x speedup

27.5 Conclusions

In this paper, we further tested GPU performance for solving CFD problems. For single GPU solution, by decompose the problem and optimize the code in a GPU friendly way. We see both CUDA and OpenCL are able to accelerate program in high speedup though OpenCL ran a little slower than CUDA. For multi-GPUs solution, we focused on how to reduce the communication between CPU to CPU and CPU to GPU. The experiments have demonstrated that a substantial performance gain is achieved by using graphic hardware devices.

Future works focus on two aspects. Firstly, as we see in multi-GPUs solution, performances are lower when GPU devices are more used. It partly due to the small grid size, partly because communication increases as processor number increases. How to minimize communication cost is one essential problem we need to solve. Secondly, performance in term of double precision is less pronounced on present architecture. Recently, the next generation Fermi released by NVIDIA designed with much higher double precision performance and cache around SM. We hope to observe a leap improvement of existing codes of double precision on this new platform.

Appendix A

In this appendix, the code shows how GPU devices are assigned to different MPI processors. To implement domain decomposition strategy, GPU devices must be assigned in continuous numbers in x and y axis so that we can dispatch tasks according to their position.

```
void AssignGPUtoProcessors()
{
//step1: broadcast hostname to every MPI processors
  MPI_Bcast(&(hostnames[i]),MPI_MAX_PROCESSOR_NAME);

//step2: sort the hostname on each nodes
  qsort(hostnames, np_size);

//step3: split the processors with same hostname into
sub MPI common world, so that GPUs can be assigned to
the processors with continuous order
  for (n=0; n<np_size-1; n++){
    if(strcmp(hostname, hostnames[n]) == 0) break;
    if(strcmp(hostnames[n],hostnames[n+1])) clr++;
  }
  MPI_Comm_split(MPI_COMM_WORLD, clr,0, &nodeComm);
  MPI_Comm_rank(nodeComm, &myrank);

//step 4: get the number of GPUs device,available on
each node, store them in devloc,
      for (dev = 0; dev < deviceCount; ++dev)   {
            cudaGetDeviceProperties(&deviceProp, dev);
            if(deviceProp.minor>1 && device-
Prop.minor<9999) {
                  devloc[slot]=dev;
                  slot++;
            };
      }

//step 5:Assign GPU device to MPI processes
  cudaSetDevice(devloc[myrank]);
}
```

References

Anderson WK, Bonhaus DL (2009) Airfoil design on unstructured grids for turbulent flows. AIAA J 37(2):185–191

Baldwin BS, Lomax H (1978) Thin layer approximation and algebraic model for separated turbulent Flows. AIAA 78–257

Brandvik T, Pullan G (2007) Acceleration of a two-dimensional euler flow solver using commodity graphics hardware. J Proc Inst Mech Eng Part C: J Mech Eng Sci 221:1745–1748

Jespersen DC (2009) Acceleration of a CFD code with a GPU. NAS Technical report NAS-09-003.

Toro EF (1999) Riemann solvers and numerical methods for fluid dynamics-a practical introduction. Springer, Berlin

http://www.grc.nasa.gov/www./wind/valid/raetaf/raetaf.html

Sanders J, Kandrot E (2011) CUDA by example: an introduction to general purpose GPU programming. Addison-Wesley, Boston

Khronos openCL working group (2008) The openCL specication, V1.0

NVIDIA Corporation (2007) Compute unified device architecture programming guide. http://www.nvidia.com

Tingxing D, Xinliang L, Sen L (2010) Acceleration of computational fluid dynamic codes on GPU. In: 8th Asian computational fluid dynamics conference

Chapter 28
Efficient Rendering of Order Independent Transparency on the GPUs

Fang Liu

Abstract Order independent transparency refers to the problem of rendering scenes using alpha blending equations, which requires the primitives in the scenes to be rendered according to their distances to the viewer. It is one of the key rendering effects in many graphics applications, thus has been extensively studied. Various techniques and systems have been proposed to render order independent transparency. These techniques can be classified into three categories based on their underlying methodologies: the primitive level methods, the fragment level methods, and the screen-door methods.This article provides a comprehensive review of these existing methods, with an emphasis on the advanced techniques that have been recently developed. The background of order independent transparency is introduced at the beginning of this review. Key contributions, advantages as well as limitations of each method are summarized in three following parts, respectively. The first part focuses on the primitive level methods, which tries to solve the problem by pre-sorting primitives, then rendering them from back to front using alpha blending equations. The second part reviews the fragment level methods, which performs fragment sorting and blending on the fly, or captures all the fragments per pixel and sort fragments in post-processing before blending. The performance and memory consumption analysis is presented as a comparison between these methods. The third part introduces another catalog of methods which approximates the rendering results using screen-door techniques, which is quite practical while rendering scenes with high depth complexities, such as grass and hair. Finally, a simple conclusion is given at the end of the review, indicating the direction of the future development of order independent transparency.

F. Liu (✉)
The SuperComputing Center, Computer Network Information Center,
Chinese Academy of Sciences, Beijing, China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 437
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_28,

28.1 Background

Efficient rendering of order independent transparency have long been a challenging problem in 3D computer graphics. Correct rendering of semi-transparent geometry requires multiple layers of scenes to be sorted and blended in depth order. The alpha-blending function provided by current graphics hardware can only blend the fragments on the same pixel location in submission order, rather than the depth order. However, the fragments on the GPUs are processed in parallel in an undefined submission order which is always not consistent with the depth order, thus will generate obvious artifacts during rendering.

There are two ways to perform correct alpha blending in depth order. The most intuitive way is to composite fragments from back to front. For a certain pixel location, suppose the backmost fragment with color (C_1, A_1) is blended over the second backmost fragment with color (C_2, A_2) over a background color C_0, generating the following intermediate color C_2':

$$\begin{aligned} C_1' &= A_1{}^*C_1 + (1 - A_1)^*C_0 \\ C_2' &= A_2{}^*C_2 + (1 - A_2)^*C_1' \\ &= A_2{}^*C_2 + (1 - A_2)^*(A_1{}^*C_1 + (1 - A_1)^*C_0) \end{aligned} \tag{28.1}$$

From above equations we can summarize that fragments can be blended in front-to back order with the following over-blending equation:

$$C_{dst}' = A_{src}{}^*C_{src} + (1 - A_{src})^*C_{dst} \tag{28.2}$$

where ($C_{src,}$ A_{src}) represents the incoming source fragment color while C_{dst} represents current color stored on the destination color buffer, and C_{dst}' is the result color after blending. This process will iterate for each incoming fragment from the back to front until all the fragments on that pixel location have been captured, and the final result will be stored on the color buffer.

The above method does not use the alpha channel of the destination color buffer. When peeling and compositing fragments on the same pixel location from the front to back, the alpha blending equations in (Eq. 28.1) must be modified to the following equations:

$$\begin{aligned} C_1' &= A_1{}^*C_1 + (1 - A_1)^*C_0 \\ C_2' &= A_2{}^*C_2 + (1 - A_2)^*C_1' \\ &= A_2{}^*C_2 + (1 - A_2)^*(A_1{}^*C_1 + (1 - A_1)^*C_0) \\ &= A_2{}^*C_2 + (1 - A_2)^*A_1{}^*C_1 + (1 - A_2)^*(1 - A_1)^*C_0 \end{aligned} \tag{28.3}$$

It can be easily deduced from above equations that fragments can be blended from the front to back according to the following under-blending equations:

$$\begin{aligned} C'_{dst} &= C_{dst} + A_{dst}{}^{*}(A_{src}{}^{*}C_{src}) \\ A'_{dst} &= (1 - A_{src})^{*}A_{dst} \end{aligned} \tag{28.4}$$

where A_{dst} is initialized to 1.0. The blending process can be performed on the GPUs by pre-multiplying C_{src} by A_{src} in a fragment shader using separate blending equations for RGB and alpha channels. In the end, the blended fragments are composited with the opaque background C_{bg} with the final blending equation:

$$C'_{dst} = C_{dst} + A_{dst}{}^{*}C_{bg} \tag{28.5}$$

Both above methods are quite easy to understand but hard to be directly implemented on the GPUs. The main reason is that in both methods fragments are processed in depth order, either from front to back or from back to front. It implicitly requires depth-sorted traversal of the input scenes and the triangles should not be intersecting with each other.

However, most objects in the scenes are complex and have different transformation hierarchies. Thus most natural order of scene traversal rarely satisfies the above requirements since triangles of the scenes are rasterized on the GPUs in parallel and the generated fragments are always processed in undefined order. Therefore it becomes quite intractable to guarantee the fragments on the same pixel location to arrive in the same order as the pre-sorted primitives. In addition, new primitives might be generated in geometry shaders on advanced GPUs and inserted to the graphics pipeline, making it more difficult to keep the depth order of fragments per pixel along with the pre-sorted primitives while rasterization.

Therefore there comes out three catalogs of methods trying to sort the geometry on the primitive level or on the fragment level. The primitive level methods presorts the primitives in a back-to-front order. Then the furthest layers are drawn first and each successive transparent layer will be blended over the previous one. It is intuitive that pre-sorting of primitives can help to assure the fragments per pixel to be blended consequently in depth order. In contract, the fragment level methods directly rasterize the scenes without pre-sorting and try to capture and sort all the fragments per pixel. The fragments will be blended on the fly or in an additional post-processing pass. The third catalog of screen-door methods approximate order independent transparency using a set of pixels that are fully on or off. The sub-pixel variation of the technique encodes alpha information by implicitly turning samples on or off. All these methods have been well developed in recent decades and many great accomplishments have been achieved. We will give details to these methods in detail in the following sections, respectively.

28.2 Primitive Level Methods

The most intuitive catalog of solutions to order-independent transparency works at the primitive level. The main idea is to pre-sort the geometries according to the distances from there centers to the viewers. Then the geometries are rendered in a back-to-front order. There have developed in the past several methods as detailed below.

28.2.1 The Painter's Algorithm

The most typical primitive level method is the painter's algorithm, which is also known as a priority fill. It was first used to solve the visibility problem in 3D computer graphics: when projecting geometries from 3D object space onto a 2D screen plane, it is necessary to decide which parts are visible, and which are hidden while painting. The painter's algorithm derives ideas from the way a painter employed to create a painting. It starts by sorting geometries in the scenes from back to front as viewed from the camera. The furthest surfaces are rendered first on the screen, and each successive layer will paint over the previous invisible parts. The strategy can solve the visibility problem at the cost of having painted redundant areas of distant objects. The algorithm can be easily extended to solve the problem of order independent transparency. The primitives can be pre-sorted and blended over each other according to their alpha values from furthest to nearest to assure the correct depth order.

The algorithm is simple to implement but might fail in some special cases, including cyclic overlap or piercing primitives. In the former case of cyclic overlap, the primitives overlap each other. The resulting cycle makes it impossible to determine which primitive is above the others. Numerous methods in the field of computational geometry have been proposed to solve this problem. One of the existing solutions is to cut the primitives to assure an unique depth order, as first proposed in Newell's algorithm (Newell and Simon 1972). The algorithm performs a series of tests between two primitives in order of increasing computational difficulty. If all the results of the tests are false, one of the primitives must be split into halves along the line of intersection with the other one. The algorithm will continue to be performed on other pairs of primitives until all of them have passed the tests. In the latter case of piercing primitives when one primitive intersects another, the problem may also be resolved by cutting the offending polygons in the same way as used for cyclic overlap.

Besides above limitations, the painter's algorithm might be inefficient in basic implementations. It forces the graphics pipeline to repaint a pixel when it is coved by more than one primitive, even if the primitive is occluded by others in the finished scene. This means that, for scenes with rich details, the computer hardware has to waste much time on these unnecessary computations. Thus a reverse painter's algorithm can be used to alleviate this problem. The objects nearest to the viewer are painted first and the latter paint must never be applied those areas that are already

painted. This strategy can be very efficient for rendering pipeline since it helps to avoid unnecessary calculations for shading on invisible pixels on the finished scene, such as lighting or texturing.

Though the reverse algorithm suffers from the same problems as the standard version, it gave a hint to the early development of Z-buffer techniques. The Z-buffer techniques can be considered as an extension to the painter's algorithm. It resolves the visibility problem on the fragment level using a fixed-precision Z-buffer (i.e., depth-buffer) implemented in hardware. A typical usage of depth buffer starts by clearing the buffer to the maximum depth value. During rasterization, only the front-most layer are kept on the frame buffer since it will pass all the depth tests. The technique helps to resolve the visibility problem and avoids unnecessary update to the frame buffer. Even more, the unnecessary computing for pixel shading and texturing of the invisible fragments could also be avoided using the recently developed early-Z culling technique. However, the algorithm cannot be directly used for order independent transparency unless rendering the scenes multiple times using a state-of-art method namely depth peeling, as will be detailed in the next section.

28.2.2 The Vis-sort Algorithm

The vis-sort algorithm (Govindaraju et al. 2005) accelerates the painter's algorithm in complex and dynamic environments by exploiting the temporal coherence between successive frames. It assumes that there are no cyclic overlap or piercing primitives among the input objects, and there is an unique sorting order among them for any viewport.

The algorithm starts by rearranging the primitives in depth order from a starting viewpoint. In many interactive applications, the depth values as well as the relative order of the primitives often do not vary much due to the spatial and temporal coherent motion of the viewer or primitives. If the primitives are ordered in one frame, the next frame often needs to perform sorting on a nearly-sorted sequences of primitives. Therefore, the authors described a sorting algorithm to exploit the coherency between successive frames. It can be well mapped onto the GPUs and exhibits linear-time performance in environments with high coherence.

The 1D version of the sorting algorithm starts with an unsorted sequence of 1D elements and an empty set for the output sequence. During each iteration, the algorithm operates on the unsorted input list and computes a sequence of consecutive quantiles beginning with the minimum of the input. These sorted values are appended to the ordered sequence in the output set. The process iterates on the remaining data values until the input data set is left empty. 3D sorting is a variation of the 1D version though a bit more intricate with two overlap constraints.

The sorting algorithm can be well mapped to the commodity graphics hardware. The comparisons of data elements are performed using the depth test functionality of the GPUs and the resulting minimum value is kept in the depth buffer. Our sorting algorithm has a best-case run-time complexity of O(N) but a worst-case run-time

complexity of $O(N^2)$ for reverse sequences. Fortunately, the performance could be improved by using a multi-stage ordering approach to avoid the worst case scenarios with low coherence. Meanwhile, the cyclic primitives could be detected and the visibility order could be resolved using this approach.

The vis-sort algorithm can generate transparency effects interactively for complex models in dynamic environments. During each frame, the algorithm computes an initial back-to-front order of the primitives in the scene and uses the sorted sequence in the previous frame as an input for the current frame. In most interactive applications, it operates on nearly-sorted lists and exhibits linear-time performance. However, the algorithm has one major limitation: it makes assumptions that the input objects are non-overlapping and there is a sorting order among them. The cyclic primitives can only be detected but not solved unless exploiting other existing solutions.

28.2.3 Coherent Layer Peeling

The coherent layer peeling algorithm (Carr et al. 2008) also exploits the property of sorted coherency between successive frames for transparent high-depth-complexity scenes. But it successfully solves the overlapping problem using a multi-pass approach. In each iteration, the algorithm efficiently detects and renders a subsequence of fragments that have been in order for a given pixel using the depth test combined with the stencil test on the GPUs.

For each frame, the algorithm starts by a simple bucket sort of triangle centroids to pre-sort the primitives in linear-time performance. Suppose the opaque surfaces are separated from the scenes and rendered first, generating a depth map called *opaque_depth*. The first surface following the last peeled layer s_c is denoted as s_n with a depth value $D(s_n)$, respectively. The iteration to render the transparency effects can be broken down into three steps with 2 passes each as described below.

In the first step, the algorithm render the scene twice to find the nearest out-of-order surface *z_min_prev* within the depth range $(D(s_n);\infty]$ which comes before s_n in the list but has a greater depth value than s_n. It works as a threshold to discard all the further fragments which are definitely out-of-order, thus can narrow down the fragments in order into a loose range within [D(sn); *z_min_prev*). The second step also uses two passes to find the nearest out-of-order surface *z_min_post* in the depth range $[D(s_n)$; min(*z_min_prev*; *opaque_depth*) that follows s_n in sequence. Similarly, fragments further than this threshold are also out-of-order and should be discarded. The two steps above are composed to a tight bound on the peel range given by: $(D(s_n)$; min(*z_min_prev*; *z_min_post*; *opaque_depth*)]. The surfaces that lie in the above interval are guaranteed to be in a coherent depth order, thus can be composed into the final image in step 3 by a standard composing pass. Note the surface at the upper bound of the peel range may be out of order, thus can be individually captured using a z equals test in an additional pass. This three-step process repeats until all the layers have been peeled which can be detected using occlusion queries on the GPUs.

The 3-steps process described above can generate correct result only in a single iteration with 6 passes in the best case when the data is perfectly sorted. It gains great speedup especially for moderate-sized scenes with quite high depth complexities. However, for the worst case of reserve order, the algorithm can peel away two layers per iteration, making it less efficient. In addition, the sort of triangle centroids per frame will degrade the performance of the algorithm especially for large irregular scenes. Though the authors have proposed an ideal system to reduce the number of geometry passes per iteration down to two, it needs some modifications to current graphics hardware, leaving an open problem in the future.

28.3 Fragment Level Methods

The second catalog of solutions to order independent transparency works at the fragment level. These methods always need no pre-sorting of the primitives as the primitive level methods. Fragments on the same pixel location are sorted and composed on the fly during rasterizaiton. In another way, the fragments can be captured and stored on the GPU memory, and post-sorted before blending in a full-screen pass. The details of each method will be described in the following subsections as below.

28.3.1 Depth Peeling

The depth peeling algorithm is a simple and robust fragment-level technique that makes order independent transparency possible on commodity graphics hardware. It was first described by Abraham Mammen using virtual pixel maps (Mammen 1989) and by Paul Diefenbach using a dual depth buffer (Deefenbach 1996). The technique was first implemented on the graphics hardware by Cass Everitt (Everitt et al. 2001), which was an multi-pass extension of aforementioned Z-buffer technique. The standard depth test function on the GPUs peels off the nearest fragment at each pixel without imposing any ordering restrictions. However, for multi-layered scenes, the second nearest or further surfaces will be discarded and cannot be captured in a single pass. The depth peeling technique solve this problem by exploiting the depth test function on the GPUs in a really straightforward way. It can peel away the front-most n layers in depth order with n passes over a scene.

The algorithm starts by rendering the scene in a normal procedure in the first pass, and the nearest surface is captured and stored in the depth buffer. Meanwhile, the color attribute of the nearest layer would be simultaneously captured and stored in a color buffer. In the second pass, the depth layer peeled away in the previous pass is bound to the fragment shader as a texture parameter. For each incoming fragment, the fragment shader would sample the input texture and fetch the depth value of the nearest layer stored on that pixel location. The value would be used as a reference,

and all the fragments with a depth value less than or equal to the reference would be discarded at the beginning of the shader. Those fragments that pass this test would be shaded and the output fragment will generate a second depth buffer if it passes the following depth test. Meanwhile, the color attribute of this layer could be blended into the previously captured layer stored in the color buffer on the fly using the front-to-back blending Eq. 28.2. Similarly, the second nearest depth layer will be used in the third pass as a reference to delaminate both the first and second nearest surfaces that have been captured in the previous two passes. The pattern described above is fairly simple but needs dual depth tests per fragment each pass. The first test is performed in the fragment shader while the second one is supported by the built-in depth test function unit on the GPUs. This procedure continues and the previously nearest layer captured in pass i will be used as the reference texture in pass $i + 1$ and the fragments that pass the dual depth tests will be blended into the current color buffer. The algorithm will terminate until all the fragments have been captured which can be detected by occlusion query on current GPUs. The final result can be generated by blending with the background layer in an additional composing pass. The reverse depth peeling could also be performed using the back-to-front blending Eq. 28.4 as described in (Thibieroz 2007).

The technique presented above is a straightforward and robust way to render scenes that contain transparent objects. It makes good use of graphics hardware and has no requirement for the scene to be rendered in pre-sorted depth order. The algorithm works fairly well for simple scenes, but for large scenes with high complexity, multiple rasterizations of the geometry will become a performance bottleneck.

Theoretical Extensions to depth peeling include the A-buffer (Carpenter 1984), the R-buffer (Wittenbrink 2001), and the Z^3 algorithm (Jouppi and Chang 1999). Unfortunately, all of these extensions need hardware modifications. The F-Buffer (Mark and Proudfoot 2001) maintains FIFO buffers to hold fragments in between rendering passes and need additional post-sort passes, with implementations available on ATI's graphics hardware (Houston et al. 2005).

28.3.2 Dual Depth Peeling

A practical extension to the depth peeling technique namely dual depth peeling was recently proposed by Bavoil and Myers (2008). It succeeds to capture both the nearest and furthest layers each pass, making the theoretical acceleration to classical depth peeling as a factor of two. The depth peeling process is interleaved with alpha blending of the two layers, thus can avoid additional memory consumption to store the captured layers.

Conceptually, dual depth peeling peels away two layers at a time each pass and blend them in both back-to-front and front-to back directions using a min–max depth buffer with an RG32F color texture. The min–max depth test is performed to capture both the nearest and furthest layers using the 32-bit floating point MAX/MIN blending functions supported by GeForce 8. Take the MAX blending as an example,

the min–max depth buffer is cleared to $(-1, -1)$, and the MAX blending is turned on in substitution for the read-modify-write part of the min–max depth test. Each incoming fragment will update the buffer with a *float2*(depth, −depth) in the shader. After the first pass, the maximum depth layer will be captured and stored in the first channel of the min–max depth buffer as it always win the comparison in MAX blending. Meanwhile, the minimum one can be resorted by negating the value inside the second channel. Afterwards the min–max depth values will be used as two references in the following geometry pass to capture the second nearest and furthest layers in a similar way as depth peeling. At the same time, the peeled fragments from the front and back layers in the previous pass are blended into two separate color buffers using the under-blending and over-blending equations, respectively. The process continues until the front and back advancing fronts meet, and the middle fragment may be processed as either front or back one. The final result is composed of both images in two directions using under-blending. This method can generate correct results.

For a scenes with N layers, the depth peeling algorithm needs N passes, while dual depth peeling reduces the number to $N/2 + 1$, which means a two-time speedup at most for geometry bound applications. The authors also describe a weight-average method to approximate order independent transparency with a single geometry pass. The method simplifies the alpha blending equation using a per-pixel weighted average over the pixel to replace the RGBA color. It can render plausible transparent effect in a single geometry pass only for scenes with nearly uniform distributed colors, i.e. the color of each pixel is close to the final average.

28.3.3 K-Buffer and its Multi-Pass Extension

K-buffer (Bavoil et al. 2007; Liu et al. 2006) presents another extension to the Z-buffer technique. The algorithm exploits multiple render target (MRT) buffers as a read-modify-write pool of k entries, namely k-buffer, that makes it possible to efficiently capture up to k fragments in a single pass. The concept of k-buffer was first proposed by Callahan et al. (2005). It is defined as a fixed size buffer of fragments per pixel that is maintained in GPU memory and is effective for sorting and compositing fragments in the special case of direct volume rendering on current GPUs. The definition can be generalized to be a pool of fragments per pixel that allows fragments to be compared, ordered, blended, and discarded in a streaming manner. The order independent transparency can take this advantage to be rendered in a single geometry pass.

The k-buffer is firstly cleared to set all the elements in the k-buffer to zero or one. For each incoming fragment, the algorithm takes three steps in fragment shader for dynamic sorting. The first step loads all the elements on the corresponding pixel location in the k-buffer into a temporary local array. Then the current fragment is inserted into the local array by insertion sort algorithm in ascending or descending order of their depth values. Finally, the local array containing the current fragment is written back to the k-buffer on the GPU memory. The color attribute of the fragments

will be captured and sorted together with the depth value for transparency. The final result can be composed by either over-blending or under-blending equations in post-processing.

The theoretical acceleration rate of the algorithm is k times in comparison to depth peeling. However, multiple fragments might simultaneously update the same pixel location on k-buffer due to the parallel execution of the fragment program, thus results in read-modify-write (RMW) hazards on the k-buffer. The problem could be improved by dynamic pre-sorting of geometries, or triangle batches as proposed by the authors (Bavoil et al. 2007). Both modifications dramatically slow down the algorithm since the former one needs pre-sorting of triangles each frame and the latter one will break the pipeline. Liu et al. (2009) tries to alleviate the RMW hazards by multi-pass approach. The k-buffer is first cleared to the maximum depth values. Each fragment will load the elements to a local array and choose the first entry with the maximum depth value of the array for storage. The modifications guarantees at least one fragment be captured when collision happens. The discarded fragments will be captured in the following passes using the previous captured fragments as references. The strategy solve the RMW hazards at the cost of multi-passes. As the depth complexity of the scenes increases, the RMW hazards will also increase and the performance of the algorithm will degrade rapidly, which quite limits the utility of the algorithm. In addition, the benefit from the development of GPU is limited since the RMW hazards will also increase as the parallel degree of GPU increases, thus leads to more extra passes.

28.3.4 Stencil-Routed A-Buffer

The stencil routed a-buffer (Myers and Bavoil 2007) algorithm uses multisampling anti-aliasing (MSAA) buffers to store a vector of elements per pixel. This is accomplished by rendering the scene in aliased mode, in which all sub-pixel fragments would receive the same value computed for that pixel. The sub-pixel stencil routing (Purcell et al. 2003) is first introduced to route the fragments into sub-pixels of the MSAA buffer at the sample level according to the D3D10 specification.

At the beginning, the stencil test is set to D3D10_COMPARISON_EQUAL and the stencil operation is D3D10 STENCIL_OP_DECR_SAT. The 8 sub-pixels of the stencil buffer is first initialized to 2–9 respectively as a stencil mask for a single pixel, and the reference value in the stencil test is set to 2. For a certain pixel, the first incoming fragment will only update one unique sub-pixels whose current stencil values equals to the reference value, i.e. the fragment is routed to the sub-pixel with a stencil value equals to 2. The stencil values of all sub-pixels will decrease after the update, and the next fragment will update a new sub-pixel whose stencil value is just decreased to 2. The process continues until an overflow occurs after 8 fragments have hit the same pixel. It can be detected in an additional fullscreen pass using occlusion query and stencil test to check whether any values of the last sub-pixel has been decrease to 0. The above procedure can capture up to eight layers in one geometry

pass. For complex scenes with more layers, the multi-pass can also be exploited to capture all the layers. In post-processing for transparency, fragments will be sorted in the fragment shader using the Load intrinsic to read each individual samples from the MSAA buffer. Afterwards a bitonic sort can be used in the shader before all the fragments are finally blended into the final image.

The algorithm completely avoids the RMW hazards that occurs in the k-buffer algorithm. And it gains 8 times speedup in comparison to depth peeling, which is limited by the maximum number of MSAA samples in current GPUs. It can capture at most 254 fragments per pixel due to the low resolution of the 8-bit stencil buffer. Another limitation is that the implementation requires hardware multisample antialiasing (MSAA) to be disabled while rendering. For scenes with high depth complexity, the non-linear bitonic sort in post-processing might be expensive, and it has to capture all the layers before post-sorting, that might cause memory exhaustion before post-sorting.

28.3.5 Bucket Depth Peeling

The bucket depth peeling algorithm (Liu et al. 2009) exploits the same feature of MAX/MIN blending on Geforce 8 as dual depth peeling. It succeeds to capture up to 32 fragments in a single geometry pass using bucket sort on MRT buffers. The potential fragment collisions can be alleviated by an additional pass or more. The algorithm gains significant speed improvement over depth peeling and the result is plausible for uniform distributed scenes. An adaptive scheme with a preprocessing pass is also introduced to substantially reduce the collisions.

The basic idea of the algorithm is to perform a bucket sort on the fragment level. The MRT buffers are exploited as a bucket array for each pixel location, and each bucket is capable of holding only one fragment. All the buckets can be concurrently updated using MAX/MIN blending, which assures atomic update of a specific channel of MRT while keeping the others unchanged. Take MAX blending as example, a bounding volume or a coarse visual hull is first rendered to get the approximate depth range of each pixel location. While rendering the scene, the depth range is divided into consecutive subintervals uniformly and all the MRT buffers are cleared. In the geometry pass, a linear bucket sort is performed on the pixel shader, i.e., fragments within each subinterval will be routed into the corresponding buckets. Meanwhile, the rest buckets will be updated by the default zeros in order to keep unchanged. In a following fullscreen pass, the bucket array per pixel can be sequentially accessed to get the sorted fragments for post-processing. When multiple fragments are routed to the same bucket, i.e. a collision happens, the bucket will only hold the maximum one and the rest will be discarded, and artifacts will arise. A multi-pass extension can alleviate the problem as depth peeling or dual depth peeling. However, only one more pass might be supported most of time due to the limited memory on current graphics hardware.

Inspired by the image histogram equalization, the authors have also developed a two-pass approach to further reduce the collisions, namely adaptive bucket depth peeling. After the initial pass to render the bounding volume of the scenes, the depth range is divided into 1024 uniform subintervals. A depth histogram is defined as an auxiliary array with each entry indicating the number of fragments falling into the corresponding depth subinterval, thus is a probability distribution of the geometry. It is constructed using MRT buffers with pixel format GL_RGBA32UI_EXT. In the first geometry pass, an incoming fragment within the kth subinterval will set the kth bit of the depth histogram to 1 using the OpenGL's 32-bit logic operation GR_OR. After the first pass, each bit of the histogram will indicate the presence of fragments in that subinterval or not. The depth histogram is equalized in a following fullscreen pass to ensure the one-to-one correspondence between fragments and subintervals. Therefore in the second geometry pass, there will be only one fragment routed into each bucket, and collisions will be reduced substantially. We perform the bucket sort in the second geometry pass. The equalized histogram is passed to the pixel shader as input textures and the upper bounds in the input equalized histogram will redivide the depth range into non-uniform subintervals with almost one-to-one correspondence between fragments and subintervals. As a result, there will be only one fragment falling into each bucket on most occasions, thus collisions can be reduced substantially.

The bucket depth peeling algorithm has a linear-time performance and produces a great speedup to the classic depth peeling especially for large scenes. It turns out to be a good approximation when performance is preferred to the visual quality. The adaptive extension of the algorithm substantially improves the rendering results but needs a non-trivial memory overhead for depth histogram, especially for applications with high screen resolutions. Meanwhile, the performance will be halved due to the additional pass for histogram equalization.

28.3.6 The FreePipe System for Transparency

Order-independent transparency is difficult to be solved in a single geometry pass because it requires programmability on the blending stage. The FreePipe system was first proposed as a programmable parallel rendering architecture (Liu et al. 2010) implemented in CUDA (NVIDIA 2008). It can run entirely on current graphics hardware while having performance comparable with the traditional graphics pipeline. Within the framework, two efficient schemes namely multi-depth test scheme and fixed-size A-buffer scheme are proposed to render transparency in a single geometry pass by modifying the blending stage. Both schemes have achieved significant speedups compared to the aforementioned methods that are based on traditional graphics pipelines.

The multi-depth test scheme performs fragment sorting on the GPU using a novel sorting algorithm for unpredictable data via CUDA *atomicMin* operation. The algorithm begins by allocating a fixed size integer array per pixel as storage in global

memory and initialized to the maximum integer. Each incoming fragment will loop through the pixel array at corresponding pixel locations and store the depth value to the first empty entry via *atomicMin* operation. For a non-empty entry, if its value is smaller than the current depth value, the entry will be left unchanged and the fragment will continue to test the next entry. Otherwise, *atomicMin* operation assures the current depth value to be stored into that entry and the returned source value will be used to test the next entry. This operation loops until the fragment is stored into the first empty entry. Otherwise, it will be discarded at the end of the array. In post-processing, the sorted depth array per pixel can be passed to a following CUDA kernel for post-processing. The results can be rendered into a pixel buffer object and directly drawn onto the screen using texture mapping. The strategy can store the front most N layers tightly thus is memory efficient. It performs fragment sorting in global memory in a time complexity of $O(N^2)$, therefore will suffer from high memory latency for complex scenes.

The fixed-size A-buffer scheme allocates a *struct* array per pixel in global memory. A fragment counter is set up for each pixel location and initialized to 0. Each incoming fragment will first increase the corresponding counter by 1 using the *atomicInc* operation. In other words, the kth incoming fragment of a pixel will atomically increase the counter to $k + 1$ and return the source value k. Then the depth value and the additional attributes of the kth fragment can be stored into the kth entry of the array in submission order without read-modify-write hazards. The captured fragments will be sent to a post-sort CUDA kernel, with each thread handling only one pixel. All fragments per pixel will be loaded into an array allocated in the registers of that thread and sorted by insert sort or bitonic sort. The results then can be directly accessed in correct depth order for deferred shading in post-processing. The fragment sorting is performed at the register level without memory latency, that is more efficient for scenes with high depth complexity. However, the memory might exhaust to capture all the layers for those applications that only need several front most layers, such as shadow mapping (Bavoil et al. 2008).

The rasterization strategy used in FreePipe system is only efficient when input triangles occupy a small number of pixels. When the scene contains large triangles, each thread will loop through a large number of covered pixels, thus will result in heavy serialization. In addition, when scene geometry contains fewer triangles, the performance will degrade rapidly. Thirdly, both schemes for transparency will result in serious performance bottleneck and memory consumption while handling geometries with too many layers, such as hair, grass and transparent particle effects. Despite above limitations, the system gives a promising direction to solve transparency in a single pass in the future.

28.3.7 Linked List for Transparency on DX11

The per-pixel linked list on DX11 (Hensley 2010) is a break-through for order independent transparency on the GPUs. All the fragments can be captured in a single

geometry pass provided enough memory for storage. The algorithm requires two buffers to construct the fragment linked list. Both buffers are allocated as UAV buffers (RWStructuredBuffer) on the GPUs on DX11. The first kind the screen sized head pointer buffer, which stores the start addresses of the last node written for every pixel location, and is initialized to end-of-list value. The other kind is the node buffer, i.e., the linked list, which contains data and links for all visible transparent fragments. It must be large enough to accommodate all the transparent fragments.

The algorithm starts by rendering the opaque scene objects as usual. When the transparent scene objects are rendered, each shader thread will retrieve and increase a global counter value to allocate a new node in the linked lists. The operation calls the standard *IncrementCounter*() memory function to atomically update the counter so that each incoming fragment could be allocated an unique offset in the node buffer. The offset works as the pointer to the current last fragment on that pixel location. It is exchanged into the corresponding head pointer buffer using a standard *InterlockedExchange*() function, and the previous offset in the head buffer is returned. Then a new node will be constructed by the depth and color attribute of the fragment, together with the returned pointer as a link to the second last fragment. Finally, the node will be added into the node buffer at a linear address computed by the global counter value. In post-processing, the algorithm draws a screen quad to sort and resolve the associated per-pixel linked list, and composite it with background for the final output.

The algorithm is kind of similar to the A-buffer scheme in FreePipe system [] but with a much tighter memory layout. The implementation on DX11 shows great speedup to depth peeling algorithm. However, the fragment sorting in post-processing still requires a fixed-size array per pixel, which imposes the same "max overdraw" limitation as in the A-buffer scheme.

28.3.8 Evaluations

The performance and memory consumption of the above fragment level methods can be evaluated by on a set of typical models. All the scenes are rendered in a 512×512 viewport and the frame rates are taken on a commodity PC of Intel Duo Core 2.4 GHz with 3 GB memory, and NVIDIA Geforce 280 GTX (compute capability 1.3) with Windows XP Pro SP2 32-bit and CUDA version 2.1(181.20). In the experiment, the k-buffer algorithm and its multi-pass extention (Liu)[], bucket depth peeling (BDP), the multi-depth test scheme(MDTS) and the A-buffer(ABS) scheme in FreePipe system, the stencil-routed A-buffer (SRAB) and dual depth peeling (DDP) are chosen in comparison to the classical depth peeling(DP).

For a scene with depth complexity N, suppose the vertex rasterization needs time T_v for each geometry pass, and the fragment shader in these methods costs T_f, the accelerate rate can be computed by:

$$K = \frac{N^* T_v}{T_v + T_f} \tag{28.6}$$

According to the above Eq. 28.5, the classical depth peeling needs N passes to peel off all the N layers. While in contrast, the MDTS and ABS methods in FreePipe system only need one pass thus can gain an accelerate rate of about N times compared to depth peeling, especially for large scenes (See Table 28.1). The k-buffer and bucket depth peeling have similar speedup but the former suffers from severe read-modify-write hazards while the latter is subject to fragment collisions. The multi-pass extension of k-buffer assures correct results at the cost of extra geometry passes, thus the performance is substantially degraded. For simple scenes where the extra cost T_f on fragment shader cannot be ignored, such as bunny model, the aforementioned methods may not perform as efficiently as for large scenes, even slower than DP. The SRAB method captures 8 layers each pass on current graphics hardware, so the speedup is up to N/8 times in comparison to depth peeling. The dual depth peeling peels two layers simultaneously, and the theoretical speedup is two times.

For memory analysis in above experiment, only the memory for fragment storage are taken into consideration for simplicity. Suppose the depth information (32 FP) of each layer needs P MB memory. As for the scenes in the experiment, P equals to 1 MB at a 512×512 viewpoint. Each layer also needs P MB for color attribute (RGBA8). The depth peeling algorithm only needs 4*P MB memory for the nearest layer in the previous pass and the current pass to store both the color and depth information. The dual depth peeling doubles the performance of depth peeling at the cost of double memory consumption, which is 8*P MB for both the nearest and furthest layers. The memory consumption of above two methods is independent on the depth complexity thus is efficient. Suppose the maximum depth complexity is N, which is not more than 16 for above scenes. Both k-buffer and its extension needs 2*N*P MB memory for fragment storage, which is acceptable on current graphics hardware. The stencil-routed A-buffer needs the same amount as k-buffer since all the layers have to be captured before composition. In FreePipe system, the memory consumption of the render target array by multi-depth test scheme is 2*N*P MB. A-buffer scheme only requires P MB extra memory for the fragment counter per pixel. The bucket depth peeling algorithm needs as much memory as possible to alleviate collisions, thus will consume 32*P MB memory on current graphics hardware due to the massive usage of MRTs. Note the depth information is no longer stored in bucket depth peeling since the fragments are naturally in order when stored into the buckets. In

Table 28.1 Comparison of frame rates (fps) by different methods

Model	Tri. No.	K-buffer (fps)	Liu (fps)	BDP (fps)	MDTS (fps)	FABS (fps)	SRAB (fps)	DDP (fps)	DP (fps)	N
Bunny	70 K	875	183	645	455	616	334	160	141	8
Armadillo	349 K	294	46	258	339	422	168	41	20	13
Dragon	871 K	262	60	256	244	316	106	40	29	10
Buddha	1.0 M	248	42	253	208	275	89	34	24	10
Lion	1.3 M	163	15	183	152	209	42	21	11	16
Lucy	2.0 M	149	20	160	152	98	49	23	18	12
Neptune	4.0 M	46	6	47	87	105	21	8	5	8

summary, the fragment level methods except depth peeling are a trade-off between memory consumption and performance. For high definition resolutions and irregular distributed scenes, these methods may have a huge memory consumption in render targets that is a big memory overhead.

28.4 Screen-Door Transparency

A third catalog the algorithm tries to approximate order independent transparency using a screen-door technique (Fuchs et al. 1985). The algorithm replaces transparent surfaces with a set of pixels that are fully on or off. The sub-pixel variation of the algorithm has been implemented in hardware as the alpha-to-coverage technique, which implicitly encodes alpha information by turning samples on or off. However, the fixed dithering pattern makes the layers occluding each other rather than blending in depth order, leading to serious noise and visual artifacts. Because alpha-to-coverage uses a fixed dithering pattern, multiple layers occlude each other instead of blending correctly, leading to visual artifacts. The object-level analogy to screen-door transparency drops triangles can be dropped from a mesh in proportion to transparency (Sen et al. 2003), but requires dynamic tessellation for scalable objects in the scene. In order to keep rendering times in proportion to screen size, Callahan et al. (2005) drop some polygons and increase the opacity of the remaining ones.

Stochastic transparency (Eenderton et al. 2010) is a simple extension to screen-door transparency using a randomized sub-pixel stipple pattern to reduce noise substantially. The basic algorithm with alpha correction starts by rendering the opaque background into a multi-sampled z-buffer. In the first geometry pass, the total alpha is rendered into a separate multi-sampled buffer. And the transparent primitives are rendered into a separate multi-sampled buffer in a following second pass. The samples are culled by the opaque z-buffer, and discarded by stochastic alpha-to-coverage. In the final compositing pass, the opaque background is first dimmed by one minus total alpha at each sample. Then the filtered transparent color is read and corrected to the filtered total alpha, and blended over the filtered, dimmed background. The alpha correction used above can eliminate noise to zero in areas where all layers have an unique color, but might do more harm than good for scenes with varying colors.

The authors also proposed a depth-based stochastic transparency which is more accurate in complex regions. The visibility of each ray is encoded using the ratio of samples that have passed the multi-sampled depth test. The algorithm also starts by rendering the opaque background and gets the total alpha in the first geometry pass. In the second pass, the transparent primitives are rendered, discarding samples by stochastic alpha-to-coverage, but only the depth values of the remained samples are stored. Then the transparent primitives are rendered once again into a separate multi-sampled color buffer in additive blending mode, only accumulating those samples that have smaller depth values than the corresponding one in the combined z-buffer

from the previous step. Finally, the filtered accumulated sum is corrected and blended over the filtered, dimmed background.

Stochastic transparency provides an excellent solution to order independent transparency. The algorithm avoids read-modify-write loops which is common in traditional z-buffer based methods, making it quite practical for modern GPU hardware.

Conclusions and Future Work

There are also other methods for order-independent transparency on other platforms such the ray tracing and REYES systems. They always have more photorealistic rendering results, such as described in Wexler et al. (2005) and Zhou et al. (2009). Though these methods can generate high quality rendering results, the performance is still not satisfactory for real time applications. Recently, Intel has announced Larrabee (Seiler et al. 2008), a highly parallel model that makes the rendering pipeline completely programmable. The blending stage would be open to programmers thus can be optimized for transparency.

Though efficient rendering of order independent transparency is still a changeling problem on graphics applications. We are sure there are other new possible solutions that have not been thought of. More efficient sorting algorithms might be designed and implemented on the GPUs for primitive sorting before blending. The linked-list method on DX11 gives the most advanced fragment level method which has a very tight memory layout. However, it will consume a huge amount of memory for scenes with high depth complexity. The future trend is to develop the screen-door techniques to give a better approximation and reasonable correction for complex scenes.

Hopefully this review article can provide a good basis for those wanting to develop new unforeseen methods for transparency or new desirable applications using these methods. As the fast evolvement of the graphics hardware, it is promising that order independent transparency could be well solved on the GPUs in the near future.

References

Bavoil L, Myers K (2008) Order independent transparency with dual depth peeling. Technical report, NVIDIA Corporation

Bavoil L, Callahan SP, Lefohn A, Comba JLD, Silva CT (2007) Multi-fragment effects on the GPU using the k-buffer. In: Gooch B, Sloan PP (eds) ACM SIGGRAPH symposium on interactive 3D graphics games, ACM, New York, pp 97–104

Bavoil L, Callahan SP, Silva CT (2008) Robust soft shadow mapping with back projection depth peeling. J Graphics GPU Game Tools 13(1):19–30

Callahan SP, Ikits M, Comba JLD, Silva CT (2005) Hardware assisted visibility sorting for unstructured volume rendering. IEEE Trans Visual Comput Graphics 11(3):285–295

Carpenter L (1984) The A-buffer, an antialiased hidden surface method. In: Proceedings of the 11th annual conference on computer graphics interactive techniques, pp 103–108

Carr N, Mech R, Miller G (2008) Coherent layer peeling for transparent high-depth-complexity scenes. In: Molnar S, Doggett M (eds) The 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware. ACM, New York, pp 33–40

Catmall EE (1974) A subdivision algorithm for computer display of curved surfaces. Ph.D. thesis, University of Utah

Deefenbach P (1996) Pipeline rendering: interaction realism through hardware-based multipassrendering. Ph.D. Dissertation, Department of Computer Science, University of Pennsylvania

Eenderton E, Sintorn E, Shirley P, Lubeke D (2010) Stochastic transparency. In: Proceedings of the symposium on interactive 3D graphics games, 2010. ACM, New York, pp 157–164

Fuchs H, Goldfeather J, Hultquist J, Spach S, Austin J, Brooks JRF, Eyles J, Poulton J (1985) Fast spheres, shadows, textures, transparencies, image enhancements in pixel-planes. In: Proceedings of SIGGRAPH. ACM, New York, pp 111–120

Govindaraju NK, Henson M, Lin MC, Manocha D (2005) Interactive visibility ordering transparency computations among geometric primitives in complex environments. In: Proceedings of the 2005 symposium on interactive 3D graphics games, pp 49–56

Grun H, Thibieroz N (2010) OIT indirect illumination using DX11 linked lists. In: Proceedings of game developers conference, 2010

Jouppi NP, Chang CF (1999) z3: an economical hardware technique for high-quality antialiasing. Transparency, pp 85–93

Liu BQ, Wei LY, Xu YQ (2006) Multi-layer depth peeling via fragment sort. Technical report, Microsoft Research Asia

Liu F, Huang MC, Liu XH, Wu EH (2009) Efficient depth peeling via bucket sort. In: Stephen N, McAllister D, Pharr Intel M, Wald I (eds) The 1st high performance graphics conference. ACM, New York, pp 51–57

Liu F, Huang MC, Liu XH, Wu EH (2010) Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In: Varshney A, Wyman C (eds) ACM SIGGRAPH symposium on interactive 3D graphics and games. ACM, New York, pp 75–82

Mammen A (1989) Transparency antialiasing algorithms implemented with the virtual pixel maps technique. IEEE Comput Graph Appl 9(4):43–55

Mark WR, Proudfoot K (2001) The F-buffer: a rasterization-order fifo buffer for multi-pass rendering. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware, pp 57–64

Morein S (2001) ATI Radeon-HyperZ technology. In: Proceedings of the hot 3D workshop on graphics hardware

Myers K, Bavoil L (2007) Stencil routed A-buffer. In: ACM SIGGRAPH, 2007 technical sketch program. ACM, New York

Newell A, Simon HA (1972) Human problem solving. Prentice Hall, Englewood Cliffs

NVIDIA (2008) NVIDIA CUDA: Compute unified device architecture. NVIDIA Corporation

Purcell TJ, Donner C, Cammarano M, Jensen HW, Anrahan P (2003) Photon mapping on programmable graphics hardware

Seiler L, Carmean D, Sprangle E, Forsyth T, Abrash M, Dubey P, Junkins S, Lake A, Sugerman J, Cavin R, Espasa R, Grochowski E, Juan T, Hanrahan P (2008) Larrabee: a many-core x86 architecture for visual computing. ACM Trans Graph 27(3):18:1–18:15

Sen O, Chemudugunta C, Gopi M (2003) Silhouette opaque transparency rendering. In: Sixth IASTED international conference on computer graphics Imaging, pp 153–158

Thibieroz N (2007) Robust order-independent transparency via reverse depth peeling in DirectX® 10. In: Engel W (ed) ShaderX6—advanced rendering techniques. A.K. Peters, Natick

Wexler D, Gritz L, Enderton E, Rice J (2005) GPU-accelerated high-quality hidden surface removal. In: Harris M, Luebke D (eds) ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware. ACM, New York, pp 7–14

Wittenbrink CM (2001) R-buffer: a pointerless a buffer hardware architecture. In: Pfister H (ed) ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware. ACM, New York, pp 73–80

Yang J, Hensley J, Grun H, Thibieroz N (2010) Real-time concurrent linked list construction on the GPU. In: Eurographics symposium on rendering, vol 29(4) pp 1297–1304

Zhou K, Hou Q, Ren Z, Gong MM, Sun X, GUO BN (2009) Renderants: interactive REYES rendering on GPUs. In: Inakage M (ed) ACM transactions on graphics, vol 28, issue 5. ACM, New York, pp 155:11–55:11

Chapter 29
Performance Evaluation of Fast Fourier Transform Application on Heterogeneous Platforms

Xiaojun Li, Yang Gao, Xinyu Ma and Ying Liu

Abstract Heterogeneous platforms, integrating SMPs, clusters, GPUs, FPGAs, etc. are becoming the most popular architectures of supercomputers. Achieving high performance on CPUs or GPUs requires careful consideration of their different architectures, which challenges the capability and skills of programmers. In order to overcome the portability problem, OpenCL, a free cross-platform programming standard, is proposed by Khronos Compute Working Group. However, the performance of OpenCL-based programs has not been thoroughly studied yet. Therefore, in this paper, we first design *OpenFFT-Bench*, an FFT application with OpenCL-based FFT and OpenGL-based real-time spectrum visualization as the benchmark. We evaluate its performance on four OpenCL programming platforms including NVIDIA CUDA, ATI Stream (GPU), ATI Stream (CPU), and Intel OpenCL. Characteristics of OpenFFT-Bench are investigated with varied FFT size. Experimental results show that OpenCL and OpenGL-based applications can not only run on multiple heterogeneous platforms, but also achieve relatively high performance on GPU-based platforms.

X. Li · Y. Gao
School of Information Science and Engineering, Graduate University of Chinese Academy of Sciences, Beijing, China
e-mail: lixj09@mails.gucas.ac.cn

Y. Gao
e-mail: gaoyang0309@mails.gucas.ac.cn

Y. Li (✉)
University of Chinese Academy of Sciences, Beijing, China
e-mail: yingliu@ucas.ac.cn

X. Ma
Agilent Technologies, Inc., Beijing, China
e-mail: xin-yu_ma@agilent.com

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_29,

29.1 Introduction

High-performance computing is currently undergoing a period of rapid revolution. Multi-core processors were established as the most popular CPU architecture, and since then, we are witnessing a rapid transition to many-core systems. Many-core Graphics Processing Units (GPUs) offer a tremendous computing power that is an order of magnitude larger than the fastest multi-core CPUs. They are becoming the most attractive platforms for high performance computing due to their high performance/cost ratios.

In 2006, NVIDIA released the Compute Unified Device Architecture (CUDA) for its GPUs, which specifies extensions of C programming language for writing program kernels directly targeting GPUs. CUDA enhances the viability of GPUs as a general-purpose computing platform by providing a straightforward programming model and language. In the same year, AMD released its ATI Stream architecture. Since then, many promising applications have been implemented on GPUs in almost all domains and applications.

With the prosperity of GPUs, nowadays, the architectures of the world's top supercomputers are becoming heterogeneous platforms, usually integrating SMPs, clusters, GPUs, FPGAs. Heterogeneous platform is obviously the developing trend in the next decades. However, achieving high performance on CPUs or GPUs requires careful consideration of their architectures and rigorous optimization efforts. A more serious problem is that the programming specification on CPUs and GPUs are so different that programs running on one platform may not be compatible with another platform. Therefore, programmers may have to re-write the code to meet its targeting platform, which will incur tremendous amount of challenging, tedious and boring work. This serous weakness will eventually hinder the development of high performance computing on heterogeneous platforms.

In order to overcome the above portability problem, Open computing language (OpenCL) is proposed by Khronos Compute Working Group. OpenCL is designed for parallel computing cross heterogeneous platforms consisting of CPUs, GPUs or other processors. It supplies API library to manage the platform and defines a language based on C99 for writing kernels executed on OpenCL processors. It supports both task-based and data-based parallelism and specifies OpenCL devices' memory hierarchy. People can write high performance algorithms running on all OpenCL platforms without knowing their original programming language model, such as CUDA, TBB, Brook++, etc. However, performance study of OpenCL-based programs has not seen yet. A benchmark that measures the performance of different OpenCL programming platforms is in demand.

Fast Fourier Transform (FFT) is one of the most widely used algorithms for engineering and scientific computation. FFTs are of great significance to a broad variety of applications, including signal processing, electronic measurement, image analysis, biological sequence analysis, etc. Therefore, in this paper, we try to investigate an FFT application with real-time visualization and its characteristics on a serial of

heterogeneous high performance computing platforms. Specifically, in this paper we make the following contributions:

(1) We first propose *OpenFFT-Bench*, an FFT application which includes OpenCL-based FFT and OpenGL-based spectrum visualization;
(2) We measure the characteristics of OpenFFT-Bench on multiple programming platforms;
(3) We analyze the architectural properties of different platforms and highlight important bottlenecks.

Floating-point computation capability (GFLOPS), overall execution time and frame rate are measured in our experiments. OpenCL + OpenGL make sure OpenFFT-Bench run across multiple platforms without any effort of re-writing the code. The experimental results show that the performance of OpenFFT-Bench on GPUs is always higher than that on CPUs, although CUFFT outperforms OpenCL-based FFT on GPUs due to its customized optimization on specific hardware. NVIDIA CUDA OpenCL platform outperforms other platforms in all aspects. It indicates that OpenCL and OpenGL cross-platform specifications not only contribute to portability, but also provide relatively high performance.

The rest of this paper is organized as follows. In Sect. 29.2, we briefly introduce some benchmarks on parallel computing platforms. Section 29.3 describes the background knowledge of this paper. In Sect. 29.4, we present the details of our application. In Sect. 29.5, we present the detailed setups of our evaluation platforms and report our experimental results. In Sect. 29.6, we conclude our work.

29.2 Related Work

The NAS Parallel Benchmarks (NPB) (Baliley et al. 1991) are a set of benchmarks targeting highly parallel supercomputers. They are developed and maintained by the NASA Advanced Supercomputing (NAS) Division. The LINPACK Benchmarks (Dongarra 1988a,b) are a measure of the floating point computing power of a system introduced by Jack Dongarra. The perfect club benchmarks (Berry et al. 1989) are also well-known performance evaluation suits on supercomputer. They represent codes in a number of engineering applications and computational research.

Shane proposed optimization principles and evaluated a multithreaded GPU (Geforce 8800) using CUDA (Ryoo et al. 2008). Barrachina evaluated and tuned Level 3 CUBLAS on CUDA (Barrachina et al. 2008). Vasily presented the performance of dense linear algebra on multiple GPUs including GeForece GTX280, GeForce 9800 GTX, GeForce 8800 GTX, and GeForce 8600 GTS (Volkov and Demmel 2008).

Up to now, as far as to our knowledge, there is no such benchmark suite evaluating the performance of various OpenCL programming platforms. Therefore, in this paper, we propose a 1D FFT application, OpenFFT-Bench, to evaluate the performance of multiple programming platforms.

29.3 Background Knowledge

29.3.1 Parallel Computing

Parallel computing is a form of computation where many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. The parallel computer architecture has rapidly changed since the past decades.

29.3.1.1 Conventional Parallel Computing Architecture

Conventional parallel computer architectures include SMP, MPP, clusters, etc. A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory via a bus and I/O devices. A massively parallel processor (MPP) is a single computer with many networked processors. A cluster is a group of loosely coupled computers that work together closely, so that they can be regarded as a single computer in some respects. Clusters consist of multiple standalone machines connected by a network. There are still other conventional parallel computers, such as distributed memory multiprocessor (DSM), parallel vector processors (PVP), etc.

29.3.1.2 General Purpose Computing Graphic Processing Units

Tesla C1060 is a typical NVIDIA GPU, which consists of 30 stream multiprocessors. Each stream multiprocessor contains 8 stream processors, 2 special function units, 32 KB shared memory, control unit registers, etc., Memory hierarchy consists of shared memory, texture memory, constant memory and device memory, where all memory is physically in the device memory except that shared memory is on die.

ATI GPU consists of several stream processors (5 in ATI Radeon 3870). Eachstream processor consists of groups of SIMD engines (4 in ATI Radeon 3870). Each SIMD engine has numerous thread processors (16 in Radeon 3870) for executing kernels, each on a different stream. A thread processor is arranged as a five-way very long instruction word processor. Up to 5 scalar operations can be co-issued in VLIW instruction. Stream cores can execute integer or single-precision floating operations. One of the five stream cores can handle transcendental operations.

29.3.1.3 OpenCL Standard

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving

software developers portable and efficient access to the power of these heterogeneous processing platforms (OpenCL, version 1.0). OpenCL supports both task parallelism and data parallelism. OpenCL standards include a subset of C99 language with extensions for parallelism, a configuration profile for handheld and embedded devices, a requirement on numerical number based on IEEE 754, solution to efficiently interoperate with OpenGL, OpenGL ES and other graphics APIs.

OpenCL platforms consist of a host connected to one or more OpenCL compute devices. Each compute device consists of several compute units and each compute unit has numerous processing elements. Tasks to be processed by OpenCL device are submitted to a command queue maintained by OpenCL.

The basic element of OpenCL parallelism is kernel function. Instances of kernel functions are executed concurrently on multiple processors on different part of a data stream. An instance is called a *work item* with a unique identifier. Work items are organized into *work groups*, providing a more coarse-gained parallelism. OpenCL specification defines OpenCL devices' memory hierarchy, including private memory, local memory, global memory, constant memory, as in Fig. 29.1. Private memory is owned by a unique work item. Local memory belongs to a work group, which can be used for communication and synchronization. Global memory can be read and written by work items in all work groups. Constant memory is a special region of global memory where data cannot be modified.

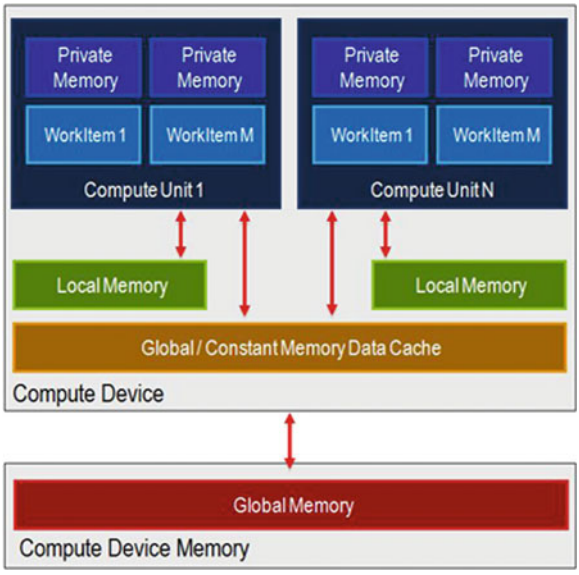


Fig. 29.1 OpenCL memory hierarchy

29.3.2 Overview of Fast Fourier Transform (FFT)

29.3.2.1 FFT

The forward Discrete Fourier Transform (DFT) of a complex sequence $x = x_0, x_1 \cdots x_{N-1}$ is an N-point complex sequence, $X = X_0, X_1, \cdots X_{N-1}$, where $X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N}$. A naïve implementation of DFT requires $O(N^2)$ operations and can be expensive. FFT algorithms compute the DFT in $O(N \log N)$ operations. Sorensen and Burrus compiled a database of over 3400 entries on efficient algorithms for FFT (Sorensen and Burrus 1995).

Cooley-Turkey FFT (Cooley and Tukey 1965) is a commonly used FFT algorithm. It expresses the discrete Fourier transform (DFT) of an arbitrary composite size $N = N_1 N_2$ in terms of smaller DFTs of sizes N_1 and N_2. Radix-2 is the simplest and most common form of it. The Pseudo Code of Cooley-Turkey FFT is in Fig. 29.2.

29.3.2.2 Parallel FFT on CPUs

Franz Franchetti et al. presented a parallel FFT program generator for shared memory parallel machines, such as SMPs and multi-core CPU systems (Franchetti et al. 2006). It offers perfect load-balance and avoids false sharing. Both Intel MKL libraries and Intel IPP libraries provide FFT applications, which are highly optimized on Intel hardware. Intel MKL FFT is for engineering and scientific applications and Intel IPP FFT is for media and communication applications. Only IA-32 and IA-64 CPU architectures are supported by Intel libraries. In addition, a great deal of work related to parallel FFT implementations has been done on IBM cell processors (Bader and Agarwal 2007; Williams et al. 2006; Frigo and Johnson 2007). Other FFT implementations on CPUs include Dmitruk et al. (2001), Li et al. (2003) using MPI, and Goedecker et al. (2003) using combined OpenMP and MPI, etc.

29.3.2.3 Parallel FFT on GPUs

Naga K. Govindaraju et al. presented novel algorithms for computing discrete Fourier transforms on CUDA GPUs (Govindaraju et al. 2008). With tricky skills and tuning the application to memory hierarchy, it achieves 2–4× speedup to CUFFT and 8–40× speedup to Intel MKL FFT on different FFT sizes. Cleomar Pereira da Silva et al. implemented a three-dimensional FFT using multiple GPUs (da Silva et al. 2010).

$X_{0,\ldots,N-1}$: Input array of length N, N is a power of 2

$Y_{0,\ldots,N-1}$: Output array of length N

S is the stride of the input array, $X+S$ denoting the array staring X_s

$Y_{0,\ldots,N-1} \leftarrow FFT2(X,N,S)$

if $N = 1$ then

$Y_0 \leftarrow X_0$

else

$Y_{0,\ldots,N/2-1} \leftarrow FFT2(X,N/2,2S)$

$Y_{N/2,\ldots,N-1} \leftarrow FFT2(X+S, N/2, 2s)$

for k = 1 to $N/2 - 1$

$t \leftarrow Y_k$

$Y_k \leftarrow t + \exp(-2\pi ik/N) Y_{k+N/2}$

$Y_{k+N/2} \leftarrow t - \exp(2\pi ik/N) Y_{k+N/2}$

end for for

end for if

Fig. 29.2 Pseudo Code of Cooley-Turkey FFT

29.3.3 Overview of OpenGL for High Performance Graphics

Visualization has ever-expanding applications in science, education, engineering (e.g. product visualization), interactive multimedia, medicine, etc. The supporting technique of visualization is computer graphics. However, in early days, the development of computer graphics is hindered due to the lack of graphics power. With the emergence of Graphics Processing Units (GPUs), computer graphics has been undergoing a revolutionary progress.

In addition to support general purpose computing, GPU is a specialized microprocessor that offloads and accelerates graphics rendering from the central processor. GPU implements a number of graphics primitive operations in a way that makes running them much faster than drawing directly to the screen with the host CPU. GPUs are mainly used for playing 3D games or high-end 3D rendering.

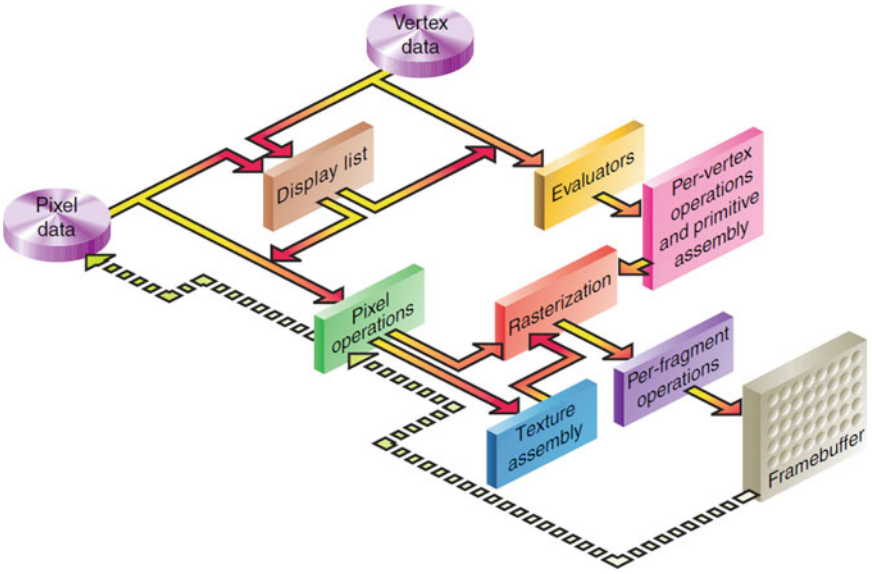


Fig. 29.3 Order of operations in OpenGL

OpenGL (Open Graphics Library) is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics. The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is successfully and widely used in CAD, virtual reality, scientific visualization, flight simulation, video games, etc.

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering is not a strict rule about how OpenGL is implemented, but it provides a reliable guide for predicting what OpenGL will do. Figure 29.3 shows how OpenGL processes data. Geometric data (vertices, lines, and polygons) follow the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer.

OpenGL maintains the vertex data in vertex buffer, which locates in GPU global memory. The vertex buffer is optimized to exchange data with host memory and interoperate with OpenCL, then, the vertex data is rasterized into pixels or fragments.

29.3.4 Benefits of OpenCL Plus OpenGL for Visualization

OpenCL standard supports OpenGL extensions which allow OpenCL applications to use OpenGL buffer, texture objects, rend buffer objects as OpenCL memory objects. Thus, when data is located in the global memory of GPU, it is not necessary to interrupt the host CPU when GPU transfers data. This mechanism contributes to achieve high performance when OpenCL applications run on GPUs, etc.

29.4 Application

In this paper, we propose OpenFFT-Bench. It is an FFT application, consisting of OpenCL-based FFT part and OpenGL-based real-time spectrum visualization part. Section 29.4.1 presents the details of the OpenCL-based FFT and Sect. 29.4.2 describes efficient visualization method we proposed.

29.4.1 Parallel FFT Using OpenCL

Little research has been done on OpenCL-based FFT. There are only two implementations available: Apple OpenCL FFT implementation (https://developer.apple.com/library/mac/#samplecode/OpenCL_FFT/Introduction/Intro.html) and bealto OpenCL FFT implementation (Bainville 2010). Due to the robustness of Apple OpenCL FFT implementation, we choose it as the FFT solution of OpenFFT-Bench.

In Apple OpenCL FFT implementation, for any N points, it decomposes N into the product of several factors. Factors are sorted such that the first one is the largest. It determines the size of local memory used by each work item. Products of the remaining factors determine the number of work items needed per group. For example, assuming $N = 1024$, it is decomposed into $1024 = 16 \times 16 \times 4$. It needs 64 work items per group and each work item needs $16 \times 2 \times$ size of (float) local memory. The first kernel performs 64 length-16 FFTs with 64 work times working in parallel, and then transposes the result in local memory. Then, it performs another 64 length-16 FFTs and transposes, followed by 256 length-4 FFTs (https://developer.apple.com/library/mac/#samplecode/OpenCL_FFT/Listings/fft_kernelstring_cpp.html).

29.4.2 Visualization

29.4.2.1 3D Display Mode

In our application, the screen is divided into three parts. The 3D view of 1D FFT dynamical spectrums are presented on the left part of the screen, where x-axis denotes

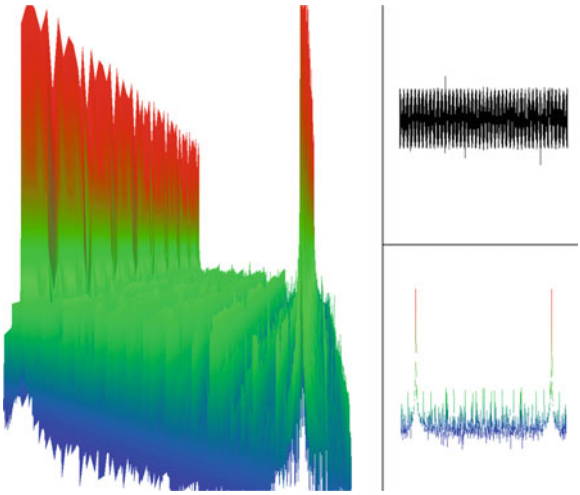


Fig. 29.4 Visualization of 1D-FFT spectrums

the order of the points in a single period, *y*-axis denotes the amplitude of every point in this period, and *z*-axis denotes time, that is, different periods. Linear interpolation of blue, green and red is used to shade along the positive direction of *y*-axis. The viewport at the top right corner of the screen presents the source signal data. The one at the lower right corner presents the 2D view of the given 1D FFT spectrum. A screenshot is shown in Fig. 29.4.

29.4.2.2 Vertex Shader for Color Shading on y Axis

Along the positive direction of *y*-axis, there is a gradual color change from blue to green, then to red. In order to achieve this feature, we only need to support vertex shader, because OpenGL shading pipeline provides linear interpolation between two vertices. The code for vertex shader is presented in Fig. 29.5. To be convenient, the max value on *y*-axis is fixed at 250.

29.4.3 OpenCL and OpenGL Interoperability

29.4.3.1 Shared Buffer for Computing and Visualization

A new mechanism is provided by OpenCL that a buffer, up to 768 KB, can be shared between OpenCL and OpenGL. In our application, by utilizing the shared buffer, 64×1024 points to be visualized (*x, y, z* coordinates) are stored at any given time. 1024 points are retrieved and plotted on the screen each time. Whenever a new period

```
#define MAX 250
void main(){
       int half = MAX / 2;
       float value = abs(gl_Vertex.y);
       if (value > half){
              value -= half;
              gl_FrontColor.r = value / half;
              gl_FrontColor.g = 1.0 - value / half;
              gl_FrontColor.b = 0.0;
       }
       else {
              gl_FrontColor.r = 0.0;
              gl_FrontColor.g = value / half;
              gl_FrontColor.b = 1.0 - value / half;
       }
       gl_FrontColor.a = 1.0;
       gl_BackColor = gl_FrontColor;
       gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fig. 29.5 Code for vertex shader for coloring

of points (1024 points) is generated by OpenCL FFT, it is pushed into the shared buffer and the oldest period in the buffer is deleted.

29.4.3.2 Points Reduction

Since the resolution of a typical screen is 1280 × 1024, there is no way to plot more than 1024 points even if FFT size exceeds 1024, such as 2048, 4096, or 32768. Therefore, we have to reduce the number of points to be plotted at a given time to 1024. When FFT size exceeds 1024, we divide the points into 1024 groups consecutively and take the sum of the amplitude of the points within a group as the value on *y*-axis.

29.5 Performance Evaluation

29.5.1 Evaluation Methodology

29.5.1.1 Experimental Data

In our experiments, a sine signal is generated as the source signal. Whenever a sine signal is generated, it is transformed by OpenCL FFT on the devices, and the

corresponding spectrum will be displayed on the screen in real time. The computation is expensive in complexity because N sine calculations are required to generate one sine signal.

29.5.1.2 Timing

Although OpenCL standards provide a timing tool, it can only be applied in limited environment. Here, we need a timing tool which can be used in all environments on all platforms. So, we choose the Windows timing tool, *QueryPerformanceCounter*, which retrieves the current value of high-resolution performance counter on CPU. The resolution is in microseconds.

29.5.1.3 Parameters

We will evaluate two parameters on each platform: *GFLOPS* and *frame rate*. GFLOPS represents billions of floating-point operations per second, that is, the computing capability of each platform under OpenCL framework. Frame rate is the number of frames or images that are projected or displayed per second on the screen, which reflects the efficiency of our visualization solution on each platform.

29.5.2 Hardware Platforms

We will evaluate the parameters on two computers, an Intel i7 920 CPU with a NVIDIA GPU card, referred as C1, an Intel i7 860 CPU with an ATI GPU card, referred as C2. The setups of C1 and C2 are listed in Table 29.1.

29.5.3 Benchmark Characteristics

In this section, we analyze several characteristics of our proposed benchmark. For each characteristic, we analyze how the results vary when we change the input data.

Table 29.1 Setups of our hardware platforms

Computer	CPU	CPU memory	Operating system	GPU	GPU SDK
C1	i7 920, 4 cores 2.67 GHz	6 GB	Windows 7 64-bit	NVIDIA Tesla C2050	CUDA SDK 3.2
C2	i7 860, 4 cores 2.8 GHz	4 GB	Windows 7 64-bit	ATI HD5870	Stream SDK V2.2

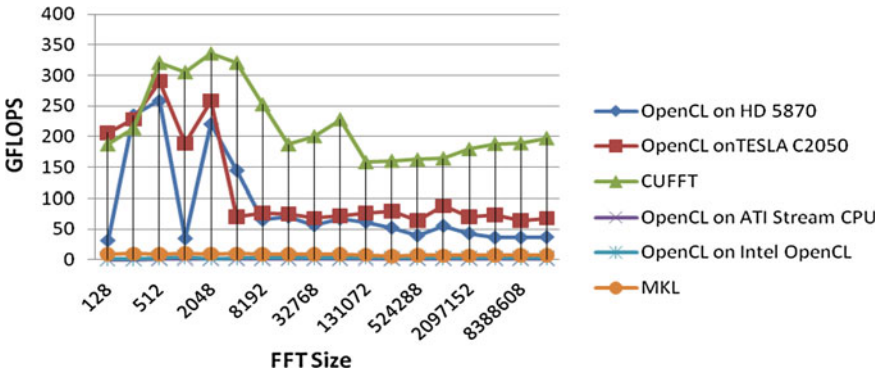


Fig. 29.6 OpenCL FFT floating-point computation capability (GFLOPS) on different programming platforms with varying FFT size

Our measures of interest include GFLOPS of the OpenCL FFT part, the overall execution time, and the frame rate of the visualization part.

To evaluate the performance of the OpenCL FFT part of OpenFFT-Bench, four programming platforms are used: (1) NVIDIA CUDA (2) ATI Stream (GPU) (3) ATI Stream (X86 CPU) (4) Intel OpenCL. Note: The executable code generated by programming platforms will be run on hardware platforms, C1 and C2.

29.5.3.1 Floating-Point Computation Capability (GFLOPS)

GFLOPS of the OpenCL FFT part of OpenFFT-Bench is measured as follows: First of all, 256 MB data is allocated into OpenCL device memory; secondly, Fast Fourier Transform is performed on the data, varying FFT size from 128 to 16777216. To calculate GFLOPS, formula $5^{*}N^{*}\log_2(N)/T$ is used where N denotes FFT size and T denotes the execution time.

Figure 29.6 presents GFLOPS of OpenCL FFT of OpenFFT-Bench. The corresponding programming platforms and hardware platforms are: platform (1) on NVIDIA Tesla C2050 GPU card, platform (2) on ATI HD 5870 GPU card, platform (3) on Intel i7 920 CPU, platform (4) on Intel i7 920 CPU. x denotes FFT size and y denotes GFLOPS. GFLOPS of CUFFT and MKL FFT are also measured.

From Fig. 29.6 we can see that NVIDIA Tesla C2050 outperforms ATI HD5870 in terms of GFLOPS, where the peak GFLOPS value, 280, is observed at FFT size 512.

It is also observed that GFLOPS of NVIDIA CUFFT is even higher than that of OpenCL FFT on NVIDIA Tesla C2050. This is not surprising, because in order to be a cross-platform programming model, OpenCL does not optimize the usage of some specific resources on specific hardwares, such as on-die shared memory, registers, etc., while CUFFT is highly optimized to NVIDIA device.

Table 29.2 Overall execution time (in millisecond) on four programming platforms

Programming platforms	1024 points	32768 points	1048576 points
NVIDIA CUDA	1.68	3.37	7.94
ATI Stream (GPU)	9.09	9.35	12.99
ATI Stream (CPU)	17.86	20.00	250.00
Intel OpenCL	3.12	8.77	142.86

GFLOPS of OpenCL FFT on Intel OpenCL is 1–3× higher than that on ATI stream (CPU) because Intel fully utilizes the computation resources of its own product. GFLOPS of MKL is 2–10× higher than that of OpenCL on the same CPU. This is not surprising either, because MKL uses more customized optimization techniques on Intel hardware, such as optimization on L1 cache, L2 cache, etc.

Our experimental results show that computation capability of OpenCL on GPU always outperforms CPU-based platforms, for example, on Tesla C2050 GPU, GFLOPS is 25–170× higher than that on Intel i7 920 including Intel MKL running on Intel CPU. This observation indicates that the cross-platform portability of OpenCL standard is valuable in high performance computing on heterogeneous platforms because higher GFLOPS can be achieved without any effort of rewriting code when an application is transplanted from one platform to another.

29.5.3.2 Overall Execution Time

The overall execution time of our benchmark consists of time for signal generation, FFT, points reduction, and visualization. The overall time measured in milliseconds is listed in Table 29.2.

From Table 29.2, we can see OpenFFT-Bench consumes the longest time on ATI Stream (X86 CPU). When we transplant it to other programming platforms, NVIDIA CUDA, ATI stream (GPU), or Intel OpenGL, much less time is consumed, where NVIDIA CUDA platform consumes the least time. Again, this observation indicates that the portability of OpenCL and OpenGL is valubale in high performance computing on hetergenourous platforms.

29.5.3.3 Frame Rate

Frame rate is measured as follows: First of all, a complex signal is generated using OpenCL on i7 920 CPU, NVIDIA Tesla C2050 GPU card or ATI HD 5870 GPU card depending on which is being evaluated; secondly, calculate its corresponding coordinates on the screen, transform the signal and calculate the corresponding signal's spectrum. We reduce the number of points to 1024 due to the limitation of the resolution of the screen when FFT size is larger than 1024. A new period of

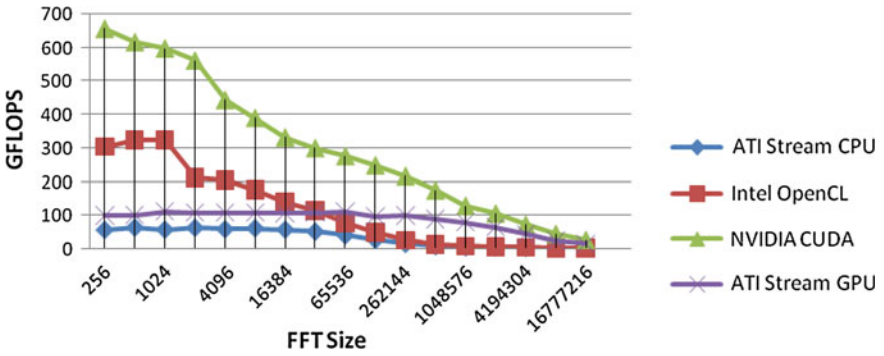


Fig. 29.7 Frame rate on four programming platforms with varying FFT size

points are plotted to replace the points currently displayed on the screen. Figure 29.7 presents the frame rate of the visualization part of OpenFFT-Bench on four different programming platforms.

From Fig. 29.7, we can see that the frame rate of OpenFFT-Bench on NVIDIA Tesla C2050 GPU is 1–7× higher than that on ATI HD 5870 GPU where the peak value, 685 frames per second, is observed at FFT size 128. There is little difference between all the platforms when FFT size is over 1048756. Frame rate on Intel OpenCL platform is 1–7× higher than that on ATI stream (CPU).

Frame rate of OpenFFT-Bench on ATI Stream (GPU) is higher than that on CPU in most cases except Intel OpenCL platform with FFT size less than 32768. Again, it verifies that the portability of OpenCL and OpenGL are valuable in computing on heterogeneous platforms.

29.6 Conclusion

In this paper, we proposed a cross-platform benchmark application, OpenFFT-Bench. It consists of OpenCL-based FFT and OpenGL-based real-time spectrum visualization. Performance of OpenFFT-Bench were evaluated on four OpenCL programming platforms including NVIDIA CUDA, ATI Stream (GPU), ATI Stream (CPU), Intel OpenCL, where two computers were used as the hardware platforms, an Intel i7 920 CPU with NVIDIA GPU, and an Intel i7 860 CPU with ATI GPU. Overall execution time, GFLOPS and frame rate were measured on the four OpenCL programming platforms. Experimental results show that NVIDIA CUDA OpenCL platform outperforms other platforms in all aspects, where the peak GFLOPS value, 280, is observed at FFT size 512, and 685 frames per second is observed at FFT size 128. Intel OpenCL outperforms ATI stream (CPU) in all aspects too. The performance of OpenFFT-Bench on ATI stream (GPU) is higher than that on CPU in most cases except Intel OpenCL platform with FFT size less than 32768. Our results indicate

that OpenCL and OpenGL-based applications can not only run on multiple heterogenerous platforms, but also achieve relatively high performance. Its portability and efficiency will definitely release programmers from rigorous optimization on each individual platform. The efficiency of programmers and the robustness of programs will benefit from these cross-platform specifications as well. OpenCL and OpenGL specifications will contribute to the success of the development of high performance heterogeneous platforms.

Appendix

1. ***OpenCL parallel FFT code***

```
//OpenCL FFT code(128 points)


#ifndef M_PI
#define M_PI 0x1.921fb54442d18p+1
#endif
#define complexMul(a,b) ((float2)(mad(-(a).y, (b).y, (a).x * (b).x), mad((a).y,(b).x, (a).x * (b).y)))
#define conj(a) ((float2)((a).x, -(a).y))
#define conjTransp(a) ((float2)(-(a).y, (a).x))

#define fftKernel2(a,dir) \
{ \
   float2 c = (a)[0]; \
   (a)[0] = c + (a)[1]; \
   (a)[1] = c - (a)[1]; \
}

#define fftKernel2S(d1,d2,dir) \
{ \
   float2 c = (d1); \
   (d1) = c + (d2); \
   (d2) = c - (d2); \
}

#define fftKernel4(a,dir) \
```

```
{ \
   fftKernel2S((a)[0], (a)[2], dir); \
   fftKernel2S((a)[1], (a)[3], dir); \
   fftKernel2S((a)[0], (a)[1], dir); \
   (a)[3] = (float2)(dir)*(conjTransp((a)[3])); \
   fftKernel2S((a)[2], (a)[3], dir); \
   float2 c = (a)[1]; \
   (a)[1] = (a)[2]; \
   (a)[2] = c; \
}

#define fftKernel4s(a0,a1,a2,a3,dir) \
{ \
   fftKernel2S((a0), (a2), dir); \
   fftKernel2S((a1), (a3), dir); \
   fftKernel2S((a0), (a1), dir); \
   (a3) = (float2)(dir)*(conjTransp((a3))); \
   fftKernel2S((a2), (a3), dir); \
   float2 c = (a1); \
   (a1) = (a2); \
   (a2) = c; \
}

#define bitreverse8(a) \
{ \
   float2 c; \
   c = (a)[1]; \
   (a)[1] = (a)[4]; \
   (a)[4] = c; \
   c = (a)[3]; \
   (a)[3] = (a)[6]; \
   (a)[6] = c; \
}

#define fftKernel8(a,dir) \
{ \
```

```
const float2 w1 = (float2)(0x1.6a09e6p-1f, dir*0x1.6a09e6p-1f); \
const float2 w3 = (float2)(-0x1.6a09e6p-1f, dir*0x1.6a09e6p-1f); \
float2 c; \
fftKernel2S((a)[0], (a)[4], dir); \
fftKernel2S((a)[1], (a)[5], dir); \
fftKernel2S((a)[2], (a)[6], dir); \
fftKernel2S((a)[3], (a)[7], dir); \
(a)[5] = complexMul(w1, (a)[5]); \
(a)[6] = (float2)(dir)*(conjTransp((a)[6])); \
(a)[7] = complexMul(w3, (a)[7]); \
fftKernel2S((a)[0], (a)[2], dir); \
fftKernel2S((a)[1], (a)[3], dir); \
fftKernel2S((a)[4], (a)[6], dir); \
fftKernel2S((a)[5], (a)[7], dir); \
(a)[3] = (float2)(dir)*(conjTransp((a)[3])); \
(a)[7] = (float2)(dir)*(conjTransp((a)[7])); \
fftKernel2S((a)[0], (a)[1], dir); \
fftKernel2S((a)[2], (a)[3], dir); \
fftKernel2S((a)[4], (a)[5], dir); \
fftKernel2S((a)[6], (a)[7], dir); \
bitreverse8((a)); \
}

#define bitreverse4x4(a) \
{ \
float2 c; \
c = (a)[1]; (a)[1] = (a)[4]; (a)[4] = c; \
c = (a)[2]; (a)[2] = (a)[8]; (a)[8] = c; \
c = (a)[3]; (a)[3] = (a)[12]; (a)[12] = c; \
c = (a)[6]; (a)[6] = (a)[9]; (a)[9] = c; \
c = (a)[7]; (a)[7] = (a)[13]; (a)[13] = c; \
c = (a)[11]; (a)[11] = (a)[14]; (a)[14] = c; \
}

#define fftKernel16(a,dir) \
{ \
```

```
    const float w0 = 0x1.d906bcp-1f; \
    const float w1 = 0x1.87de2ap-2f; \
    const float w2 = 0x1.6a09e6p-1f; \
    fftKernel4s((a)[0], (a)[4], (a)[8], (a)[12], dir); \
    fftKernel4s((a)[1], (a)[5], (a)[9], (a)[13], dir); \
    fftKernel4s((a)[2], (a)[6], (a)[10], (a)[14], dir); \
    fftKernel4s((a)[3], (a)[7], (a)[11], (a)[15], dir); \
    (a)[5] = complexMul((a)[5], (float2)(w0, dir*w1)); \
    (a)[6] = complexMul((a)[6], (float2)(w2, dir*w2)); \
    (a)[7] = complexMul((a)[7], (float2)(w1, dir*w0)); \
    (a)[9] = complexMul((a)[9], (float2)(w2, dir*w2)); \
    (a)[10] = (float2)(dir)*(conjTransp((a)[10])); \
    (a)[11] = complexMul((a)[11], (float2)(-w2, dir*w2)); \
    (a)[13] = complexMul((a)[13], (float2)(w1, dir*w0)); \
    (a)[14] = complexMul((a)[14], (float2)(-w2, dir*w2)); \
    (a)[15] = complexMul((a)[15], (float2)(-w0, dir*-w1)); \
    fftKernel4((a), dir); \
    fftKernel4((a) + 4, dir); \
    fftKernel4((a) + 8, dir); \
    fftKernel4((a) + 12, dir); \
    bitreverse4x4((a)); \
}

#define bitreverse32(a) \
{ \
float2 c1, c2; \
c1 = (a)[2]; (a)[2] = (a)[1]; c2 = (a)[4]; (a)[4] = c1; c1 = (a)[8]; (a)[8] = c2; c2 = (a)[16]; (a)[16] = c1; (a)[1] = c2; \
c1 = (a)[6]; (a)[6] = (a)[3]; c2 = (a)[12]; (a)[12] = c1; c1 = (a)[24]; (a)[24] = c2; c2 = (a)[17]; (a)[17] = c1; (a)[3] = c2; \
c1 = (a)[10]; (a)[10] = (a)[5]; c2 = (a)[20]; (a)[20] = c1; c1 = (a)[9]; (a)[9] = c2; c2 = (a)[18]; (a)[18] = c1; (a)[5] = c2; \
c1 = (a)[14]; (a)[14] = (a)[7]; c2 = (a)[28]; (a)[28] = c1; c1 = (a)[25]; (a)[25] = c2; c2 = (a)[19]; (a)[19] = c1; (a)[7] = c2; \
c1 = (a)[22]; (a)[22] = (a)[11]; c2 = (a)[13]; (a)[13] = c1; c1 = (a)[26]; (a)[26] = c2; c2 = (a)[21]; (a)[21] = c1; (a)[11] = c2; \
c1 = (a)[30]; (a)[30] = (a)[15]; c2 = (a)[29]; (a)[29] = c1; c1 = (a)[27]; (a)[27] =
```

```
c2; c2 = (a)[23]; (a)[23] = c1; (a)[15] = c2; \
    }

    #define fftKernel32(a,dir) \
    { \
        fftKernel2S((a)[0], (a)[16], dir); \
        fftKernel2S((a)[1], (a)[17], dir); \
        fftKernel2S((a)[2], (a)[18], dir); \
        fftKernel2S((a)[3], (a)[19], dir); \
        fftKernel2S((a)[4], (a)[20], dir); \
        fftKernel2S((a)[5], (a)[21], dir); \
        fftKernel2S((a)[6], (a)[22], dir); \
        fftKernel2S((a)[7], (a)[23], dir); \
        fftKernel2S((a)[8], (a)[24], dir); \
        fftKernel2S((a)[9], (a)[25], dir); \
        fftKernel2S((a)[10], (a)[26], dir); \
        fftKernel2S((a)[11], (a)[27], dir); \
        fftKernel2S((a)[12], (a)[28], dir); \
        fftKernel2S((a)[13], (a)[29], dir); \
        fftKernel2S((a)[14], (a)[30], dir); \
        fftKernel2S((a)[15], (a)[31], dir); \
        (a)[17] = complexMul((a)[17], (float2)(0x1.f6297cp-1f, dir*0x1.8f8b84p-3f));
\
        (a)[18] = complexMul((a)[18], (float2)(0x1.d906bcp-1f, dir*0x1.87de2ap-2f));
\
        (a)[19] = complexMul((a)[19], (float2)(0x1.a9b662p-1f, dir*0x1.1c73b4p-1f));
\
        (a)[20] = complexMul((a)[20], (float2)(0x1.6a09e6p-1f, dir*0x1.6a09e6p-1f));
\
        (a)[21] = complexMul((a)[21], (float2)(0x1.1c73b4p-1f, dir*0x1.a9b662p-1f));
\
        (a)[22] = complexMul((a)[22], (float2)(0x1.87de2ap-2f, dir*0x1.d906bcp-1f));
\
        (a)[23] = complexMul((a)[23], (float2)(0x1.8f8b84p-3f, dir*0x1.f6297cp-1f));
\
        (a)[24] = complexMul((a)[24], (float2)(0x0p+0f, dir*0x1p+0f)); \
         (a)[25] = complexMul((a)[25], (float2)(-0x1.8f8b84p-3f, dir*0x1.f6297cp-
1f)); \
```

```
    (a)[26] = complexMul((a)[26], (float2)(-0x1.87de2ap-2f, dir*0x1.d906bcp-1f)); \
    (a)[27] = complexMul((a)[27], (float2)(-0x1.1c73b4p-1f, dir*0x1.a9b662p-1f)); \
    (a)[28] = complexMul((a)[28], (float2)(-0x1.6a09e6p-1f, dir*0x1.6a09e6p-1f)); \
    (a)[29] = complexMul((a)[29], (float2)(-0x1.a9b662p-1f, dir*0x1.1c73b4p-1f)); \
    (a)[30] = complexMul((a)[30], (float2)(-0x1.d906bcp-1f, dir*0x1.87de2ap-2f)); \
    (a)[31] = complexMul((a)[31], (float2)(-0x1.f6297cp-1f, dir*0x1.8f8b84p-3f)); \
    fftKernel16((a), dir); \
    fftKernel16((a) + 16, dir); \
    bitreverse32((a)); \
}

__kernel void fft0(__global float *in_real, __global float *in_imag, __global float *out_real, __global float *out_imag, int dir, int S)
{
    __local float sMem[640];
    int i, j, r, indexIn, indexOut, index, tid, bNum, xNum, k, l;
    int s, ii, jj, offset;
    float2 w;
    float ang, angf, ang1;
    __local float *lMemStore, *lMemLoad;
    float2 a[8];
    int lId = get_local_id( 0 );
    int groupId = get_group_id( 0 );
      s = S & 3;
    ii = lId & 15;
    jj = lId >> 4;
    if( !s || (groupId < get_num_groups(0)-1) || (jj < s) ) {
      offset = mad24( mad24(groupId, 4, jj), 128, ii );
      in_real += offset;
      in_imag += offset;
      out_real += offset;
      out_imag += offset;
```

```
        a[0].x = in_real[0];
        a[0].y = in_imag[0];
        a[1].x = in_real[16];
        a[1].y = in_imag[16];
        a[2].x = in_real[32];
        a[2].y = in_imag[32];
        a[3].x = in_real[48];
        a[3].y = in_imag[48];
        a[4].x = in_real[64];
        a[4].y = in_imag[64];
        a[5].x = in_real[80];
        a[5].y = in_imag[80];
        a[6].x = in_real[96];
        a[6].y = in_imag[96];
        a[7].x = in_real[112];
        a[7].y = in_imag[112];
    }
    fftKernel8(a+0, dir);
    angf = (float) ii;
    ang = dir * ( 2.0f * M_PI * 1.0f / 128.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
    a[1] = complexMul(a[1], w);
    ang = dir * ( 2.0f * M_PI * 2.0f / 128.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
    a[2] = complexMul(a[2], w);
    ang = dir * ( 2.0f * M_PI * 3.0f / 128.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
    a[3] = complexMul(a[3], w);
    ang = dir * ( 2.0f * M_PI * 4.0f / 128.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
    a[4] = complexMul(a[4], w);
    ang = dir * ( 2.0f * M_PI * 5.0f / 128.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
    a[5] = complexMul(a[5], w);
    ang = dir * ( 2.0f * M_PI * 6.0f / 128.0f ) * angf;
    w = (float2)(native_cos(ang), native_sin(ang));
```

```
a[6] = complexMul(a[6], w);
ang = dir * ( 2.0f * M_PI * 7.0f / 128.0f ) * angf;
w = (float2)(native_cos(ang), native_sin(ang));
a[7] = complexMul(a[7], w);
lMemStore = sMem + mad24(jj, 144, ii);
j = ii & 7;
i = ii >> 3;
i = mad24(jj, 144, i);
lMemLoad = sMem + mad24(j, 18, i);
lMemStore[0] = a[0].x;
lMemStore[18] = a[1].x;
lMemStore[36] = a[2].x;
lMemStore[54] = a[3].x;
lMemStore[72] = a[4].x;
lMemStore[90] = a[5].x;
lMemStore[108] = a[6].x;
lMemStore[126] = a[7].x;
barrier(CLK_LOCAL_MEM_FENCE);
a[0].x = lMemLoad[0];
a[1].x = lMemLoad[4];
a[2].x = lMemLoad[8];
a[3].x = lMemLoad[12];
a[4].x = lMemLoad[2];
a[5].x = lMemLoad[6];
a[6].x = lMemLoad[10];
a[7].x = lMemLoad[14];
barrier(CLK_LOCAL_MEM_FENCE);
lMemStore[0] = a[0].y;
lMemStore[18] = a[1].y;
lMemStore[36] = a[2].y;
lMemStore[54] = a[3].y;
lMemStore[72] = a[4].y;
lMemStore[90] = a[5].y;
lMemStore[108] = a[6].y;
lMemStore[126] = a[7].y;
barrier(CLK_LOCAL_MEM_FENCE);
```

```
a[0].y = lMemLoad[0];
a[1].y = lMemLoad[4];
a[2].y = lMemLoad[8];
a[3].y = lMemLoad[12];
a[4].y = lMemLoad[2];
a[5].y = lMemLoad[6];
a[6].y = lMemLoad[10];
a[7].y = lMemLoad[14];
barrier(CLK_LOCAL_MEM_FENCE);
fftKernel4(a+0, dir);
fftKernel4(a+4, dir);
angf = (float) (ii >> 3);
ang = dir * ( 2.0f * M_PI * 1.0f / 16.0f ) * angf;
w = (float2)(native_cos(ang), native_sin(ang));
a[1] = complexMul(a[1], w);
ang = dir * ( 2.0f * M_PI * 2.0f / 16.0f ) * angf;
w = (float2)(native_cos(ang), native_sin(ang));
a[2] = complexMul(a[2], w);
ang = dir * ( 2.0f * M_PI * 3.0f / 16.0f ) * angf;
w = (float2)(native_cos(ang), native_sin(ang));
a[3] = complexMul(a[3], w);
angf = (float) ((16 + ii) >>3);
ang = dir * ( 2.0f * M_PI * 1.0f / 16.0f ) * angf;
w = (float2)(native_cos(ang), native_sin(ang));
a[5] = complexMul(a[5], w);
ang = dir * ( 2.0f * M_PI * 2.0f / 16.0f ) * angf;
w = (float2)(native_cos(ang), native_sin(ang));
a[6] = complexMul(a[6], w);
ang = dir * ( 2.0f * M_PI * 3.0f / 16.0f ) * angf;
w = (float2)(native_cos(ang), native_sin(ang));
a[7] = complexMul(a[7], w);
lMemStore = sMem + mad24(jj, 160, ii);
j = ii >> 3;
i = ii & 7;
i = mad24(jj, 160, i);
lMemLoad = sMem + mad24(j, 40, i);
```

```
lMemStore[0] = a[0].x;
lMemStore[40] = a[1].x;
lMemStore[80] = a[2].x;
lMemStore[120] = a[3].x;
lMemStore[16] = a[4].x;
lMemStore[56] = a[5].x;
lMemStore[96] = a[6].x;
lMemStore[136] = a[7].x;
barrier(CLK_LOCAL_MEM_FENCE);
a[0].x = lMemLoad[0];
a[1].x = lMemLoad[8];
a[2].x = lMemLoad[16];
a[3].x = lMemLoad[24];
a[4].x = lMemLoad[80];
a[5].x = lMemLoad[88];
a[6].x = lMemLoad[96];
a[7].x = lMemLoad[104];
barrier(CLK_LOCAL_MEM_FENCE);
lMemStore[0] = a[0].y;
lMemStore[40] = a[1].y;
lMemStore[80] = a[2].y;
lMemStore[120] = a[3].y;
lMemStore[16] = a[4].y;
lMemStore[56] = a[5].y;
lMemStore[96] = a[6].y;
lMemStore[136] = a[7].y;
barrier(CLK_LOCAL_MEM_FENCE);
a[0].y = lMemLoad[0];
a[1].y = lMemLoad[8];
a[2].y = lMemLoad[16];
a[3].y = lMemLoad[24];
a[4].y = lMemLoad[80];
a[5].y = lMemLoad[88];
a[6].y = lMemLoad[96];
a[7].y = lMemLoad[104];
barrier(CLK_LOCAL_MEM_FENCE);
```

```
fftKernel4(a+0, dir);
fftKernel4(a+4, dir);
if( !s || (groupId < get_num_groups(0)-1) || (jj < s) ) {
out_real[0] = a[0].x;
out_imag[0] = a[0].y;
out_real[16] = a[4].x;
out_imag[16] = a[4].y;
out_real[32] = a[1].x;
out_imag[32] = a[1].y;
out_real[48] = a[5].x;
out_imag[48] = a[5].y;
out_real[64] = a[2].x;
out_imag[64] = a[2].y;
out_real[80] = a[6].x;
out_imag[80] = a[6].y;
out_real[96] = a[3].x;
out_imag[96] = a[3].y;
out_real[112] = a[7].x;
out_imag[112] = a[7].y;
 }
}
```

2. ***FFT Rendering code***

```
// OpenGL vertex buffers interoprated with OpenCL
GLuint buffer;
GLuint bufferSource;

// OpenCL bind the two buffers
resbuffer = clCreateFromGLBuffer(context, CL_MEM_READ_WRITE, buffer,
&errcode);
signalbuffer = clCreateFromGLBuffer(context, CL_MEM_READ_WRITE,
bufferSource, &errcode);

// GLUT call back function
void display(){
        // do FFT on next period of data in OpenCL
        err = clEnqueueAcquireGLObjects(queue, 2, mem_objects, 0, NULL,
```

```
NULL);
        err = clSetKernelArg(fftsinkernel, 2, sizeof(int), &rnd);
        err = clEnqueueNDRangeKernel(queue, fftsinkernel, 1,0, globalworksize, localworksize, 0, NULL, NULL);
        err = clEnqueueNDRangeKernel(queue, extractkernel, 1, 0, globalworksize, localworksize, 0, NULL, NULL);
        err = clEnqueueNDRangeKernel(queue, signalkernel, 1, 0, localworksize, localworksize, 0, NULL, NULL);
        clEnqueueReadBuffer(queue, signalbuffer, 1, 0, 3 * 1024 * sizeof(float), res, 0, NULL, NULL);

        clFFT_ExecutePlannar(queue, fft_plan, batchSize, dir, realbuffer, imagbuffer, realbuffer, imagbuffer, 0, NULL, NULL);

        err = clEnqueueNDRangeKernel(queue, modulekernel, 1, 0, globalworksize, localworksize, 0, NULL, NULL);
        err = clEnqueueNDRangeKernel(queue, normkernel, 1, 0, globalworksize, localworksize, 0, NULL, NULL);
        err = clEnqueueNDRangeKernel(queue, transferkernel, 1, 0, localworksize, localworksize, 0, NULL, NULL);

        clEnqueueReleaseGLObjects(queue, 2, mem_objects, 0, NULL, NULL);

        // Clear the color buffer and depth buffer
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Set the state of OpenGL to display
        glViewport(0, 0, SCREEN_WIDTH * 3 / 4, SCREEN_HEIGHT);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glFrustum(-0.01, 0.01, 0 - 0.01 * SCREEN_HEIGHT / SCREEN_WIDTH, 0.01 * SCREEN_HEIGHT / SCREEN_WIDTH, 0.01, 100);
        gluLookAt(EYE.x, EYE.y, EYE.z, CENTER.x, CENTER.y, CENTER.z,
        UP.x, UP.y, UP.z);
        glScalef(3.0, 0.01, 5.0);

        // Display FFT spectrums in 3D mode
        if (drawPoints){
```

```
            glColor3f(0.0, 0.0, 0.0);
            glBindBuffer(GL_ARRAY_BUFFER, buffer);
            glEnableClientState(GL_VERTEX_ARRAY);
            glVertexPointer(DIM, GL_FLOAT, 0, BUFFER_OFFSET(0));
            glDrawArrays(GL_POINTS, 0, REDUCE_NUM * DATA_NUM);
        }
        else {
            shader->useProgram();
            glBindBuffer(GL_ARRAY_BUFFER, buffer);
            glEnableClientState(GL_VERTEX_ARRAY);
            glVertexPointer(DIM, GL_FLOAT, 0, BUFFER_OFFSET(0));

            glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
            glEnableClientState(GL_INDEX_ARRAY);
            glIndexPointer(GL_FLOAT, 0, BUFFER_OFFSET(0));

            glDrawElements(GL_TRIANGLE_STRIP, ((REDUCE_NUM *
2) + 2) * (DATA_NUM - 1), GL_UNSIGNED_INT, 0);

            glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
            glDisableClientState(GL_INDEX_ARRAY);
            shader->cancelProgram();
        }

        // Display FFT spectrum in 2D mode
        glViewport(SCREEN_WIDTH * 3 / 4, 0, SCREEN_WIDTH / 4,
        SCREEN_HEIGHT / 2);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        gluOrtho2D(-0.2, 2.0, -0.7, 2.8);
        glScalef(1.8, 0.007, 1.0);
        if (drawPoints){
            glDrawArrays(GL_POINTS, 0, REDUCE_NUM);
        }
        else {
            shader->useProgram();
```

```
                glLineWidth(1.0);
                glDrawArrays(GL_LINES, 0, REDUCE_NUM);
                shader->cancelProgram();
        }

        // Display source signal data in 2D mode
         glViewport(SCREEN_WIDTH * 3 / 4, SCREEN_HEIGHT / 2,
SCREEN_WIDTH / 4, SCREEN_HEIGHT / 2);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        gluOrtho2D(-0.5, 5.5, -1.5, 1.5);
        glScalef(5.0, 100.0, 1.0);
        glColor3f(0.0, 0.0, 0.0);
        glBindBuffer(GL_ARRAY_BUFFER, bufferSource);
        glEnableClientState(GL_VERTEX_ARRAY);
        glVertexPointer(DIM, GL_FLOAT, 0, BUFFER_OFFSET(0));
        if (drawPoints){
                glDrawArrays(GL_POINTS, 0, REDUCE_NUM);
        }
        else {
                glLineWidth(1.0);
                glDrawArrays(GL_LINE_STRIP, 0, REDUCE_NUM);
        }

        // Double buffer swap
        glutSwapBuffers();
}
```

References

Bader DA, Agarwal V (2007) FFTC: fastest fourier transform for the IBM cell broadband engine. In: Proceedings of the 14th IEEE international conference on high performance computing (HIPC). Lecture notes in computer science vol 4873:2007

Bainville E (2010). http://www.bealto.com/gpu-fft_opencl-1.html

Baliley DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, Simon HD, Venkatakrishnan V, Weeratunga SK (1991) The NAS parallel benchmarks-summary and preliminary results. In: Proceeding of supercomputing '91

Barrachina S, Castillo M (2008) Igual FD, Mayo R, Quintana-Ortí ES (2008) Evaluation and tuning of the level 3 CUBLAS for graphics processors. IEEE international symposium on parallel and distributed processing, In

Berry M, Chen D, Koss P, Kuck D, Lo S, Pang Y, Pointer L, Roloff R, Sameh A, Clementi E, Chin S, Schnerder D, Fox G, Messina P, Walker D, Hsiung C, Schwarzmeier J, Lue K, Orszag S, Seidl F, Johnson O, Goodrum R (1989) The PERFECT club benchmarks: effective performance evaluation of supercomputers. Int J Supercomput Appl 3:5–40

Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex fourier series. Math Comput 19:297–301

da Silva CP, Cupertino LF, Chevitarese D, Pacheco MAC (2010) Exploring data streaming to improve 3D FFT implementation on multiple GPUs. In: 22nd international symposium on computer architecture and high performance computing workshops.

Dmitruk P, Wang L-P, Matthaeus WH, Zhang R, Seckel D (2001) Scalable parallel FFT for spectral simulations on a Beowulf cluster. Parallel Comput 27(14):1921–1936

Dongarra J (1988a) The LINPACK benchmarks: an explanation, supercomputing. Springer, Berlin, pp 10–14

Dongarra J (1988b) Performance of various computers using strandard linear equations software in a fortran environment. Technical report MCSRD 23, Argonne National Laboratory, March 1988.

Franchetti F, Voronenko Y, Puschel M (2006) FFT Program generation for shared memory: SMP and multicore. Proceedings of supercomputing, In

Frigo M, Johnson SG (2007) FFTW on the cell processor. http://www.Fftw.org/cell/index.html

Goedecker S, Boulet M, Deutsch T (2003) An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes. Comput Phys Commun 154:105–110

Govindaraju NK, Lloyd B, Dotsenko Y, Smith B, Manferdelli J (2008) High performance discrete fourier transforms on graphics processors. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing

Li LW, Wang YJ, Li EP (2003) MPI based parallelized precorrected FFT algorithm for analyzing scattering by arbitrarily shaped three-dimensional objects. J Electromagn Waves Appl 42:247–259

Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu WW (2008) Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming

Sorensen HV, Burrus CS (1995) Fast fourier transform. PWS Publishing, Boston

The OpenCL specification version 1.0. http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf, https://www.developer.nvidia.com/cuda-downloads

Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing

Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K (2006) The potential of the cell processor for scientific computing. In: Proceedings of the 3rd conference on computing, Frontiers, pp 9–20

Chapter 30
Accurate Evaluation of Local Averages on GPGPUs

Dmitry A. Karpeev, Matthew G. Knepley and Peter R. Brune

Abstract We discuss fast and accurate evaluation of local averages on GPGPU. This work was motivated by the need to calculate reference fluid densities in the classical density functional theory (DFT) of electrolytes proposed in Gillespie et al. (2002). In Knepley et al. (2010) we developed efficient algorithms for the minimization of three-dimensional DFT models of biological ion channel permeation and selectivity. One of the essential bottlenecks of 3D DFT models is the evaluation of local *screening averages* of the chemical species' densities. But the problem has wider applicability and fast evaluation of averages over the local spherical screening neighborhood of every grid point are typically inaccurate due to the use of collocation approximations of densities on Cartesian grids. Accurate evaluations based spectral quadrature were proposed and used in Knepley et al. (2010), but they are significantly more computationally expensive because of their nonlocal nature in the Fourier space. Here we show that the passage to the Fourier space can, in fact, make the averaging calculation much more amenable to efficient implementation on GPGPU architectures. This allows us to take advantage of both improved accuracy and hardware acceleration to arrive at a fast and accurate screening calculations.

D. A. Karpeev (✉) · P. R. Brune
Mathematics and Computer Science Division, Argonne National Laboratory,
Argonne, IL 60439, USA
e-mail: karpeev@mcs.anl.gov

P. R. Brune
e-mail: prbrune@gmail.com

M. G. Knepley
Computation Institute, University of Chicago, Chicago, IL 60637, USA
e-mail: knepley@ci.uchicago.edu

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 487
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_30,

30.1 Introduction

The problem of *fast* and *accurate* evaluation of local averages is ubiquitous in computational science. One such calculation arises in computational physical chemistry: *repeated* evaluation of reference fluid densities in the classical density functional theory (DFT) of electrolytes (see Gillespie et al. 2002 for details). However, this type of calculation arises in many contexts whenever a density ρ, piecewise constant over the cells of a regular grid, needs to be averaged over a small ball B^i about each grid cell i. This is a discrete version of the following continuous mapping $\rho \mapsto \overline{\rho}$:

$$\overline{\rho}(\boldsymbol{x}) = \frac{1}{|B^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}|} \int_{B^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}} \rho(\boldsymbol{y}) d\boldsymbol{y} = \int \rho(\boldsymbol{y}) \chi^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}(y) d\boldsymbol{y} = \int \rho(\boldsymbol{y}) \chi_{R_{\boldsymbol{x}}}(\boldsymbol{y} - \boldsymbol{x}) d\boldsymbol{y}. \tag{30.1}$$

The notation in the above equation is as follows: $B^{\boldsymbol{x}}_{R_{\boldsymbol{x}}} = \{\boldsymbol{y} : |\boldsymbol{y} - \boldsymbol{x}| < R_{\boldsymbol{x}}\}$ is an averaging ball of radius $R_{\boldsymbol{x}}$ about $\boldsymbol{x}$, $|B^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}|$ is the volume of the ball, and $\chi^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}$ denotes the normalized[1] characteristic function of this averaging ball, while $\chi_{R_{\boldsymbol{x}}}$ is the normalized characteristic function of a ball of the same size at the origin $B_{R_{\boldsymbol{x}}}$. We can write $\chi_{R_{\boldsymbol{x}}}$ explicitly as follows:

$$\chi_{R_{\boldsymbol{x}}} = \frac{\theta(R_{\boldsymbol{x}} - |\boldsymbol{x}|)}{\frac{4}{3} R^3_{\boldsymbol{x}}}, \tag{30.2}$$

where $\theta(x)$ is the Heaviside step function (http://en.wikipedia.org/wiki/Heaviside_step_function), which is the characteristic function of the set $\{x > 1\} \subset \mathbb{R}$, equal to 1 for $x > 0$ and to 0 otherwise.

The integral in Eq. (30.1) is a convolution, but only if $R_{\boldsymbol{x}} = R \equiv const$. Many efficient computational techniques are available for calculating convolutions, most notably, the Fast Fourier Transform (FFT) algorithm (Cooley and Tukey 2000). In many applications, however, $R_{\boldsymbol{x}}$ can vary substantially from point to point, so other approaches are necessary for an accurate numerical evaluation of (30.1).

30.2 Discrete Averaging Problem

Here we define a discrete averaging calculation by applying (30.1) to a piecewise constant density defined on a regular grid with N cells. This can be easily generalized to other discretizations of ρ. Schematically discrete averaging is illustrated in Fig. 30.1: the piecewise constant density is averaged over a ball about cell i and the average is assigned to i. This calculation amounts to an application of a sparse matrix $\mathbf{A} = (A_{ij})$ to the vector $\boldsymbol{\rho} = \left(\rho_i\right)$ of cell density values:

[1] $\int \chi^{\boldsymbol{x}}_{R_{\boldsymbol{x}}} d\boldsymbol{x} = 1$.

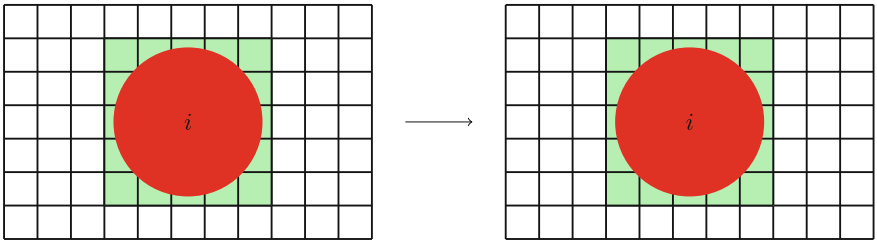


Fig. 30.1 Averaging of piecewise constant cell values over a ball about i

$$\rho_i = \sum_j A_{ij}\rho_j . \tag{30.3}$$

30.2.1 Runtime Complexity

The elements the ith row of $\mathbf{A}$ constitute *quadrature* weights approximating integral (30.1) at the ith cell center $\boldsymbol{x}_i$ over the ball $B^i = B^{\boldsymbol{x}_i}_{R_{\boldsymbol{x}_i}}$ using cell density values $\boldsymbol{\rho}$. For a piecewise constant ρ each element A_{ij} is the volume of a spherical segment—the intersection of B^i with grid cell j. We assume that the averaging ball radius $R_i = R_{\boldsymbol{x}_i}$ is small compared to the size of the computational grid, so the averaging is a local operation and $\mathbf{A}$ is sparse. In particular, the bandwidth of the matrix is determined by the number of grid cells intersecting the largest averaging ball B^i (Fig. 30.2). If the maximal averaging radius $\sup_{\boldsymbol{x}} R_{\boldsymbol{x}}$ is small and bounded independently of the problem size N (the number of grid points), the averaging operation (30.3) will be a sparse matrix-vector product (SpMV) and will have an optimal asymptotic complexity $\mathcal{O}(N)$. In the end, however, it is the wallclock time of the calculation that matters, not the theoretical complexity. It is well-known that on modern hardware architectures SpMV rarely achieves more than 10 % of the peak hardware performance and is limited by the memory bandwidth (Vuduc et al. 2005). This is because the *a*rithmetic intensity—the ratio of arithmetic operations to load/store operations—is rather low for SpMV and most of the time is spent waiting on the data to travel from the memory to the CPU. Thus, alternative approaches to SpMV should be explored, even if they result in a higher theoretical complexity. Indeed, the higher asymptotic complexity can be potentially offset by the higher peak performance of the alternative averaging method.

One approach to improving the overall wallclock time could be *h*ardware acceleration, for example, using a (general purpose) graphics processing unit, (GP)GPU. Here we specifically consider the NVIDIA GPU architecture and their CUDA programming model (http://developer.nvidia.com/cuda-gpus), and assume the reader has a basic familiarity with them. The memory bandwidth bottleneck affects calculations on the GPU just as well as on the CPU: each streaming processor (SP) running a thread from a given block will have to spend most of the time waiting for the data

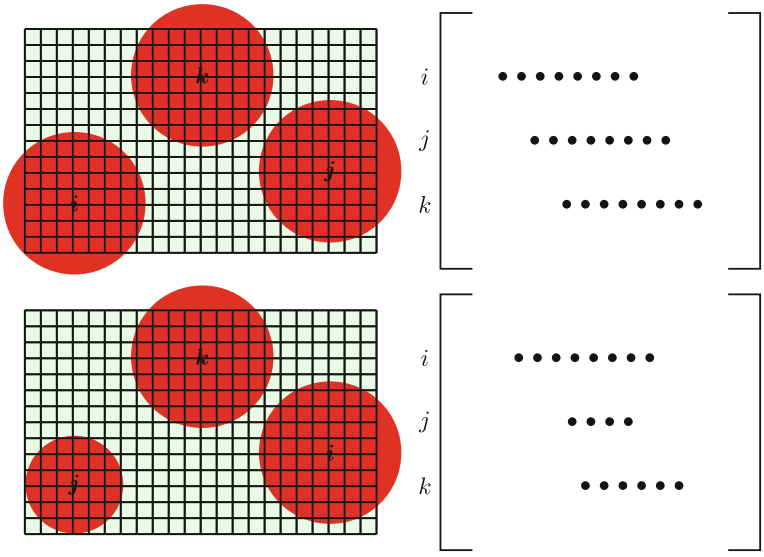


Fig. 30.2 Schematic structure of the averaging matrix with (*top*) and without (*bottom*) a constant averaging radius; • represents nonzero matrix entries

to be delivered. Paralleling the computation, which is essentially free compared to the memory load/store time, will result in a modest improvement at best. Even with $\mathcal{O}(N)$ SPs the calculation will still take $\mathcal{O}(N)$ time, instead of $\mathcal{O}(1)$, since there is only one memory bus and most of the time is spent on its delivery of data.

Convolution averaging. Another stumbling block on the way to an efficient deployment of GPUs is the irregular sparsity pattern of the matrix rows (see Fig. 30.2 (*bottom*)). Indeed, assume each CUDA thread is responsible for the calculation of a single output entry ρ_i—a single inner product between the ith row of $\mathbf{A}$ and $\boldsymbol{\rho}$. Different matrix rows have different numbers of nonzero elements, hence, different amounts of work, but the calculation in a block is limited by the thread executing the longest inner product, while the other threads waste SP cycles. One scenario, in which the sparsity pattern is constant for all rows is the case of $R_{\boldsymbol{x}} = R \equiv const$ (see Fig. 30.2 (*top*)). Here (30.1) becomes a convolution and there are efficient ways of carrying out its discrete version without even forming matrix $\mathbf{A}$. In fact, averaging is best done in the Fourier space using the convolution theorem:

$$\overline{\rho} = \rho \star \chi_R = \left(\rho^{\vee} \chi_R^{\vee}\right)^{\wedge}, \tag{30.4}$$

where $f^{\wedge}$ is the Fourier transform of f and $f^{\vee}$ is the inverse Fourier transform; in particular, $f = f^{\wedge\vee} = f^{\vee\wedge}$. This version of the averaging calculation is essentially a dense reformulation of the SpMV algorithm (30.3). Its total time and storage complexity of (30.4) is $\mathcal{O}(N \log N)$ and $\mathcal{O}(N)$ respectively, but it can be carried out on a GPU efficiently using CUFFT, a CUDA implementation

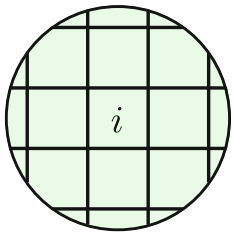


Fig. 30.3 Cell segments cut out from the grid by an averaging ball about cell i

of the FFT algorithm (http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUFFT_Library_3.1.pdf). This is the first indication of how we might achieve high performance of the averaging operation on a GPU: reformulating the problem as a dense linear algebra operation and utilizing the full hardware power of a GPU.

30.2.2 Accuracy

Before matrix **A** can be applied, however, its entries must be accurately calculated, since this will impact the accuracy of the overall discrete averaging computation. Having a fast but an inaccurate averaging operation is of little value, and for some applications, such as the DFT reference density calculation, it is entirely worthless, as we will indicate below. How can A_{ij} be computed accurately?

Spherical segments. Depending on the radius R_i of the ith averaging ball, the spherical segments resulting from its intersections with grid cells can have different shapes (see Fig. 30.3). To compute the volume of each spherical segment we must determine which grid planes intersect the averaging ball B^i, which requires accurate evaluation of geometric predicates (e.g., deciding whether a point lies on one side of the plane or the other). Then, for each grid cell intersecting the averaging ball the shape of the resulting spherical segment must be determined and its volume calculated. Typically, the volume calculation can will lead to a many expensive evaluations of trigonometric functions. Overall, the calculation of spherical segments of different shapes will take a lot of floating point arithmetic and logic to implement the shape-dependent flow control. If the averaging operation with the *same* radii $\boldsymbol{R} = (R_i)$ needs to be applied many times, the relatively expensive calculation of A_{ij} can be amortized over the many matrix-vector multiplications. In the case of a sparse **A**, precomputing and storing the matrix can be an acceptable solution, despite complex and error-prone nature of the direct calculation algorithm. In the problems we are mostly interested in, in particular in the DFT problem described below, the averaging radii change in the course of the overall calculation: it may be helpful to think about R_x as being time dependent and averaging occurring every timestep. Then the calculations of the spherical segment volumes have to be carried out every iteration.

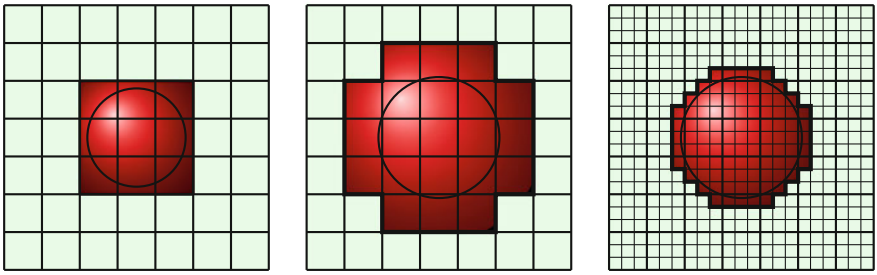


Fig. 30.4 Pixelized ball approximation on an increasingly fine grid

Pixelized balls. As we mentioned above, the accuracy of averaging takes precedence over the speed, so we must compute $\mathbf{A}$ accurately, before it can be applied to $\boldsymbol{\rho}$. One approach that avoid most of the complexities of spherical segment volume calculations is to approximate each averaging ball B^i by a "pixelized ball"—the set of all grid cells meeting B^i (see Fig. 30.3). Since whole cells and not spherical segments are included in the calculation, the algorithm becomes simpler and much more robust, but diminishes in accuracy. To remedy that the grid could be refined to include $\overline{N} > N$ points to improve the approximation of the B^i by pixelized balls (see Fig. 30.4). This will result in a larger matrix $\overline{\mathbf{A}}$ with an much higher number of nonzeros, since for a fixed R_i the number of nonzeros in the ith row grows as $\mathcal{O}(\overline{N})$ (see Fig. 30.5).

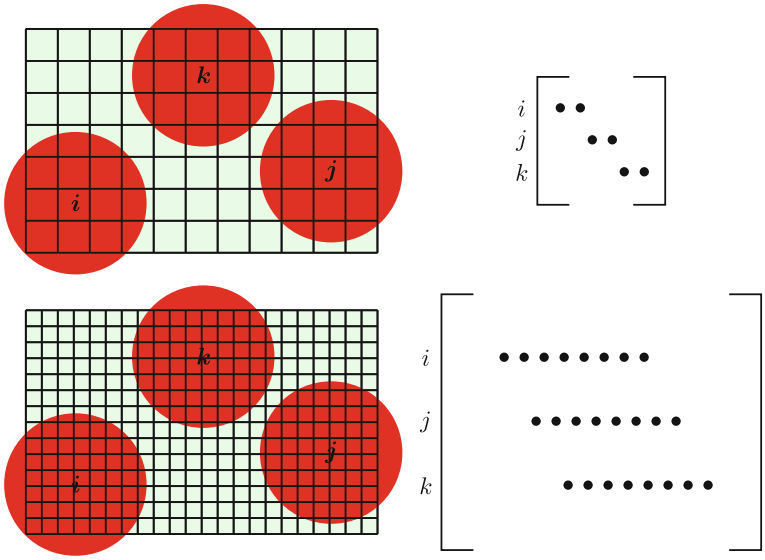


Fig. 30.5 Schematic structure of the averaging matrix and its change under grid refinement. • represents nonzero matrix entries. Refinement of the grid by a factor of 2 in each direction results in the quadrupling of the matrix storage (*bottom*) relative to the unrefined grid (*top*)

This results in a denser calculation, so the use of GPU can be considered again. The trade off between accuracy and speed remains poor, however: the accuracy of $\mathbf{A}$ improves roughly as $\mathcal{O}(1/\overline{N})$, while the computational complexity grows as $\mathcal{O}(\overline{N}^2)$. The accuracy return on refinement saturates near $\overline{N} = \mathcal{O}(N^2)$, since at that point the error in the approximation of B^i by a pixelized ball will become dominated by the error in the discrete approximation of ρ by $\boldsymbol{\rho}$: indeed, grid refinement does not improve the quality of $\boldsymbol{\rho}$ initially given on a grid of size N. Finally, the irregular nature of the calculation and memory access in Eq. (30.3) are still detrimental to its deployment on GPUs: while the calculation becomes denser, it does not become more regular. In particular, the matrix-free approach of (30.4) is still out of reach.

30.3 Spectral Quadrature

The idea of our approach to averaging is to combine the improvements in accuracy with the improvements in the structure of the algorithm that favor its deployment on GPUs, even at the expense of some increase in asymptotic complexity. After all, the algorithms discussed above, while sufficiently accurate are either too fragile and error prone (calculation of $\mathbf{A}$ by spherical segment volumes), or have effective complexity much higher than $\mathcal{O}(N)$ (pixelized ball approximation on a refined grid $\overline{N} \gg N$). Neither one is particularly suitable for an efficient deployment on a GPU. Instead, we propose a reformulation inspired by the Fourier space approach to averaging (30.4).

30.3.1 Convolution by Spectral Quadrature

Observe that in the application of (30.4) the issue of the accuracy of the spherical segment volumes was sidestepped altogether: the quadrature was replaced by an inner product of $\hat{\boldsymbol{\rho}}$, the discrete Fourier transform of $\boldsymbol{\rho}$, with $\hat{\boldsymbol{\chi}}_R = \left(\hat{\chi}_R(\boldsymbol{k}_i)\right)$, the discretized Fourier transform of χ_R. Unlike $\hat{\boldsymbol{\rho}}$, we can obtain $\hat{\boldsymbol{\chi}}_R$ by computing $\hat{\chi}_R$ *exactly* and then discretizing it, rather than applying FFT to $\boldsymbol{\chi}_R = \left(\boldsymbol{\chi}_R(\boldsymbol{x}_i)\right)$, a discretization of χ_R. This is possible because an analytic form of $\hat{\chi}_R$ can be obtained as follows:

$$\hat{\chi}_R(\boldsymbol{k}) = \frac{1}{(2\pi)^{\frac{3}{2}}} \int_{\mathbb{R}^3} \chi_R(\boldsymbol{y}) e^{-i\boldsymbol{k}\cdot\boldsymbol{y}} d\boldsymbol{y} = \frac{3}{4\pi \left(R\sqrt{2\pi}\right)^3} \int_{\mathbb{R}^3} \theta(R - |\boldsymbol{y}|) e^{-i\boldsymbol{k}\cdot\boldsymbol{y}} d\boldsymbol{y}.$$

This integral can be evaluated noting that the integrand depends only on the absolute values $k = |\boldsymbol{k}|$ and $r = |\boldsymbol{y}|$, and the angle θ between $\boldsymbol{k}$ and $\boldsymbol{y}$. Passing to the spherical coordinates with $\boldsymbol{k}$ as the axis (see Fig. 30.6) and ϕ the azimuthal angle in the plane orthogonal to $\boldsymbol{k}$, we compute as follows:

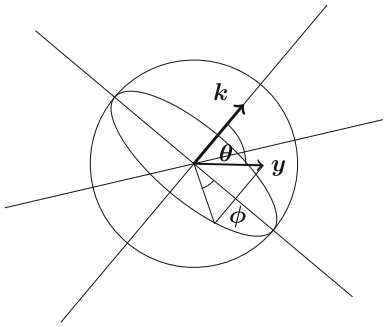


Fig. 30.6 Spherical coordinates relative to the axis through $\boldsymbol{k}$

$$\begin{aligned}
\hat{\chi}_R(\boldsymbol{k}) &= \frac{3}{4\pi\left(R\sqrt{2\pi}\right)^3}\int_0^{2\pi} d\phi \int_0^{\pi} \sin\theta d\theta \int_{\mathbb{R}} \theta(R-r) r^2 e^{-ikr\cos\theta} dr \\
&= \frac{6\pi}{4\pi\left(R\sqrt{2\pi}\right)^3}\int_0^{R} r^2 dr \int_0^{\pi} e^{-ikr\cos\theta}\sin\theta d\theta \\
&= \frac{3}{2\left(R\sqrt{2\pi}\right)^3}\int_0^{R} r^2 dr \int_{-1}^{1} e^{ikru} du \\
&= \frac{3}{2\left(R\sqrt{2\pi}\right)^3}\int_0^{R} \frac{2r}{k}\sin krdr
\end{aligned}$$

where we used a change of coordinates $u = \cos\theta$.The last integral can be easing evaluated using integration by parts:

$$\int_0^R r\sin krdr = \left[\frac{r}{k}\cos kr\right]_R^0 + \frac{1}{k}\int_0^R \cos krdr = -\frac{R\cos kR}{k} + \frac{\sin kR}{k^2}.$$

Putting it all together we obtain an analytic expression for $\hat{\chi}_R$

$$\hat{\chi}_R(\boldsymbol{k}) = \frac{3}{\left(\sqrt{2\pi}\right)^3}\left(-\frac{\cos kR}{k^2R^2} + \frac{\sin kR}{k^3R^3}\right) \tag{30.5}$$

which allows us to compute $\hat{\boldsymbol{\chi}}_R$ to machine precision by evaluating it at the Fourier grid cell centers $\boldsymbol{k}_i$. In fact, $\hat{\boldsymbol{\chi}}_R$ need not be precomputed or stored: each $\hat{\boldsymbol{\chi}}_R = \hat{\chi}_R(\boldsymbol{k}_j)$ can be computed on the fly while evaluating the inner product with $\hat{\boldsymbol{\rho}}$ using only j, h (grid cell size) and R, effectively halving the required bandwidth to the memory. An evaluation of (30.3) by discretizing (30.4) can be viewed as *spectral quadrature* of (30.1).

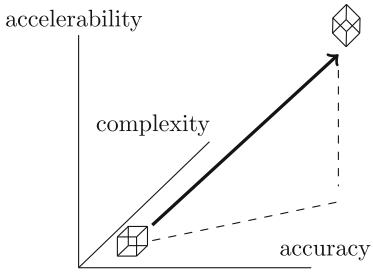


Fig. 30.7 Algorithm reformulation to improve accuracy at the expense of arithmetic complexity, offsetting the latter by better accelerability

We can think of the Fourier transform as a software *accelerator* that speeds up the actual performance of the averaging algorithm (30.4). Here is the general scheme we want to apply to the general averaging procedure: replace algorithm of **low** *accuracy* but also low computational *complexity* and low amenability to *acceleration*, with algorithm of **higher** *accuracy*, but possibly higher *complexity*, which is offset by accelerating the algorithm using software or hardware (see Fig. 30.7).

30.3.2 General Averaging by Spectral Quadrature

In the convolution case Fourier methods were used to both improve the accuracy of the quadrature, and to enable a software acceleration method of the matrix-vector product. Passing to the Fourier space and using a discretization of the analytic Fourier transform of $\hat{\chi}_R$ pushed the accuracy essentially to its natural limit, but replaced a sparse and local (relative to the grid) matrix calculation (30.3) with a dense and global calculation—the Fourier transform. Unlike grid refinement for example, the Fourier-based "densification" has a regular computational structure that could be exploited by the accelerator—the FFT algorithm. In the general averaging case something similar happens: a dense spectral quadrature calculation replaces (30.3), but, since it does not have enough regularity, FFT cannot be used. Still, enough regularity exists in the computation that it can be effectively accelerated by hardware—GPU.

In order to derive the spectral quadrature in the general case we no longer appeal to the convolution theorem, but, rather, to more a pair of more basic results: (1) the L^2-isometry property of the Fourier transform, and (2) the multiplicative action of the shift on the Fourier image.[2] The first result is more fundamental and states that the L^2 inner product of two functions is equal to the L^2 inner product of their Fourier images. Using the second equality in Eq. (30.1) we then obtain

$$\overline{\rho} = \int \rho(\boldsymbol{y})\chi^{\boldsymbol{x}}_{R_x}(\boldsymbol{y}) = \int \hat{\chi}^{\boldsymbol{x}}_{R_x}(\boldsymbol{k})\hat{\rho}(\boldsymbol{k}). \tag{30.6}$$

This is the essence of spectral quadrature.

[2] The convolution theorem can be derived from these two results.

The second result allows us to reduce the calculation of $\hat{\chi}^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}$ to that of χ_R as follows:

$$\begin{aligned}\hat{\chi}^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}(\boldsymbol{k}) &= \frac{1}{(2\pi)^{\frac{3}{2}}}\int \chi^{\boldsymbol{x}}_{R_{\boldsymbol{x}}}(\boldsymbol{y})e^{-i\boldsymbol{k}\cdot\boldsymbol{y}}d\boldsymbol{y} = \frac{1}{(2\pi)^{\frac{3}{2}}}\int \chi_{R_{\boldsymbol{x}}}(\boldsymbol{x}-\boldsymbol{y})e^{-i\boldsymbol{k}\cdot\boldsymbol{y}}d\boldsymbol{y} \\ &= \frac{e^{i\boldsymbol{k}\cdot\boldsymbol{x}}}{(2\pi)^{\frac{3}{2}}}\int \chi_{R_{\boldsymbol{x}}}(\boldsymbol{y})e^{-i\boldsymbol{k}\cdot\boldsymbol{y}}d\boldsymbol{y} = \frac{e^{i\boldsymbol{x}\cdot\boldsymbol{k}}}{(2\pi)^{\frac{3}{2}}}\hat{\chi}_{R_{\boldsymbol{x}}},\end{aligned} \tag{30.7}$$

where $\hat{\chi}_{R_{\boldsymbol{x}}}$ has form (30.5), since that expression is insensitive to the dependence of R on $\boldsymbol{x}$. Putting together a discrete version of (30.6) now we obtain

$$\overline{\rho}_i = \sum_j \frac{e^{i\boldsymbol{x}_i\cdot\boldsymbol{k}_j}}{(2\pi)^{\frac{3}{2}}}\hat{\chi}_{R_i}(\boldsymbol{k}_j)\hat{\rho}_j, \quad \text{or} \quad \overline{\boldsymbol{\rho}} = \mathbf{L}\hat{\boldsymbol{\rho}}, \tag{30.8}$$

where matrix $\mathbf{L}$ has elements $L_{ij} = \frac{e^{i\boldsymbol{x}_i\cdot\boldsymbol{k}_j}}{(2\pi)^{\frac{3}{2}}}\hat{\chi}_{R_i}(\boldsymbol{k}_j)$, $\hat{\boldsymbol{\rho}} = \mathbf{F}\boldsymbol{\rho}$ and $\mathbf{F}$ is the matrix of the discrete Fourier transform operator with the elements $F_{ij} = \frac{e^{i\boldsymbol{x}_i\cdot\boldsymbol{k}_j}}{(2\pi)^{\frac{3}{2}}}$.

How does this compare to the case $R_{\boldsymbol{x}} = R \equiv const$? In this case $R = R_i$ and equation (30.8) can be factored using a diagonal matrix $\hat{\boldsymbol{K}}$ with the elements $\hat{K}_{ij} = \delta_{ij}\hat{\chi}_R(\boldsymbol{k}_j)$:

$$\overline{\boldsymbol{\rho}} = \mathbf{F}^T\hat{\boldsymbol{K}}\hat{\boldsymbol{\rho}} = \mathbf{F}^T\hat{\boldsymbol{K}}\mathbf{F}\boldsymbol{\rho}. \tag{30.9}$$

This is a discrete version of the convolution theorem, as expected. Since $\hat{\boldsymbol{K}}$ is diagonal, it is cheap to apply—$\mathcal{O}(N)$, while F and F^T have enough structure to make their application relatively fast using the FFT algorithm—$\mathcal{O}(N\log N)$. When $R_i \not\equiv const$ such a factorization is impossible and the combined matrix $\mathbf{L}$ is dense and has no particular structure to enable a fast matrix-vector product algorithm. This matrix, however, can be apply in the matrix-free manner since all of its matrix elements can be computed on the fly using only $\mathcal{O}(N)$ data $\boldsymbol{R} = (R_i)$, significantly reducing the memory bandwidth requirement. Furthermore, for the total memory access requirement in Eq. (30.8), which is $\mathcal{O}(N)$ both for $\boldsymbol{R}$ and $\hat{\boldsymbol{\rho}}$, we have $\mathcal{O}(N^2)$ flops, as in any dense matrix-vector product. Finally, each inner product between $\hat{\boldsymbol{\rho}}$ and a row of $\mathbf{L}$ can be efficiently parallelized on a GPU.

30.4 CUDA Spectral Quadrature

In this section we present a method for evaluating (30.8) on GPU that can be easily implemented in CUDA. The main bottleneck in computing (30.8) from $\boldsymbol{\rho}$ is in the application of $\mathbf{L}$, since, as mentioned above, $\hat{\boldsymbol{\rho}}$ can be obtained in $\mathcal{O}(N\log N)$ using

CUDAFFT (http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUFFT_Library_3.1.pdf). While (30.8) increases the complexity to $\mathcal{O}(N^2)$, this is essentially unavoidable for accuracy reasons. The question now is how to best utilize the GPU hardware to offset this increase in complexity (see Fig. 30.7); in other words, how can we increase the *accelerability* of the algorithm. We first present the algorithm, and then illustrate its speedup by comparing a GPU calculation to a CPU calculation.

30.4.1 Algorithm

The algorithm we present is reminiscent of the standard CUDA parallelization of the matrix-matrix product. In particular, it achieves a similar reuse of the local data. We use a simple, yet generic illustration in Fig. 30.8. The complete algorithm starts with ρ and R in the CPU memory, and ends up with $\overline{\rho}$ in the CPU memory. The Fourier space density $\hat{\rho}$ is computed first. It can be done on the CPU using any of the standard FFT implementations and transferred to the GPU, or the transform itself can be done on the GPU, leaving $\hat{\rho}$ in the GPU's main memory; R is transferred to the GPU main memory in either case. We now discuss the core part of the algorithm corresponding to Eq. (30.8)—the calculation of $\overline{\rho}$ from $\hat{\rho}$ and R on the GPU.

Using shared memory. The averaged density $\overline{\rho}$ is computed one block $\overline{\rho}_s$ at a time; the whole GPU participates in computing $\overline{\rho}_s$. The calculation is carried out by an array of B thread blocks running concurrently on the GPU. On an NVIDIA GPU, each thread block runs on a single streaming multiprocessor (SM), which consists of a number (e.g., 8) of streaming processors (SP) illustrated by yellow squares in Fig. 30.8. The typical number of threads per block is 32. Each SM has fast but small local shared memory (`shmem` in Fig. 30.8), which can be accessed within about a single clock cycle by all of the threads within the block. The strategy is to minimize the number of accesses to the global GPU memory and keep most of the threads busy with the data in `shmem`. Since the calculation of $\overline{\rho}_s$ requires the averaging radii R_s supported on the same grid block s, each thread block loads R_s into its own `shmem` (the gray rectangles inside `shmem` in Fig. 30.8). The reading is done concurrently by all of the threads (Fig. 30.8 indicates the thread responsible for producing the corresponding `shmem` value—by loading from global memory or via a calculation).

Subdomain calculation. Once R_s is in `shmem`, $\overline{\rho}_s$ can be computed. Each $\hat{\rho}_j$ contributes to each $\overline{\rho}_{s_i}$, $s_i \in s$, but this calculation—the sum on j in Eq. (30.8)—will be broken up into partial sums with j varying over small subdomains $a, b, c, d, \ldots$ of the Fourier grid. The subdomains are made small enough to fit into a single thread block's `shmem` and distributed to the thread blocks in a round robin fashion (colored rectangles inside `shmem` in Fig. 30.8). Each thread t of the thread block owning a calculates the contribution from $\hat{\rho}_a$ to a single $\overline{\rho}_{s_t}$ by carrying out the sum in Eq. (30.8) for $j \in a$. Observe that the matrix elements $\hat{\chi}^{\boldsymbol{x}_{s_t}}_{R_{s_t}}(\boldsymbol{k}_j) = \frac{e^{i\boldsymbol{x}_{s_t}\cdot\boldsymbol{k}_j}}{(2\pi)^{\frac{3}{2}}}\hat{\chi}_{R_{s_t}}(\boldsymbol{k}_j)$ never need to be stored, but can be calculated on the fly, once by each thread t from

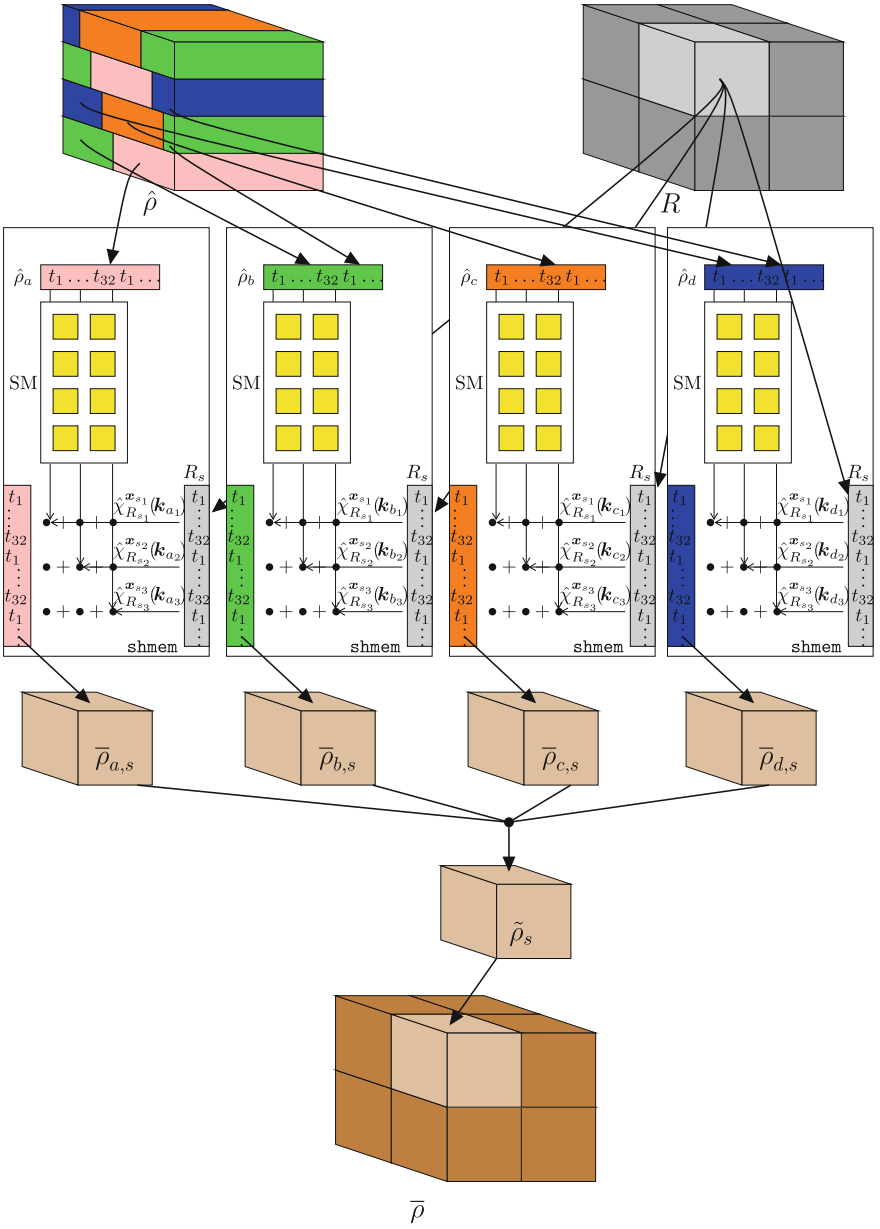


Fig. 30.8 Illustration of the averaging algorithm on a 3D domain. *Arrows* denote data flow, colors label different GPU thread blocks ($t_1 \ldots t_{32}$), with 32 typically being the largest number of simultaneously running threads per SM that the hardware can support

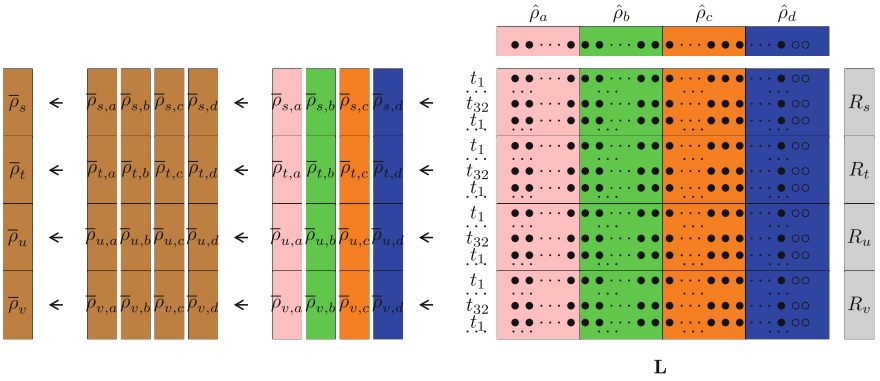


Fig. 30.9 Illustration of the data and work decomposition of the matrix form of quadrature (Eq. (30.8)). *Filled circles* represent grid points, *unfilled circles* represent padding to complete the thread blocks

s_t, R_{s_t} and j. Thus, the block achieves about $\beta|a|$ flops per $|a| + |s|$ loads from memory. The prefactor β is the cost of calculating each matrix element $\hat{\chi}_{R_{s_t}}^{\boldsymbol{x}_{s_t}}(\boldsymbol{k}_j)$, which can be significant, since each such calculation involves evaluation of trigonometric functions (see Eqs. (30.5, 30.7)). Most importantly, these calculations can be performed concurrently by the hundreds of available SPs. The result is B partial sum contributions to $\overline{\rho}_s$: $\overline{\rho}_{s,a}, \overline{\rho}_{s,b}, \overline{\rho}_{s,c}, \overline{\rho}_{s,d}, \ldots$.

Reduction and accumulation. The partial sums must be added to ρ_s, which is initially 0. This operation of an array summation is known as a *reduction* and is well-documented along with several of its GPU implementations (see, e.g., http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf). The reduction is carried out by a different set of thread blocks. For that the partial sums are copied from `shmem` to the GPU global memory (brown cubes in Fig. 30.8). Then a separate GPU calculation is carried out to sum up the partial sums down to a single block $\tilde{\rho}_s$. Notation $\tilde{\rho}_s$ refers to the fact that the result of this reduction is not necessarily all of $\overline{\rho}_s$, but only the contribution from the B blocks $a, b, c, d, \ldots$ processed in one iteration. This contribution is added to $\overline{\rho}$ in GPUs global memory, and the process repeated with the same real space grid block s until all of Fourier space grid blocks have been processed and, thus, all of ρ_s has been computed. After that the calculation moves on to other real space blocks $t, u, v, \ldots$, until the whole real space grid has been covered.

Before concluding the description of the algorithm, we briefly illustrate its matrix interpretation in Fig. 30.9. In effect, the columns of matrix $\mathbf{L}$ are partitioned and distributed to different thread blocks—corresponding to the partitioning of the Fourier space grid into subdomains $a, b, c, d, \ldots$. These columns are applied to small blocks of row values $\hat{\rho}$ corresponding to the real space grid subdomains $s, t, u, v, \ldots$. The partial output vectors $\overline{\rho}_{s,a}, \overline{\rho}_{s,b}, \overline{\rho}_{s,c}, \overline{\rho}_{s,d}, \ldots$ correspondingto the application of

the column blocks to $\hat{\rho}$ are summed up to produce $\overline{\rho}_s$. Finally, the values of the partitioned matrix are never stored by are computed on the fly from the element indices (i, j) and the averaging radius values R_i.

30.5 Results and Discussion

To test our algorithm we draw on a previously mentioned problem from physical chemistry—the calculation of the screened density of ions in a solution (Gillespie et al. 2002). A numerical approach to this problem was developed in Knepley et al. (2010), in which the authors discovered that a repeated application of the averaging operator is the main bottleneck in the calculation of the equilibrium ionic density ρ. In that algorithm the averaged density $\overline{\rho}$ is used as an important, but intermediate calculation in a large iterative solution loop. Certain approximations to $\overline{\rho}$ turn out to be permissible, while maintaining the convergence rate of the iterative procedure. Other approximations, however, badly reduce the convergence rate. In particular, approximating screening balls with cubes (e.g., Fig. 30.4 (left)) is not allowed, but it is permissible to carry out the averaging procedure in single precision, while the rest of the calculation is done in double precision.

Since the timing of the algorithm depends only on the size of the domain and is completely independent of the numerical values of ρ or R, we carry out our tests on the case of constant R. We carry out the same summation corresponding to Eq. (30.8) on the GPU using the developed algorithm (in single precision), and on the CPU (in double precision). The total number of grid points in our test is $N = 32^3$.

For the test we used a computer with a Intel Core 2 Duo E8400 3.0 GHz CPU and 4 GB of RAM. The GPU was an NVIDIA GTX285 (http://www.nvidia.com/object/product_geforce_gtx_285_us.html) with 1 GB of global memory and 240 SP cores organized into SMs of 8 cores with 16k of shared memory between them.

The GPU implementation of the averaging algorithm was done using NVIDIA's CUDA programming environment. In particular, we used PyCuda (http://mathema.tician.de/software/pycuda), a Python wrapper for the CUDA language (http://developer.nvidia.com/cuda-gpus) that has a large number of built-in utilities for basic operations and program setup, allowing for easy prototyping of the algorithm. The reference CPU implementation of the averaging algorithm was written using the PETSc numerical linear algebra library (http://www.mcs.anl.gov/petsc).

To demonstrate the effectiveness of our approach to averaging on GPU we carried out a simple numerical study on cubic grids of varying size $N = n^3$, where n is the number of grid points in one dimension. The results are summarized in Table 30.1. The CUDA implementations show a speedup in excess of 300× over the CPU implementation for our test cases. In fact, CPU runtime was so prohibitively slow for $n = 64, 80, 96$ that the calculation was aborted (indicated by — in the table). Still, a 300× speedup is somewhat nonintuitive, as the number of SP cores is 240, suggesting a 240× speedup possible at the most. Recall, however, that the GPU computation is done in single precision, affording a further 2× to 4×-fold speedup over the CPU calculation. With the more recent NVIDIA GPUs more double-precision

Table 30.1 Runtime results of the averaging algorithm

n	$N = n^3$	GPU (s)	CPU (s)
16	4096	0.1	6.3
32	32768	1.5	448.4
48	110592	14.9	4481.7
64	262144	76.2	—
80	512000	297.2	—
96	884736	870.3	—

Here n is the linear of the cubic computational grid, and $N = n^3$ is total number of grid points

units are available per SM, so that the same calculation can be carried out on the GPU in double precision. We do not expect that this modification would alter the results significantly.

How far is our algorithm from the full utilization of the GPU hardware? Using a benchmark calculation we estimated that the evaluation of the trigonometric functions sin() and cos() takes about 24 floating point operations (flops) on the GPU we used. This is the main component of the core averaging algorithm described in the previous section. From this and the clock speed of NVIDIA GTX285 we estimate that the averaging algorithm achieves a rate of about 124 Gigaflop/s (GF/s), while the GPU card has a theoretical peak (neglecting the special function unit) of approximately 660 GF.

Clearly, there is room for improvement for our averaging algorithm. Even so, we show that by reformulating the problem in a way that simultaneously increases accuracy and accelerability (Fig. 30.7), we can beat the increased computational complexity of the algorithm. In fact, any sufficiently accurate algorithm would have computational complexity comparable to our parallel algorithm—a manifestation of the well-known dictum that the best parallel algorithm is also *a* best serial algorithm. From all of the equally *best* serial algorithms (consider, for example, an equally accurate and expensive refinement algorithm in Fig. 30.4) however, we must select the one that affords the most concurrency and, therefore, the most accelerability by GPU hardware.

Using the massive parallelism available on commodity GPU hardware, we were able to accelerate the averaging calculation by more than 300×. In particular, for the biological problem in (Knepley et al. 2010) this results in the acceleration of a key component in the simulation of biological ion channels, making possible simulations of realistic channels that were previously out of reach.

References

Gillespie D, Nonner W, Eisenberg RS (2002) J Phys Condens Matter 14:12129

Knepley M, Karpeev D, Davidovits S, Eisenberg R, Gillespie D (2010) J Comp Phys 132(12): 124101–124112

Cooley JW, Tukey JW (1965) Math Comput 19(90):297–301

Vuduc R, Demmel JW, Yelick KA (2005) J Phys Conf Ser 16:521 (Proceedings of SciDAC 2005)

Chapter 31
Accelerating Swarm Intelligence Algorithms with GPU-Computing

Robin M. Weiss

Abstract Swarm intelligence describes the ability of groups of social animals and insects to exhibit highly organized and complex problem-solving behaviors that allow the group as a whole to accomplish tasks which are beyond the capabilities of any individual. This phenomenon found in nature is the inspiration for swarm intelligence algorithms—systems that utilize the emergent patterns found in natural swarms to solve computational problems. In this paper, we will show that due to their implicitly parallel structure, swarm intelligence algorithms of all sorts can benefit from GPU-based implementations. To this end, we present the ClusterFlockGPU algorithm, a swarm intelligence data mining algorithm for partitional cluster analysis based on the flocking behaviors of birds and implemented with CUDA. Our results indicate that ClusterFlockGPU is competitive with other swarm intelligence and traditional clustering methods. Furthermore, the algorithm exhibits a nearly linear time complexity with respect to the number of data points being analyzed and running time is not affected by the dimensionality of the data being clustered, thus making it well-suited for high-dimensional data sets. With the GPU-based implementation adopted here, we find that ClusterFlockGPU is up to 55x times faster than a sequential implementation and its time complexity is significantly reduced to nearly $O(n)$.

R. M. Weiss (✉)
Department of Geology and Geophysics,
University of Minnesota,
Minneapolis, USA
e-mail: Weiss.Robin.M@gmail.com

R. M. Weiss
Minnesota Supercomputing Institute,
University of Minnesota,
Minneapolis, USA

R. M. Weiss
Department of Computer Science,
Macalester College,
Saint Paul, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 503
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_31,

31.1 Introduction

Swarm intelligence describes the phenomenon of self-organization that emerges in groups of decentralized agents. In nature, swarm behavior allows colonies of ants to efficiently locate and gather food, flocks of birds to coordinate their movements and herds to evade predators. These highly organized global-behaviors exhibited by the group as a whole emerge from the actions of individuals (local-behaviors) and direct and indirect communication between swarm members. So-called swarm intelligence algorithms use virtual swarms to reproduce these swarm behaviors with the goal of solving computational problems.

In the past decade, the field of swarm intelligence has become a hot topic in the areas of computer science, collective intelligence, and robotics. Swarm intelligence algorithms based on the behaviors of ant colonies have been shown to be able to tackle a wide range of hard optimization problems including the Traveling Salesman, Quadratic Assignment, and Network Routing Problems (Dorigo and Caro 1999; Dorigo and Stützle 1999, 2004). Bird-flocking algorithms have been successfully applied in the areas of computer graphics, data mining, and distributed vehicle control (Veenhuis and Köppen 2006; Olfati-Saber 2004).

There have been a number of publications regarding the application of swarm intelligence to data mining problems. These algorithms have been shown to produce results that are competitive with traditional techniques (Palathingal et al. 2005; Parpinelli et al. 2001; Merwe and Engelbrecht 2003; Cui and Potok 2006; Zhang et al. 2011), and some even provide useful features not found in other methods (Grosan et al. 2006; Charles et al. 2007). We have found that with the exception of Charles et al. (2007) and Zhang et al. (2011), none of these algorithms have multi-threaded let alone GPU-based implementations.

Swarm intelligence systems are characterized by groups of independent agents collectively working to solve some problem and, as such, contain a large amount of implicit parallelism which we believe is a good match for the GPU computing model. To demonstrate, we present our ClusterFlockGPU algorithm, a GPU-based flocking algorithm for partitional cluster analysis. The goal of partitional cluster analysis is to partition a collection of data points into "clusters" where data points in one cluster exhibit maximal similarity to data points in that cluster, while exhibiting minimal similarity to data points in every other cluster.

The ClusterFlockGPU algorithm is a synthesis of the flock-based MSF and DSC clustering algorithms proposed in Cui and Potok (2006) and Veenhuis and Köppen (2006), respectively. These algorithms all capitalize on the "birds of a feather flock together" effect to achieve a partitional clustering of data points. In ClusterFlockGPU, a flock of boids is generated where each boid represents a single data point from the input data set. The natural flocking behaviors of the boids eventually lead to the formation of sub-flocks where boids in a given sub-flock are similar to one another, and dissimilar to those in all other sub-flocks. Then, by using a reverse look up to map each boid back to the data point it represents, the clusters in the data set can be identified.

The primary advantage of a flock-based approach to partitional clustering is that the number of clusters does not need to be known at runtime (a major limitation of the k-means and other algorithms). With flock-based clustering, the number of clusters simply appears from the interactions of the flock agents. Additionally, we show that the ClusterFlockGPU algorithm is not affected by the dimensionality of the data being analyzed making it well suited for high-dimensional data sets.

The focus of this paper is solely a GPU-based approach for bird flocking algorithms. However, we wish to underscore that the GPU architecture is well-suited for swarm intelligence algorithms of all sorts. For an in-depth analysis of a GPU-based ant colony algorithm for rule-based classification, the reader is referred to Weiss (2011).

31.2 Flock Algorithms

The term "flock algorithm" refers to any algorithm that models the collective behaviors exhibited by flocks of birds, schools of fish, or herds of animals in nature. The basis for nearly all flock algorithms is derived from the work of Craig Reynolds and his seminal work "Flocks, Herds, and Schools: A Distributed Behavioral Model" (Reynolds 1987). In Reynolds (1987), Reynolds proposes that the global behaviors of a flock of birds can be achieved in a virtual environment by having each bird agent (or "boid") adhere to three rules as it flies. These are:

1. **Collision Avoidance**: Boids steer away from any nearby boid that is within some threshold distance.
2. **Velocity Matching**: Boids modify their velocities to match the average velocity of their *k*-nearest flockmates.
3. **Flock Centering**: Boids steer toward the centroid of their *k*-nearest flockmates.

Reynolds shows that by repeatedly applying these three flocking rules to each boid in the flock, a realistic flocking behavior of the swarm as a whole can be achieved. The overall flocking algorithm can be summarized as follows:

```
create N boid agents and initialize with random position and velocity
for each boid:
     vAvoid = collision avoidance force
     vMatching = velocity matching force
     vCentering = flock centering force
     boid.velocity += (c1 * vAvoid) + (c2 * vMatching) + (c3 * vCentering)
     boid.position += boid.velocity
done
```

Listing 1: Pseudocode of the generalized flocking algorithm

In the above algorithm, `c1`, `c2` and `c3` are constants that weight the relative influence of the three components of each boids' velocity updating. Although not depicted above, also note that in most implementations the magnitude of each boids' velocity is limited to some maximum to prevent chaotic and out-of-control behavior.

31.3 Swarm Intelligence and GPU Computing

This project proposes that swarm intelligence systems are well-suited for implementation on GPU devices. With a sequential implementation, the population of a swarm has a direct and often very large impact on overall running time as each swarm agent's "thought process" occurs one after the next on the CPU. This makes algorithms that require very large populations of swarm agents unacceptable due to the associated complexity. We propose that the massive multi-threading capabilities of GPU devices can be used to achieve much larger populations of swarm agents without severely degrading running time.

There is clearly a large amount of implicit parallelism in the algorithm from Listing 1. In particular, all N boid agents must complete the same calculations in each iteration based on the characteristics of their neighbors (as dictated by Reynold's flocking rules). Because of this, we can easily fit this algorithm into the SPMD execution pattern of CUDA.

A straightforward way of parallelizing the algorithm from Listing 1 is to allocate one GPU thread for each of the N agents. This allows us to unroll the for-loop and significantly "flatten" the algorithm. In such an implementation, each agent concurrently calculates it's flocking rules and updates its velocity and position. With this approach the complexity of the flock algorithm is significantly reduced.

Furthermore, because all boid agents in flock algorithms operate on local data (the properties of the boids in each boid's local neighborhood), an intelligent nearest-neighbor algorithm can be used to reduce the number of boids that each boid must consider in each iteration of the algorithm. As noted in CUDA documentation, memory latency is often the largest bottleneck with GPU-based algorithms. Since each boid only needs to consider it neighbors, the impact of the long-latency DRAM memory operations on the GPU can be mitigated.

31.4 ClusterFlockGPU

ClusterFlockGPU is a synthesis of the MSF and DSC algorithms presented in Cui and Potok (2006) and Veenhuis and Köppen (2006). Like these methods, the ClusterFlockGPU algorithm introduces the notion of similarity into Reynolds' algorithm to achieve a partitional clustering effect. With each boid representing a data point from the input data set, we repeatedly apply the following rules to determine the motion of each boid in a 2-dimensional virtual environment:

1. **Avoidance**: Each boid attempts to move away from the most dissimilar boid in its vicinity with the magnitude of this force proportional to how dissimilar the boids are.
2. **Velocity Matching**: Each boid modifies its velocity to match the velocity of the boid which it is most similar to.

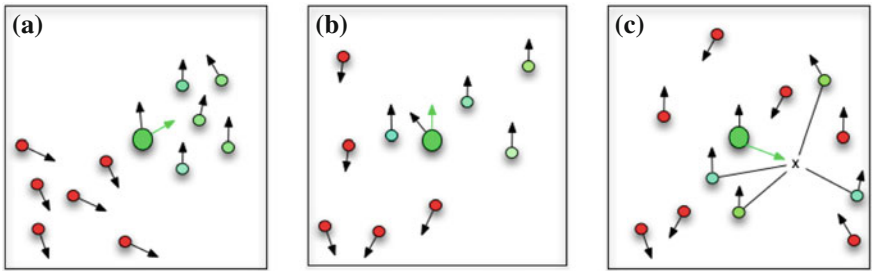


Fig. 31.1 The three flocking rules each boid adheres to as it moves. **a** Shows the avoidance force, **b** the alignment force, and **c** the flock centering force

3. **Flock Centering**: Each boid moves toward the centroid of its *k* most similar other boids with a force proportional to the similarity between the boid and the most similar boid of the *k* most similar other boids in its neighborhood.

In Fig. 31.1 we show how a given boid (the large boid in the center of each sub-figure) will update its velocity based on these rules and the characteristics of its neighbors. The black vectors emanating from each boid shows that boid's current velocity. The green vector emanating from the large boid shows the vector which satisfies the given flocking rule. Keep in mind that each boid will calculate these three flocking rules for itself and based on its own and its neighbors characteristics.

The ClusterFlockGPU algorithm works by first generating a population of boids, each representing one of the input data points. The input data itself is loaded into the GPU and based on the data, a similarity value between each pair of boids is calculated via the application of some similarity metric. ClusterFlockGPU is compatible with any similarity metric `sim(boidA, boidB)` given that `sim` produces a real-valued result in the range [0,1] where 0 signifies the data points represented by `boidA` and `boidB` are completely opposite, and 1 indicates `boidA` is identical to `boidB`.

The similarity values between all boids are computed in the very first stage of the algorithm and are based on the data points that each boid represents. Because the later stages of the algorithm depend only on this similarity value and not on the data that was used to calculate it, the dimensionality of data being clustered has no impact on the bulk of the algorithm. Furthermore, after pair-wise similarity is computed, the data itself can be unloaded from GPU memory as only the similarity values need be referenced.

For some predefined number of iterations, the flocking rules presented above are applied to each boid. When the number of completed iterations is greater than the parameter Ω (usually set to 30 % of the maximum number of iterations), the environment of the boids is shrunk by 2 % each iteration until the environment is just 2 % larger in the x- and y-directions than the threshold distance which defines the

boids' local neighborhood. In this way, nearly all of the boids are brought into the immediate neighborhoods of each other. This facilitates a more complete intermixing of sub-flocks and helps prevent the situation where multiple clusters of what should be the same one form by forcing more boids into each others' immediate neighborhood.

When the maximum number of iterations has been reached, boids will have separated into discernible sub-flocks which can be interpreted as clusters. We then use a simple agglomerative algorithm to extract cluster membership from the positions of boids in their virtual environment.

Starting with a randomly chosen boid not assigned to a cluster, a new cluster label, C_i is created and all boids whose euclidean distance from the selected boid is less than some threshold distance, T_c, are grouped into C_i. This process continues by randomly selecting another unassigned boid and grouping all boids within the threshold distance into C_{i+1}, and so on.

There is an obvious weakness in this grouping strategy as the situation can arise where depending on which boid is selected as the "center" boid, clusters may be split into multiple clusters, or boids which belong in one cluster are grouped with another. Figure 31.2 shows how this situation can occur. Notice that two of the boids which should be grouped in cluster 1 have been grouped into cluster 2. Because the selection of "center" boids in the cluster extraction scheme is random, this erroneous situation is both possible and unpredictable. An improved cluster extraction method would benefit the ClutserFlockGPU algorithm and represents a distinct area for improvement.

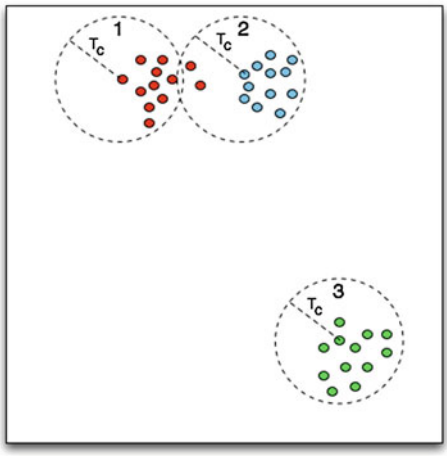


Fig. 31.2 Erroneous cluster extraction due to random "center" boid selection. Boids from cluster 1 are grouped with cluster 2

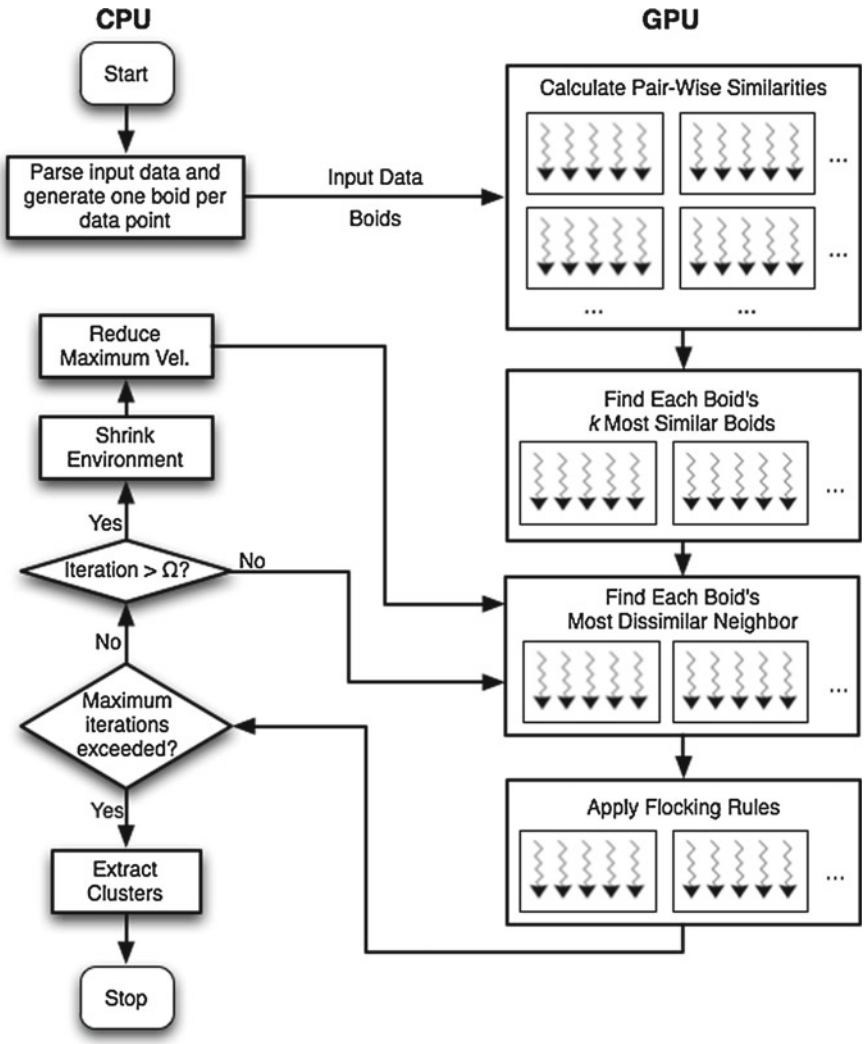


Fig. 31.3 Control flow of the ClusterFlockGPU algorithm

Listing 2 gives a pseudocode overview of the GPU implementation of the ClusterFlockGPU algorithm. We use the notation (C/G—threadCount) to indicate where the step takes place (CPU or GPU) and how many concurrent threads are used. Figure 31.3 depicts the control flow of ClusterFlockGPU diagrammatically.

```
(C)  parse input data set
(C)  generate N boids (one boid per data point)
(C)  initialize all boids with random position and velocity
(C)  copy boids to GPU
(C)  copy input data set to GPU
(G - N^2) calculate pair-wise similarity between all boids
(G - N) determine each boid`s n most similar other boids
while (maximum number of iterations not met)
     (G - N) find each boid`s neighbors and most dissimilar neighbor
     (G - N) apply flocking rules and update boid positions
     (C) if current iteration # > Omega
          (C) shrink environment and decrease maximum velocity
done
(C)  extract clusters
```

31.5 Implementation Notes

The overall implementation strategy for ClusterFlockGPU is very straightforward. We use an array of `boid structs` to hold the data associated with each boid in the flock as shown below.

```
struct Boid {
     float x;
     float y;
     float dx;
     float dy;
};
```

Since the relevant characteristics of each boid are its position and velocity in the x- and y-directions, each boid struct can be neatly aligned to 16-byte boundaries in global memory. This allows for efficient access to these data structures from global memory where they must be stored. The array of boid structs is initialized by the CPU and copied to the GPU in the first step of the algorithm.

The similarity value between each boid is calculated by a single kernel function, `computeSimMat`, which is called once before the main loop of the algorithm is entered. A `__device__` function `getSimilarity` is used to apply a similarity metric based on the data represented by each boid.

```
__global__ void computeSimMat(...){

     int a = blockIdx.x * blockDim.x + threadIdx.x;
     int b = blockIdx.y * blockDim.y + threadIdx.y;
     if (a >= numBoids || b >= numBoids) return;

     float sim = getSimilarity(a, b, inputData);
     similarities[a * numBoids + b] = sim;

}
```

Following this step, two GPU kernels, `getDisSim` and `stepTime`, are repeatedly called in each iteration to advance positions of the boids in the flock. `getDisSim` conducts an all-to-all calculation between all boids to determine which boids are in each's neighborhood and which of them is the most dissimilar. The methods `calcMatching`, `calcCentering`, and `calcAvoidance` in `stepTime` are `__device__` functions and implement each of the above detailed flocking rules.

```
__global__ void stepTime(...){

      int boidID = blockIdx.x * blockDim.x + threadIdx.x;
      if (boidID >= numBoids) return;

      // Calculate this boid's new velocity factors
      float2 vMatching = calcMatching(...);
      float2 vCentering = calcCentering(...);
      float2 vAvoid = calcAvoidance(...);
      ...

      vCentering *= similarities[idx(me, mostSim)]);
      vAvoid *= (1.0f - similarities[idx(me, mostDisSim)]);

      boids[boidID].dx += vMatching.x + vCenter.x + vAvoid.x;
      boids[boidID].dy += vMatching.y + vCenter.y + vAvoid.y;

      // Scale new velocity to keep within maxVal if necessary
      ...

      // Move the boid one time step according to its velocity
      ...

      boids[boidID].x += boids[boidID].dx;
      boids[boidID].y += boids[boidID].dy;

}
```

31.6 Results and Analysis

We begin our analysis of the performance of ClusterFlockGPU by assessing its running time on data sets of varying sizes and dimensionality. To this end, we run the algorithm on differently sized synthetic data sets of randomly generated data points. Running ClusterFlockGPU on these random data sets serves two purposes. First, it allows us to generate data sets of arbitrary size such that we can compare the effect of both data count and dimensionality on performance. Second, if the algorithm is performing a true clustering, no distinct clusters should emerge from the random data as we assume no strong groupings of data points to be present. Our results for this experiment are shown in Fig. 31.4a, b.

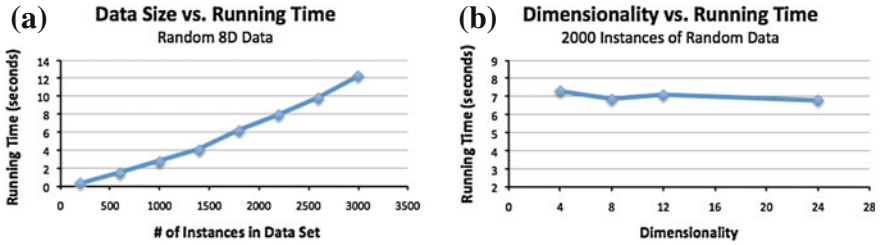


Fig. 31.4 Performance analysis of the ClusterFlockGPU algorithm. Graph **a** illustrates the impact of data size on overall running time. Graph **b** shows the effect of dimensionality on overall running time

The test platform used for the results given below is a was a desktop PC equipped with a 2.4 GHz Intel Core 2 Quad processor and 3 GB of DDR3 memory. The GPU device used was an NVIDIA Tesla C1060.

We see in Fig. 31.4a that the running time of the ClusterFlockGPU algorithm scales in a nearly linear fashion with the size of the input data set. A profile of the algorithm reveals that running time is dominated by the time spent identifying the nearest neighbors of each boid as needed for calculating the avoidance force. This step is completed by having every boid calculate its distance to every other boid and those within some threshold distance are taken to be "nearby."

The neighbor detection step has complexity $O(n^2)$ when performed sequentially but is carried out in roughly linear time in our implementation because we use n concurrent threads. This is an admittedly naive implementation strategy and a more intelligent nearest-neighbor detection method that does not require this type of all-to-all comparison approach would benefit the ClusterFlockGPU algorithm greatly.

Figure 31.4b shows that running time for ClusterFlockGPU is constant with respect to dimensionality. The reason for this is that after the pair-wise similarity values are computed in the early phase of the algorithm, the input data set is discarded. Therefore, the dimensionality of the input data set has no bearing on the later stages of the algorithm as only these similarity values are used later on.

To assess the speedup achieved by this GPU-based implementation, we created a sequential CPU-based version of ClusterFlockGPU and ran it on the same machine used to test our GPU version. Once again we use randomly generated 8-dimensional data sets for testing purposes. Timing data gathered from the CPU-based version along with the GPU-implementation is given in Fig. 31.5a, b.

We find a marked improvement in running time of the GPU version over the CPU implementation. Figure 31.6 shows the speedup achieved by the GPU implementation for varying sized data sets. The speedup begins to decrease the datasets increase in size. While we have not conducted an extensive study into the cause of this slowing behavior, we hypothesize that this is being caused by the latency in accessing the global memory locations where the boids' positions and velocities are being stored. As the number of boids increases, as does the number of required memory transactions and thus greater latency is introduced.

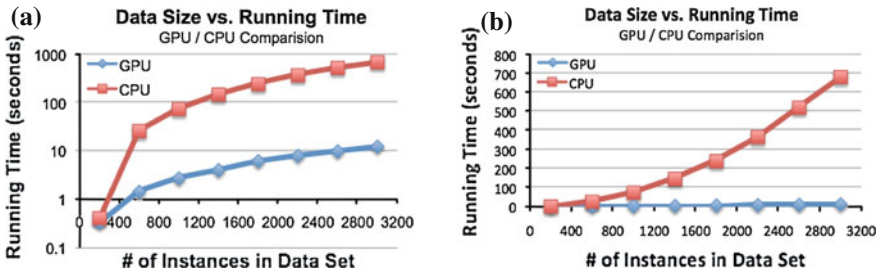


Fig. 31.5 Timing comparison of GPU and CPU implementations of ClusterFlockGPU

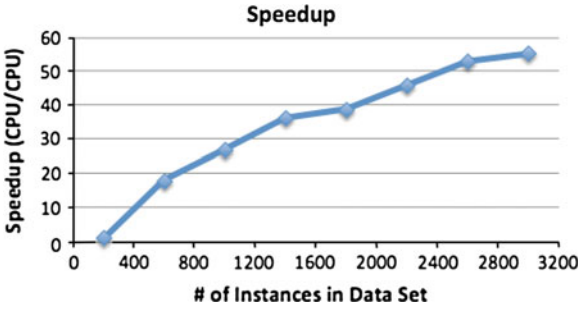


Fig. 31.6 The speedup with respect to data set size of the GPU-based implementation of ClusterFlockGPU over the CPU implementation

Synthetic Random Data Sets

Instances:	500	1000	2000	5000
# of Clusters	460	823	1526	3278

Fig. 31.7 The number of clusters found by ClusterFlockGPU in synthetic data sets of randomly generated 8-dimensional data

Figure 31.7 shows the number of clusters that are formed when ClusterFlockGPU is run on random synthetic data sets of varying size. We find that the number of clusters is large in each instance showing that no strong clusters are found in the data sets which indicates false clusters are not being found.

To compare the performance of ClusterFlockGPU to other partitional clustering algorithms we use the F-Measure, a commonly used measure of accuracy in the machine learning and information retrieval literature. Because ClusterFlockGPU is a stochastic algorithm, results can vary from run to run. For this reason, the values presented in Figs. 31.8 and 31.9 are averages over 10 independent runs on the given data set. Comparison values for the swarm intelligence based PSO, Ant-Based, and DSC clustering algorithms are taken from Merwe and Engelbrecht (2003), Handl et al. (2003), and Veenhuis and Köppen (2006) respectively. Results for k-means

Iris Data Set

	ClusterFlockGPU (Pearson Similarity)	PSO	Ant-Based	DSC	k-Means (Euclidean Similarity)
# of Clusters	3	3 (given a priori)	3.02	4	3 (given a priori)
F-Measure	0.966637	**0.970201**	0.816812	0.830683	0.824521

Fig. 31.8 Comparison of F-Measure for the Iris data set with various clustering methods. Comparison data is taken from Merwe and Engelbrecht (2003), Handl et al. (2003), and Veenhuis and Köppen (2006)

WBC

	ClusterFlockGPU (Euclidean Similarity)	PSO	Ant-Based	DSC	k-Means (Euclidean Similarity)
# of Clusters	5.4	2 (given a priori)	2	6	2 (given a priori)
F-Measure	0.736112	**0.976535**	0.967604	0.685732	0.965825

Fig. 31.9 Comparison of F-Measure for the WBC data set with various clustering methods. Comparison data is taken from Merwe and Engelbrecht (2003), Handl et al. (2003), and Veenhuis and Köppen (2006)

were generated with the WEKA software package. Because Merwe and Engelbrecht (2003), Handl et al. (2003), and Veenhuis and Köppen (2006) all provide results for the performance of their respective algorithms on the Iris and Wisconsin Breast Cancer (WBC) data sets, these are the data sets adopted for testing here. Both data sets were downloaded from the UCI Machine Learning Repository (Irvine 2010).

ClusterFlockGPU proves to be a very competitive method for clustering the Iris data set. We see however that on the WBC data set, the ClusterFlockGPU suffers from the same weakness that the DSC algorithm has, namely that too many clusters are formed. Also, we note that the performance of both ClusterFlockGPU and DSC is poor for the WBC data set. The reason for this is unknown but we posit that there is some characteristic of the "shape" of the WBC data set that makes it poorly suited for flock-based clustering.

31.7 Conclusion

In this work we explored the features of swarm intelligence systems that make them well-suited for GPU-based computing and presented our ClusterFlockGPU algorithm for partitional cluster analysis. The ClusterFlockGPU algorithm is able to produce slightly better results than the DSC algorithm on which it is based and achieves a nearly linear time complexity. Furthermore, we show that the algorithm is not affected by the dimensionality of the data being clustered making is well-suited for high-dimensional data sets.

The prospect of using GPU devices as the compute platform for swarm intelligence applications is indeed quite promising. By utilizing the massive multi-threading capabilities of GPU devices, swarm intelligence algorithms can feasibly achieve very large populations of agents. This aspect is particularly important for particle swarm and ant-colony swarm intelligence systems as these algorithms require a large number of explorations of solution space (and an equally large number of swarm agents) to achieve good results. With GPU-based implementations, ant-colony systems can support significantly larger ant populations and can theoretically achieve greater accuracy while also decreasing running time. An examination of the GPU-based approach to ant-colony optimization algorithms is provided in Weiss (2011).

Finally, we would like to point out that by allocating one or more threads per agent in a swarm intelligence system leads to an implementation that can more accurately reproduce the asynchronous nature of the systems which are the inspiration for swarm algorithms. The implications of being able to very accurately simulate large distributed and self-organizing systems are far reaching and deserving of further study.

References

Charles JS, Potok TE, Patton RM, Cui X (2007) Flocking-based document clustering on the graphics processing unit. NICSO, In, pp 27–37

Cui X, Potok TE (2006) A distributed agent implementation of multiple species flocking model for document partitioning clustering. Lect Notes Comput Sci 4149:124–137

Dorigo M, Caro GD (1999) Ant algorithms for discrete optimization. Artif Life 5:137–172

Dorigo M, Stützle T (1999) The ant colony optimization meta-heuristic. New ideas in optimization, McGraw Hill, London, pp 11–32

Dorigo M, Stützle T (2004) Ant colony optimization. MIT Press, Cambridge

Grosan C, Abraham A, Chis M (2006) Swarm intelligence in data mining. Stud Comput Intell 34:1–20

Handl J, Knowles J, Dorigo M (2003) On the performance of ant-based clustering. In: 3rd International conference on hybrid intelligent systems, IOS Press, Amsterdam, pp 204–213

Irvine UC (2010) Uci machine learning repository. http://archive.ics.uci.edu/ml/

Merwe DVD, Engelbrecht A (2003) Data clustering using particle swarm optimization. Proceedings of IEEE congress on evolutionary computation, IEEE, Canberra, In, pp 215–220

Olfati-Saber R (2004) Flocking for multi-agent dynamic systems: algorithms and theory. Technical report CIT-CDS 2004–005, California Institute of Technology

Palathingal P, Cui X, Potok TE (2005) Document clustering using particle swarm optimization. Special issue on Efficient heuristics for information, organization, (Special issue)

Parpinelli RS, Lopes HS, Freitas AA (2001) An ant colony based system for data mining: application to medial data. In: Proceedings of the genetic and evolutionary computation conference 2001, Morgan Kaufmann Publishers, San Francisco, pp 791–797

Reynolds CW (1987) Flocks, herds, and schools: a distributed behavioral model. Comput Graph 21(4), (Anaheim, California, ACM SIGGRAPH '87 Conference Proceedings), pp 25–34

Veenhuis C, Köppen M (2006) Data swarm clustering. Stud. Comput Intell 34:221–241

Weiss RM (2011) GPU-accelerated ant colony optimization. In: Wen-mei Hwu W (ed) GPU computing gems emerald, Chap. 22. Morgan Kaufmann Publishing. ISBN: 978-0-12-384988-5

Zhang Y, Mueller F, Cui X, Potok T (2011) Data-intensive document clustering on graphics processing unit (GPU) clusters. J Parallel Distrib Comput Data Intensive Comput 71(2):211–224

Chapter 32
Asynchronous Parallel Logic Simulation on Modern Graphics Processors

Yangdong Deng, Yuhao Zhu and Wang Bo

Logic simulation has become the bottleneck of today's integrated circuit (IC) design projects (Rashinkar et al. 2000). For instance, over 80 % of the IC design turn-around time of NVIDIA is spent on logic simulation even with NVIDIA's proprietary supercomputing facility (Huang 2010). It is thus essential to develop parallel simulation solutions to maintain the momentum of increasing IC integration capacity. Inspired by the supreme parallel computing power of modern GPUs (Blythe 2008), in this chapter we reported our recent work on using GPU to accelerate the time-consuming IC verification process by developing a massively parallel gate-level logical simulator. To the best of authors' knowledge, this work is the first one to leverage the power of the modern GPUs to successfully unleash the massive parallelism of a conservative discrete event driven algorithm, CMB algorithm (Chandy and Misra 1979, 1981; Chandy et al. 1979). Based on a novel data-parallel algorithmic mapping strategy, both the data structure and processing flow of the CMB protocol are re-designed to better exploit the potential of modern GPUs. A dynamic memory management mechanism is developed to efficiently utilize the relatively limited GPU memory resource. Experimental results prove that our GPU based simulator outperforms a CPU baseline event-driven simulator by a factor of 47.4X on average. This work demonstrates that the CMB algorithm can be efficiently and effectively deployed on GPUs without the performance overhead that had hindered its successful applications on previous parallel architectures.

Y. Deng (✉)
Institute of Microelectronics, Tsinghua University,
Beijing, China

Y. Zhu
Department of Computer Science, University of Texas,
Austin, TX, USA

W. Bo
Department of Electrical Engineering, Stanford University,
Stanford, CA, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 517
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_32,

32.1 Introduction

After almost 40 years' development, integrated circuits already become the most complex machines that people have made. However, we still see a strong drive for increasing integration capacity. One example is the recently released NVIDIA graphics processing units (GPUs), Fermi, which consist of 3 billion transistors. In fact, the pace at which GPU integration complexity increases is even higher than that predicted by the Moore's Law (Huang 2010). Since today IC architects depend on electronic design automation (EDA) tools, an essential requirement for EDA tools is to provide a scalable computing throughput so that we can maintain the momentum of IC functionality increasing.

Meanwhile, designing a cutting-edge IC is a time-consuming and thus costly process. A typical IC design project at the 45 nm technology node would cost around $20–50 M. Such a prohibitive cost implies that a first-silicon-success, i.e., a working IC after the first tape-out, is crucial for IC design companies, for most companies could afford a re-spin of the IC development project. To guarantee a first-silicon-success, IC designers have to perform intensive verification computations to validate if a design implementation correctly captures the design intention. In other words, IC verification engineers want to achieve a higher level of design coverage, although an exhaustive verification is already impractical.

With the above two factors combined together, verification has become the bottleneck of today's IC design process. Among various verification tools, logic simulation is the fundamental and indispensible means to evaluate various design options and verify the correctness of IC designs. The usage model of logic simulation is extremely simple: IC designers first construct a model of the design implementation usually in a hardware description language like SystemC and Verilog, apply input patterns to the primary inputs of the model, and then observe the outputs of the model. If the outputs have the expected value at proper moments, then the design implementation is correct (for the given input stimuli). Otherwise, there must be something wrong and designers have to investigate what causes the mismatch. Despite the simple usage model, logic simulation can be an extremely time-consuming process because the number of patterns required for an exhaustive simulation is an exponential function of the number of internal stages (i.e., number of register bits) plus the number of input pins. Today we are not pursuing complete simulation coverage, but the number of patterns to cover all major usage scenarios is still daunting. For instance, the gate level simulation of NVIDIA's Fermi GPU would take months even on NVIDIA's proprietary supercomputing facility (Huang 2010). In addition, the trend of integrating the whole system with multi-processors into a single chip (MPSoC) makes overall system verification even more difficult. In fact, logic simulation could take over 70 % of the total design time in a typical SoC design and 80 % of the NRE cost (Rashinkar et al. 2000).

A large body of research work has been devoted to accelerate the logic simulation process (Fujimoto 2000). Besides improving the efficiency of algorithms, parallel computing has long been widely considered as the essential solution to provide a

scalable simulation throughput. Unfortunately, logic simulation also has been one of the most difficult problems for parallelization due to the irregularity of problems and the hard constraints of maintaining causal relations (Bailey et al. 1994). Today commercial logic simulation tools offer multi-core CPUs and cluster based solutions by mainly exploiting the task level parallelism. Large simulation farms could consist of thousands of CPU nodes, which would be expensive and power hungry. Meanwhile, the communication overhead might finally overweigh the performance improvement through integration of more machines.

Recently graphics processing units (GPUs) are emerging as a powerful but economical high performance computing platform (Blythe 2008). With hundreds of small cores installed on the same chip, GPUs could sustain both a computing throughput and a memory bandwidth that are one order of magnitude higher than those of CPUs. On applications that can be properly mapped to the parallel computing resources, modern GPUs can outperform CPUs by a factor of up to a few hundreds (Blythe 2008). Motivated by the success of GPUs in many domains, several research teams have developed simulation solutions to unleash the power of GPUs for the logic simulation. Gulati and Khatri first presented a GPU based fault simulator, which used an oblivious algorithm where all gates are evaluated at every simulation cycle (Gulati and Khatri 2008). The oblivious simulation has the additional advantage of being capable to manipulate multiple input patterns in parallel. In Chatterjee et al. (2009a), an oblivious gate-level logic simulator was built on GPUs and achieved an average speed-up of tenfolds. The first GPU based event-driven logic simulator was introduced in Chatterjee et al. (2009b). This work utilizes a synchronous approach in which events happen in the same time step are evaluated in parallel.

Different from previous works using synchronous simulation protocols, we propose a GPU accelerated logic simulator based on an asynchronous event-driven algorithm, CMB algorithm, in this paper. We present a fine-grain mapping strategy to effectively assign simulation activities to the computing elements of GPUs for a higher level of available data level parallelism. The original CMB algorithm deploys a priority queue to store events for every gate and such a data structure incurs significant performance penalty due to the divergent branches. We develop a distributed data structure so that the events can be separately stored into FIFOs attached on pins. A dynamic GPU memory management is also proposed for efficient utilization of the relatively limited GPU memory. We also build a CPU/GPU cooperation mechanism to effectively sustain the computation power of GPUs. By combing the above techniques, we are able to achieve a speed up of 47.4X on GPUs on average. The contributions of this paper are summarized as follows.

- To the best of the authors' knowledge, this is the first GPU-based implementation of CMB algorithm for logic simulation.
- A dynamic GPU memory management mechanism is proposed for GPU processing. It supports dynamic memory allocation and recycling to guarantee efficient memory usage in spite of the irregular memory patterns that is inevitable in logic simulations. It can also be deployed in applications from other domains.

- We re-designed GPU friendly data structures for the CMB algorithm for efficient and effective mapping to the data parallel computing model of modern GPUs.
- We presented how the input dependency, logic depth, and type of gates can be incorporated in the re-ordering mechanism to reduce divergent processing in the simulation process.
- We described a closely coupled CPU and GPU simulation system in utilizing asynchronous execution and zero copy mechanisms in CUDA with a low overhead.

The remainder of this paper is organized as follows. Section 32.2 introduces the background of parallel logic simulation. Section 32.3 presents the basic processing flow of our GPU based logic simulator. Various optimization techniques that are essential for simulation throughput are outlined in Sect. 32.4. Section 32.5 presents the experimental results and detailed analysis. Related works are review in Sect. 32.6. In Sect. 32.7, we conclude the paper and propose future research directions.

32.2 Preliminaries

In this section, we first briefly present the algorithmic framework of parallel discrete event-driven simulation with a focus on the CMB algorithm.

Today the event driven simulation algorithm is the workhorse of virtually all real-world simulation engines (Fujimoto 2000). The event driven simulation algorithm is built on the basis of the event, which is composed of both a value and a *timestamp*. An event indicates that the simulation state would have a transition at the time of *timestamp*. Another important notion is the *physical time*. Different from the simulation *time*, the *physical time* is the time passed in the physical system that is being simulated. The logic simulation of VLSI circuits can be naturally performed by an event driven simulator. Generally, a logic simulator would use a global event queue to store all pending logic transitions. The event queue can be implemented as a priority queue so that the events are automatically chronologically ordered as they are inserted into the queue. At every simulation cycle, a logic simulator fetches the first event in the queue and performs evaluation according to the type of gate at which the event happens. The evaluation may create new events, which are again inserted into the event queue. The process is repeated until no events are available any more.

The event driven logic simulation is a very efficient sequential algorithm. However, the parallelization of the event driven logic simulation algorithm turns out to be challenging (Chandy and Misra 1979). In fact, it could not extract sufficient parallelism by simply evaluating events happened at the same timestamp (Chandy and Misra 1979). The concept of distributed time, event driven simulation, was developed by Chandy and Misra (1979) and Bryant (1977) and designated as the CMB algorithm. Such an approach is based on the concept of *logic processes* (*LP*). In a parallel simulation process, different modules of a simulated system are abstracted as logic processes. A *LP* maintains its local simulation time and could have several

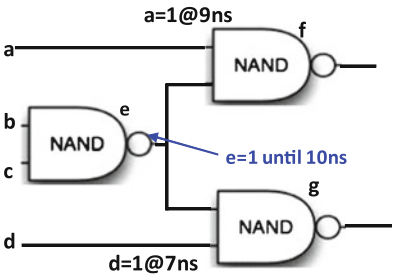


Fig. 32.1 The basic concept of CMB simulation algorithm

inputs and an output. At each simulation step, a *LP* may receive several new events from its inputs and generate an event at its output. The key idea of CMB algorithm is that different *LPs* could independently push forward their evaluation as long as the causal relations are not violated. Figure 32.1 is an illustration of the CMB algorithm. In the 3-gate circuit, the gate e has a pending event at 10 ns and this fact is known to gates f and g. Meanwhile, input pin a has an awaiting event at 9 ns, while input pin d has one at 7 ns. Obviously, since e would not change until 10 ns, the states of f and g would be completely determined by a and d before 10 ns. In addition, gates f and g can be evaluated because a and d are independent. In other words, the safe evaluation time, $T_{min,}$ of gates f and g are 10 ns. After evaluating the events of a and d, the local time of gates f and g would be 9 and 7 ns, respectively. The CMB simulation could run into a deadlock due to the distributed time mechanism (Chandy and Misra 1979). The most commonly used approach to prevent deadlock is through the usage of null events (Bryant 1977).

The parallel, distributed time simulation algorithms fall into two categories, conservative and optimistic. The conservative approach always selects the earliest event to evaluate, which means that no event with a timestamp smaller than the evaluated ones would be received in the succeeding simulation. This protocol enforces the causal relation of the simulation. On the other hand, the optimistic approach may process events whose timestamps may be larger than those of later events. When a prior evaluation turns out to be incorrect, a rollback mechanism or a reverse computation will be initialized to restore the results before the incorrect evaluation. In this paper, we adopt the conservative approach to avoid the complex flow control of the optimistic approach that would be inefficient to implement on GPUs.

To avoid deadlock in the CMB algorithm, we adopt the commonly used technique of null events. When a LP, A, would not generate new events (i.e., logic value remains the same) after an evaluation, it will instead send a null event with timestamp T_{null}. When another LP, B, receives the null event, its simulation time can then proceed to T_{null}. It can be formally proved that the CMB algorithm can correctly proceed until completion with the help of null messages (Chandy and Misra 1979).

32.3 Asynchronous Parallel Logic Simulation

This section explains the details of our GPU-based massively parallel simulator. First the basic simulation flow is presented. Next we show how to sustain a high level parallelism through carefully manipulating the communication of messages. In the section, we propose a novel technique for dynamic memory management on GPUs and adapt issuing of input pattern according to the memory pages utilization. They are essential to guarantee a robust and efficient simulation flow for real VLSI designs.

32.3.1 General Framework

During the process of parallel simulation, the input circuit is converted into an internal representation, in which every gate is mapped to a logic process (LP). During simulation, a LP receives events from their inputs, performs logic evaluations, and then creates and sends events to outputs. In such a manner, the simulation process proceeds like waves travel forward. In order to handle both internal and I/O signals of a circuit in a uniform manner, every PI (primary input) is treated as a virtual gate. The simulation flow is organized as three consecutively executed primitives, *extract*, *fetch*, and *evaluate*. First, virtual gates *extract* pending input patterns and insert them into their corresponding event queues. Logically these stimuli are equivalent to the outputs of virtual gates. Next, all gates (including virtual gates) send the output events to the queues of the pins at the succeeding level. This phase can also be regarded as a *fetch* primitive from the viewpoint of an input pin. In real designs, we use fetch rather than send to extract more parallelism, because the number of pins is much larger than that of gates. Finally, in compliance with timing orderings, the real gates (i.e., all gates excluding virtual gates) *evaluate* the inputs to generate new events and add them into the event queues of the corresponding output pins.

Although the above three primitives exhibit sufficient parallelisms that are suitable for GPU implementation, two problems remain to be resolved. First, the workload has to be efficiently distributed to the hundred of cores installed on a graphic processor such that the fine-grained data level parallelism can be effectively exposed. Second, an efficient memory management mechanism is essential, because GPUs are only backed up by a limited capacity of memory storage. To address the first problem, every pin (gate) is assigned to a single thread in the *fetch* (*evaluate*) kernel. In the evaluate kernel, each thread actually serves as a LP. For the second problem, a dynamic memory management mechanism is indispensable to best utilize the memory resource. Because the current CUDA releases only offer preliminary support for flexible memory management, we designed an efficient paging mechanism for memory allocation and recycling. To further cushion the pressure on the memory usage, the issuing rate of input stimuli can be determined in an adaptive manner by considering the memory occupancy. The overall GPU based simulation flow is

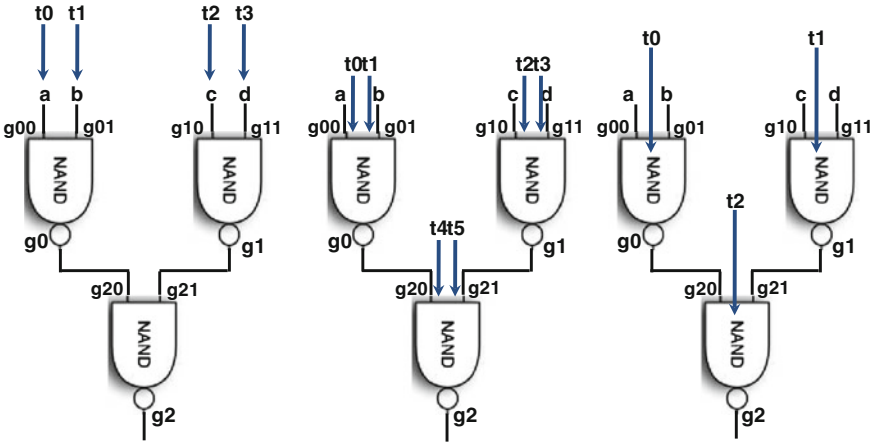
Fig. 32.2 GPU based logic simulation flow

```
while completion requirements not meet do
for each PI do
    if memory to allocate is enough
        extract the stimuli to the PI output pins
    else
     issue null message the PI output pins
end for each
for each input pin do
    fetch messages from output pins
end for each
allocate memory if needed
for each gate do
    insert new events from its input pins to the event FIFOs
    evaluate the earliest message in its event FIFOs
    write evaluation result to the output pin
end for each
release memory if possible
end while
```

outlined in Fig. 32.2. The "**for each**" structure indicates that the following operation would be executed in parallel.

In Fig. 32.3, the fine-grained mapping mechanism is illustrated with a simple example circuit consisting of three 2-input NAND gates, *g0*, *g1*, and *g2*. The four primary inputs are *a*, *b*, *c*, and *d*. The input pins for a given gate *gi* are labeled as *gi0* and *gi1*, where $i = 0, 1,$ or 2. The arrows indicate where the processing happens in each primitive. In the *extract* primitive, the primary inputs are concurrently handled by multiple threads labeled as *t0–t4*. In the *fetch* primitive, each thread handles a different input pin. And finally in the *evaluate* primitive, one thread is assigned to every gate.

List 1 is the code snippet of the main loop of logic simulation. The three primitives are repeatedly called until a given number of cycles have been simulated. A dynamic memory allocation operation is performed before the *fetch* primitive, while a memory releasing is called after the *evaluate* primitive. The dynamic memory operations will be explained in Sect. 32.3.3. Sample code for each primitive is listed in the appendix.



(a) Extract primitive **(b)** Fetch primitive **(c)** Evaluate primitive

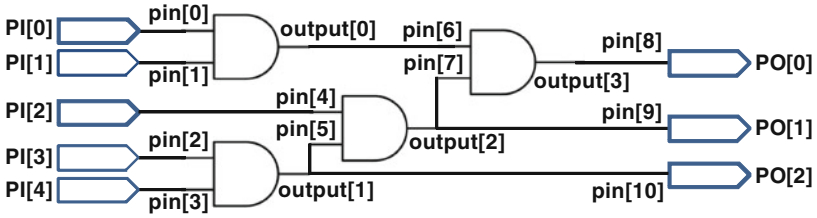
Fig. 32.3 Fine-grain mapping of workload. **a** Extract primitive. **b** Fetch primitive. **c** Evaluate primitive

```
while(current_run < NUM_RUN){
    //prepare data to device
    …

    //Extract primitive – get events on primary inputs
    opoq_sim_update_pi<<<update_pi_grid, update_pi_threads>>>(d_pi_input_event, d_gate_output_event,
                            pause_flag, num_pis, d_pi_head, d_pi_size, pi_pitch);
    cudaThreadSynchronize();

    //allocate memory pool on GPU for future dynamic memory operations
    alloc_cell(alloc_cell_array, &avail_cell_fifo, num_pins);

    //Fetch primitive – update pins for each gate
    opoq_sim_update_pin<<<update_pin_grid, update_pin_threads, 0, stream[0]>>> (d_gate_output_event,
                                d_pin_input_event, d_gate_clock_value, d_pin_tail_cell, d_pin_tail_offset,
                                d_pin_size, d_msg_array, d_alloc_cell_array, num_pins, pin_pitch, d_input_id,
                                d_input_inv_ptr, d_pin_in_gate);
    cudaThreadSynchronize();

    //Evaluate primitive – logic evaluation
    opoq_evaluate<<<evaluate_grid, evaluate_threads, 0, stream[1]>>> (current_run, d_pin_input_event,
                                d_gate_clock_value, d_gate_val, d_gate_output_event, d_msg_array,
                                d_release_cell_array, d_pin_head_cell, d_pin_tail_cell, d_pin_head_offset,
                                d_pin_tail_offset, d_pin_size, num_gates, num_pis, num_pins, pin_pitch,
                    d_input_ptr, d_gate_type, d_gate);
    cudaThreadSynchronize();

    //release unused GPU memory
    release_cell(release_cell_array, &avail_cell_fifo, num_pins);
}
```

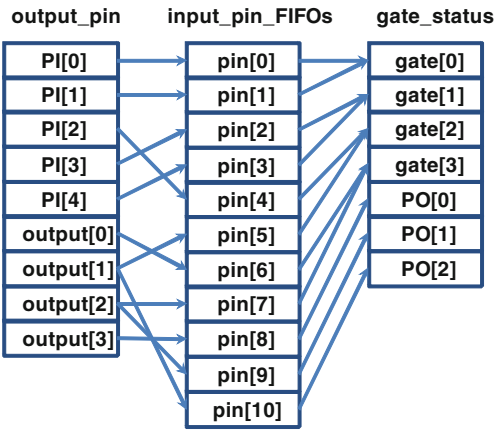
32.3.2 Data Structures for Fine-Grained Message Passing

The CMB algorithm is an event-driven algorithm and thus intrinsically suitable for a message-passing model, such as MPI (2000). However, modern GPUs are based on a shared memory architecture where several multiprocessors uniformly access the same global memory. Therefore, we emulate a message passing mechanism in our simulator through manipulating three arrays, namely *output_pin*, *event_queue* and *gate_status*. The *output_pin* array reserves an entry for the output pin of every gate and stores events generated during gate evaluation. Note that the events for virtual gates are extracted from the input stimuli in the extract primitive rather than from evaluation. The *event_queue* array stores the events to be evaluated by a gate. The *gate_status* array records related information for each gate.

Figure 32.4a is a sample circuit whose corresponding data structures are illustrated in Fig. 32.4b. At the beginning of simulation, the virtual gates, i.e., primary inputs, extract stimuli into the corresponding portion of *output_ pin*. Then all events stored in *output_ pin* are written into event_queue to emulate one round of message passing.



(a) A sample circuit



(b) The corresponding data structure of the circuit in (a) for parallel logic simulation

Fig. 32.4 Sample circuit and its simulation data structures. **a** A sample circuit. **b** The corresponding data structure of the circuit in (**a**) for parallel logic simulation

Finally, the event with the earliest timestamp T_{pin} is chosen from its *event_queue*. As formulated in the original CMB algorithm, if T_{pin} is smaller than Tmin, that event could be safely fetched for evaluation.

To maintain the ordering of the events passed to a gate, the event_queue mentioned above has to be a priority queue. However, it is extremely challenging to find an efficient GPU implementation for managing a heap based priority queue due to the large number of branches incurred by heap operations. To avoid this bottleneck, we decompose the event_queue of a gate into several FIFOs, with one FIFO for an input pin. Such FIFOs are designated as input_pin_FIFOs. The advantages of such an approach are trifold. First, by decomposing the gate-wise priority queue into a group of light-weight distributed FIFOs, the insertion operation for a priority queue is much simpler. In fact, those events arriving at a given input pin are naturally chronological and thus newly arrived events can be safely appended at the end of the FIFO. Otherwise, the insertion operation of a priority queue would incur serious performance overhead because different queues would have different structures, which would inevitably lead to diverse program execution paths would vary. Meanwhile, the distributed FIFO mechanism enhances the potential parallelism because the insertion operations for different FIFOs are independent. Otherwise, a centralized, gate-wise storage would result in resource contention and restrict the maximum parallelism to be the number of gates. Additionally, if a newly arrived event has the same event as the latest one in the FIFO, it will be discarded and only the last-come event timestamp is updated. This mechanism greatly reduces the number of active events in the simulation and is proved to be essential in the experiments.

32.3.3 Memory Paging Mechanism

The limited GPU memory would pose an obstacle for the simulation of large-scale circuits. First of all, industry-strength, large-scale circuits would require considerable space to store necessary information. In addition, a significant number of events would be created and sent in the simulation process. Such events also need to be buffered in memory before they are processed. Current GPUs do not support dynamic memory management features such as *malloc*()/*free*() in C language and *new*()/*delete*() in C++. Accordingly, a straightforward implementation would allocate a fixed size for the FIFO, i.e., *input_ pin_FIFO*, for each input. However, there is no easy way to optimally determine the FIFO size before runtime. A smaller FIFO size would lead to frequent FIFO overflow, while a larger size suffers low utilization of memory storage. To overcome this problem, we introduce a memory paging mechanism to efficiently manage the GPU memory.

The feasibility and advantage of such a dynamic memory management mechanism is justified by two basic observations in the CMB based simulation experiments. First, we found that the numbers of events received at different pins would fluctuate dramatically. A relatively small number of pins are very “hot” in the sense that they receive many more events than those “cold” ones by orders of magnitude.

This fact certainly suggests that a similar non-uniform pattern should be applied to the allocation of FIFO size on each input pin to avoid both starvations and overflow. The second key observation is that, in different period, the occupation of a FIFO varies noticeably. Thus, the management should be able to recycle the memory storages.

Based on these observations, we designed a paging mechanism for dynamic management. A large bulk of memory *MEM_SPACE* with *MEM_SIZE* bytes is allocated on GPU before the simulation. It is uniformly divided into small memory pages with a constant size of *PAGE_SIZE*, each of which can hold up to *EVENT_NUM* events. A page is the minimum unit of memory management and each page is uniquely indexed in a continuous manner. This memory space is reserved for the *input_ pin_FIFO* arrays of all the inputs pins. Initially, a specific number of pages, INIT_PAGE_NUM, are allocated to every FIFO of input pins. Indexes of all remaining pages are inserted into a global FIFO called *available_ pages*, which is used to record the index of available pages. As the simulation process moves on, preallocated pages are gradually consumed. As a result, free pages are needed to store newly created events, while pages storing processed events can be recycled. In our simulator, we deal with the requests for allocating new pages and releasing useless pages after every execution of fetch and evaluate primitives, respectively.

The FIFO structure for a pin is defined as Fig. 32.5. The *page_queue* stores the indexes of pages that have been allocated to that pin. The events are extracted and stored in *MEM_SPACE* according to the FIFO pointers. As illustrated in Fig. 32.6, the locations of first and last events in the FIFO are pointed by *page_queue[head_ page] * PAGE_SIZE + head_offset* and*page_queue[tail_ page] * PAGE_SIZE + tail_offset*, respectively.

A few additional data structures are needed to guarantee the correct working of the paging mechanism. Two index arrays, *page_to_allocate* and *page_to_release*, are employed to support page release and allocation. Every pin has an entry in both arrays. Elements in *page_to_allocate* denote the page indexes that are to be allocated to corresponding pins, while elements in page_to_release hold the page indexes of those pins that can be released. When pin *i* runs out of pages, the GPU kernel fetches the *i*th value in *page_to_allocate* and inserts it into pin *i*'s *page_queue*. Similarly, when events on a page of pin *i* are all processed, GPU kernel writes the index of that page to the *i*th entry of *page_to_release*, and pops that page index from *page_queue*. Because all events information is stored in *MEM_SPACE*, no explicit copy has to

Fig. 32.5 FIFO data structure for GPU dynamic memory management

```
typedef struct{
    unsigned int   size;
    unsigned int   *page_queue;
    unsigned int   head_page;
    unsigned int   head_offset;
    unsigned int   tail_page;
    unsigned int   tail_offset;
}FIFO_T;
```

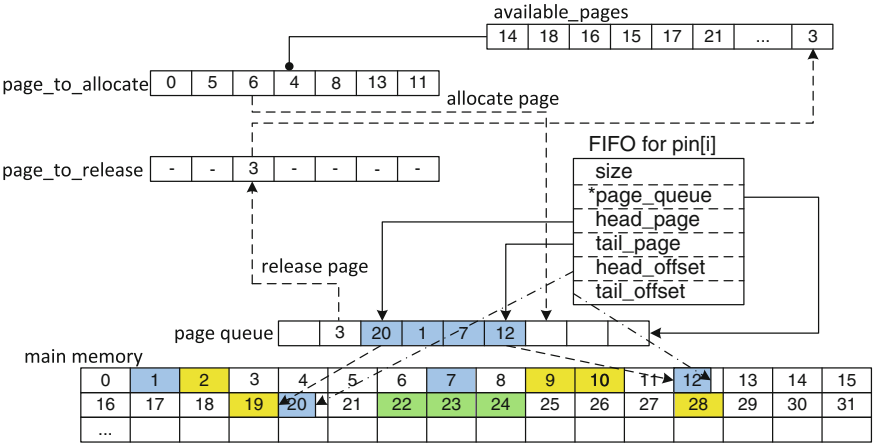


Fig. 32.6 Illustration of a sample GPU memory layout

be performed for release and allocation. The only overhead is introduced when the control flow returns to the CPU side. At that time, the host thread would recycle the pages in *page_to_release* by inserting them into *available_ pages*, and designate pages to be instantly allocable by writing indexes of those pages into related entries in *page_to_allocate*. The update of these two arrays to the *available_ pages* is done sequentially on the host CPU. Further improvement of this step will be discussed in the next section.

Finally, in the experiments, we found that under proper conditions, events can be created at a much higher rate than that of evaluation. Under such circumstance, all pages in *MEM_SPACE* may be occupied by events that have not been evaluated. This would prevent the new input stimuli to be issued due to the lack of memory space. To attack this problem, we propose an adaptive strategy to control the issuing speed of input stimuli.

Before issuing stimuli, one host side thread is responsible for checking the number of allocable pages residing in *available_ page*. If the number is under a pre-define threshold, the issue of stimuli is paused. Instead, null events are issued with the same timestamp as the last valid event. These null events will not be inserted into pin FIFOs but can push forward the simulation process. Those events ready to be evaluated will be processed simultaneously and some pages may be released. When the number of obtainable pages in *available_ page* is once again above the threshold, normal issue would continue. In this work, we choose a relatively conservative threshold value such that it is no smaller than the number of pins since it is possible that all pins require a page in next simulation iteration.

32.4 GPU Oriented Performance Optimization

The techniques presented in Sect. 32.4 provide a basic flow for a massively parallel logic simulator. However, performance tuning and optimization are essential to successfully unleash the computing power of GPUs. For GPUs, major hurdles for computing performance include divergent program execution paths, communication overhead between CPU and GPU, as well as un-coalesced memory access. In this section, we discuss a set of optimizations to improve the simulation throughput.

32.4.1 Gate Reordering

The efficient execution of a GPU program requires the data to be organized with good locality and predictability. When the threads are organized into warps, ideally these threads should take the same actions. However, gates in an input circuit are generally arbitrarily ordered and the ordering is used to assign gates to threads. As a result, generally threads in a warp would follow different instruction paths when performing the table lookup for gate delay and logic output value as well as the calculation of T_{pin}.

To reduce the divergent paths, we proposed a gate re-ordering heuristic according to circuit information. The ordering process can be performed prior to simulation. Gates are sorted and then ordered with regard to gate types and number of fan-ins. When the gates are sequentially assigned to threads, threads in a warp are very likely to follow the same execution path in the *evaluate* primitive. Our experiments show that this heuristic could enable an additional speedup up to 1.5X on some designs.

32.4.2 Minimizing Divergent Execution Paths

Although hardware techniques like dynamic warp formation (Fung et al. 2007) have been proposed to minimize the overhead of intra-warp divergent branches, current off-the-shelf GPUs are still sensitive to flow control instructions such as *if*, *switch* and *while*. These instructions would potentially lead to a serialization of parallel threads. As stated in Sect. 32.3.2, the design of the fundamental data structure already mitigates the divergent path problem by replacing the gate-wise priority queue with multiple distributed pin-wise FIFOs so as to avoid the insertion of priority queues.

32.4.3 Hiding Host-Device Interaction Overhead

The only interaction between host and device threads during simulation is introduced by the paging mechanism. To dynamically allocate and recycle GPU memory, it requires explicit copy of data (*page_to_allocate* and *page_to_release*) and the corresponding sequential processing in every iteration. The resultant overhead is

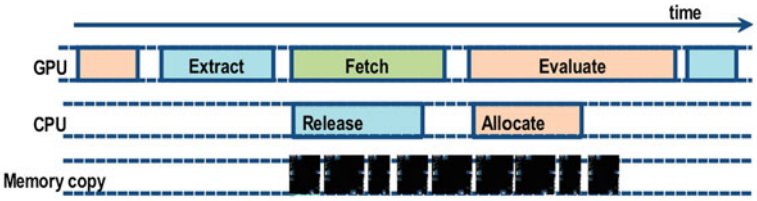


Fig. 32.7 Illustration of ideal overlap

twofold. First, data transfer between GPU and CPU is through a PCI Express (PCI-SIG 2010) interface, which usually cannot meet the consumption speed of the GPU hardware. The second overhead is due to the sequential processing on the CPU. In fact, Amdahl's Law (1967) suggests that sequential part of a program could severely drags down the performance of a parallel program. In fact, the CPU processing might overweigh the performance gain offered by the massively parallel execution on the GPU.

Certainly the ideal situation would be to overlap the three parts as illustrated in Fig. 32.7. Such an overlapping is indeed possible through a careful organization of the primitives. A key observation is that the *fetch* primitive that inserts events into event FIFOs only need to execute the memory allocation operations when the allocated pages for that FIFO are full. In contrast, the *evaluate* primitive that evaluates events from FIFOs just requires the memory release operations, because some allocated pages may have be completely processed. Therefore, the CPU processing of *page_to_allocate* and the *evaluate* kernel can be overlapped. The same goes with the processing of *page_to_release* and the *receive* kernel. This can be straightforwardly realized with the *asynchronous execution* feature provided in CUDA. We create two streams and assign *fetch*/*release* within one stream, and *evaluate*/*allocate* in another. In this way, the *receive* primitive is executed on GPUs while the *release* operation runs on the CPU host simultaneously. The *evaluate* primitive and *allocation* operation work in the same way. Because the amount of data transferred is relatively small, we take advantage of a new feature called *zero copy* NVIDIA (2009) released in CUDA 2.2 and later version. With zero copy technique, data transfer is invoked within GPU kernels, rather than on the host size. Zero copy has the advantage of avoiding superfluous copies since only the request data will be transferred. Another advantage is that it also schedules those kernel-originated data transfer implicitly along with the kernel execution, without the need for explicit programmer interventions.

To further overlap CPU computation time with GPU processing time, we propose a *group flag* strategy. Every 32 (i.e. the warp size in CUDA) pins are aggregated into a group. Every group has two flags, one for release, and one for allocation. During GPU processing, if any pin in the group requires a new page or releases a useless page, corresponding flags are marked. Therefore, CPU thread only needs to check those groups of threads whose flags are marked and skip those unmarked ones. This strategy functions like a skip-list and proves to be important for a high level of efficiency.

32.4.4 Enhancing Memory Access Efficiency

As suggested in CUDA Best Practice Guide (NVIDIA 2009), one of the top concerns for CUDA programming is how to guarantee the efficiency of memory access. Accordingly, we perform extensive optimizations on the memory usage. For instance, a FIFO has a set of descriptive data organized as a structure with multiple members such as *head* pointer, *tail* pointer, *size* value, and *offset* value. To avoid the structure accesses that are hard to coalesce, we actually use a structure of array (SOA) to store the data. In addition, we minimize the access to long-latency global memory by storing read-only data in constant and texture memories, which are both supported by on-chip cache. The circuit topology related data, which can be determined before simulation, are loaded into texture memory. Static circuit data structures like truth table are located in constant memory by declaring them as *__constant__*. With these storage patterns, the penalty of irregular memory accessing is significantly reduced.

32.5 Results and Analysis

In this section, we report the experimental results of our GPU based logic simulator. Starting with the experimental setup, we present experimental observations that justify our choices of optimization techniques. Then a detailed performance evaluation is introduced.

32.5.1 Experimental Framework

We choose ten open-source IC designs from OpenCores.org (OpenCores 2010) and ITC99 Benchmarks (2nd Release) (ITC99 Benchmarks 1999) to test our GPU based simulator. Critical parameters of these circuits are listed in Table 32.1. Among these designs, b18 was released as a gate level netlist. The other nine designs were downloaded as RTL level Verilog code and then synthesized with Synopsys Design Compiler using a 0.13 μm TSMC standard cell library. Similar to the common industry practices, we use two sets of simulation stimuli, deterministic test patterns released with the design test bench and randomly generated patterns. Usually the deterministic patterns can be used to validate if a circuit delivers the expected behavior, while the random patterns are used to cover possible design corners. For randomly generated stimuli, we set the maximum simulation time as 250,000 cycles. Every 5 cycles would the random generator create a new true or false value so that valid stimuli actually occupy 50,000 cycles.

To evaluate the performance of our GPU simulator, we also hand-coded a baseline CPU simulator as a reference for comparison. The baseline simulator uses a classical event-driven algorithm instead of CMB algorithm, because CMB algorithm is

Table 32.1 Statistics of simulation benchmarks

Design	#Gates	#Pins	Description
AES	14511	35184	AES encryption core
DES	61203	138419	DES3 encryption core
M1	17700	42139	3-stage pipelined ARM core
SHA1	6212	13913	Secure Hashing algorithm core
R2000	10451	27927	MIPS 2000 CPU core
JPEG	117701	299663	JPEG image encoder
B18	78051	158127	2 Viper processors and 6 80386 processors
NOC	71333	181793	Network-on-chip simulator
LDPC	75035	148022	LDPC encoder
K68	11683	34341	RISC processor

Table 32.2 Distribution of number of events received by pins

#Events	AES	DES	M1	SHA1	R2000	NOC	K68	b18
0–99	34444	138317	39708	13591	26106	157202	32669	158086
100–9999	737	102	2431	321	1764	24591	1672	40
≥10000	3	0	0	1	3	0	0	1

intrinsically slower on sequential machines due to the extra work of message passing and local time maintaining (Soule and Gupta 1991). Our baseline simulator is up to 2X faster than Synopsys VCS simulator (Synopsys 2010), because such commercial simulators have to handle complex verification computations that incur performance overhead.

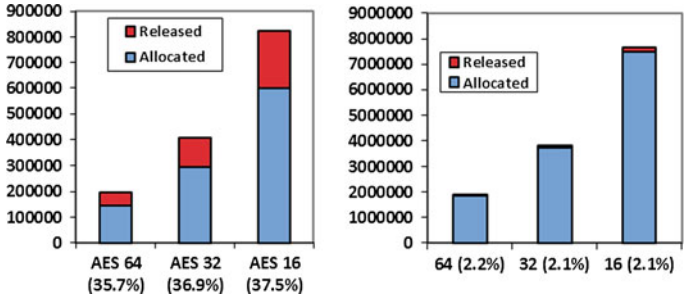
32.5.2 Justification of the Optimization Techniques

We mentioned in Sect. 32.3.3 that there exist significant variations in the events received by each pins. This fluctuation can be several orders of magnitude across one specific design. Detailed statistics for a group of benchmark circuits are listed in Table 32.2. In Table 32.2, pins are grouped into three categories with regard to the peak number of event that they could receive during a complete run of simulation. The data are collected after 50,000 simulation cycles using randomly generated input stimuli. It can be seen from the table that the activity intensities of different pins are under dramatic deviation, which implies that the dynamic paging mechanism is imperative.

We also conducted experiments to quantitatively analyze the efficiency of GPU memory paging. The page recycling algorithm is detailed in Sect. 32.3.3 Here we choose four circuits to apply deterministic input pattern and five circuits (marked with a letter 'R') to exert randomly generated stimuli. Table 32.3 quantifies the numbers of pages released and allocated when simulating each circuit. The 4th column is the ratio of released pages to allocated pages.

Table 32.3 Memory page recycling

Design	Allocated pages	Released pages	Recycled ratio (%)
AES	145085	51793	35.69
DES	429312	93312	21.74
R2000	27336	1938	7.09
JPEG	1045482	46292	4.43
SHA1 (R)	245596	14355	5.84
M1 (R)	1868242	40075	2.15
b18 (R)	45742	21935	47.95
NOC (R)	2072609	167061	8.06
K68 (R)	1430792	128857	9.00



(a) AES with released stimuli **(b)** M1 with random stimuli

Fig. 32.8 Page recycling with different configurations of $PAGE\_SIZE$. **a** AES with released stimuli. **b** M1 with random stimuli

The data in Table 32.3 are collected by configuring that *PAGE_SIZE* as 32 bytes. Circuit topology and stimuli pattern would largely affect the progress of simulation, and thus the consumption of memory pages, therefore some cases may recycle pages frequently while others exhibit a much inactive behavior. On average, about 14.2 % pages could be recycled. While the *PAGE_SIZE* decreases, the number of recycled pages would increases as illustrated in Fig. 32.8 where the Y axis represents the number of page. But there is no noteworthy difference in the relative ratio, because both released and re-allocated pages vary in a similar trend. However, a small page size means that more pages are need. A larger number of pages in turn results in more frequent page recycling and would potentially drag down the overall performance. On the other hand, a large page size enlarges the granularity of the page recycling process and thus lowers the flexibility of memory management. The extreme situation would be so large a *PAGE_SIZE* that every pin can only have one page. Considering the contradicting impacts, a *PAGE_SIZE* of 64 offers a balanced solution empirically. Finally, we justify the overlap of CPU processing, GPU processing, and memory copy. A decomposition of CPU time consumed by these three components is illustrated in Fig. 32.9. The most significant observation is that memory copy account for most of the whole simulation time. This necessitates the use of *zero copy* to reduce

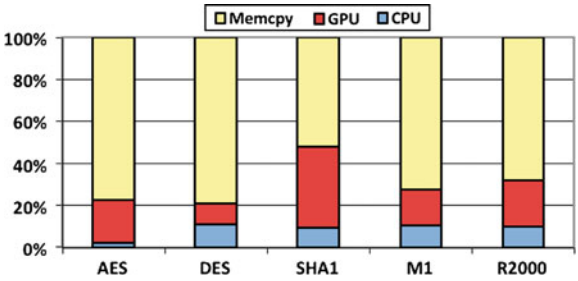


Fig. 32.9 Detailed timing analysis

redundant data copy and overlap computation with communication. Besides, CPU processing plays a non-trivial role in all designs. In DES, CPU time even outweighs GPU part. Therefore, we need to not only overlap CPU processing with GPU computation, but also employ the *group flag* mechanism to further reduce the overhead on CPU in achieving an optimal performance.

32.5.3 Performance Evaluation

In this section, we compare the performance of our GPU simulator against the CPU simulator described in Sect. 32.6. The CPU baseline simulator is compiled by gcc 4.2.4 with—O3 optimization. The GPU based simulator reported in this paper is realized in CUDA 2.2 version. The experiments were conducted on a 2.66 GHz Intel Core2 Duo server with an NVidia GTX 280 graphics card. We present results on both deterministic and random stimulus. Table 32.4 reports results of all ten designs under randomly generated input patterns. Because some designs (K68 and b18) have no testbench released, and other two designs (LDPC and NOC) are inherently tested by random patterns. Table 32.5 only reports six designs simulated under the released deterministic stimuli. Thanks to the highly parallel GPU architecture described in Sect. 32.2, the GPU simulator attains an average speed of 47.4X on average for random patterns, and up to 59X speedup for four designs with deterministic patterns. Meanwhile, there do exist two designs on which the GPU simulator is slower under deterministic patterns and the reasons will be explained in the next section.

32.5.4 Relation Between Speed-up and Activity Intensity of Input Patterns

It is worth-noting that GPU based simulator is slightly slower than its CPU counterpart for M1 and SHA1 when applying the deterministic stimuli released to the designs. However, when applying random patterns, a significant speedup can be observed. In fact, for all six designs listed in Table 32.5, the speedup on random stimuli is much more significant. The reasons are twofold. First, we find that the

Table 32.4 Simulation results with randomly generated stimuli

Design	Baseline simulator (s)	GPU based simulator (s)	Speedup (column 2/column 3)
AES	676.96	9.45	71.64
DES	881.68	16.38	53.83
M1	88.85	8.57	10.37
SHA1	51.48	5.57	9.24
R2000	223.93	8.33	26.89
JPEG	10030.64	37.11	270.29
LDPC	51.91	10.41	4.99
K68	54.46	6.74	8.08
NOC	43.32	9.71	4.46
b18	299.92	21.70	13.82

Table 32.5 Simulation results with officially released stimuli

Design	Baseline simulator (s)	GPU based simulator (s)	Simulated cycles	Speedup (column 2/column 3)
AES	90.50	5.01	42,935,000	18.06
DES	17.38	8.06	307,300,000	2.16
M1	13.56	22.11	99,998,019	0.61
SHA1	0.33	0.42	2,275,000	0.79
R2000	4.594	0.937	5,570,000	4.90
JPEG	2121.71	35.71	2,613,200	59.42

deterministic stimuli are applied in a rather "sparse" manner. For example, with the deterministic stimuli of M1, input values only receive new values in 176,500 cycles (total 99,998,019 cycles simulated). In other words, the primary inputs receive new patterns in about every 500 cycles. On the other hand, new assignments are applied very frequently in the random patterns (every 5 cycles for the results reported in Table 32.4). The simulator is thus kept much busier by the random stimuli. Second, when the deterministic input patterns are applied, one pin would receive the same value consecutively. Again taking circuit M1 as an example, a large number of pins repeatedly receive the same true or false value throughout the simulation. As described before, events with the same value as the previous one will not be added into FIFO. So the total number of events that are to be evaluated is actually even more "sparse". With randomly generated stimuli, however, pins would randomly receive a new assignment. As a result, the number of events would be significantly higher. Considering the above two factors as well as the parallelization overhead, it is not strange that GPU simulator would be sometimes slower on deterministic patterns, but still could achieve a dramatic speed-up on random patterns. In other words, the lack of activities in the deterministic input stimulus for M1 and SHA1 is the major reason for the insufficient acceleration. In fact, when using a GPU simulator, IC designers could use high-activity testbenches to extract sufficient parallelism for an even higher throughput as exemplified in Fig. 32.10. On the benchmark circuits AES,

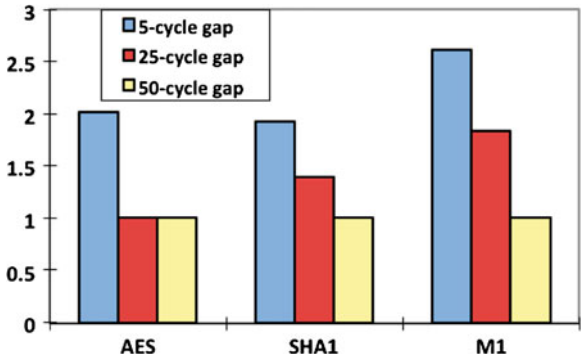


Fig. 32.10 Speed-up of our GPU simulator over CPU baseline with varying input activity intensities

SHA1, and M1, we created 3 sets of random test patterns by gradually reducing the gap between two patterns from 50 to 5 cycles. We normalize the relative speedup to the 50-cycle one and show them in Fig. 32.10. Clearly, when the input patterns have a higher level of activity, the GPU would perform better with a larger acceleration ratio.

32.6 Related Work

There are two major approaches to perform simulations: *oblivious* and *event-driven* (Fujimoto 2000). The oblivious simulator evaluates all gates at every clock cycle, while the event-driven one only evaluates those whose inputs have changed. Though oblivious simulation is simple, it is not adopted in practice. Due to the fact that only a small fraction of gates are active, i.e. the inputs of them change, every clock cycle in a large design, event-driven approaches are the most appropriate and practical methods. In fact, most current commercial simulators are based on event-driven approaches.

Considering the enormous time required for the simulation of large VLSI circuits, much emphasis has been paid to accelerate logic simulation by leveraging parallel and distributed computer architectures. Modern parallel logic simulators are typically based on event driven simulation algorithms, which can be divided into two basic categories: synchronous algorithms and asynchronous algorithms. In a synchronous simulator, multiple LPs progress in a step-locked manner as coordinated by a global clock. The events are stored in a global data structure for shared access. Such synchronous algorithms can be implemented in SIMD architectures (e.g., Bataineh et al. 1992) in a straightforward manner. To extract a higher level of parallelism, an asynchronous simulator like the one implemented in this work assigns every LP with a local clock. The basic parallel simulation protocol was first proposed by in Chandy

and Misra (1979) and Bryant (1977), and thus designated as the CMB algorithm. Each LP asynchronously evaluates its events as long as no causal relations would be violated. The overall asynchronous flow is amenable to parallel implementations because the synchronization overhead can be avoided. However, it poses challenges for a SIMD implementation since LPs tend to follow different execution paths. In addition, the communication cost of passing messages could be too expensive for CPUs located on different chips.

Later it was found that the CMB algorithm could lead to deadlock (Chandy et al. 1979). Various techniques such as probe message (Peacock et al. 1979; Holmes 1978), virtual time (Jefferson 1985), and null message (Bryant 1977) were subsequently introduced to prevent or recover from deadlock. Our simulator is based on the original idea of distributed simulation and employs the null message to avoid deadlock.

The emergence of GPU makes it possible to perform massively parallel logic simulation because it provides a large number of scalar processors with fast inter-processor communication. The work proposed in Gulati and Khatri (2008) use GPU to accelerate fault simulation by simulating multiple input patterns in parallel. A GPU-based simulator, GCS, was presented in Chatterjee et al. (2009a) to carry out high performance gate-level logical simulation. It is based on an oblivious algorithm in which every gate is evaluated at every clock cycle. The authors of Chatterjee et al. (2009b) proposed a GPU-based logic simulator by synchronously evaluating multiple events happened simultaneously. Different from the above works, our GPU simulator utilizes the CMB algorithm and does not require a global clock.

Outside the logic simulation domain, there are several related works also using GPUs for a specific application (e.g., Xu and Bagrodia 2007; Park et al. 2008; Rybacki et al. 2009), or a general framework (e.g., Perumalla 2006a,b; Park and Fishwick 2009). One early work reported in Xu and Bagrodia (2007) focuses on high fidelity network modeling and adopts a synchronous event-driven algorithm. In Park et al. (2008), a discrete-event simulation framework is proposed. Also based on synchronous event-driven algorithm, that paper developed a distributed FEL (Future Event List) structure to avoid the time-consuming global sorting. We build a different approach to solve the similar problem in this paper. Basically, the priority queue of a gate is distributed to the gate inputs such that the sorting no longer necessary. The mechanism is enhanced with a dynamic memory management mechanism to overcome the obstacle of a limited GPU memory capacity.

32.7 Conclusion and Future Work

This paper proposes the first CMB based logic simulation algorithm on modern GPUs. The CMB algorithm is efficiently and effectively mapped to GPUs in a fine-grain manner. To guarantee a high level of parallelism in the simulation, we re-designed the fundamental data structures of the CMB algorithm to overcome the previous obstacle that hinders an efficient implementation on previous shared memory architectures. A gate reordering heuristic is introduced to improve the data locality

and reduce the execution divergence on GPUs. To support robust simulation of large circuits, we developed a dynamic GPU memory allocation mechanism as well as an adaptive pattern-issuing method. A complete set of optimization techniques were also proposed to maximize the effective memory bandwidth. Our GPU simulator is tested on a group of real VLSI designs with both deterministic and random stimuli. The experimental results show that a speedup of 47.4X could be achieved against a CPU baseline simulator using a classical event-driven algorithm.

In the future, we plan to extend this work in several directions. First, the simulator will be enhanced to support RTL. Complex processes would need to be mapped to computing resources under such a context. A process decomposition mechanism would be necessary. Second, we will generalize our simulator to perform simulation in other application domains such network and transportation simulations. Another interesting topic is to explore the possibility of parallelizing SystemC's simulation kernel (IEEE 2005) with our techniques.

Appendix. Sample Code Snippets

1. The *Extract* Primitive

```
__global__ void opoq_sim_update_pi(…)
{
    unsigned tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < num_pis){      // pi value -> gate output
        unsigned *thd_extract_pi_input_event = (unsigned int *)((char *)pi_input_event +
                                                    pi_head[tid] * pi_pitch);
        if(issue_pause) { //GPU congested – let's pause issuing input patterns
            unsigned last_time = GET_TIME(gate_output_event[tid]);
            gate_output_event[tid] = MAKE_PAIR(last_time, NULL_MSG);
        } else {
            unsigned msg = FIFO_EXTRACT_D(thd_extract_pi_input_event,
                                                    pi_head, pi_size, tid);
            gate_output_event[tid] = msg;
        }
    }
}
```

2. The *Fetch* Primitive

```
__global__ void opoq_sim_update_pin(…)
{
    unsigned tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < num_pins){    //Fetch input updates
        unsigned *thd_pin_input_event = (unsigned int *)((char *)pin_input_event + tid * pin_pitch);
        unsigned id = d_input_id[tid];
        unsigned gate_index = d_input_inv_ptr[tid];
        unsigned value = gate_output_event[id];//(time, msg)
        unsigned pin_num = d_pin_in_gate[tid];
        unsigned new_time = GET_TIME(value);
        unsigned msg = GET_MSG(value);
        switch(pin_num) {
            case 0 : gate_clock_value[gate_index].x = new_time; break;
            case 1 : gate_clock_value[gate_index].y = new_time; break;
            case 2 : gate_clock_value[gate_index].z = new_time; break;
            case 3 : gate_clock_value[gate_index].w = new_time; break;
        }
        if(msg != NULL_MSG) {
            unsigned fifo_last_value = D_FIFO_LAST_D(thd_pin_input_event, msg_array,
                                                     pin_size, pin_tail_cell, pin_tail_offset, tid);
            if(msg != GET_MSG(fifo_last_value)) {
                D_FIFO_INSERT_D(thd_pin_input_event, msg_array, alloc_cell_array,
                                pin_tail_cell, pin_tail_offset, pin_size, new_time, msg,
                                num_pins, tid);
            }
        }
    }
}
```

3. The *Evaluate* Primitive

```
__global__ void opoq_evaluate(…)
{
    unsigned tid = blockDim.x * blockIdx.x + threadIdx.x;
    //set up the local simulation time
    …

    // if one or more event can be evaluated
    if(time < min_time) {
    input_val    = h_gate_val[tid];
    gate_val.x = input_val & 0x1;
    gate_val.y = (input_val >> 8) & 0x1;
    gate_val.z = (input_val >> 16) & 0x1;
    gate_val.w = (input_val >> 24) & 0x1;
```

```
        // update the gate input value
        unsigned pin_num = end - begin;
        for(unsigned pid = 0; pid < pin_num; pid++){
            switch(pid) {
                case 0: if(time != GET_TIME(event0)) continue; break;
                    case 1: if(time != GET_TIME(event1)) continue; break;
                    case 2: if(time != GET_TIME(event2)) continue; break;
                    case 3: if(time != GET_TIME(event3)) continue; break;
            }
            thd_pin_input_event = (unsigned int *)((char *)pin_input_event + (begin + pid) * pin_pitch);
            event_temp = D_FIFO_EXTRACT_D(thd_pin_input_event, msg_array, release_cell_array,
                                          pin_head_cell, pin_head_offset, pin_size, tid,
                                          begin + pid, num_pins);
            msg = GET_MSG(event_temp);
            //message handling
            …
        }

        // evaluate the result
        result = TRUTH_TABLE[GATE_OFFSET[type] + (gate_val.x << 3) + (gate_val.y << 2) +
            (gate_val.z << 1) + gate_val.w];

        …

        //reassign gate_val
        h_gate_val[tid] = (gate_val.w << 24) + (gate_val.z << 16) + (gate_val.y << 8) + gate_val.x;
    }
}
```

References

Amdahl GM (1967) Validity of the single-processor approach to achieving large-scale computing capabilities. In: American federation of information processing societies conference, AFIPS Press, pp 483–485

Bailey ML, Briner JV Jr, Chamberlain RD (1994) Parallel logic simulation of VLSI systems. ACM Comput Surv 26(3):255–294

Bataineh A, Özgüner F, Szauter I (1992) Parallel logic and fault simulation algorithms for shared memory vector machines. In: International conference on computer-aided design

Blythe D (2008) Rise of the graphics processor. Proc IEEE 96(5):761–778

Bryant RE (1977) Simulation of packet communications architecture computer system. MIT-LCS-TR-188, MIT

Chandy KM, Misra J (1979) Distributed simulation: a case study in design and verification of distributed programs. IEEE Trans Softw Eng SE-5(5):440–452

Chandy KM, Misra J (1981) Asynchronous distributed simulation via a sequence of parallel computations. Commun ACM 24(4):198–206

Chandy KM, Misra J, Holmes V (1979) Distributed simulation of networks. Comput Netw 3:105–113

Chatterjee D, DeOrio A, Bertacco V (2009a) Event-driven gate-level simulation with GP-GPUs. In: Design automation conference

Chatterjee D, DeOrio A, Bertacco V (2009b) High-performance gate-level simulation with GP-GPUs. In: Design automation test Europe

Fujimoto RM (2000) Parallel and distributed simulation systems. Wiley-Interscience, New York
Fung WWL, Sham I, Yuan G, Aamodt TM (2007) Dynamic warp formation and scheduling for efficient GPU control flow. In: International symposium on microarchitecture, Chicago, pp 407–418
Gulati K, Khatri S (2008) Towards acceleration of fault simulation using graphics processing units. In: Design automation conference
Holmes V (1978) Parallel algorithms on multiple processor architectures. Ph.D. dissertation, Computer Science Department, University of Texas, Austin
Huang JH (2010) Keynote speech. In: Mini GPU technology conference, Beijing
IEEE (2005) IEEE Std. 1666–2005, Standard for SystemC
ITC99 Benchmarks (1999) http://www.cad.polito.it/tools/itc99.html
Jefferson DR (1985) Virtual time. ACM Trans Prog Lang Syst 7(3):404–425
MPI (2000) http://www.mpi-forum.org/docs/
NVIDIA (2009) CUDA Programming Guide 2.3
NVIDIA (2010) White paper. NVIDIA's next generation CUDA™ compute architecture: Fermi
OpenCores (2010) http://www.opencores.org/
Park H, Fishwick PA (2008) A fast hybrid time-synchronous/event approach to parallel discrete event simulation of queuing networks. In: Conference on winter simulation
Park H, Fishwick PA (2009) A GPU-based application framework supporting fast discrete-event simulation. Simulation. doi:10.1177/0037549709340781
PCI-SIG (2010) PCIe base 3.0 specification.http://www.pcisig.com/specifications/pciexpress/base3
Peacock JK, Wong JW, Manning EG (1979) Distributed simulation using a network of processors. Comput Netw 3(1):44–56
Perumalla KS (2006a) Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In: Workshop on principles of advanced and distributed simulation
Perumalla KS (2006b) Parallel and distributed simulation: traditional techniques and recent advances. In: Conference on winter simulation
Rashinkar P, Paterson P, Singh L (2000) System-on-a-chip verification: methodology and techniques. Kluwer Academic Publishers, Dordrecht
Rybacki S, Himmelspach J, Uhrmacher AM (2009) Experiments with single core, multi-core, and GPU based computation of cellular automata. In: Advances in international conference system simulation, pp 62–67
Soule L, Gupta, A (1991) An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation. ACM Trans Model Comput Simul 1(4):308–347
Synopsys (2010) VCS: multicore-enabled functional verification solution. http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx
Xu Z, Bagrodia R (2007) GPU-accelerated evaluation platform for high fidelity network modeling. In: International workshop on principles of advanced and distributed simulation

Chapter 33
Implementations of Main Algorithms for Generalized Symmetric Eigenproblem on GPU Accelerator

Yonghua Zhao, Fang Liu, Yangang Wang and Xuebin Chi

Abstract To solve a generalized eigensystem problem, we firstly need to transform the generalized eigenproblem to a standard eigenproblem, and then reduce a matrix to tridiagonal form. These are based on both blocked Cholesky decomposition and blocked Householder tridiagonalization method. We present parallel implementations of standard transformation which combines the Cholesky into the transformation from generalized to standard form, and reduction of a dense matrix to tridiagonal form on GPU accelerator using CUBLAS. Experimental results clearly demonstrate the potential of data-parallel coprocessors for scientific computations. When comparing against the CPU implementation, the GPU implementations achieve above 16-fold and 20-fold speedups in double precision respectively.

Keywords Tridiagonalization · Cholesky · Eigenvalue · GPU · Symmetric matrix

33.1 Introduction

Many scientific and engineering applications lead to large symmetric matrix eigenvalue problems. Examples come from computational quantum chemistry, finite element modeling, multivariate statistics, and density functional theory. With the

Y. Zhao(✉) · F. Liu · Y. Wang · X. Chi
Supercomputing Center, Computer Network Information Center,
Chinese Academy of Sciences, 100190 Beijing, China
e-mail: yhzhao@sccas.cn

F. Liu
e-mail: Liufang@sccas.cn

Y. Wang
e-mail: Wangyg@sccas.cn

X. Chi
e-mail: chi@sccas.cn

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 543
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_33,

increasing complexity of mathematical models and the availability of various high performance systems, such as multi-core and general purpose GPU(GPGPU), there is a growing demand to solve large-scale eigenvalue problem.

Some researches have already been done on the implementation of the different dense linear algebra algorithms on GPU (Tomov et al. 2010; Barrachina et al. 2009). Volkov et al. give efficient implementation of the LU, QR and Cholesky factorization using vector capabilities of the graphic processors (Volkov and Demmel 2008). The MAGMA (Tomov et al. 2009) project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid (CPU + GPU) architectures. Several important implementations of the linear algebra solutions are developed, like QR factorization (Agullo et al. 2010) using multiple GPUs, Cholesky factorization (Ltaief et al. 2009) on multi-core systems with GPU accelerators and Hessenberg reduction (Tomov et al. 2010) for hybrid GPU-based systems.

Parallel solution on the generalized symmetric eigenproblem relates to five translations (Anderson et al. 1999), and many researchers have studied the parallel algorithms on traditional architectures such as single-core machines or distributed memory parallel machines (Anderson et al. 1999; Blackford et al. 1997). But the lack of the implementation of the eigenvalue solvers on heterogeneous/hybrid (CPU + GPU) architectures is still present. Volkov et al. give the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices using GPUs (Volkov and Demmel 2008). Lessig et al. present the MRRR parallel algorithm (Bientinesi et al. (2005)) for data-parallel coprocessors (Lessig and Bientinesi 2009).

CUBLAS NVIDIA (2008); Igual et al. (2009) is NVIDIAs implementation of the basic linear algebra algorithms. It includes the BLAS 1, 2 & 3 routines redesigned using CUDA for execution on the NVIDIA graphic cards. This paper focuses on the Blocked algorithm from generalized eigenproblem to standard eigenproblem (Cao et al. 2002) and blocked Householder tridiagonalization method (Bischof and Loan 1985), using the current generation of GPUs and CUBLAS. In algorithm transforming the generalized eigenproblem into standard form, we present a new parallelization which combines the Cholesky into the transformation from generalized to standard form.

The rest of the paper is structured as follows: Sect. 33.1 describes generalized symmetric matrix eigenvalue problem. Section 33.3 gives the blocked algorithms of standardization combining the Cholesky on CPU and GPU. Section 33.4 describes the blocked tridiagonalization of a symmetric matrix on CPU and GPU. Section 33.5 gives the comparison performance results on CPU and CPU+GPU. Finally, Sect. 33.6 gives the conclusion.

33.2 Generalized Symmetric Matrix Eigenvalue Problem

We consider the solution of the symmetric matrix eigenvalue problem

$$AX = \Lambda BX \tag{33.1}$$

Table 33.1 Five stages for the solution of generalized symmetric eigenproblem

Operation steps	Operations
Cholesky decomposition	$L^T L = B$
Transformation from generalized eigenproblem to standard form	$C = L^{-1}\, A(L^{-1})^T$
Reduction to tridiagonalization	$QTQ^T = A$
Solve tridiagonalization problem	$ZDZ^T = T$
Back transformation is applied to the eigenvectors of T	$Z = QZ$
Back transformation is applied to the eigenvectors of C	$X = L^{-1}\, Z$

where $A \in R^{n\times n}$ is a symmetric matrix and $B \in R^{n\times n}$ is symmetric positive definite, $\Lambda = diag(\lambda_1, \lambda_2, \cdots, \lambda_n) \in R^{n\times n}$ is a diagonal matrix containing the eigenvalues of eigenvalue problem (1), and the j-th column of the orthogonal matrix $X \in R^{n\times n}$ is an eigenvector associated with λ_j. Given the matrix A and B, the objective is to compute its eigenvalues or a subset thereof and, if requested, the associated eigenvectors as well.

Efficient algorithms for the solution of generalized symmetric eigenvalue problems usually consist of five stages (Table 33.1) and eigenproblem of a tridiagonal matrix T. The MR^3 algorithm (Dhillon and Parlett 2004) is applied to matrix T to accurately compute its eigenvalues and, optionally, the associated eigenvectors.

In next several sections, we give the Blocked algorithm from generalized eigenproblem to standard form Based on Cholesky decomposition and blocked Householder tridiagonalization method, using the current generation of GPUs and CUBLAS.

33.3 The Blocked Algorithm Transforming Generalized Eigenproblem into Standard Form

33.3.1 The Blocked Algorithm Based on Cholesky Decomposition on CPU

For Generalized symmetric eigenproblem $Ax = \lambda Bx$, since B is symmetric positive definite, so the Cholesky decomposition of B exists, i.e., there exists an upper triangular matrix R, such that $B = R^T R$. Let $C = RAR^T$, and $y = Rx$. Then the generalized eigenproblem can be changed into a standard eigenvalue problem $Cy = \lambda y$. Algorithm 1 presents the blocked algorithm that combines the Cholesky decomposition into transformation from a generalized eigenproblem to the standardization eigenproblem.

Algorithm 1. Blocked algorithm standardization based on Cholesky decomposition , blocked size is *nb* .

begin

Partitioning matrices:

$$A=\left(\begin{array}{c|c} A_{TL} & * \\ \hline A_{BL} & A_{BR} \end{array}\right),\ B=\left(\begin{array}{c|c} B_{TL} & * \\ \hline B_{BL} & B_{BR} \end{array}\right)$$

Where A_{TL} and B_{TL} is 0×0.

do until A_{BR} is 0×0 matrix

b=min(nb, A_{BR})

$$A=\left(\begin{array}{c||c} A_{TL} & * \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)=\left(\begin{array}{c|c||c} A_{00} & * & * \\ \hline\hline A_{10} & A_{11} & * \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

$$B=\left(\begin{array}{c||c} B_{TL} & * \\ \hline\hline B_{BL} & B_{BR} \end{array}\right)=\left(\begin{array}{c|c||c} B_{00} & * & * \\ \hline\hline B_{10} & B_{11} & * \\ \hline B_{20} & B_{21} & B_{22} \end{array}\right)$$

(1) B_{11}:=L_{11}=Chol(B_{11}) (2)A_{11}:=C_{11}=sygst(A_{11}, L_{11})

(3) B_{21}:=L_{21}=$A_{21}L_{11}^{-T}$

(4) $A_{21}:=A_{21}L_{11}^{-T}$ (5) $A_{21}:=A_{21}-\frac{1}{2}L_{21}C_{11}$

(6) $B_{22}=B_{22}-L_{21}L_{21}^{T}$ (7) $A_{22}:=A_{22}-L_{21}A_{21}^{T}-A_{21}L_{21}^{T}$

(8) $A_{21}=A_{21}-\frac{1}{2}L_{21}C_{11}$ (9) $A_{21}:=C_{21}=L_{22}^{-1}A_{21}$

$$A=\left(\begin{array}{c||c} A_{TL} & * \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)=\left(\begin{array}{c|c||c} A_{00} & * & * \\ \hline\hline A_{10} & A_{11} & * \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

$$B=\left(\begin{array}{c||c} B_{TL} & * \\ \hline\hline B_{BL} & B_{BR} \end{array}\right)=\left(\begin{array}{c|c||c} B_{00} & * & * \\ \hline\hline B_{10} & B_{11} & * \\ \hline B_{20} & B_{21} & B_{22} \end{array}\right)$$

end do

end

33.3.2 The Blocked Algorithm Based on Cholesky Decomposition on GPU

Recent work on the implementation of BLAS and the major factorization routines for the solution of linear systems has demonstrated the potential of GPUs to yield high performance on dense linear algebra operations which can be cast in terms of matrix-matrix products. In this subsection we describe how to exploit the GPU in Algorithm 1, orchestrating the computations carefully to reduce the number of data transfers between the host and the GPU. During the reduction to standard eigenproblem, in

general, b will be small so that both computation chol(A_{11}) and sygst(A_{11}, L_{11}) are likely to be better suited for the CPU, and other operations may be computed on the GPU using CUBLAS.

Now, assume that the entire matrix resides in the GPU memory initially. Algorithm 2 gives in pseudo-code the GPU algorithm, Prefix '*d_*', standing for device, before a matrix denotes that the matrix resides on the GPU memory.

Algorithm 2 Blocking algorithm of tridiagonalization

/*Let b denote the algorithmic block size*/

Send matrix A and B from the CPU to matrix dA and dB on the GPU.

Begin

Partitioning matrices:

$$d\_A=\left(\begin{array}{c|c} d\_A_{TL} & * \\ \hline d\_A_{BL} & d\_A_{BR} \end{array}\right),\ d\_B=\left(\begin{array}{c|c} d\_B_{TL} & * \\ \hline d\_B_{BL} & d\_B_{BR} \end{array}\right)$$

Where d_A_{TL} and d_B_{TL} is 0×0.

do until d_A_{BR} is 0×0 matrix

b=min(nb, d_A_{BR})

$$dev\_A=\left(\begin{array}{c|c} dev\_A_{TL} & * \\ \hline dev\_A_{BL} & dev\_A_{BR} \end{array}\right)=\left(\begin{array}{c|c|c} dev\_A_{00} & * & * \\ \hline dev\_A_{10} & dev\_A_{11} & * \\ \hline dev\_A_{20} & dev\_A_{21} & dev\_A_{22} \end{array}\right)$$

$$dev\_B=\left(\begin{array}{c|c} dev\_B_{TL} & * \\ \hline dev\_B_{BL} & dev\_B_{BR} \end{array}\right)=\left(\begin{array}{c|c|c} dev\_B_{00} & * & * \\ \hline dev\_AB_{10} & dev\_B_{11} & * \\ \hline dev\_B_{20} & dev\_B_{21} & dev\_B_{22} \end{array}\right)$$

Send d_B_{11} and d_A_{11} from GPU to $B_{11}$and A_{11} on CPU

(1)B_{11}:=L_{11}=Chol(B_{11}) (2)A_{11}:=C_{11}=sygst(A_{11}, L_{11})

Send B_{11} and A_{11} from GPU to d_B_{11} and d_A_{11} on CPU

(3) d_B_{21}:=d_L_{21}=d_A_{21}*$d\_L_{11}^{-T}$

(4) $d\_A_{21}:=d\_A_{21}*d\_L_{11}^{-T}$ (5) $d\_A_{21}:=d\_A_{21}-\frac{1}{2}d\_L_{21}*d\_C_{11}$

(6) $d\_B_{22}=d\_B_{22}-d\_L_{21}*d\_L_{21}^{T}$ (7) $d\_A_{22}:=d\_A_{22}-d\_L_{21}*d\_A_{21}^{T}-d\_A_{21}*d\_L_{21}^{T}$

(8) $d\_A_{21}=d\_A_{21}-\frac{1}{2}d\_L_{21}*d\_C_{11}$ (9) $d\_A_{21}:=d\_C_{21}=d\_L_{22}^{-1}*d\_A_{21}$

$$dev\_A=\left(\begin{array}{c|c} dev\_A_{TL} & * \\ \hline dev\_A_{BL} & dev\_A_{BR} \end{array}\right)=\left(\begin{array}{c|c|c} dev\_A_{00} & * & * \\ \hline dev\_A_{10} & dev\_A_{11} & * \\ \hline dev\_A_{20} & dev\_A_{21} & dev\_A_{22} \end{array}\right)$$

$$dev\_B=\left(\begin{array}{c|c} dev\_B_{TL} & * \\ \hline dev\_B_{BL} & dev\_B_{BR} \end{array}\right)=\left(\begin{array}{c|c|c} dev\_B_{00} & * & * \\ \hline dev\_AB_{10} & dev\_B_{11} & * \\ \hline dev\_B_{20} & dev\_B_{21} & dev\_B_{22} \end{array}\right)$$

end do

end

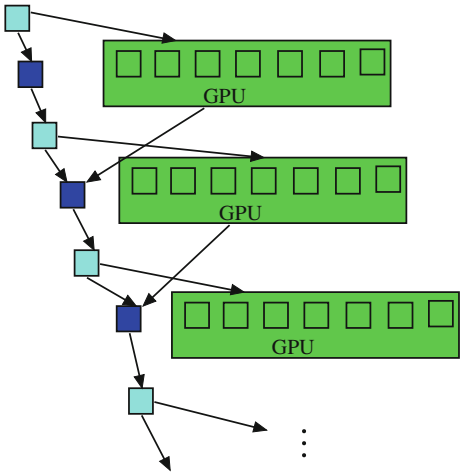


Fig. 33.1 The computations and communications between the CPU and GPU

Figure 33.1 illustrates the computations and communications between the CPU and GPU during the standard transformation. The total communications from CPU to GPU and from GPU to CPU are about $2n^2+2nb$ and $2nb$ in algorithm 2 , respectively.

33.4 The Blocked Tridiagonalization of a Symmetric Matrix

33.4.1 The Householder Transformation

Basic operations utilized by the reduction to tridiagonal form are the computation and application of Householder transformations.

Theorem 1 *Given a vector $z \in R^n$, we can find a vector $u \in R^n$ and a scalar $\beta \in R$ such that*

$$(I - \beta uu^T)\mathrm{z} = (z_1, \cdots, z_{k-1}, \sigma_k, 0, \cdots, 0)^T$$

Where $\sigma_k = sign(z_k)\,\|z_{k+1:n}\|_2$, The scalar $\beta \equiv 2/||u||_2^2$ and the reflector vector $u \equiv (0, \cdots 0, z_k + \sigma_k, z_{k+1}, \cdots, z_n)^T$ are a pair of quantities which satisfy the above theorem, and I denotes the identity matrix of appropriate order. The reflection $(I - \beta uu^T)z$ in Theorem 1 is called Householder transformation. This reflection does not affect the elements $z_1, \cdots, z_k$.

Given a symmetric matrix A, we can compute $H_j = I - \beta_j u_j u_j^T (j = 1, 2, \cdots, n-2)$,where $u_j \in R^n$ with the first j entries zero, such that $T = H_{n-2}\ H_1 A H_1\ \cdots\ H_{n-2}$ is a tridiagonal symmetric matrix. Applying a both-sided Householder transformation to A, we can get

$$(I - \beta uu^T)A(I - \beta uu^T) = A - uw^T - wu^T \tag{33.2}$$

where $w = v - \frac{\beta}{2}(v^T u)u$, $v = \beta Au$. The right side of the Eq. 33.2 is called rank-2 modification of A. In order to exploit the symmetry, only the lower (or the upper) triangular part of this matrix is updated.

33.4.2 A Blocking Algorithm of Tridiagonalization on CPU

It is known that blocking algorithms are more efficient than a non-blocking algorithm. The basic principle of the blocking algorithm is to accumulate a certain number of the vectors u and w into a block update. If we reserve b pairs of vectors u, w, then we can gather b rank-2 modifications of A to be a rank-$2b$ modification using the BLAS-3 operation. Here, we suppose the matrix is divided into column panels with width b, then the operations of the blocking algorithm include $\lceil n/p \rceil$ column panel reduce steps, and an out-panel matrix modification follows each panel reduction.

Figure 33.2 provides a schematic of the column panel reduction. During the column panel reduction, the Householder transformation is performed to the part in a column panel, and reserves vectors w and u produced during the transformation to form $n \times b$ matrices W and U that are used to modify the rest parts A_{22} of the matrix. The updating of A_{22} is performed by running the rank-$2b$ modification

$$A_{22} = A_{22} - UW^T - WU^T$$

after panel operations are complete. Subsequently, the same procedure is applied to the new modified submatrix A_{22} until the column panel is not existed. The delayed modification of A_{22} makes the producing of vector v more complicated because the elements stored in matrix A_{22} are not updated in time during the panel transformation.

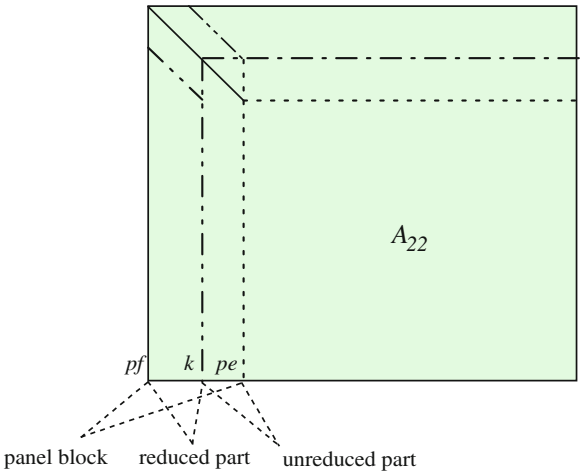


Fig. 33.2 Schematic of the column panel reduction

So when solving w, the computation $v = \beta Au$ needs to be replaced by

$$v = (Au - UW^T u - WU^T u)$$

where $UW^T u$ and $WU^T u$ correspond to out-panel corrections of A.

Algorithm 3 gives Blocking algorithm of tridiagonalization. For each reduced column in the panel, a new column of W is generated. This requires four panel-vector multiplications and one symmetric matrix-vector multiplication with the submatrix A_{22} as operand. The latter operation, computed with the BLAS-2 kernel *dsymv*, is the most expensive one, requiring roughly $2(n - j)^2 b$ flops. Step 8 also requires $2(n - j)^2 b$ flops, but is entirely performed by the BLAS-3 kernel *syr2k* for the symmetric rank-$2b$ update. The overall cost of performing the reduction from A to T in algorithm1 is therefore $4n^3/3$ flops provided that $b \ll n$.

Algorithm 3 Blocking algorithm of tridiagonalization

/*Let b denote the algorithmic block size*/

$nb = \lceil (n-2)/b \rceil$

for j = 1 : nb

/*Get first and last column in the panel block */

$pf = (i-1) \times b + 1$ $pe = \min(pf + b - 1, n - 2)$

$W = [\emptyset]$ $U = [\emptyset]$

/* The current panel is reduced to tridiagonal form in next loop and builds matrices W and U*/

for $k = pf : pe$

(1) Generate $u(k+1: n)$, β of H_k from vector $A(k: n, j)$

(2) Matrix-vector production: z = $A(pe+1{:}n, pe+1{:}n)\, u$

/* correct for out of date entries of A */

(3) $x(1{:}k\text{-}pf) = U^T_{(pe+1:n,\,1:k-pf)} u(\text{pe+1:n})$

(4) $y(1{:}k\text{-}pf) = W^T_{(pe+1:n,\,1:k-pf)} u(\text{pe+1:n})$

(5) $z = z - W(pe+1{:}n, 1{:}k-pf)x(1{:}k-pf) - W(pe+1{:}n,\ 1{:}k-pf)y(1{:}k-pf)$

(6) $w = z - (\beta z^T u/2)u$

$W = [W|w^T]$ $U = [U|u^T]$

/* Update remainder of panel block */

(7) $A(k+1{:}n, k+1{:}pe) = A(k+1{:}n, k+1{:}pe) - u^T_{(k+1:n)} w(k+1{:}pe) - w^T_{(k+1:n)} u(k+1{:}pe)$

endfor

/*Rank-$2b$ update of the submatrix A_{22} */

(8) $A(pe+1{:}n, pe+1{:}n) = A(pe+1{:}n, pe+1{:}n) - U(pe+1{:}n, 1{:}pe-pf+1)W^T_{(pe+1:n,1:pe-pf+1)} - W(pe+1{:}n, 1{:}pe-pf+1)U^T_{(pe+1:n,1:pe-pf+1)}$

endfor

end

Note that there is no need to construct the orthogonal factor $Q = H_1 H_2 \cdots H_{n-2}$ explicitly. Instead, the vectors u_j defining the Householder reflectors H_j are stored in the annihilated entries of A. If the eigenvectors are required, the back-transform QX^T is computed in $2n^3$ flops without ever forming Q. Using the compact WY representation (Tomov et al. 2010), this operation can be performed almost entirely in terms of calls to BLAS-3 kernels.

33.4.3 A Blocking Algorithm of Tridiagonalization on GPU

In this subsection we describe how to exploit the GPU in the reduction of a matrix to tridiagonalization matrix, orchestrating the computations carefully to reduce the number of data transfers between the host and the GPU. During the reduction to tridiagonalization form in algorithm 1, Step 2 and 8 are a natural candidate for being computed on the GPU, while, due to the kernels involved in step 3, 4, and 5 (mainly narrow matrix-vector products), these computations are better suited for the CPU. Step 7 can be performed either on the CPU or the GPU, but in general, b will be small so that this computation is likely better suited for the CPU.

Now, assume that the entire matrix resides in the GPU memory initially. We can then proceed to compute the reduced form by repeating the following three steps for each column block:

(a) Transfer column panel back from GPU memory to main memory. Perform Operations 1on the CPU.

(b) Transfer Householder vector u from CPU to the GPU. Perform step 2 on GPU.

(c) Transfer z from GPU to CPU. Perform step 3, 4, 5, 6 and 7 on CPU.

(d) Transfer W from main memory to the GPU. Compute Operation 8 on the GPU.

Algorithm 4 gives in pseudo-code the GPU tridiagonalization algorithm, prefix '*d_*', standing for device, before a matrix denotes that the matrix resides on the GPU memory.

Algorithm 4 Blocking algorithm of tridiagonalization

/*Let b denote the algorithmic block size*/
Send matrix A from the CPU to matrix dA on the GPU
$nb = \lceil (n-2)/b \rceil$
for $j = 1 : nb$
$pf = (i-1) \times b + 1$ $pe = \min(pf + b - 1, n - 2)$
$W = [\emptyset]$ $d\_U = [\emptyset]$
Send the current panel matrix $d\_A(pf{:}\ n, pf : pe)$ from the GPU to matrix $A(pf{:}\ n, pf{:}\ pe)$ on the CPU
for k = $pf : pe$
generate $u(k{+}1{:}\ n), \beta$ of H_k from vector $A(k{:}\ n, k)$
Send $u\ (k{+}1{:}\ n)$ *form the CPU to* $d\_u(k{+}1{:}\ n)$ *on the GPU*
Matrix-vector production: $d\_z = d\_A(pe{+}1{:}n, pe{+}1{:}n)\ d\_u(pe{+}1{:}\ n)$

$x(1{:}k\text{-}pf) = U^T_{(pe+1:n,1:k-pf)} u(\text{pe}+1{:}\text{n})$

$y(1{:}k\text{-}pf) = W^T_{(pe+1:n,1:k-pf)} u(\text{pe}+1{:}\text{n})$

Send *dev_z* from the G*PU to z on the CPU*

$z = z - (W(pe+1{:}n,\ 1{:}k-pf)x(1{:}k-pf) - U(pe+1{:}n,\ 1{:}k-pf)\,y(1{:}k-pf))$

$w = z - (\beta z^T u/2)u$

$W = [W|w^T]$ $\quad dev\_U = [dev\_U|dev\_u^T]$

(* update remainder of panel block *)

$A(k{+}1{:}n, k{+}1{:}pe) = A(k{+}1{:}n, k{+}1{:}pe) -$

$$u^T_{(k+1:n)} w(k{+}1{:}pe) - w^T_{(k+1:n)} u(k{+}1{:}pe)$$

endfor

/* rank-2*b* update of the submatrix A_{22} */

Send *W* from the *CPU to d_W* on the GPU

$$d\_A(pe+1{:}n, pe+1{:}n) = d\_A(pe+1{:}n, pe+1{:}n) - d\_U(pe+1{:}n, 1{:}pe-pf+1) d\_W^T_{(pe+1:n,1:pe-pf+1)} - d\_W(pe+1{:}n, 1{:}pe-pf+1) d\_U^T_{(pe+1:n,1:pe-pf+1)}$$

end for

Figure 33.3 illustrates the communications between the CPU and GPU for inner/outer iteration during the tridiagonalization of matrix. The total communications from CPU to GPU and from GPU to CPU are about $3(1+n)*n/2$ and $(n+1)*n$ in algorithm 4, respectively.

33.5 Experimental Results

The performance results that we provide in this section use NVIDIA GeForce Tesla c1060 GPU and a quad-core Intel Xeon running at 2.33 GHz. On the CPU we use LAPACK and BLAS from MKL 10.0.0, and on the GPU using CUBLAS 2.3

Table 33.2 gives performance of transformation from generalized eigenproblem to standard form on CPU and GPU. The basic implementation is for 1 core + 1GPU, and uses CUBLAS 2.3. The orders of test matrices are 3,000, 6,000, 8,000, 10,000 and 16,000, respectively. The results show that we achieve an enormous above 21× speedup compared to the implementation running on 1 core in double precision.

Figure 33.4 shows the performance of the blocked tridiagonalization of symmetric Matrices on CPU and GPU. The basic implementation is for 1 core + 1GPU, and uses CUBLAS 2.1. The result shows that we achieve an enormous 16× speedup compared to the implementation running on 1 core in double precision.

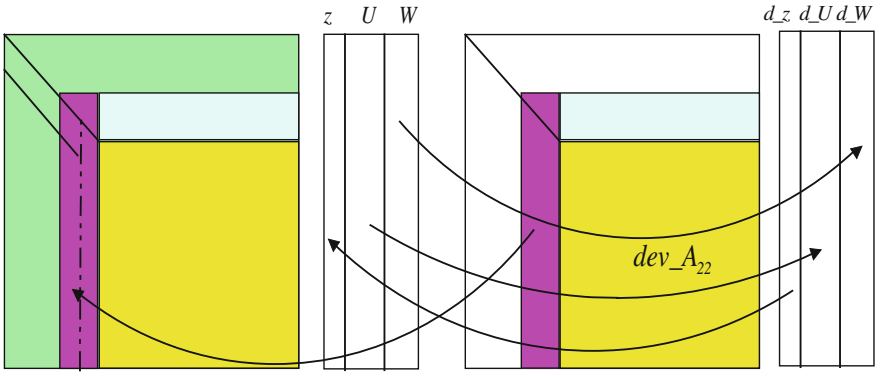


Fig. 33.3 CPU/GPU communications for tridiagonalization in inner/outer iteration

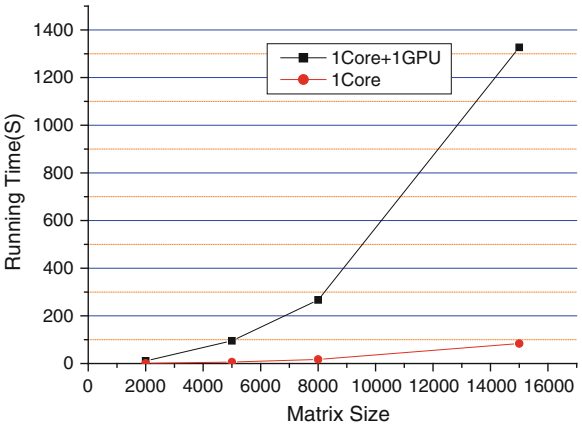


Fig. 33.4 Performance of CPU versus GPU about reducing matrix to tridiagonal form

Table 33.2 Performance of CPU versus GPU about transforming generalized eigenproblem to standard

	6000*6000	8000*8000	12000*12000	15000*15000
1 core	106.4	234	640	1604
1core+1 GPU	5.02	10.4	27.12	61.68
Speedup	21.2	22.5	23.6	26.2

33.6 Conclusion

We have evaluated the performance for both blocked transformation algorithm from generalized eigenproblem to standard eigenproblem based on Cholesky decomposition and the reduction algorithm of a dense matrix to tridiagonal form on GPU. They are two main transformations for computing the generalized symmetric matrix

eigenvalue. Experimental results on NVIDIA GeForce Tesla c1060 GPU using CUBLAS clearly show the potential of data-parallel coprocessors for scientific computations. The GPU acceleration of two algorithms achieves 16-fold and 20-fold speedups in double precision respectively.

Acknowledgments This work is supported by the National Science Foundation of China (Grant No. 60873113) and 863 Program (2009AA01A134, 2010AA012301).

References

Agullo E, Augonnet C, Dongarra J, Faverge M, Ltaief H, Thibault S, Tomov S (2010) QR factorization on a multicore node enhanced with multiple GPU accelerators, University of Tennessee Computer Science, Tech. Rep. ICL-UT-10-04

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A et al. (1999), LAPAck Users' Guide SIAM

Barrachina S, Castillo M, Igual FD, Mayo R, Quintana-Ort‘ı ES, Quintana-Ort‘ı G, Exploiting the capabilities of modern GPUs for dense matrix computations, Concurrency and Computation: Practice and Experience, 21(18):2457–2477, 2009. (Online). Available: http://dx.doi.org/10.1002/cpe.1472

Bientinesi P, Dhillon IS, van de Geijn RA (2005) A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. SIAM J Sci Comput 27(1):43–66

Bischof C, Van Loan C (1987) The WY representation for products of Householder matrices, SIAM J. Sci. Stat. Comp. 8, no. 1, S2–S13, Parallel processing for scientific computing (Norfolk, Va., 1985). MR 88f:65070

Blackford L, Cleary A, Choi J, d'Azevedo d'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A et al. (1997), ScaLAPACK users' guide. Society for Industrial Mathematics

Cao X, Chi X, Gu N (2002) Parallel solving symmetric eigenproblems. 5th international conference on algorithms and architectures for parallel processing. IEEE, Beijing, China

Dhillon Inderjit S, Parlett Beresford N (2004) Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. Linear Algebra Appl 387:1–28

Igual FD, Quintana-Ort'ı G, van de Geijn R (2009) Level-3 BLAS on a GPU: Picking the low hanging fruit. FLAME Working Note # 37. DICC 2009–04-01, Universitat Jaume I. Dept. ICC

Lessig C, Bientinesi P (2009) On parallelizing the MRRR algorithm for data-parallel coprocessors, In: Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J (eds) in PPAM (1), ser. Lecture Notes in Computer Science, vol 6067 Springer, Heidelberg, pp 396–402 (Online)

Ltaief H, Tomov S, Nath R, Du P, Dongarra J (2009) A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators, Tech. report, LAPACK Working Note 223, Tech. Rep

Nvidia C (2008) CUBLAS library. NVIDIA Corporation, Santa Clara, California

Tomov S, Nath R, Ltaief H, Dongarra J, (2010) Dense linear algebra solvers for multicore with GPU accelerators, IEEE international symposium on parallel and distributed processing, workshops and Ph.D. forum (IPDPSW), pp 1–8

Tomov S, Nath R, Du P, Dongarra J (2009) MAGMA version 0.2 User Guide

Tomov S, Nath R, Dongarra J (2010) Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. Parallel Comput 36(12):645–654

Volkov V, Demmel J, LU, QR and Cholesky factorizations using vector capabilities of GPUs, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, pp 2008–49

Volkov V, Demmel JW (2008) Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices, Department of Computer Science, University of Tennessee, Knoxville, inst-UT-CS:adr, LAPACK Working Note 197, (Online)

Chapter 34
Using Mixed Precision Algorithm for LINPACK Benchmark on AMD GPU

Xianyi Zhang, Yunquan Zhang and Lei Wang

Abstract LINPACK is a de facto benchmark for supercomputers. Nowadays, the CPU and GPU heterogenous cluster becomes an important trendy of supercomputers. Because of high performance of mixed precision algorithm, we had developed a mixed precision high performance LINPACK software package GHPL on NVIDIA GPU cluster. In this paper, we will introduce the recent work about porting and optimizing GHPL on AMD GPU. On AMD GPU platform, we implemented a hybrid of CPU and GPU GEMM function by ACML-GPU and GotoBLAS library. According to our results, the speedup of GHPL over HPL was 3.21. In addition, we would point out the limitations of ACML-GPU library.

34.1 Introduction

LINPACK is a de facto benchmark for supercomputers. Based on LINPACK benchmark, TOP500 website (http://www.top500.org) lists top 500 high performance computers in whole world every half year. In TOP500 2010 Oct list, because of

X. Zhang · Y. Zhang · L. Wang
Lab of Parallel Software and Computational Science, Institute of Software,
Chinese Academy of Sciences, 100190 Beijing, China
e-mail: xianyi@iscas.ac.cn

L. Wang
e-mail: lei.beststones@gmail.com

X. Zhang · L. Wang
Graduate University of Chinese Academy of Sciences, 100190 Beijing, China

Y. Zhang
State Key Lab of Computing Science, Chinese Academy of Sciences,
100190 Beijing, China
e-mail: zyq@mail.rdcps.ac.cn

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 555
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_34,

huge computational capacity of GPU, there are many CPU and GPU heterogeneous supercomputer, for example, NUDT Tianhe-1A, Dawning Nebula and IPE Mole-8.5.

Nowadays, single precision floating operations are usually faster than double precision on CPU and GPU. Thus, many researchers studied on the mixed precision algorithm which combined the single precision and double precision operations to obtain the higher performance with the similar double precision accuracy. For example, Moler (1967) and Wilkinson (1965) analyzed the mixed precision approach which was suitable for many problems in linear algebra. Langou et al. (2006) presented their results of mixed precision algorithm for systems of dense linear equations on CPU cluster and IBM CELL processor. Meanwhile, Kurzak and Dongarra (2006) implemented the mixed precision LINPACK benchmark on IBM CELL processor.

In our previous work (Wang et al. 2010), we implemented the mixed precision high performance LINPACK software package (GHPL) on CPU and NVIDIA GPU heterogeneous cluster. In 32 NVIDIA Tesla C1060 GPUs, the average speedup of GHPL over HPL is 3.06. In this paper, we will present our recent work about porting and optimizing GHPL package on AMD GPU. The rest of this paper is organized as follows. In Sect. 34.2 we review the mixed precision algorithm briefly. In Sect. 34.3 we describe our implementation and optimization on AMD GPU. In Sect. 34.4 we analyze the performance of GHPL and Sect. 34.5 is our conclusions and the future work.

34.2 Mixed Precision Algorithm for LINPACK Benchmark

Figure 34.1 shows the mixed precision algorithm to solve the linear system in Langou et al. (2006), Kurzak and Dongarra (2006), Wang et al. (2010). According to this algorithm, the LU factorization of coefficient A which needs the most operations is performed by single precision arithmetic. Moreover, it only requires using double precision in the residual calculation and the iterative refinement of solution. Therefore, the operations with time complexity $O(n^3)$ are executed in single precision and the operations with time complexity $O(n^2)$ are performed in double precision. The limitation of the mixed precision algorithm is that the condition number of the coefficient matrix should not exceed the reciprocal of the accuracy of the single precision.

34.3 Implementation on AMD GPU

In our previous GHPL (Wang et al. 2010), we already implemented mixed precision algorithm such as LU factorization in single precision, solving $Ly = Pb$, iterative refinement and convergence checking. Thus, we just needed move some computation workload to AMD GPU.

1: $LU \leftarrow PA$ (SP)

2:solve $Ly = Pb$ (SP)

3:solve $Ux_0 = y$ (SP)

do k = 1, 2, ……..

4: $r_k \leftarrow b - Ax_{k-1}$ (DP)

5:solve $Ly = Pr_k$ (SP)

6:solve $Uz_k = y$ (SP)

7: $x_k \leftarrow x_{k-1} + z_k$ (DP)

check convergence

done

Fig. 34.1 Mixed precision algorithm for Gaussian elimination. SP stands for single precision and DP stands for double precision

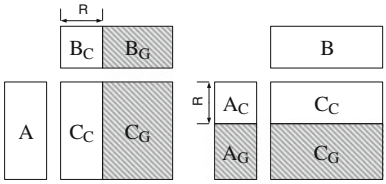


Fig. 34.2 The hybrid of CPU and GPU GEMM (The shadow sub matrix workload is executed on AMD GPU)

GEMM subroutine is the bottleneck in both double precision HPL and mixed precision GHPL software package. Likewise GHPL on NVIDIA GPU, we combined CPU and AMD GPU to accelerate GEMM function as Fig. 34.2 shown. For example, we split matrix B to sub matrix B_C and sub matrix B_G based on ratio R. Then, we calculated C_C on CPU and C_G on AMD GPU.

On AMD GPU, we used ACML-GPU library provided by GPU vendor. However, this library only supported GEMM function running on GPU and other BLAS subroutines were CPU version. Even worse, those CPU BLAS functions were single thread. As a result, we cannot sufficiently exploit multi cores on CPU. Thus, we used other optimized BLAS library GotoBLAS instead of ACML-GPU on CPU. Because of the conflict of function definition in GotoBLAS and ACML-GPU, we changed

Table 34.1 The configuration of test bed

CPU	AMD Phenom II X4 940 (3.0 GHz)
Memory	8 GB DDR2
GPU	AMD Radeon HD 5850
OS	Ubuntu 9.04
Compiler	GCC-4.1.3
GPU BLAS library	ACML-GPU 1.1
CPU BLAS library	GotoBLAS2 1.13
MPI library	OpenMPI 1.2.8

the interface of GotoBLAS by adding a postfix to every BLAS subroutine name. Meanwhile, we modified the calling CPU BLAS codes in GHPL to use GotoBLAS directly.

Furthermore, GEMM subroutine in ACML-GPU was a blocking function so that it could not return immediately and called GEMM subroutine in CPU BLAS library. To achieve GEMM running on CPU and AMD GPU at the same time, we used multi thread technique to start a helper thread calling CPU GEMM function.

34.4 Performance Evaluation

As Table 34.1 shown, the test bed equipped with AMD quad cores CPU, AMD Radeon HD 5850 GPU and 8 GB memory. We used GotoBLAS2 1.13 on CPU and ACML-GPU 1.1 on GPU.

Firstly, we tested our hybrid of CPU and GPU GEMM function with different ratio R and using different CPU BLAS threads in both single and double precision. From Fig. 34.3 it can be observed that using 3 CPU threads could reach the best performance on single and double precision square matrix multiplication. Thus, we suspected that ACML-GPU library should use 1 CPU core to control GPU. When we used 4 CPU BLAS threads, there would be a confliction making the performance low. The optimal ratios of single and double precision were around 10 and 7.5 %, respectively.

Secondly, we tested hybrid GEMM function with typical matrix multiplication in LINPACK benchmark, which was multiplying $(N - nb) \times nb$ and $nb \times (N - nb + 1)$ matrices. In this workload, N presents the problem size and nb means block size. Figure 34.4 shows us that the performance of calculating typical matrix multiplication was significant slow than square matrices multiplication in both single precision and double precision. Because of this issue of ACML-GPU implementation, it would limit our GHPL performance.

At last, we compared our GHPL to double precision HPL software package which also called our CPU and GPU hybrid DGEMM function to gain the benefit from AMD GPU. Then, we tuned the input data file HPL.dat of HPL and GHPL on our CPU and AMD GPU test platform. According to our results in Table 34.2, the speedup of GHPL over HPL was about 3.21.

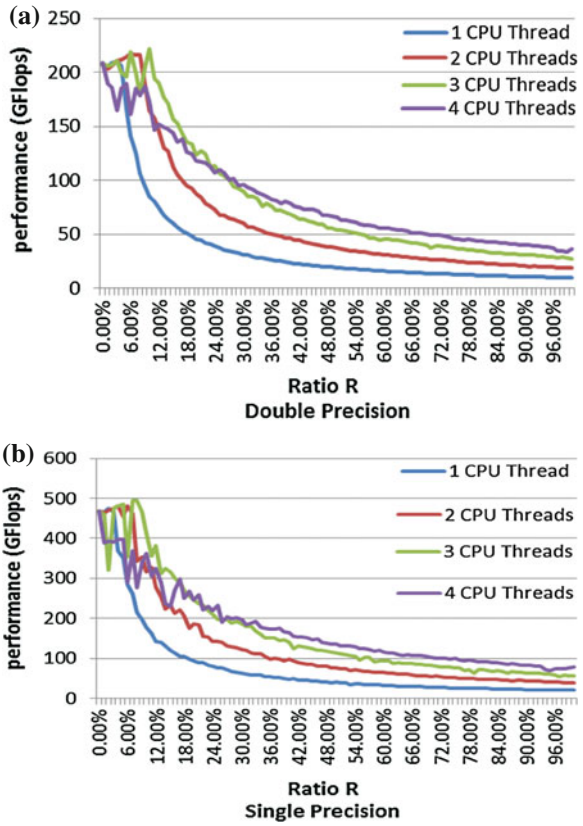


Fig. 34.3 The hybrid of CPU and GPU GEMM performance on square matrices (m = n = k = 4096)

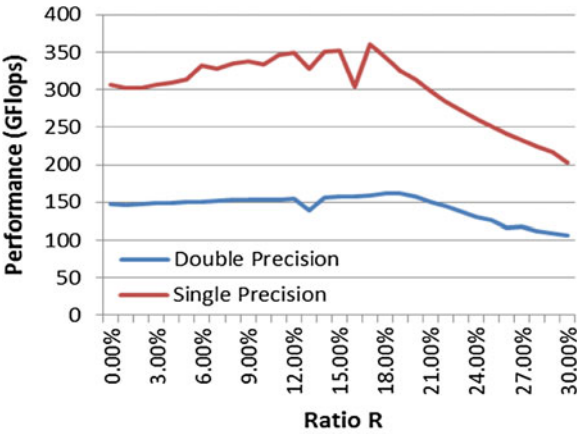


Fig. 34.4 The hybrid of CPU (3 CPU threads) and AMD GPU GEMM performance on typical matrices of LINPACK benchmark(m = N − nb, n = N − nb + 1, k = nb, N = 203040, nb = 1536)

Table 34.2 Optimal results of HPL and GHPL

	T/V	N	NB	P × Q	Gflops
HPL	R00R2C8	28160	1536	1 × 1	95.29
GHPL	WR00R2R16	23040	1792	1 × 1	306.5

34.5 Conclusions

In this paper, we ported our previous mixed precision high performance LINPACK software package GHPL to AMD GPU. By the hybrid of CPU and GPU GEMM function, we could use AMD GPU to accelerate LINPACK benchmark. The speedup of GHPL over HPL was 3.21 on our test bed. In addition, we analyzed the performance of ACML-GPU library and figured out the limitation, low performance of multiplying $(N-nb) \times nb$ and $nb \times (N-nb+1)$ matrices which was used in updating tail matrix step in HPL.

In future, we could try other AMD GPU BLAS library such as GATLAS (http://golem5.org/gatlas/) and AAPPML (http://developer.amd.com/gpu/appmathlibs/Pages/default.aspx) to improve the performance. Meanwhile, we could apply adaptive tuning technique to fine the optimal ratio R based on the variable workload.

Acknowledgments This work is partly supported by the National 863 Plan of China (No.2006AA01A125, No. 2009AA01A129, No.2009AA01A134), the China HGJ Project (No. 2009ZX01036-001-002), the Knowledge Innovation Program of the Chinese Academy of Sciences (No.KGCX1-YW-13), the Ministry of Finance (No. ZDYZ2008-2).

References

Kurzak J, Dongarra J (2006) Implementation of the mixed-precision high performance LINPACK benchmark on the CELL Processor. University of Tennessee Computer Science, Technical report UT-CS-06-580, LAPACK Working Note 177, Sept 2006

Langou J, Langou J, Luszcek P, Kurzak J, Buttari A, Dongarra JJ (2006) Exploiting the performance of 32bit floating point arithmetic in obtatining 64 bit accuracy. In: Proceedings of the 2006 ACM/IEEE conference on supercomputing, Tampa, 2006

Moler CB (1967) Iterative refinement in floating point. J ACM 14(2):316–321

Wang L, Zhang Y, Zhang X, Liu F (2010) Accelerating linpack performance with mixed precision algorithm on CPU+GPGPU heterogeneous cluster, In: Proceedings of the 10th IEEE international conference on computer and information technology, 2010, pp 1169–1174

Wilkinson JH (1965) The algebraic eigenvalue problem. Clarendon, Oxford

Chapter 35
Parallel Lattice Boltzmann Method on CUDA Architecture

Weibing Feng, Wu Zhang, Bing He and Kai Wang

Abstract In this article, an implementation of 2D Lattice Boltzmann Method by CUDA is presented. The simulation has been finished by GPU very well, which has the latest core Tesla C1060. From the results, the speedup of simulation on GPU is 30 times more than on a CPU, and the summit speedup is 41 times.

Keywords CUDA · Lattice Boltzmann Method

35.1 Introduction

Since 2003, there were many implements of complex computation by GPU. In the early time, somebody used graphics APIs and Cg language to manipulate the GPU operator in order to compute the data from memory which had stored in the Video RAM. From that way, they have gotten 4.6 times faster than their CPU cluster implementation (Fan et al. 2004). After the CUDA-enable GPU having been designed by Nvidia Corporation, more and more applications had been finished on this kind of GPU. Multilevel memory architecture and multi-threads parallel computation method bring flexible programming ability to us. Somebody have used GPU to get more than 10 times faster than their CPU implementation in some computation fields. Such as, Daniel Cederman and Philippas Tsigas had finished the GPU-Quicksort by CUDA-enable GPU (Daniel Cederman et al. 2008), Gunther etc had shown a method allowing ray tracing with CUDA (Gunther et al. 2007), Aspuru-Guzik had completed computational chemistry using GPUs (Leslie Vogt et al. 2008), and so on.

Due to the facts that Lattice Boltzmann Method operate on a finite difference grid, are explicit in nature and require only next neighbor interaction they are very suitable for the implementation on GPU. A speedup of at least one order of magnitude could be

W. Feng (✉) · W. Zhang · B. He · K. Wang
School of Computer Science and Engineering,
Shanghai University, Shanghai 200072, China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 561
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_35,

achieved by a GPU compared to an implementation on a CPU. Applications for LBM simulations in graphics hardware range from real-time ink dispersion in absorbent paper (Nelson et al. 2005), dispersion simulation and visualization for urban security (Qiu et al. 2004), simulation for soap bubbles (Wei et al. 2004), melting and flowing in multiphase environment (Zhao et al. 2006) and visual simulation of heat shimmering and mirage (Zhao et al. 2007).

This paper is organized as follows: Sect. 35.2 illustrates the architecture of CUDA and the performance of Tesla C1060 which will be used in the experiments. Sect. 35.3 introduces the Lattice Boltzmann Method and the D2Q9 model. In sect. 35.4, we detail the simulation algorithm of LBM and the program frame. Sect. 35.5 presents the experiments' results and the analysis of them.

35.2 The CUDA of Nvidia

35.2.1 Introduction of CUDA

CUDA, which is short for Compute Unified Device Architecture, has been designed to complete parallel computation with GPU by Nvidia Corporation. Users can through a new programming interface to complete computationally intensive applications with GPU. The parallel computation programs can be strongly simplified by using the standard C language and some C extension APIs. CUDA Toolkit give users some complete software development solutions for programming CUDA-enabled GPU, such as standard FFT and BLAS libraries for GPU. CUDA programs are transplantable among almost all Operating System, such as Windows, Linux and Mac OS (Nvidia 2001). We used the version 2.0 for this application.

35.2.2 Efficient Parts in CUDA

CUDA has a lot of APIs for GPU computation, which are all C language extension. But a small subset of the APIs following needed for our LBM kernel program is discussed. The GPU is looked as a parallel computation device which can execute a large number of threads in the same time. The CPU controls the whole program as a host and calls the GPU to do some computation jobs. After the parallel computation job being finished, CPU uses the result to do other things. When the GPU is running the parallel program, CPU can do other things at the same time. Both the host and the device maintain their own DRAM, named as host memory and device memory, respectively. Users can transport the data from one DRAM to another, such as host to host, host to device, device to host or device to device. And one procedure only needs optimized API calls that utilize the device's high performance Direct Memory Access (DMA) engines.

There are two most important design thoughts in CUDA. One is the layout of threads. Threads in GPU are divided into grids, and every grid is divided into blocks.

The size of grid and blocks could be set in the front main function of a CUDA program. Usually, the size of height and width in grid or block are the same, so that could provide an easy way to calculate the number of threads in a grid or block. A thread block is a batch of threads that can cooperate together by sharing data in shared memory. And they are all finish their tasks under the specifying instruction of synchronization API in the kernel function. Every thread is identified by its thread ID, which can be calculated from grid ID, block ID and the width and height of computation area. An application can specify a block as a 2D or 3D array and identify each thread using a 3-component index. The layout of a block is specified in a function call to the device by a variable type dim3, which contains three integers defining the extensions in x, y, z. If the block's dimension is 2, the component z can be omitted. There is a limit that a block can contain maximum 512 threads in the current CUDA version. In the program, we partition the running threads by block size and run these threads by kernel functions.

The other important role is the multilevel device memory. There are four levels memory on the chip of GPU. The global type memory is like the CPU DRAM, and every thread could access data from global memory. There is a special memory for graphics storage named texture memory, but users only can read data from it. The shared memory can be used as a faster DRAM in each thread block, and every thread in this block can access it. Threads in one block can access the same data in shared memory. In this way, the data in shared memory can be reuse instead of access from global memory to improve the efficient of computation. There are lots of register for each thread, these register only prepare for one thread as a local memory. In the kernel function, the declared variables are stored in the registers to provide the fastest access.

35.3 Lattice Boltzmann Method

Lattice Boltzmann Method is a numerical method to solve the Navier-Stokes equations, which have a remarkable ability to simulate single and multiphase fluids (Michael et al. 2006). Lattice Boltzmann models vastly simplify Boltzmann's original positions and microscopic momentums from a continuum to just a handful and similarly discrete time into distinct steps. Because Lattice Boltzmann Method has some advantages, such as flexible geometrical characteristic, natural parallel, easy to be implement and high precision (Guo Zhaoli et al. 2002).

There are equations to describe the method as follow (Guo Zhaoli et al. 2002):

$$\rho = \sum_{a=0}^{8} f_a, \tag{35.1}$$

$$\boldsymbol{u} = \frac{1}{\rho} \sum_{a=0}^{8} f_a \boldsymbol{e}_a, \tag{35.2}$$

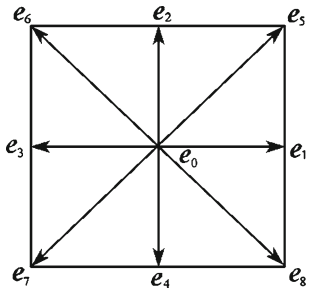


Fig. 35.1 D2Q9 model has 9 velocity directions

stand for the density of fluid and the velocity vector respectively,

$$f_a(\boldsymbol{x} + \boldsymbol{e}_a \Delta t, t + \Delta t) = f_a(\boldsymbol{x}, t) - \frac{[f_a(\boldsymbol{x}, t) - f_a^{eq}(\boldsymbol{x}, t)]}{\tau}, \tag{35.3}$$

is BGK relaxation function for the collision, where $\frac{[f_a(\boldsymbol{x},t)-f_a^{eq}(\boldsymbol{x},t)]}{\tau}$ becomes to the collision item. Though they can be combined in a simple procedure, collision and propagation steps must be separated when the solid boundary comes up, for the rebound boundary condition is a separate collision. The balance status distributing function is as follow:

$$f_a^{(eq)} = \omega_a \rho \left[1 + \frac{\boldsymbol{c}_a \cdot \boldsymbol{u}}{c_s^2} + \frac{(\boldsymbol{c}_a \cdot \boldsymbol{u})^2}{2c_s^4} - \frac{u^2}{2c_s^2} \right], \tag{35.4}$$

In this equation, $c_s = c/\sqrt{3}$, ω_a stands for the weight coefficient which concerns the discrete velocity vector.

In our implementation, we use D2Q9 model which has 2 dimensions and contains 9 velocities, as following: (Fig. 35.1)

Each direction has its collision equation and sub-velocity propagation direction. After finishing each iterative step, there is a formula to calculate the density and velocity of each lattice. And the density and velocity are used to calculate the equate equilibrium function, which is for updating the lattice's status.

35.4 Simulation of 2D Square Cavity Fluid

In our experiments, we choose a classic fluid to simulate which is 2D square cavity fluid. There is a flow inlet boundary in this model, other three boundaries are solid. This inlet with a low velocity drives the whole calm fluid in square cavity in order to change the old condition to the new. When the newer condition has tiny change from the old one, this fluid could be treated as a stable fluid. In this paper, we are mainly talking about the speedup of simulation with GPU instead of how to get final stable fluid. Certainly, the speedup stands for the computation speed a GPU than a CPU under the precondition of correct result.

There are four steps in the main stream:

- allocate the memory on CPU and GPU, initial data on CPU and GPU;
- set the grid size and block size;
- simulating the fluid in an iterative procedure;
- get the result from GPU and free the memory on CPU and GPU.

Each iterative procedure can be divided into:

- Collision kernel function;
- copying data on GPU;
- Propagation kernel function;
- storing data as the next step's initial data.

Because threads in one block running at the same time and transporting the velocity from one to another will cover the nearby lattice in the propagation step. So we need allocate two same size memory blocks to store the whole fluid, one block set as the resource and one set as the destination.

In the Collision kernel function, we need judge the thread ID whether on the boundary or not in order to set the correspond values to computation equations. In the solid boundary part, we use the rebound format to exchange the data in opposite direction.

In the Propagation kernel function, we use the two memories to set as the resource and the destination. After propagating the velocity, we should copy the data in destination to the resource one to keep they be the same, and set the resource one as the next collision's initial data, set the destination one as the next propagation's target.

35.5 Experiments

35.5.1 Hardware Platform and Software Platform

We use the Tesla C1060 GPU on the HP workstation 8600 to complete this application. Tesla C1060 has a GTX280 GPU core of 240 multiprocessors and 4 GB video memory. The bit width of this device is GDDR3 512bit, the video bandwidth is 102 GB per second and the frequency of memory is 800 MHz (Nvidia 2001).

There are four CPU cores on this workstation, each of them is Xeon E5410 2.66 GHz.

In this experiment, we use CUDA 2.0 Toolkit final vision and CUDA SDK 10 vision under the Linux operating system. The stream line pictures are painted with ParaView Vision 3.2.

35.5.2 Experiment Results Under GPU Comparing With CPU

There are two pictures to show the same simulation results after 10,000 iterative steps with grid 256×256 from CPU and GPU respectively: (Fig. 35.2)

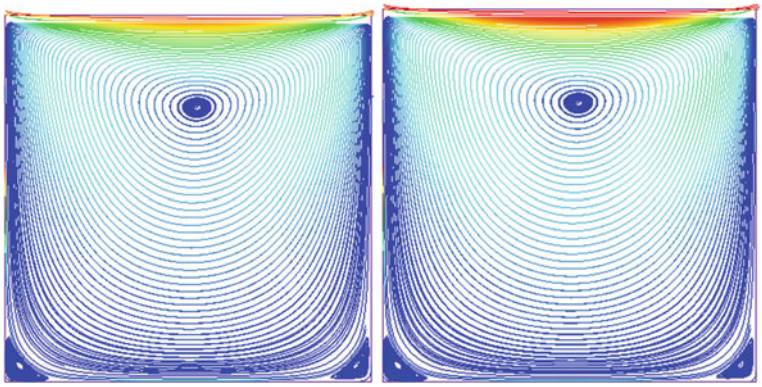


Fig. 35.2 The left is CPU, the right is GPU, and they are the same (Re = 50)

Table 35.1 Speedup of GPU-CPU after 10,000 iteration steps

GPU-CPU Speedup					
Grid size of fluid		256×256	512×512	1024×1024	2048×2048
GPU	4	18.67X	19.17X	14.24X	12.21X
Block	8	26.97X	29.34X	24.89X	19.58X
Size	16	31.15X	41.20X	41.65X	32.11X

There is a table show the partial result from our experiments which include the fluid grid 256 × 256, 512 × 512, 1024 × 1024. The sizes of grids are the multiple of 16 in order to match the blocks fully which is the best size to be divided the threads' grid in these experiments. And the block including 256 threads is the nearest size under maximum limit 512 threads per block. (Table 35.1)

From the tables, we can get the maximum GPU-CPU speedup is 41. And the speedups of each kind of grid size are the same between different iteration steps. So we can get the conclusion that the iteration steps can't impact the speedup, and the block size 16 is the best to divide the thread grid, and under the block size 16 the speedups are more than 30X. In this way, we can save much time to simulate a 2-dimension fluid.

35.6 Conclusions

In this paper, we have introduced a way to simulate 2D square cavity fluid in Lattice Boltzmann Method by using CUDA-enabled GPU Tesla C1060. From the results, we can get well speedup of GPU to CPU. Although our D2Q9 model of LBM using rebound format to solve the fluid boundary, we believe that computation of the boundary solution format the more flexible the more speedup of GPU to CPU we can get. At the end of December of 2008, CUDA had been a part of OpenCL being established by (KHRONOS 2003) , which is an industry consortium creating

open standards for the authoring and acceleration of parallel computing, graphics and dynamic media on a wide variety of platforms and devices. And we believe GPU will play an important role in parallel computation.

Appendix: The kernel Function of Computing Collision

```
#ifndef _COLLISION_KERNEL_H_
#define _COLLISION_KERNEL_H_
#define BLOCK_DIM 16
__global__ void Collision(
            float *k_fNew0, float *k_fNew1, float *k_fNew2,
            float *k_fNew3, float *k_fNew4, float *k_fNew5,
            float *k_fNew6, float *k_fNew7, float *k_fNew8)
{
      const float rho_0 = 1.0f;
      const float u_0 = 0.1f;
      const float tau = (1.0f * 6.0 + 1.0) / 2.0;
      const float tau_inv = 1.0 / tau;
      const int fluidsize = 2048;
      unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
      unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
      unsigned int index = yIndex * fluidsize + xIndex;

      float rho, vx, vy, temp;
      float feq0, feq1, feq2, feq3, feq4, feq5, feq6, feq7, feq8;

      if(index >= fluidsize * (fluidsize - 1))
      {
            rho = rho_0;
            vx = u_0;
            vy = 0.0;

            feq0 = 4.0 / 9.0 * rho * (1.0 - 1.5 * (vx * vx + vy * vy));
            feq1 = rho *(1.0/9.0+1.0/3.0*vx+1.0/2.0*vx*vx-1.0/6.0*(vx*vx+vy*vy));
            ……
            feq8 = rho*(1.0/36.0-1.0/12.0*(vy-vx)+1.0/8.0*(vy-vx)*(vy-vx)
                  -1.0/24.0*(vx*vx+vy*vy));

            //compute new distributing function
            k_fNew0[index] += (feq0 - k_fNew0[index]) * tau_inv;
            k_fNew1[index] += (feq1 - k_fNew1[index]) * tau_inv;
            ……
```

```
        k_fNew8[index] += (feq8 - k_fNew8[index]) * tau_inv;
    }
    else if(((index + 1) % fluidsize == 0) || ((index % fluidsize) == 0) || (index < fluidsize))
    {
        temp = k_fNew1[index];
        k_fNew1[index] = k_fNew3[index];
        k_fNew3[index] = temp;

        temp = k_fNew2[index];
        k_fNew2[index] = k_fNew4[index];
        k_fNew4[index] = temp;

        temp = k_fNew5[index];
        k_fNew5[index] = k_fNew7[index];
        k_fNew7[index] = temp;

        temp = k_fNew6[index];
        k_fNew6[index] = k_fNew8[index];
        k_fNew8[index] = temp;
    }
    else
    {
        // compute the density and the velocity
        rho = k_fNew0[index] + k_fNew1[index] + k_fNew2[index] +
              k_fNew3[index] + k_fNew4[index] + k_fNew5[index] +
              k_fNew6[index] + k_fNew7[index] + k_fNew8[index];
        vx = (k_fNew1[index] - k_fNew3[index] + k_fNew5[index] -
              k_fNew6[index] - k_fNew7[index] + k_fNew8[index]) / rho;
        vy = (k_fNew2[index] - k_fNew4[index] + k_fNew5[index] +
              k_fNew6[index] - k_fNew7[index] - k_fNew8[index]) / rho;

        //compute the balance status distributing function
        feq0 = 4.0/9.0*rho*(1.0-1.5*(vx*vx+vy*vy));
        feq1 = rho*(1.0/9.0+1.0/3.0*vx+1.0/2.0*vx*vx-1.0/6.0*(vx*vx+vy*vy));
        ……
        feq8 = rho*(1.0/36.0-1.0/12.0*(vy-vx)+1.0/8.0*(vy-vx)*(vy-vx)
               -1.0/24.0*(vx*vx+vy*vy));
        //compute new distributing function again
```

```
            k_fNew0[index] += (feq0 - k_fNew0[index]) * tau_inv;
            k_fNew1[index] += (feq1 - k_fNew1[index]) * tau_inv;
            ……
            k_fNew8[index] += (feq8 - k_fNew8[index]) * tau_inv;
        }
        //the synchronization in block
        __syncthreads();
    }
    #endif
```

References

Cederman D, Tsigas P (2008) A Practical quicksort algorithm for graphics processors. In: Lecture Notes in Computer Science, vol 5193/2008. Springer, Heidelberg, pp 246–258

Chu NS-H, Tai C-L (2005) Moxi: Real-time ink dispersion in absorbent paper. ACM Trans Grap 24(3):504–511

Fan Z, Qiu F, Kaufman AE, Yoakum-Stover S (2004) GPU cluster for high performance computing. In: Proceedings of ACM/IEEE supercomputing conference 47–59 2004

Gunther J, Popov S, Seidel H-P, Slusallek P (2007) Realtime ray tracing on GPU with BVH-based packet traversal. In: Proceedings of the IEEE/Eurographics symposium on interactive ray tracing 113–118 2007

KHRONOS web site. www.khronos.org. <!– Missing/Wrong Year –>

Nvidia web site. www.nvidia.com. <!– Missing/Wrong Year –>

Qiu F, Zhao Y, Fan Z, Wei X, Lorenz H, Wang J, Yoakum-Stover S, Kaufman AE, Mueller K (2004) Dispersion simulation and visualization for urban security. IEEE Vis 7:553–560

Sukop MC, Thorne DT Jr (2006) Lattice Boltzmann modeling an introduction for geoscientists and engineers, 1st edn. Springer, Heidelberg

Vogt L, Olivares-Amaya R, Kermes S, Shao Y, Amador-Bedolla C, Aspuru-Guzik A (2008) Accelerating resolution-of-the-identity second-order Møller-Plesset quantum chemistry calculations with graphical processing units. J Phys Chem A 112(10):2049–2057

Wei X, Zhao Y, Fan Z, Li W, Qiu F, Yoakum-Stover S, Kaufman A (2004) Lattice-based flow field modeling. IEEE Trans Vis Comput Grap 10(6):719–729

Zhao Y, Han Y, Fan Z, Qiu F, Kuo YC, Kaufman A, Mueller K (2007) Visual simulation of heat shimmering and mirage. IEEE Trans Vis Comput Graph 13(1):179–189

Zhao Y, Wang L, Qiu F, Kaufman A, Mueller K (2006) Melting and flowing in multiphase environments. Comput Graph 30(4):519–528

Zhaoli G, Chuguang Z, Qing L (2002) Lattice Boltzmann method for hydrodynamics. Hubei Science and Technology Press, China

Part VIII
Visualization

Chapter 36
Iterative Deblurring of Large 3D Datasets from Cryomicrotome Imaging Using an Array of GPUs

Thomas Geenen, Pepijn van Horssen, Jos A.E. Spaan, Maria Siebes and Jeroen P.H.M. van den Wijngaard

Abstract The aim was to enhance vessel like features of large 3D datasets ($4000 \times 4000 \times 4000$ pixels) resulting from cryomicrotome images using a system specific point spread function (PSF). An iterative (Gauss-Seidel) spatial convolution strategy for GPU arrays was developed to enhance the vessels. The PSF is small and spatially invariant and resides in fast constant memory of the GPU while the unfiltered data reside in slower global memory but are prefetched by blocks of threads in shared GPU memory. Filtering is achieved by a series of unrolled loops in shared memory. Between iterations the filtered data is stored to disk using asynchronous MPI-IO effectively hiding the IO overhead with the kernel execution time. Our implementation reduces computational time up to 350 times on four GPU's in parallel compared to a single core CPU implementation and outperforms FFT based filtering strategies on GPU's. Although developed for filtering the complete arterial system of the heart, the method is general applicable.

36.1 Introduction

Cardiovascular disease, currently one of the leading causes of death in western society, attracts a significant amount of research devoted to early diagnosis and further disease prevention. Coronary artery disease and ensuing myocardial infarction often result from atherosclerosis which causes gradual narrowing of the vascular lumen. In case of gradual under-perfusion and development of tissue ischemia, the vascular architecture of the coronary arterial tree may alter, leading to sprouting and outgrowth

T. Geenen (✉) · P. van Horssen · J. A. E. Spaan ·
M. Siebes · J. P. H. M. van den Wijngaard
Department of Biomedical Engineering and Physics Academic Medical Center,
University of Amsterdam, Amsterdam, Netherlands
e-mail: geenen@gmail.com

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 573
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_36,

of new vasculature and growth of coronary collateral vessels connecting areas with sufficient tissue perfusion with areas that have insufficient perfusion.

To study coronary arterial tree organization, an imaging cryomicrotome was previously developed at the Department of Biomedical Engineering and Physics of the Academic Medical Center, University of Amsterdam (Spaan et al. 2005). With this imaging technique, the full arterial vascular tree of the myocardium can be imaged up to the arteriolar level. In this procedure, the vascular tree under investigation is cannulated, flushed with buffered saline and filled with a fluorescent plastic penetrating the arterial vessels up to approximately $10\,\mu m$. Subsequently, using a dedicated light source equipped with a corresponding excitation filter for the fluorescent replica material and an emission filter for the CCD camera, a thin slice of preset distance, e.g. $25\,\mu m$, is cut from the frozen sample after which a high resolution image is obtained from the remaining bulk material. These images can then be recombined as a 3 dimensional stack to represent the coronary arterial architecture. Since the imaging cryomicrotome has become available, it is possible to study the coronary circulation with increased resolution and across a variety of scales yielding high resolution 3 dimensional datasets. Important clinical applications involve the study of the growth of coronary collaterals in the diseased or failing myocardium (van den Wijngaard et al. 2010) as well as the perfusion heterogeneity illustrated by the distribution of fluorescent labeled microspheres (van Horssen et al. 2010).

The cryomicrotome images currently obtained measure 4000×4000 pixels, yielding $25\,\mu m \times 25\,\mu m \times 25\,\mu m$ cubic voxels. One dataset, describing an entire pigmyocardium, may therefore comprise about 6.4×10^{10} voxels. Subsequent data analysis includes the iterative filtering, dark current removal and the subsequent thinning of the vascular tree medial lines to obtain vascular centerlines (Palágyi 1998), thereby obtaining the topologically representative vascular tree skeleton. Currently, the enhancement and filtering of the uncompressed raw data is a time consuming process, which may take several days on a designated high end desktop personal computer.

We are able to reduce the end-to-end runtime for these operations to the order of minutes, a speedup of ca. 350, by using an iterative deblurring filter, implemented on an a array of GPU's. The skeletonization is not implemented on GPU hardware since efficient 3D skeletonization algorithms take already on the order of minutes to process an image stack (Palágyi 1998) and are typically IO bound, spending most of the walltime in reading images to and from disk, rendering them unsuitable for GPU acceleration.

36.2 Method

To filter stacks of images obtained from a cryomicrotome, we propose to use an iterative deblurring method using the PSF associated with light diffraction in the sample during imaging of the fluorescent vessels (Rolf et al. 2008). The operation of the PSF on an image can be written as $\mathbf{Ax} = \mathbf{b}$, with $\mathbf{x}$ the original image, $\mathbf{b}$ the blurred image and $\mathbf{A}$ the filter operator, PSF.

We obtain **b** from the cryomicrotome and reconstruct **x**, the original image, by applying several Gauss-Seidel type iterations, Eq. 36.1.

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j>i} a_{ij} x_j^k - \sum_{j<i} a_{ij} x_j^{k+1} \right), \quad i = 1, 2, ..., n \tag{36.1}$$

By definition a_{ii} is one and as a result of the cutting away of slices during the data acquisition **A** is asymmetric, effectively reducing the number of operations per iteration by half. This results in the following iterative scheme, Eq. 36.2.

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j>i} a_{ij} x_j^k \right), \quad i = 1, 2, ..., n \tag{36.2}$$

Since we now no longer have to deal with the case $j < 1$ we have an iterative method that can be trivially implemented in parallel. The filter operation can be written as a matrix vector product, Eq. 36.2, and performed in the spatial domain, or the PSF can be written as a vector **f** and the filter operation as a convolution $\mathbf{f} * \mathbf{x}$. Convolution operations are usually executed in the spectral domain since efficient transformation routines to the spectral domain (FFT) are readily available and the computational complexity is low, order $O((34/9)n \cdot \log_2(n))$ for a filter operation in the spectral domain (Frigo and Johnson 2005), with n the number of pixels. The computational complexity of spatial convolution increases with increasing filter size k as order $O(k \cdot n)$. This renders spatial convolution more expensive when the filter length is larger than $34/9 \cdot \log_2(n)$, usually several tens of entries (Fialka and Cadik 2006).

36.3 Implementation

36.3.1 Iterative Spectral Deconvolution

Deconvolution filters in the spectral domain are known to introduce artifacts in the filtered data since part of the spectrum that is filtered out was part of the unfiltered data (Theußl et al. 2000). This is especially relevant when sharp contrasts in intensity are present in the unfiltered data, since these sharp contrast usually are represented by a broad spectrum. In the presented case, ringing may occur, leading to false detection of spurious vessel remnants, Fig. 36.1.

To prevent ringing from occurring, the filter in the spectral domain has to be supplemented with an anti-ringing filtering step. The latter operation is an additional step compared to a filter in the spatial domain, since filtering in the spatial domain does not introduce ringing artifacts.

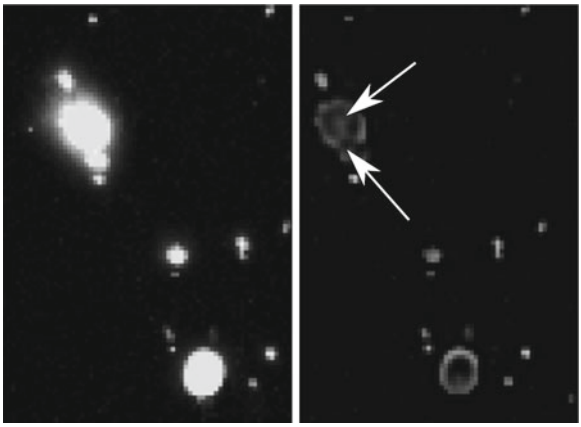


Fig. 36.1 Examples of ringing artifact after filtering without anti-ringing operation. The raw cryomicrotome data (*left panel*) display several arteries in crossection with sharp contrasts in intensity. After filtering in the spectral domain (*right panel*), ringing artifacts are visible. The *lower arrow points* at ringing that results in a breakup of the artery outer boundary. The *upper arrow* shows a typical sequence of lighter and darker bands around a sharp pixel value contrast

Apart from an increased computational load due to a more complex filter, there is an additional computational concern in spectral domain filtering of very large datasets that do not fit in GPU device memory. Since spectral transformations implicitly assume periodicity of the dataset, a periodic or reflective expansion of the data subset is required when this periodicity requirement is not met. In case of a dataset of $4000 \times 4000 \times 4000$ pixels, represented as floats in the spectral domain, more than 256 GB of data is required. Assuming all operations are performed in place and mirroring the data will increase the amount of data by a factor of 2 per slice, the data needs to be divided in 128 slices to fit the 4 GB of device memory per GPU available. In case the data is divided in blocks, the amount of data increases even 8 fold and the number of operations will increase up to 2–8 times.

An alternative strategy, common in parallel 3D FFT implementations (Dmitruk et al. 2001), is to perform a global transformation where first each slice in 2D is transformed followed by a subsequent 1D transformation in the z-direction. The downside of this approach, when implemented on GPU's, is the number of times the data has to be transported to and from the device. In detail, first the integer pixel data have to be read from disk, loaded on the device and transformed (2D FFT). Second, these data have to be copied back to the host and stored to disk. Third, the data have to be read from disk and loaded back to the device, transposed, transformed in the z-direction (1D FFT), filtered, copied back to the host and fourth, written to disk. Per pixel we need therefore four disk IO operations and four host-device transfers, compared to only two for a spatial convolution. This is even more prohibitive since the data is of float type during most of the data transfer operations occupying two to four times more space compared with raw pixel data, stored as 8-bit integers.

From the literature (Wendykier 2009), it is clear that the amount of time spent in transferring data to and from the device cannot be masked by the FFT execution time, especially for higher dimensional transforms. Additional time needed for file IO, will further increase time consumption, making the filter operation IO bound thus reducing the speedup potential.
As an alternative we propose a filter operation in the spatial domain that can be efficiently implemented on GPU hardware and requires a minimal amount of data transfer operations.

36.3.2 Iterative Spatial Deconvolution

Filtering data in the spatial domain does not introduce the artifacts described in the previous section. We can therefore implement a straightforward convolution algorithm on the GPU that will be iterative applied as a_{ij} in Eq. 36.1.
Implementation details The filter operation is illustrated in Fig. 36.2. Stacks of images are read in the background using MPI-IO and subsequently loaded to the GPU for filtering.

The filter operation is performed as a series of unrolled loops (Abrash 1989) in shared memory by blocks of 16×16 threads simultaneously (Podlozhnyuk 2009). The size of our kernel, number of threads and the requested resources per thread allow for maximum occupancy of the multiprocessors, Table 36.1. Ensuring maximum occupancy of the multiprocessors allows the thread scheduler to hide latencies by activating threads of other blocks (Nguyen 2007).

The PSF is stored in constant memory. The constant memory is a cached part of global memory of 16 KB, allowing for a considerable PSF size. If threads in a warp (a group of threads running on the same multiprocessor) access the same constant memory entry, as is the case in our implementation, it is as fast as accessing a register (Nguyen 2007).

Blocks of threads load the pixel data to shared memory. Since threads can exchange data in shared memory, we need only to load part of the pixel dataset per thread to have access to all pixel data needed during the filtering operation. Finally by having neighboring threads access neighboring pixels, strided memory access is prevented (Nguyen 2007).

As is clear from the description of our algorithm, convolution in the spatial domain allows to take full advantage of the computational resources on GPU hardware.

36.3.3 Asynchronous IO

Dealing with datasets that do not fit in system memory requires efficient disk IO methods to be implemented. We use asynchronous IO from the MPI-IO library, i.e. `MPI_File_iwrite_at` and `MPI_File_iread_at`. We show in Figs. 36.3 and 36.4 that we can effectively hide the IO with the kernel execution time. Figure 36.2

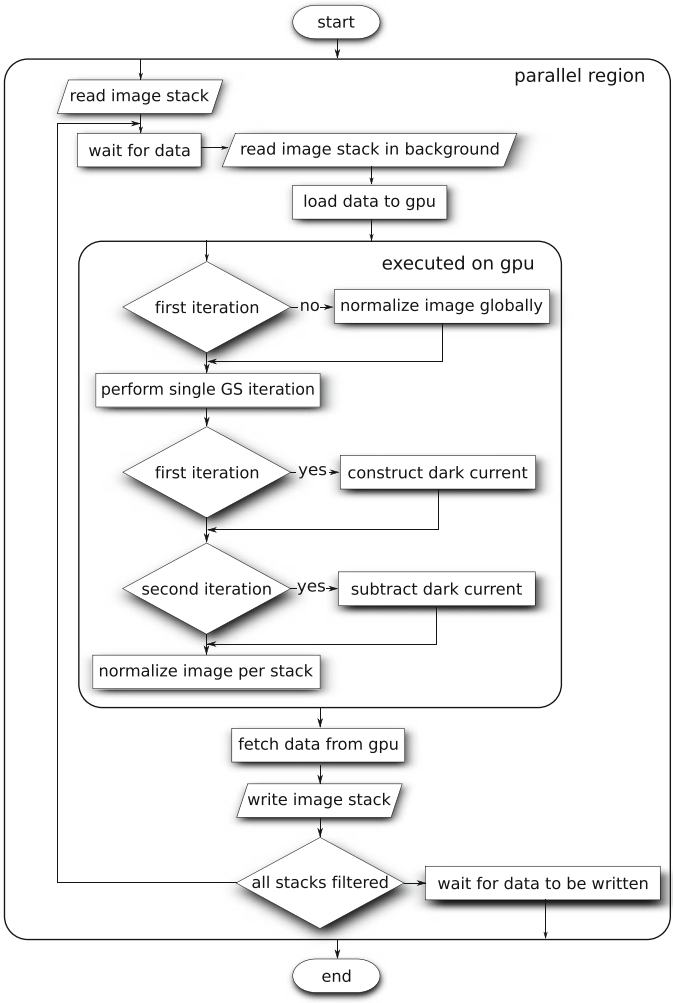


Fig. 36.2 The 3D dataset is deblurred in the spatial domain on multiple GPU's in parallel. Communication between GPU's is handled with MPI. After the application has been started and MPI initialized, we read the first stack of 2D images. When these images have been read, the data is written to the gpu and the next stack of images is read in the background with MPI_File_iread_at. After each iteration the data is normalized per stack to prevent integer overflow during the filter operation. In subsequent iterations a global normalization is performed with the global maximum pixel value. During the first iteration a so called dark current is construct. A dark current captures artifacts of the imaging system that are consistent features over a large number of slices. These artifacts are identified during the first iteration. In the next iteration these artifacts are subtracted from the data. After a stack of images is filtered the filtered data is written to disk in the background using MPI_File_iwrite_at, thus effectively hiding file IO both during read and write operations

Table 36.1 Occupancy for our GPU implementation

Threads/B,	Registers/T,	Shared memory/B,	Treads/M,	Warps/M,	Thread blocks/M
256	16	3272	1024	32	4

The size of our kernel, number of threads and the requested resources per thread allow for maximum occupancy of the multiprocessors. *B* Block, *T* Thread, *M* Multiprocessor

illustrates that IO is hidden with kernel computing time, by reading and writing consecutive blocks of data in the background while the GPU kernel is running. This is further facilitated by the spatial locality of the filter kernel, allowing us to read contiguous blocks of pixel data from disk.

36.4 Results

Comparing performance characteristics of an application implemented on both GPU and CPU hardware is easily biased by experimental setup decisions. Notably the choice of CPU and GPU platform as well as choice of optimization flags and the effort spent in optimizing the code, can influence relative performance significantly. We present performance characteristics of our GPU implementation with respect to an application that we consider representative for a sequential CPU implementation, Table 36.2.

For the spectral deconvolution we compared our GPU application with an efficient spectral implementation (Wendykier and Nagy 2008) where we considered only those parts of the application that can be considered part of the filter kernel in the CPU application. We took this approach since we did not optimize the IO for the CPU implementation as we did for the GPU implementation. Figures 36.3 and 36.4 illustrate that there is sufficient computational complexity in the filter kernel to mask the data transfer operations. Scaling characteristics for an increasing number of GPU's are presented in Figs. 36.5 and 36.6 showing almost optimal end-to-end walltime scaling for the application, for different problem sizes. The parallel effi-

Table 36.2 Speedup results for our GPU implementation relative to a CPU based spatial and spectral filter implementation

Image size	CPU spatial	CPU spectral	GPU (Tesla M1060)	GPU (gtx 285/295)	Speedup
100^2	0.01	0.14	0.629×10^{-4}		159
800^2	0.52	0.86	0.22×10^{-2}	0.16×10^{-2}	325
2000^2	3.32	4.8	0.13×10^{-1}	0.925×10^{-2}	359
4000^2	13.33		0.578×10^{-1}	0.398×10^{-1}	334

The CPU experiments were performed on a AMD Phenom II X4 995 CPU at 3.2 GHz. The spatial convolution was implemented in fortran and compiled with gfortran with optimization flags: -O3 -mtune=athlon64-sse3 –fast-math -unroll-all-loops. For the spectral implementation we refer to Wendykier and Nagy (2008)

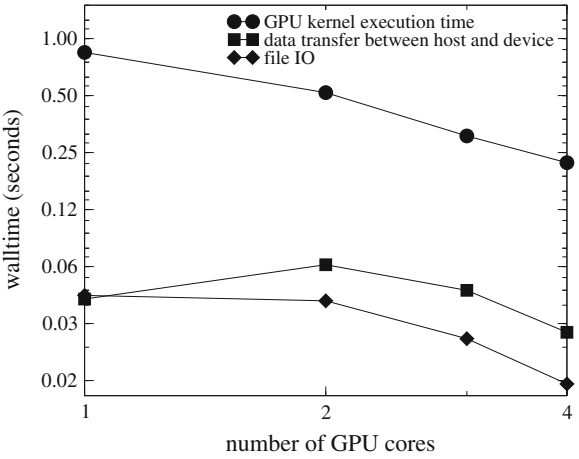


Fig. 36.3 Relative walltime for GPU kernel execution, data transfer between GPU and motherboard and file IO, for a stack of $100 \times 800 \times 800$ pixel images. We show that kernel execution time is by far the most computational demanding operation if we overlap kernel execution time with file IO. These experiments were preformed on a TeslaM106 platform

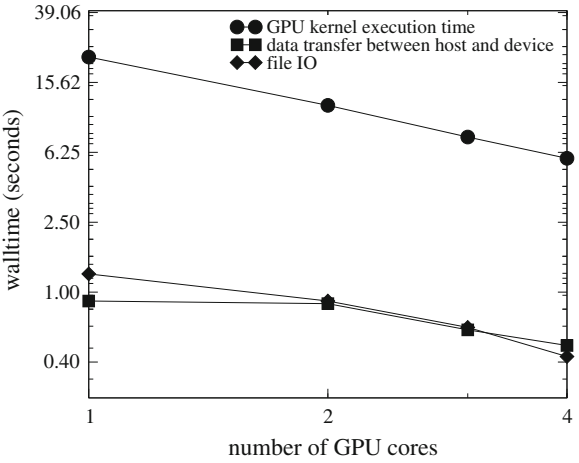


Fig. 36.4 Relative walltime for GPU kernel execution, data transfer between GPU and motherboard and file IO, for a stack of $100 \times 4000 \times 4000$ pixel images. We show that kernel execution time is by far the most computational demanding operation also for this problem size. These experiments were preformed on a TeslaM106 platform

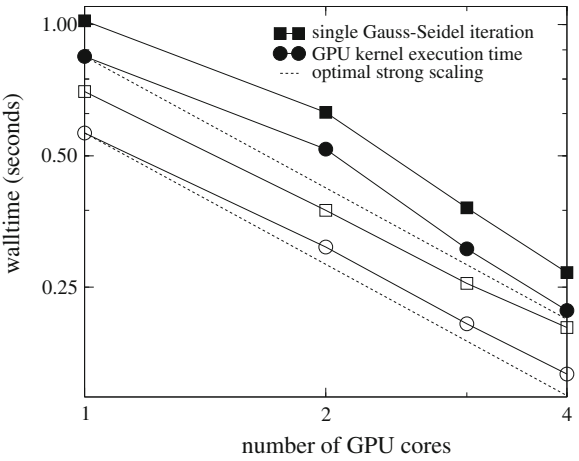


Fig. 36.5 Strong scaling relation for a stack of $100 \times 800 \times 800$ pixel images. We show that our application scales almost optimal up to four GPU's in parallel. Both the GPU kernel as well as all other parts of the filter kernel show near linear scaling, as can be seen from the parallel trend of the GPU kernel speedup compared to the speedup for a complete Gauss-Seidel iterations. Closed symbols represent experiments on a TeslaM106, open symbols experiments on gtx-285 type cards. The parallel efficiency for the experiments on these hardware platforms are 86 and 94 % respectively

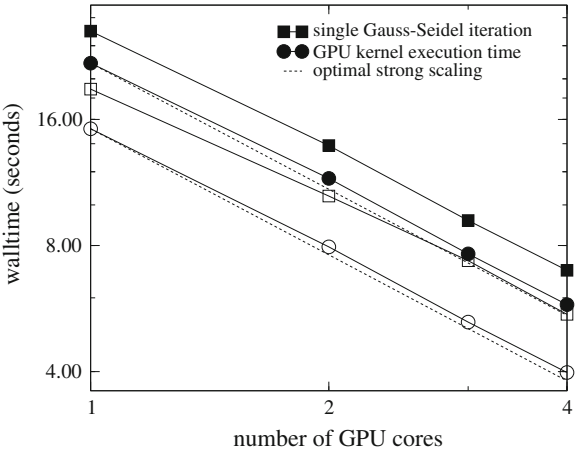


Fig. 36.6 Strong scaling relation for a stack of $100 \times 4000 \times 4000$ pixel images. Again we show that parallel scaling is almost optimal both for the GPU kernel as well as for the other parts of the filter kernel. Closed symbols represent experiments on a TeslaM106, open symbols experiments on gtx-285 type cards. The parallel efficiency for the experiments on these hardware platforms are 93 and 86 % respectively

ciency of the GPU kernels are between 86 and 94 %. We define parallel efficiency as $E_p = \frac{T_s}{T_p n} \times 100\,\%$ (with T_s the walltime for the sequential run, T_p the walltime for the parallel run and n the number of processing cores).

36.5 Discussion

We showed that a spatial deconvolution algorithm using the PSF can efficiently be implemented on graphics hardware using Cuda. We ran our application on a number of GPU's in parallel using MPI to exchange data between GPU's but also for the asynchronous IO methods that are provided by the MPI-IO library. Filtering in the spatial domain ensures data locality during the filter operations, resulting in a fast computational kernel. The data locality of the filter kernels allows prefetching data from disk in the background using asynchronous read and write operations from the MPI-IO library. In addition, this is effectively hidden by kernel execution time. Data prefetching can also be employed during the kernel execution on the GPU where blocks of threads prefetch data to shared memory, used during the filter operation.

We demonstrated that optimal scaling relations with increasing number of GPU's result in over 350 times reduction in end-to-end walltime for the filter operation compared to a single core CPU implementation.

The reduction in walltime allows for the fast reconstruction of the deconvolved three dimensional vascular bed from cryomicrotome image stacks and the reduction in computational time from several days to a few minutes allows for filter parameter optimization. This is especially relevant since a priori information about optimal filter parameters can be unavailable, requiring a series of filter operations with different settings to determine optimal filter configuration. Although developed for imaging the complete arterial system of the heart, the method is general applicable to medical imaging.

Appendix

We present some code snippets to illustrate the implementation of both the Cuda kernel and asynchronous MPI-IO. The general structure of the application is given in Fig. 36.2.

First we start reading a stack of images in the background.

Asynchronous IO

```
int ImageInfo::IreadAt(MPI::Offset offset){

 offset *= ImageSize;                    // The offset in the image stack
 if (FilePointerRead==MPI_FILE_NULL) // open file if not already opened
   FilePointerRead= MPI::File::Open(MPI_COMM_SELF, ReadFilename,
                                    MPI::MODE_RDONLY,
                                    MPI::INFO_NULL);
 RequestRead = FilePointerRead.Iread_at(offset, ReadBuffer,
                                        ReadSize, ReadDatatype);
 return(0);
}
```

Then we loop over images in the stack and filter them. After the filter operation, we scale the images on the GPU to prevent integer overflow from occurring (we use unsigned chars with an integer addressing space of 255). This operation can be combined with other simple operations such as the removal of camera imperfections, i.e. dark current, etc. After the filter operation we write the filtered data to file in the background. In the mean time we start reading the next stack of images. Before each filter operation we wait for the file read operations to be finished. Such a sequence looks in simplified form like:

Main loop

```
IreadAt(offset);
for (int i=0; i<nBlocks+1; i++){

 RequestRead.Wait()
//-----------------------------------------
// update pointers ipnt, ilayer and offset

 IreadAt(offset);
 kernelConvolve <<<grid, threads>>>(Width,
                                    Height,
                                    Filter->Width,
                                    Filter->Height,
                                    Filter->Depth,
                                    ilayer, offset,
                                    &DeviceBuffer[ipnt],
                                    DeviceFilteredImage);
 RequestWrite.Wait();
 IwriteAt(WriteOffset);
}
```

The GPU filter kernels for the Gauss-Seidel deblurring using the PSF is given in the next source-code snippet. This kernel is given in its full form in contrast to the more simplified form in which we presented the main loop.

Spatial convolution kernel

```
// ----------------------
// Define some constants

#define BLOCK_DIM 16
#define FilterLenght 294
__constant__ unsigned char DFilter[FilterLenght]; // This array holds the filter values
__constant__ float BackScale;                     // prevent integer overflow ( > 255)
__constant__ float Dfactor;                       // Sum of all filter values
// ------------------------------------------------------
// We pass several variables and arrays to the filter
// nx       = image slice width and height (for the square case)
// Filternx = the filter width (and height)
// Filternz = filter depth
// offset   = the offset in the image from where the filtering starts
// Data     = the unfiltered dataset
// Filtered = the filtered dataset

__global__ void kernelConvolve(int nx, int Filternx, int Filternz, int offset,
                               unsigned char *Data, float* Filtered)
{
 unsigned char xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x; // Thread number
 unsigned char yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y; // Thread number
 int maxFilterx=Filternx/2;                                   // Filter Heigth/Width (max)
 int minFilterx=-Filternx/2;                                  // Filter Heigth/Width (min)
 int ipnt, jpnt, kpnt, lpnt;                                  // some counters
 int nlocal;                                                  // Image Width/Heigth including edges
 float delta;                                                 // Intermediate filter result
 __shared__ float SumFilter[BLOCK_DIM][BLOCK_DIM+1];          // Intermiate filter results
 __shared__ float SData[BLOCK_DIM][2*BLOCK_DIM+1];            // Unfiltered data in shared memory
// ---------------------------------------------
// Loop over all image points and apply filter

 nlocal = nx + Filternx; // we need the width of the image without the border, width of filter
 if (xIndex < nx && yIndex < nx){                // check if we are inside image bounding box
  SumFilter[threadIdx.y][threadIdx.x] =0;        // put the temp array to zero
  for (int l=0; l<Filternz; l++){                // loop over stack of images
   ipnt=threadIdx.x+xIndex+(yIndex+maxFilterx)*nlocal+l*nlocal*nlocal;// array pointer
   jpnt=l*Filternx*Filternx+ offset + minFilterx;                     // array pointer
   for (int n=minFilterx; n<maxFilterx; n++){                         // loop rows
    lpnt = ipnt+n*nlocal;                                             // array pointer
// ---------------------------------------------------------
// Load Pixel data to shared memory by blocks of threads
// ensure coalescence both by dimension of shared memory array
// and stride in last index.

    SData[threadIdx.y][threadIdx.x]             = Data[lpnt]*BackScale;
    SData[threadIdx.y][threadIdx.x + BLOCK_DIM] = Data[lpnt + BLOCK_DIM]*BackScale;
    kpnt = jpnt + n*Filternx;                                         // array pointer
    lpnt = 0;                                                         // array pointer
// ------------------------------------------------------------------
// since we will use date fetched by other threads, sync threads

    __syncthreads();
// -------------------------------------------------------
// the inner loop for the filter operation is unrolled

    SumFilter[threadIdx.y][threadIdx.x]+=SData[threadIdx.y][threadIdx.x+lpnt++]*(DFilter[kpnt++]);
    SumFilter[threadIdx.y][threadIdx.x]+=SData[threadIdx.y][threadIdx.x+lpnt++]*(DFilter[kpnt++]);
    SumFilter[threadIdx.y][threadIdx.x]+=SData[threadIdx.y][threadIdx.x+lpnt++]*(DFilter[kpnt++]);
    SumFilter[threadIdx.y][threadIdx.x]+=SData[threadIdx.y][threadIdx.x+lpnt++]*(DFilter[kpnt++]);
    SumFilter[threadIdx.y][threadIdx.x]+=SData[threadIdx.y][threadIdx.x+lpnt++]*(DFilter[kpnt++]);
    SumFilter[threadIdx.y][threadIdx.x]+=SData[threadIdx.y][threadIdx.x+lpnt++]*(DFilter[kpnt++]);
    SumFilter[threadIdx.y][threadIdx.x]+=SData[threadIdx.y][threadIdx.x+lpnt++]*(DFilter[kpnt++]);
   }
  }
  kpnt = xIndex + (yIndex+maxFilterx)*nlocal + maxFilterx;            // array pointer
// ---------------------------------------------------------------------------------
// the filtered data is normalization and subtracted from twice the unfiltered data

  delta = 2e0*Data[kpnt] - SumFilter[threadIdx.y][threadIdx.x]*Dfactor;
  delta = fmaxf(0e0,delta);                                           // truncate
// -------------------------------
// copy filtered data to local memory

  Filtered[kpnt] =delta;
 }
}
```

References

Abrash M (1989) Michael Abrash's Graphics Programming Black Book (Special edition). Coriolis Group Books, Scottsdale

Dmitruk P, Wang LP, Matthaeus WH, Zhang R, Seckel D (2001) Scalable parallel FFT for spectral simulations on a beowulf cluster. Parallel Comput 27(14):1921–1936

Fialka O, Cadik M (2006) FFT and convolution performance in image filtering on GPU. In: Proceedings of the Conference on Information Visualization, IEEE Computer Society, pp 609–614

Frigo M, Johnson SG (2005) The design and implementation of FFTW3. In: Proceedings of the IEEE, Special issue on Program Generation, Optimization, and Platform. Adaptation 93(2):216–231

Nguyen H (2007) GPU Gems 3. Addison-Wesley Professional, Boston

Palágyi K (1998) 3D 6-subiteration thinning algorithm for extracting medial lines. Pattern Recognit Lett 19(7):613–627

Podlozhnyuk V (2009) Image convolution with CUDA. White paper, NVIDIA CUDA SDK

Rolf MP, ter Wee R, van Leeuwen TG, Spaan JAE, Streekstra GJ (2008) Diameter measurement from images of fluorescent cylinders embedded in tissue. Med Biol Eng Comput 46(6):589–596

Spaan J, ter Wee R, van Teeffelen J, Streekstra G, Siebes M, Kolyva C, Vink H, Fokkema D, van Bavel E (2005) Visualisation of intramuralcoronary vasculature by an imaging cryomicrotome suggests compartmentalisation of myocardial perfusion areas. Med Biol Eng Comput 10(4):431–435

Theußl T, Hauser H, Gröller E (2000) Mastering windows: improving reconstruction. In: Proceedings of the (2000) IEEE Symposium on Volume Visualization (VVS '00). ACM, New York, pp 101–108

van den Wijngaard J, van Horssen P, ter Wee R, Coronel R, deBakker J, de Jonge N, Siebes M, Spaan J (2010) Organization and collateralization of a subendocardial plexus in end-stage human heart failure. Am J Physiol Heart Circ Physiol 1(298):158–162

van Horssen P, Siebes M, Hoefer I, Spaan J, van den Wijngaard J (2010) Improved detection of fluorescently labeled microspheres and vessel architecture with an imaging cryomicrotome. Med Biol Eng Comput 48:735–744

Wendykier P (2009) High performance Java software for image processing. Emory University, Atlanta, PhD thesis

Wendykier P, Nagy J (2008) Large-scale image deblurring in Java. In: Computational Science ICCS 2008, pp 721–730

Chapter 37
WebViz: A Web-Based Collaborative Interactive Visualization System for Large-Scale Data Sets

Yichen Zhou, Robin M. Weiss, Elizabeth McArthur, David Sanchez, Xiang Yao, Dave Yuen, Mike R. Knox and W. Walter Czech

Abstract We have created a web-based system for multi-user collaborative interactive visualization of large data sets (on the order of terabytes) that allows users in different locations to simultaneously and collectively perform visualizations over the Internet. By leveraging an asynchronous java and XML (AJAX) web development pattern via the Google Web Toolkit (http://code.google.com/webtoolkit/), we are able to provide remote users a web portal to the University of Minnesota's Laboratory for Computational Sciences and Engineering's large-scale interactive visualization system that provides high resolution visualizations to the order of 15 million pixels. Our web application, known as WebViz (for Web-based Visualization System), provides visualization services "in the cloud" and is accessible via a range of devices including netbooks, smartphones, and other web-and JavaScript-enabled mobile devices. This paper will detail the history of the project as well as the current version of WebViz including a discussion of its implementation and system architecture. We will also discuss features, future goals, and our plans for increasing scalability of the system,

R. M. Weiss
Department of Computer Science, Macalester College,
Saint Paul, MN 55105, USA

Y. Zhou, R. M. Weiss, E. McArthur, D. Sanchez, X. Yao, D. Yuen and M. R. Knox
Minnesota Supercomputing Institute, University of Minnesota,
Minneapolis, MN 55455, USA

D. Yuen (✉)
Department of Geology and Geophysics, University of Minnesota,
Minneapolis, MN 55455, USA
e-mail: daveyuen@gmail.com

M. R. Knox
Laboratory of Computational Science and Engineering,
University of Minnesota, Minneapolis, MN 55455, USA

W. Walter Czech
Institute of Computer Science,
AGH, Krakow, Poland

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 587
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_37,

which includes a discussion of the benefits potentially, afforded us by a migration of server-side components to the Google Application Engine (http://code.google.com/appengine/).

37.1 Introduction

With the coming of larger and faster multicore and massively parallel computers to the market, coupled with the introduction of general purpose manycore GPU computing, the amount of data being produced by the scientific community in recent times and into the future is on a steep rise. We now must face the realities of visualizing large amounts of data or being drowned by the sea of unprocessed information. With a typical 3D simulation of convection in a system of $1000 \times 1000 \times 1000$ grid points yielding several Terabytes of data, the challenge becomes how we can visualize the output in an efficient and productive manner that allows for communication and sharing amongst collaborators. The coming challenges are indeed quite daunting when considering the drive for Petascale computing Dunning et al. (2009). To successfully visualize such large data sets, as well as to be able to make these visualizations accessible to the widest possible audience, the traditional methods of post-processing must be abandoned. For this reason, we have begun exploring the possibilities of a new approach—real-time interactive visualization. (Woodward et al. 2007; Damon et al. 2008; Greensky et al. 2008; McLane et al. 2008, 2009)

Our visualization system called WebViz, builds upon existing work done in the area of large-scale visualization at the Laboratory for Computational Sciences and Engineering (LCSE) at the University of Minnesota. Over the past 15 years and under the direction of Professor Paul R. Woodward, LCSE staff has constructed a visualization system consisting of 13 rendering nodes which collectively drive LCSE's PowerWall, a rear-projection display system capable of producing an overall resolution of 15 million pixels. Elder et al. (2000) This system runs custom hierarchical volume rendering software described in Porter (2002) and Porter et al. (2002), and has been employed primarily for visualizing data from simulations in astrophysics and geophysical fluid dynamics. The work we have done on WebViz is similar to that described by Billen et. al. in (2008) regarding the CAVE project which aims to bring immersive VR visualization to a broader audience by developing software that can be run on a large variety of platforms. In our case, we utilize a web-based approach for expanding the user-base of LCSE's interactive visualization system with the WebViz web application.

By providing large-scale visualization services "in the cloud", scientists are able to leverage LCSE's visualization system from anywhere in the world. Sotomayor et al. (2009) Furthermore, being a web-based system built on Java technology, WebViz is almost completely platform-independent and is accessible from any JavaScript enabled web-browser including those on iPhones and a host of other mobile devices. The WebViz project fully embraces the "Web 2.0" spirit of user-centric software design and was developed to not only be a visualization tool, but also a system

for collaboration and information sharing between users. Ubiquitous and collaborative interactive visualization is the primary objective of the WebViz project. Greensky et al. (2008); McLane et al. (2008, 2009)

In the current version of our software, we have implemented a new, highly extensible back-end framework built with Google Web Toolkit (GWT), and HTTP "server push" technology to provide a rich collaborative environment and a smooth end-user experience. The current version of the system allows users to (1) launch multiple visualizations, (2) invite other users to view their visualization in real-time (multiple observers), (3) delegate control aspects of the visualization session to others (multiple controllers), and (4) engage in collaborative chat and instant messaging with other users. These features are all in addition to a full range of essential visualization functions including 3-D camera and object orientation/position manipulation, time-stepping control, and custom color/alpha mapping.

This paper will provide an explanation of how we went about implementing the WebViz system. In Sect. 37.2 we discuss the hardware that comprises the backbone of our system. Section 37.3 presents past work, previous versions, and the history of our project. Section 37.4 details the current version of our software and provides an explanation of the overall manner of operation and implementation. Finally, Sect. 37.5 lists future improvements and additional features we would like to implement in the system, addresses some known issues and limitations, and discusses our effort to make WebViz a true cloud-computing solution by utilizing the Google Application Engine cloud-computing framework.

37.2 Review of Hardware

The visualization system in the LCSE was originally developed as a system to drive the PowerWall display in 1994. This interactive display system consists of 12 projectors which deliver an overall resolution of 15 million pixels to a three panel semi-immersive rear projection wall Elder et al. (2000). This system is different in technical objectives from the much larger display system at the San Diego Supercomputer Center, which has the capability of displaying 285 million pixels. Over the past four years LCSE's system has been improved and developed under NSF grants with the intended purpose of providing researchers with an on-demand resource for interactive computation and simultaneous visualization. A diagram of LCSE's system architecture can be seen in Fig. 37.1.

We are currently in the process of developing the interactive computation side of this visualization system. Over this past year we purchased six IBM Cell processor Tri-Blades—the same blades which compose the RoadRunner supercomputer at the Los Alamos National Laboratory. These blades are comprised of two dual Cell processor blades connected to one dual dual-core AMD Opteron blade. The blades are interconnected with a DDR Infiniband fabric. We have also purchased 24 dual quad-core Intel Nehalem processor nodes. These nodes will each have two Nvidia Tesla GPUs and 12 of the nodes will have 24 TB of locally attached disks, with all

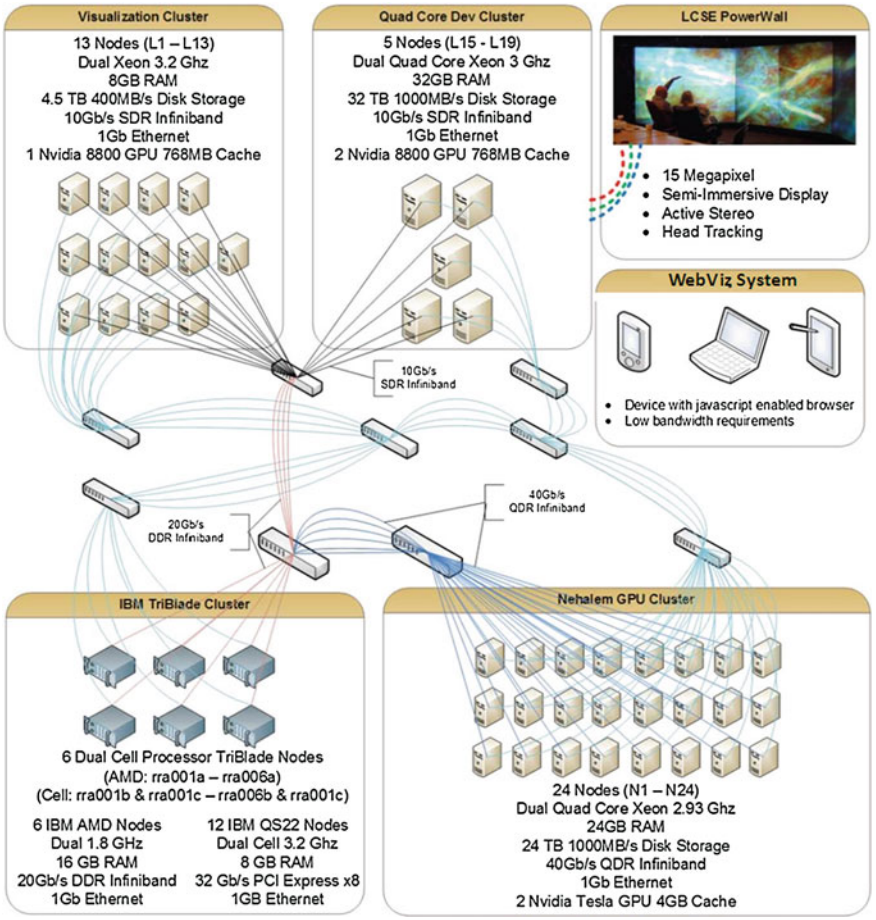


Fig. 37.1 LCSE system architecture. It should also be noted that there is an Infiniband interconnect to the local supercomputing system at the University of Minnesota which have a present capability of around 100 TFlops

of the nodes connected via a QDR Infiniband fabric. When in use, this system can compute a simulation using a combination of the Tesla GPUs, Intel/AMD CPUs, and Cell processors. While the simulation is running, the output can be visualized and analyzed using the PowerWall or the WebViz system. The unique capabilities of this system give researchers the ability to monitor the progress of their simulation, modify it if necessary, and quickly find the answers to their questions. When computing a problem of appropriate size (those that only take a few hours to compute) this system can be used interactively. We have demonstrated this capability at the several international Supercomputing conferences over the past three years.

For simulations of higher resolution we have connected the above described system to the multi-thousand core machines in the Minnesota Supercomputing

Institute (MSI). While a simulation is running on one of MSI's large machines the data can be streamed out of the compute nodes and across a 20 GB/s DDR Infiniband link to the systems and disks in the LCSE. Once the data has arrived, the machines in the LCSE can analyze and visualize it. This enables researchers to interact with a very high resolution computation. Visualizing the output of a simulation as it is running provides researchers with the ability to ensure that the simulation is proceeding as intended, and if needed, modify parameters or code to correct it. This saves CPU time as well as time needed to post-process and visualize data after the simulation has terminated. MSI implemented a reservation queue which lets a researcher reserve thousands of CPUs for a specific period of time. This queue allows for planning an interactive supercomputing session at a time that is convenient for the researches and their collaborators.

LCSE's visualization is system has also proven to be valuable for remote interactive supercomputing and visualization. Over the past three years, a web interface (the WebViz System) has been built to interface with LCSE's machines and enable remote use. Using WebViz, a researcher is able to visualize their data and monitor the progress of their simulations using any JavaScript enabled browser and is provided the full functionality of LCSE's large-scale visualization system from anywhere world via the Internet. This ability has been demonstrated at the Supercomputing 2008 conference in Austin, Texas, as well as other international conferences like in Elm, Switzerland, and is the focus of this paper.

37.3 WebViz Project History

While the current version of WebViz contains many new features and operates on a different architectural design than its predecessors, we find it important for understanding our objective to highlight the history and course of development of the system. A more thorough analysis of the previous versions of WebViz can be found in Greensky et al. (2008), McLane et al. (2008, 2009).

37.3.1 Initial Version of WebViz

The first version of WebViz was written as a standalone .NET application. This version was limited to use on Windows platforms. The application could take advantage of the rich .NET UI libraries which allowed for smooth image navigation and a highly interactive user interface. This initial prototype allowed us to test and more completely develop our interface to LCSE's rendering nodes, and to design an interface module allowing for affine transformations of images. We found however that this prototype suffered from lack of platform independence and a limited, pre-alpha feature set. Learning from this first experience, we prepared a set of objectives and goals for the WebViz application and developed a plan to implement a new version. Since

we wished to make WebViz platform-independent, collaborative, and available to the widest possible user-base, we decided to move into web technologies and started to develop our first web prototype.

37.3.2 Toward a Web-Version of WebViz

Since the LCSE PowerWall offers interactive visualization facilities to a limited number of users (i.e. those on the University of Minnesota Minneapolis campus), the WebViz initiative focuses on developing software to allow for a much wider user-base. To allow for ubiquitous access to large-scale visualization, the Web appeared to be a clear choice of medium. By designing our system to work as a web application we allow any authorized user with Internet access to use it. From the most high-level standpoint, the WebViz system is comprised of the four modules as shown in Fig. 37.2.

The first step in creating a web application for utilizing the LCSE render cluster was to create a gateway between whatever server-side code we would later implement and the actual render cluster. For this reason, Jim Greensky developed the LCSEWebServices servlet described in Greensky et al. (2008). This servlet works as an intermediary between our system and the actual render cluster and provides a level of abstraction over the raw protocol needed to communicate with the rendering software at LCSE. The servlet is able to accept an XML document known as the Key Frame Data (KFD) which specifies rendering parameters, parse the KFD appropriately, and then generate a properly formated byte stream to be sent to the render cluster to request an image. The service also has the ability to split up the requested image into distinct parts to be rendered in parallel by multiple rendering nodes (Greensky et al. 2008). The LCSEWebServices servlet is still in use in the current version of WebViz.

The components shown above leverage a combination of Java, Javascript, XML, SQL, and CSS to allow for a robust and secure system. The WebViz Server components are written in Java as a servlet and can be served by most servlet containers, in our case Apache Tomcat. Client-side code is written in javascript to provide cross-platform compatibility. On the back-end at LCSE, the render nodes run customized hierarchical volume rendering software discussed in Porter (2002) and Porter et al. (2002). A very high-level overview of the method of operation of WebViz can also be seen in Fig. 37.2. Essentially, the user interface provides a way for the user to generate input commands (by clicking buttons or setting variable values). These pieces of user input are sent to the server via an RPC request. There, the server-side WebViz code generates an XML request (by modifying the visualization session's nodes return an image fulfilling the server's request, and the server then sends that image back to the user interface for display to the user. This model, roughly, persists to the current version of WebViz.

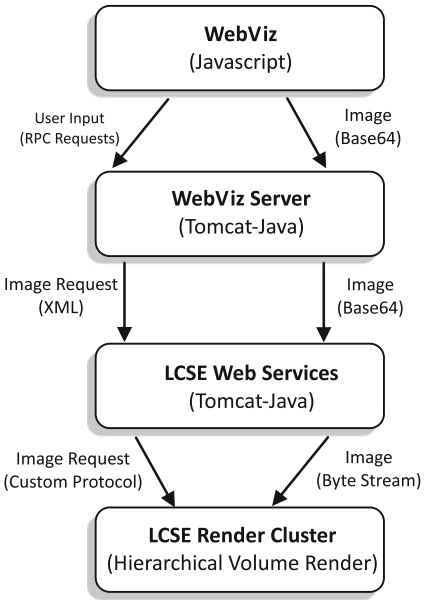


Fig. 37.2 A very high-level view of the four components comprising the WebViz system and the data-flow between them

37.3.3 WebViz Client Evolution

The client-side WebViz code has undergone many changes since the inception of the project. Since the objective of this project is to create a visualization system accessible from not only desktop computers but also mobile devices and netbooks, the size of client-side code has always been a major consideration. For this reason we decided to adopt a design pattern whereby the most intensive computational tasks would be carried out server-side leaving client-side code to deal with only simple tasks. With this idea in mind, we created two prototypes using two different web development technologies.

The first prototype of the web-based version of WebViz was built with the Echo Java framework. The web development process in Echo looks much like desktop application development using Swing or AWT, and is therefore well-suited for building prototypes as the implementation can progress quickly (McLane et al. 2009). The main advantage of Echo was that it allowed us to write a thin-client web application that took advantage of its rich set of GUI components—a vital feature in the creation of modern-looking, interactive web applications.

A second prototype was then developed using GWT. Development with GWT involves writing all client-side code to handle user interaction in Java, and then running it through a specially designed compiler to output javascript which can be embedded in a standard HTML web page. We chose GWT because it allowed for programming in a familiar and object-oriented programming language (McLane et al.

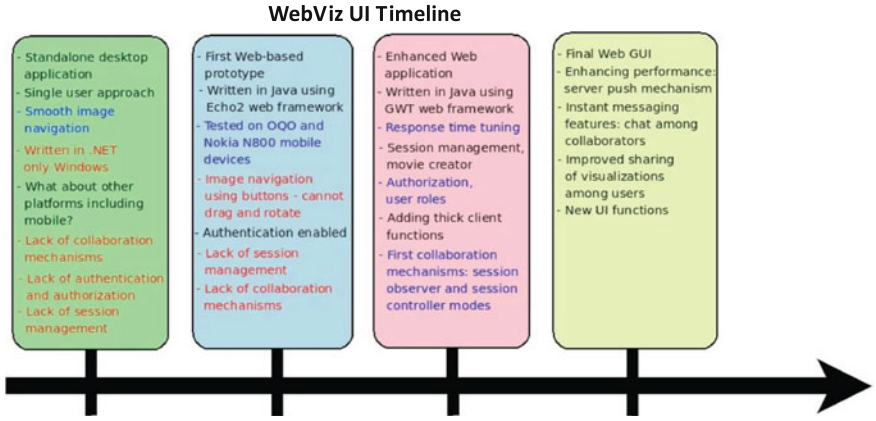


Fig. 37.3 A timeline of the history of the WebViz client. Four distinct incarnations can be seen

(2009)). An additional benefit afforded us by GWT is that the compiler can generate different versions of the web application optimized for various browsers.

Due to the special compiler in GWT development, only a subset of all Java classes are available for use but as GWT matures there are more and more classes being supported. In order to work around certain unsupported Java libraries and create the UI for the WebViz client, the developers created their own library of widgets that are both extensible and all open source. Some of these custom widgets are still in use in the current version. The first version of the GWT client exhibited a nearly identical look and feel as the Echo version which was created to work well on both regular computers and mobile devices. Upon comparing the features and limitations of these two web development technologies, the development team selected GWT as the better of the two options. The current version of WebViz is also built with GWT (Fig. 37.3).

37.3.4 WebViz Server Evolution

The first version of the server-side WebViz code was a single module HTTP servlet (see Fig. 37.4) running inside a Tomcat container. Both the Echo and GWT prototypes worked on a very simple request-response model (McLane et al. 2008, 2009). They allowed the user to adjust various visualization options such as viewing angle and the position/orientation of the data being viewed. Based on the user's input, the WebViz server would appropriately modify the KFD and send it via HTTP to the LCSE Web Services servlet where it was parsed and the information passed along to the actual rendering nodes. Upon completion of the rendering, the service encoded the image as a Base64 string which was embedded in an XML document that was sent back to the WebViz server. Once the server had the image, the client would request it and it

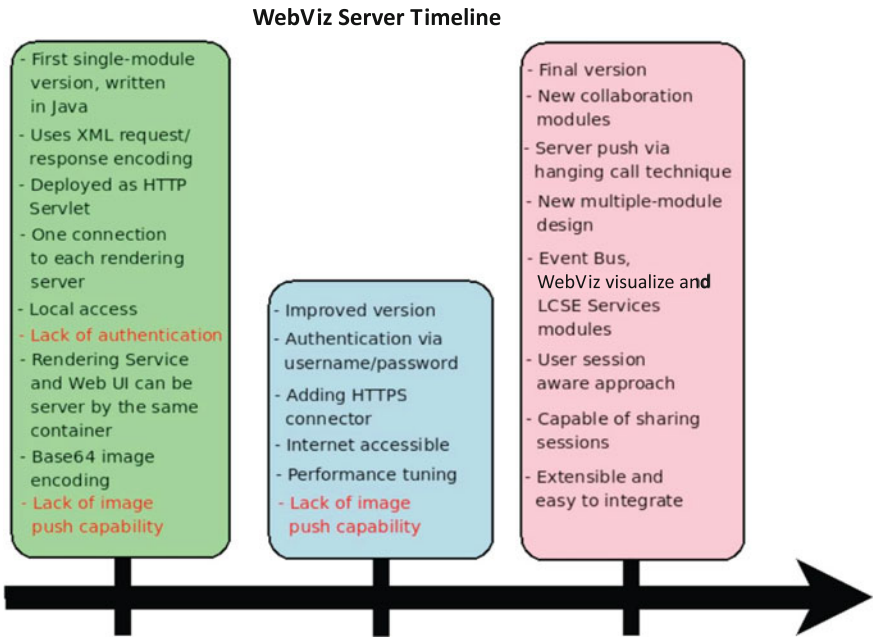


Fig. 37.4 A timeline of the history of the WebViz server

was sent to be displayed to the user. This process would repeat every time the user generated input during a visualization session.

The main drawback of this single-module approach was its inability to handle client-server communication beyond simple request-response exchanges for retrieving images. To make a more fully featured visualization application, it became clear that the client-server architecture would have to change. Therefore we decided to redesign WebViz server components using a more flexible and extensible multiple-module design. A detailed description of current architecture is provided in Section.

37.4 Current Version of WebViz

This section will detail the current version of WebViz. Section 37.4.1 describes the web application and its features, and Sect. 37.4.2 explains the overall system architecture and method of implementation.

37.4.1 Features and Product Tour

The primary function of WebViz is to allow users to launch interactive visualization sessions from any JavaScript-enabled web browser. The system was designed with the objectives of so-called "web 2.0" software development in mind. For this reason, features in the GUI are all geared toward creating a smooth end-user experience where control elements are intuitive, easy to find, and straight forward to use.

37.4.1.1 Visualization Options and Features

When a user selects "New Visualization" from the main menu, a pop-up window is generated that displays a list of directories containing data files which are available for visualization. Also in this pop-up window is a menu to select the color and alpha map that the user wishes to have applied to the visualization. Clicking OK in this dialog causes a new visualization tab to appear in the main area of the GUI. Figure 37.5 shows a screen shot of the WebViz user interface after a visualization session has been launched.

With a visualization tab open, the user has access to the visualization toolbar across the top area of the GUI (identified as area 1 in Fig. 37.5). This area is separated

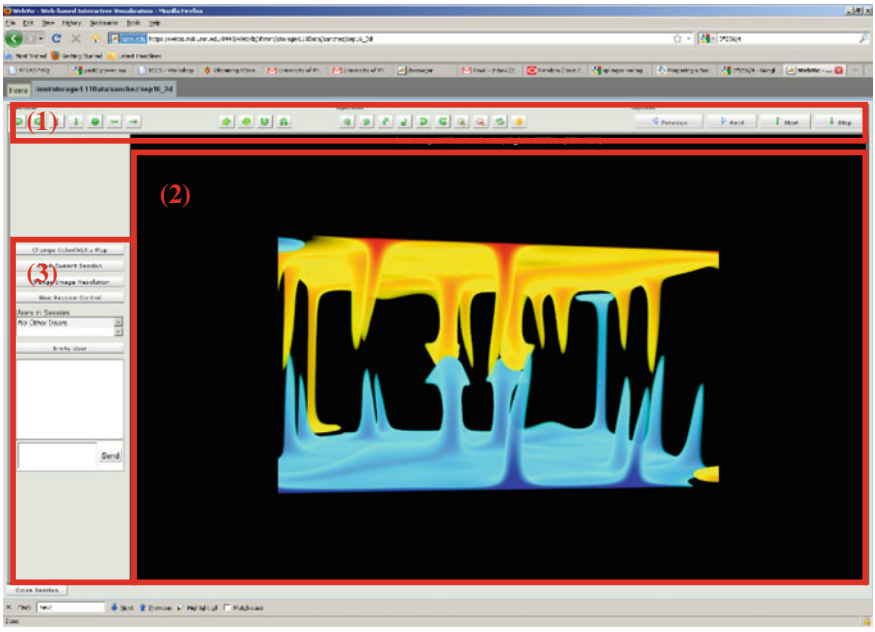


Fig. 37.5 WebViz user interface. Three areas are highlighted: *1* visualization toolbar, *2* image display area, *3* collaborative chat area

into three sub-areas representing three classes of actions that can be carried out by the user. The left-most area contains buttons for camera control. These buttons provide the user the ability to move the camera or "eye" in every direction in 3-space. This includes: rotation, pitch and yaw, and distance from object (forward/backward motion). The middle section of buttons is for controlling the position/orientation of the object or data being rendered. These buttons represent commands for rotation either left/right or up/down, and zooming in and out. Finally, the right-most set of buttons controls time-stepping. WebViz has the ability to successively render and display files contained in the selected directory in alphabetical/numerical order. The first two buttons (labeled previous and next) cause WebViz to display the previous or next time-step respectively. When clicked, the "start" button causes a dialog to pop up in which the user can initiate an auto-step function. With this, the user can specify the frequency of stepping (frame rate) they desire, as well as how many time-steps to skip between each refresh. After clicking OK, the image display area of the GUI (area 2 in Fig. 37.5) is automatically updated at the specified rate with successive time-steps found in the directory.

To access color/alpha mapping settings, the user selects the "Color/Alpha Maps" option from the main menu bar. This causes a color/alpha map editor to appear as in Fig. 37.6. From this screen, the user can edit the color/alpha map being applied to the current visualization, as well as save the map to their account for later application to other visualizations.

As same as many other web-based applications, WebViz can be accessed from any location on earth with Internet connection. Figure 37.7 shows how it works through Wi-Fi at JinMao Tower, Shanghai, which is approximate 6745 miles (10855 km) from University of Minnesota, Minneapolis, where is the LCSE located. All the examples are calculated on the Fermi C-2070 with 6 GBytes of global memory.

37.4.1.2 Collaborative Features

The features of WebViz that allow for collaborative visualization sessions set this system apart from other software in the field. With WebViz we are able to connect together many users in a single visualization session. In a collaborative session, all users are able to view the data being visualized and as changes occur, an updated image is displayed in the GUI of all connected users in real time.

With a visualization session open in the GUI, a user (the "share-er") can invite others to join by selecting "Share This Session" from the main menu bar. This action triggers a pop-up window to appear that displays a list of users that the session can be shared with. After selecting one or more users, a second pop-up window appears that allows the share-er to specify the level of control they wish to delegate to these "share-ees." The share-er can individually allow or deny access to camera control, object control, time-stepping, and/or color-mapping on a per-user basis. The share-er can also specify whether or not they wish to give those receiving the invitation the ability to share the session with yet other users. Once these forms are completed and the invitation is sent, a pop-up window appears in the GUI of the users that the

Fig. 37.6 Color/alpha map editor

share-er has elected to share with. The message alerts each share-ee that another user wishes for them to join their visualization session. If the share-ee accepts the invitation, a new tab appears in their GUI that displays the visualization. If the user does not accept the invitation immediately, they can join the session at a later time by selecting the "View Invitations" option from the main menu bar (Fig. 37.8).

The left side of the visualization GUI (area 3 in Fig. 37.5) contains the collaboration area. Here, a user is able to view a list of other users currently in the session. Below this display is a chat area where users can type messages to one another. Messages are delivered to the chat areas of all users currently in the session in real time. In addition to chat messages from users, the chat area is used by the WebViz server to deliver notification messages. For example, all users in a shared session are notified by the server when a new user joins, or a current user disconnects (Fig. 37.9).

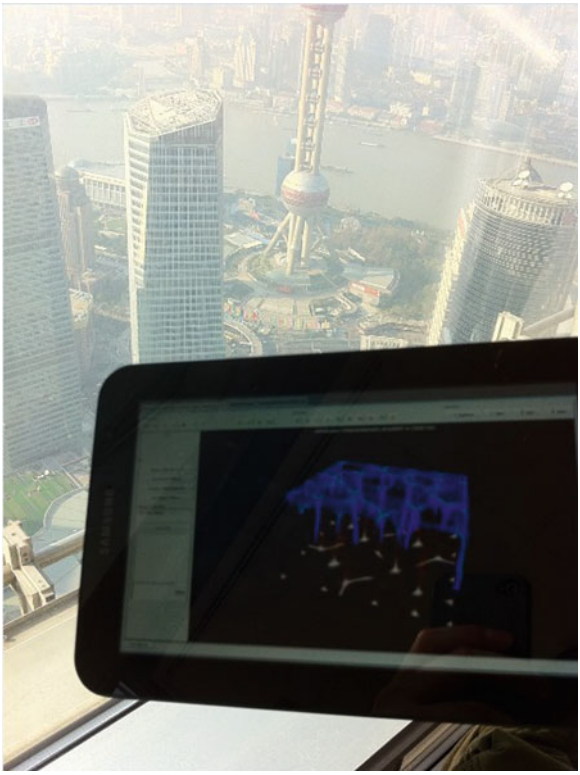


Fig. 37.7 A remote application of the visualization system in Shanghai (JinMao Tower). All visualization results are carried out on the Fermi C-2070 with 6 GB of global memory

37.4.2 Technical Overview

As mentioned in Sect. 37.3.3, the current version of WebViz is built on GWT technology. To accomplish our goals for the current version, the first obstacle we had to deal with was the highly dynamic nature of the application. For example, in the case of visualization functions, user input must be immediately processed server side, and an image rendered and returned to the user. Also, information such as chat messages must be simultaneously delivered to potentially many users in a timely fashion. For this reason, it is clear that an open channel of communication must exist at all times between the client-side user interface and the WebViz server. Complicating matters is the fact that WebViz is a web application which operates over HTTP, a protocol where only the client in a client-server pair can initiate communication.

To side-step this feature of the HTTP protocol, WebViz utilizes the so-called "HTTP server push" paradigm. The objective of server push over HTTP is to create the illusion of the server being able to initiate communication with the client and "push" information to it. This paradigm is well known in the area of web-development

Fig. 37.8 Collaborative visualization session on Samsung Galaxy and iPhone

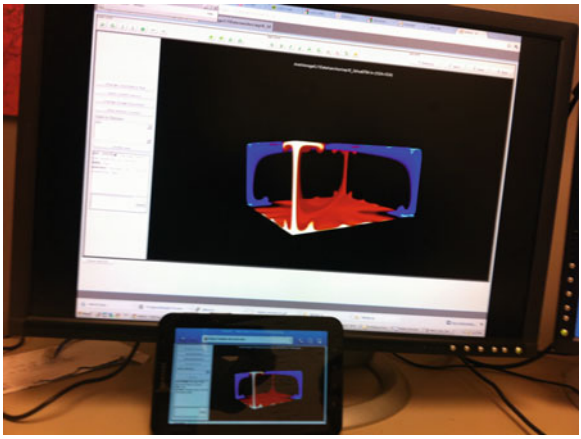


Fig. 37.9 Multiple users engaged in collaborative visualization session (PC and iPhone)

and is widely used in web applications to provide real-time client-server communication.

When implementing WebViz, much care was given to choosing a highly extensible architecture so that adding features to the system later on would be simple and straight-forward for the development team. As such, we decided to diverge from the widely used "model-view controller" (MVC) approach to software design, and instead adopted a highly "decoupled", modular design. The main departure from the MVC approach is in that GUI elements contain pertinent control functions for the type of information they handle. We also fully embrace the asynchronous nature of the GWT and allow many objects and GUI elements in the client to initiate communication with the server.

At the heart of the client-side architecture is the EventBus. At instantiation, all GUI elements in the WebViz client are "registered" with the EventBus through a design pattern known as dependency injection. The EventBus is primarily responsible for receiving updates pushed down from the server and passing them to the handleUpdate() method of the appropriate GUI element. The receiving GUI element is then responsible for appropriately processing and/or displaying the contained data.

All communication between client and server is conducted by passing EventRequest and EventResponse objects between client and server, and server and client respectively. To make these concepts concrete, let us consider the operations that occur when a user engaged in a visualization session inputs a command to alter camera angle. This process is depicted graphically in Fig. 37.10.

The process shown in Fig. 37.10 proceeds as follows:

1. The user clicks the "rotate camera right" button in the visualization toolbar.
2. An EventRequest object is generated and set to EventType IMAGE_MANIPULATION_INPUT. The payload of this object is a string representing the requested action (in this case ROTATE_CAMERA_RIGHT). Additional information is also included in the EventRequest including the unique identifier (UID) of the user's visualization session so as to inform the server which session this message pertains to. This EventRequest object is then passed to the server as the payload of an RPC request.
3. The UID included in the EventRequest allows the server to retrieve the appropriate visualization session from memory. Then, the EventType IMAGE_MANIPULATION_INPUT triggers the server to retrieve the KFD from that user's visualization session and to modify it appropriately (in this case to alter camera orientation values).

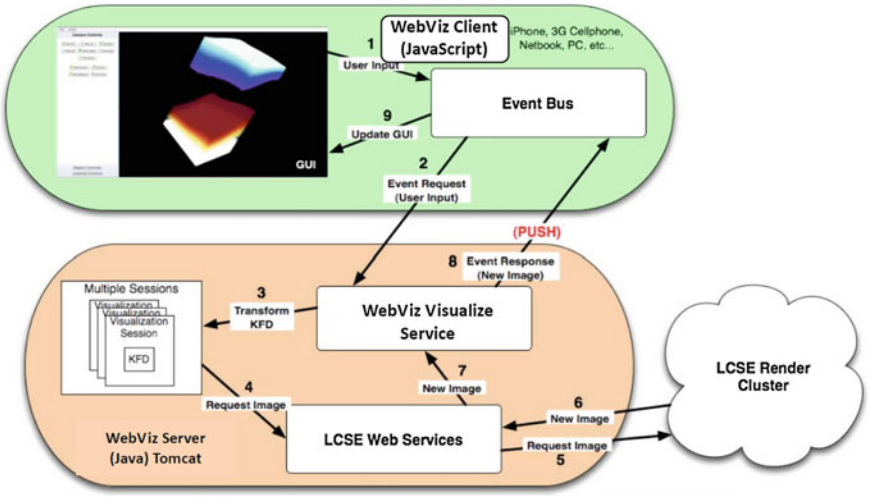


Fig. 37.10 A flow chart of the process for handling user input and updating imagery

4. The server then requests an updated image from the render cluster through the getImage() method of the LCSEWebServices servlet.
5. The LCSEWebServices servlet parses the KFD and requests an image from the render cluster.
6. Once the render cluster has generated an updated image based on the values in the KFD, the image is returned to the LCSEWebServices servlet.
7. The getImage() method called in step 4 then returns with the updated image.
8. The WebViz server now creates a new EventResponse object that is set to EventType IMAGE_UPDATE. The server then retrieves a list of all users in the visualization session that the EventRequest from step 2 pertained to and pushes a copy of the EventResponse to all connected users via HTTP push.
9. Now client-side, the EventResponse is examined by the EventBus and routed to the handleUpdate() method of the appropriate visualization tab. Noting that the EventResponse is of type IMAGE_UPDATE, the visualization tab forwards the payload (the new image) to the image display area of the GUI where it is subsequently displayed to the user.

37.4.3 Comparison Between Current and Previous Versions

The most notable differences between the previous and current versions of WebViz are the tools for collaborative use of the system. While the old system contained tools to share a session with others, it did not give share-ees the ability to manipulate visualization controls. The chat system and list of users in the session was also not present in the older version.

In terms of implementation, the new code architecture and the server push capability is a new move that we hope will continue to benefit the system as we move into subsequent versions. The advantage of an implementation such as this lies in its extensibility. As new features are implemented, any sort of client-server communication can be accomplished in terms of these EventRequest-EventResponse objects by simply creating a new EventType and creating the appropriate methods, both server- and client-side, to handle them.

37.5 Future Research

In this section, we detail new features we wish to implement in the WebViz system, as well as expose some of the current system's known issues and limitations. We also discuss our objective of deploying WebViz to the Google Application Engine cloud-computing system.

37.5.1 New Features for Future Versions

37.5.1.1 Mobile Version

While WebViz is accessible from any JavaScript-enabled web browser, including those on mobile devices, we find the GUI difficult to navigate on small-format screens (such as iPhones and mobile devices). For this reason, we intend on implementing a mobile version of WebViz called WebViz-Mobile. Once this system is in place, if a user connects from a mobile device to the main site, our web-server will automatically re-direct them to the mobile version. In WebViz-mobile, the user will have the ability to connect to visualization sessions which have been shared with them. Mobile users will exist solely as "observers" in these sessions whereby they can view the visualization in progress as well as chat messages amongst users, but will not have the ability to launch new visualizations or exercise control over the one(s) they are engaged in. Also, consider mobile devices comparative low processing ability, WebViz-mobile will lower the resolution of visualization result to improve performance. While this system does impose limitations upon functionality for mobile users, we feel it will give the highest degree of usability. There will however be an option provided mobile users to access the full-featured WebViz application should they wish.

37.5.1.2 Direct HPC Integration

In the current version of WebViz, users are only able to visualize files that are already contained within the file system of the render cluster. However, because of the highly extensible architecture of the system, it is possible to tie WebViz directly into the output stream of simulations running on nodes in the LCSE network. We also envision a customizable interface for users running a simulation to alter the parameters their simulation is using. This, coupled with the ability to view output in real time will provide users true real-time interactive visualization over the Internet.

37.5.1.3 Increased Control of Shared Sessions

A major limitation placed upon users the share-er of a session is a lack of control over their "shares." For example, once a visualization session is shared with another user, the share-er has no way of altering permissions delegated to the share-ee. For this reason, an area is needed that will display who a user has shared their session with and what degree of control these other users have over the visualization. In this area there would be a way for the share-er to alter permissions granted to other users, as well as the ability to "kick" users out of the session.

37.5.2 Known Issues and Limitations

37.5.2.1 Low Frame Rates

In the current version of the system, there is a significant delay (approximately 350 ms) between when an image is requested by the WebViz client and the image is delivered to the user.

The bottleneck has been identified as being between the LCSEWebServices servlet and the LCSE render cluster. For this reason, the development team has decided that the functionality of the LCSEWebServices servlet should be integrated directly into the WebViz code base as opposed to having it implemented as a completely separate module. By directly integrating it into WebViz, there will be a degree of latency removed by eliminating the need for the WebViz and LCSEWebServices servlets to communicate with one another. Additionally, we believe there to be some revisions to the code contained in the LCSEWebServices servlet which could be easily carried out which would increase performance metrics all around.

37.5.2.2 Scalability

The current version of WebViz runs on a single server. To increase the reliability of the system and to allow for many users, the system needs to be made scalable. The largest impediment to this is the highly stateful nature of the server-side components. For example, the server keeps track of the current state of all in-progress visualization sessions in memory. To increase scalability, the server needs to be made stateless, mainly by offloading the information stored in memory to some sort of database structure. Doing this will also prepare the system for migration to the Google Application Engine as will be discussed in Sect. 37.5.3.

37.5.2.3 Distributability

There are many aspects of the implementation in the current version that are highly customized to work with our systems. To make this piece of software into a package that could be distributed to other institutions and universities, there would have to be some refactoring performed that would “generalize” the system to allow it to work on other institution’s networks and with other rendering systems.

37.5.3 Migration to Google Application Engine

First made public in April 2008, the Google Application Engine (GAE) is a hosted web-server platform that allows developers to deploy web applications in either Python or Java in Google-managed data centers. When an application is deployed

to the GAE, it is automatically distributed amongst multiple servers in multiple data centers. As with other managed deployment solutions, we would expect to see an increase in reliability by migrating server-side components to the GAE. However, the potential benefits would not end there. The key benefit of deploying WebViz to the GAE is that the application becomes virtualized and is deployed across multiple servers in Google's world-wide infrastructure. Because of this, the server-side components of WebViz would be located in data centers geographically local to all world-wide users. This is in opposition to the current system where all resources are located at the University of Minnesota. To be sure, there would still be communication necessary between the WebViz server components and the render cluster at LCSE, however this communication would take place through Google's high-speed network backbone and only traverse the public internet on the hop from Google's network to the University of Minnesota's.

Because of the distributed nature of the GAE and the fact that a single application, once deployed, is hosted from multiple servers, special attention must be given to making server-side code compatible with this architecture. Most importantly for developing in an environment such as this is handling the fact that two requests from one user may very well be delivered to two completely different servers. For this reason, all server side components must be completely stateless. That is, the server components must treat each request as an independent transaction that does not depend on any previous communications that occurred between it and the client, except for any data storable in a database. As discussed in the previous sub-section, a great deal of information about user sessions is stored in memory on our server. To make our system stateless, we would need to leverage Google's BigTable database tool, which is offered as the datastore for use with GAE. In addition to providing a low-level API for direct interaction with the BigTable platform, JDO and JPA are supported (for the Java side) as a scheme that can run on top of the BigTable to allow for portability and ease of use. Transitioning our current system into this setup would be very simple and would turn WebViz into a true cloud-computing system that better reflects current practices for designing web applications.

Acknowledgments We would like to thank Professor Paul Woodward and Jim Greensky for their contributions to this project. This research has been supported by VLAB sponsored by the National Science Foundation and the REU program of National Science Foundation.

References

Billen MI, Kreylos O, Hamann B, Jadamec M, Kellogg LH, Staadt O, Sumner DY (2008) A geoscience perspective on immersive 3D visualization. Comput Geosci 49(9):1056–1072. doi:10.1016/j.cageo.2007.11.009

Damon MR, Kameyama MC, Knox MR, Porter DH, Yuen DA, Sevre E (2008) Interactive visualization of 3D mantle convection. Vis Geosci 49–57: doi:10.1007/s10069-007-0008-1

Dunning T, Schulten K et al (2009) Science and engineering in the petascale era. Comput Sci Eng 11(5):28–37

Elder A, Ruwart T, Allen B, Bartow A, Anderson S, Porter D (2000) The intensity powerwall: a case study for a shared file system testing framework. In: Proceedings of the 17th IEEE symposium on mass storage systems/8th NASA Goddard conference on mass storage systems and technologies

Greensky JBSG, Czech WW, Yuen DA, Knox M, Damon MR, Chen SS, Kameyama MC (2008) Ubiquitous interactive visualization of 3-D mantle convection using a web portal with Java and AJAX framework. Vis Geosci 105–115: doi:10.1007/s10069-008-0013-z

McLane J, Czech WW, Yuen DA, Knox MR, Greensky J, Kameyama MC, Wheeler V, Panday R, Senshu H (2008) Ubiquitous interactive visualization of 3-D mantle convection through web applications using Java. In: Proceedings of the international symposium on visual computing 2008 (Lecture notes in computer science), 2009 (in press)

McLane J, Czech WW, Yuen DA, Knox M, Wang SM (2009) Ubiquitous interactive visualization of large-scale simulations in geosciences over a Java-based web-portal. Concurrency and Computation: Practice and Experiences (in press)

Porter DH (2002) Volume visualization of high-resolution data using PC clusters. http://www.lcse.umn.edu/hvr/pc_vol/rend_Lpdf/

Porter DH, Woodward PR, Iyer A (2002) Initial experiences with grid-based volume visualization of fluid low simulations on PC clusters. http://www.lcse.umn.edu/dhp1/articles.html

Sotomayor B, Montero R, Llorente I, Foster I (2009) Virtual infrastructure management in private and hybrid clouds. IEEE Internet Comput 13(5):14–22

Woodward PR, Porter DH, Greensky J, Larson AJ, Knox M, Hanson J, Ravindran N, Fuchs T (2007) Interactive volume visualization of fluid flow simulation data. In: Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing. Lecture otes in computer science, vol 4699. Springer, Heidelberg, pp 659–664. http://www.lcse.umn.edu/para06

Chapter 38
Interactive Visualization Tool for Planning Cancer Treatment

R. Wcisło, W. Dzwinel, P. Gosztyla, D. A. Yuen and W. Czech

Abstract We discuss the components and main requirements of the interactive visualization and simulation system intended for better understanding the dynamics of solid tumor proliferation. The heterogeneous Complex Automata, discrete-continuum model is used as the simulation engine. It combines Cellular Automata paradigm, particle dynamics and continuum approaches to model mechanical interactions of tumor with the rest of tissue. We show that to provide interactivity, the system has to be efficiently implemented on workstations with multiple cores CPUs controlled by OpenMP interface and/or empowered by GPGPU accelerators. Currently, the computational power of modern CPU and GPU processors enable to simulate the tumors of a few millimeters in diameter in its both avascular and angiogenic phases. To validate the results of simulation with real tumors, we plan to integrate the tumor modeling simulator with the *Graph Investigator* tool. Then one can validate the simulation results on the base of topological similarity between the tumor vascular networks obtained from its direct observation and simulation. The interactive visualization system can have both educational and research aspects. It can be used as a tool for clinicians and oncologists for educational purposes and, in the nearest future, in medical in silico labs doing research in anticancer drug design and/or in planning cancer treatment.

R. Wcisło · W. Dzwinel · P. Gosztyla · W. Czech
Institute of Computer Science, AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Krakow, Poland
e-mail: wcislo@agh.edu.pl

W. Dzwinel (✉)
Institute of Teleinformatics, Cracow University of Technology, Podchorazych 1,
30-084 Kraków, Poland
e-mail: dzwinel@agh.edu.pl

D. A. Yuen
Minnesota Supercomputing Institute, University of Minnesota, Minneapolis,
MN 55455-0219, USA
e-mail: daveyuen@gmail.com

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 607
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_38,

Keywords Tumor growth · Interactive visualization · Complex automata model · Parallel computations · Complex networks · CPU vs. GPGPU

38.1 Introduction

38.1.1 Motivation

It is widely believed that cancer is responsible for about 20 % of deaths in developed countries (Jemal et al. 2010). Therefore, it is assumed to be one out of major killers in the developed world. Despite the enormous financial effort that has been devoted to research on cancer phenomenon, the most important aspects remain obscure for clinicians and experimentalists. This is the reason that many of the currently used therapeutic strategies become not entirely effective.

A better understanding of tumor formation and its proliferation can be expected from a computer (i.e., quantitative) modeling. Computer models can ultimately improve the overall clinical outcome by predicting the results of specific types of medical treatment administered at specific regions of interest and time checkpoints. During the last decade a great variety of tumor models covering various morphological and functional aspects of tumor growth has been developed. These advances have been recently reviewed (Lowengrub et al. 2010) with a focus on the classification of mathematical tools and computational algorithms.

As shown in Chaplain (2000), in silico experiments can play the role of angiogenesis assays. In Stéphanou et al. (2005), the Authors describe a computational tumor modeling framework to compare dosing schedules based on simulated therapeutic response. However, till now, despite the existence of scores of various computational models of cancer growth dynamics, a ready-to-use interactive tool for clinical application remains a dream of the future. This situation is unjustified knowing that both technologically and methodologically there is not any obstacles to create, at least, a toy interactive visualization system which might help in education of young oncologists and elucidate the fundamental mechanisms of cancer progression. The system can evolve further to a serious tools aimed to improve the therapeutic techniques currently used, to stimulate the development of new strategies and speed-up the anticancer drug design process.

In this paper we discuss the main components of a novel interactive visualization tool for simulation of cancer growth dynamics. We present preliminary results we have obtained towards development of a framework of such the system. The paper is focused on the computational model, visualization interface and implementation issues, which are the main components of such the interactive visualization system.

We present a framework of parallel 3-D model of tumor growth, which bases on Complex Automata paradigm, which combines particle dynamics with Cellular Automata and continuum models. To speed up calculations and to make on-line interactions possible we show the ways to optimize the code for multi-core CPUs and

GPGPU rather than for massively parallel systems consisting of many heterogeneous computational nodes.

38.1.2 Modeling Domain

A serious obstacle that must be overcome to simulate cancer dynamics is the intrinsic multiple scale nature of tumor growth. It involves processes occurring over a variety of time and length scales: from the tissue scale—e.g. vascular remodeling—to intracellular processes—e.g., progression through the cell-cycle.

The structure of a solid and vascularized tumor is very dynamic and heterogeneous. Its development occurs in five phases: oncogenesis (carcinogenesis), avascular growth, angiogenesis, vascular growth and metastasis. Oncogenesis is a biological process occurring on molecular level and characterized by serious disruptions in DNA reparation system mainly due to numerous mutations. Consequently, the escape of mutated cell from apoptosis is observed. The avascular stage is the earliest stage the tumor develops in the absence of blood supply. The tumor grows up to a maximum size which is limited by the amount of diffusing nutrients and oxygen reaching the tumor surface. In the third stage, hypoxic cells of this avascular tumor mass produce and release substances called tumor angiogenic factors (TAFs). They diffuse throughout the surrounding tissue, and, hitting vasculature, trigger a cascade of events which eventually lead to vascularisation of the tumor. These phases of tumor growth are depicted in Fig. 38.1. In vascular stage, tumor has access to virtually unlimited resources, so it can proliferate beyond any limits. Moreover, in this growth phase tumor acquires a means of transport for cells that penetrate into the vasculature and form metastases in any part of the host organism.

The interactive system we propose comprises only two scales of tumor growth: avascular, angiogenesis and partly vascular phases (see Fig. 38.1). The size of tumor simulated on-line, depends on the available computational power, 2-D or 3-D version of the model and its level of details. Approximately, taking into account the power of nowadays workstations, the maximum size of tumor which can be modeled using our approach does not exceed the diameter of 1 cm. This scale of interest is extremely important. Thus, once the tumor becomes vascularized, therapeutic prognoses worsen dramatically. On the other hand, in that scale, the system can be used in investigations of the role the various tumor angiogenic factors play and for defining the targets of antiangiogenic therapies.

38.2 Computational Models of Solid Tumor Growth

The computational paradigm plays a crucial role in development of in silico implementation of cancer model. In the length scales exceeding centimeters, continuum methods are acceptable for modeling tumor dynamics. Continuum model parameters

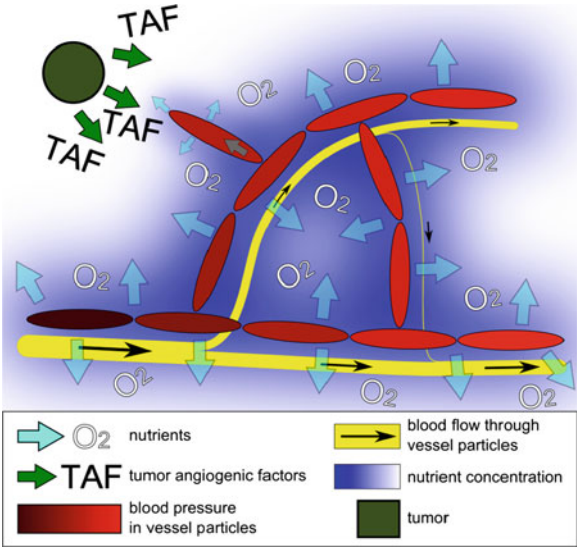


Fig. 38.1 Angiogenic phase of tumor growth

can be relatively easy to obtain, analyze and control. They may also be accessible through laboratory experimentation. However, despite they can provide significant insight into the relative role that different process play in the tumor formation they fail completely in predicting tumor microstructure. This may be important deficiency when studying the effect of genetic, cellular and microenvironment characteristics on overall tumor behavior. Within continuum models, it is not possible to capture such the important events as repeated sprout branching and the overall dendritic structure of the vascular network, compartmentalization of the tumor tissue and remodeling processes due to mechanical interactions (Lowengrub et al. 2010).

Whereas continuum models describe cell populations by means of continuous fields, discrete models deal with the dynamics of agents. The agents usually represent individual cells which are tracked and updated according to a specific set of biophysical rules taken from a discrete and finite space of states and evolve in discrete space and time. Unlike in the continuum models, discrete models can follow individual cells and can reveal more details about cell dynamics and its interaction with the tissue. There are, for example, percolation based models (Szczerba et al. 2008), Eden models (Alarcon et al. 2005; Lee et al. 2006), random walk and diffusion limited aggregation (DLA) models (Amyot et al. 2006), cell based models (Bauer et al. 2007), lattice gas models and cellular automata (Dormann and Deutsch 2002; Moreira and Deutsch 2002).

Discrete approach is particularly useful for studying spatial scales of tumor dynamics below 1 cm such as carcinogenesis, natural selection, genetic instability, interactions of individual cells with each other and the microenvironment. Analyses of cell population dynamics have also been employed in order to study biological

characteristics applying to all cells in a particular population, such as response to therapy and in studies of immunology.

According to Lowengrub et al. (2010), there are two main types of discrete models, i.e., lattice based and lattice-free. The former describes the dynamics of discrete tumor cells as automata on a grid whose states are governed by a set of deterministic or probabilistic rules. One of the most popular lattice based paradigm used for modeling tumor growth are cellular automata (CA) (see the critical overview (Dormann and Deutsch 2002)). Cellular automata deal with the dynamics of discrete elements populating the nodes of structural grid. The elements take their state from a discrete—finite or infinite—space of states and evolve in discrete space and time. The dynamics of the elements is defined in terms of local, either deterministic or probabilistic, rules. To describe four aspects of tumor growth, namely, avascular growth, vascular growth, invasion, and angiogenesis, the classical CA models should be supplied with additional graph structures defined on the top of a regular grid (Topa et al. 2006). The mechanical interactions between the tumor and healthy tissue are only partly addressed in Mansury et al. (2002).

The lattice-free approaches describe the motion of discrete cells in arbitrary locations and their interactions. It is possible to translate detailed biological processes (e.g. cell life-cycle) into rules influencing both the cell motion and their mutual interactions. However, the computational cost increases faster than linearly with the number of modeled cells, limiting these methods to the spatial and temporal scales defined by the achievable computational power. Moreover, while these models are capable of describing biophysical processes in significant detail, it may be nontrivial to obtain reliable measurements of model parameters through experiments e.g., parameters of cell interaction rules.

An important research domain involves the development of hybrid continuum–discrete models for tumor growth. These models have the potential to combine the best properties of both continuum and discrete approaches. They may provide more realistic coupling of biophysical processes across a wide range of length and time scales. Discrete models are usually hybridized with continuum approaches in the sense that both substrate concentrations and blood flow are computed using continuum approaches (i.e. oxygen, glucose, matrix-degrading enzymes concentration by solving reaction-diffusion equation and blood flow by integrating hydrodynamic equations) while the cell-based components such as migrating EC tip cells are discrete.

The oldest hybrid model, simulating two-dimensional spatial distribution of sprouts, was presented by Stokes and Lauffenburger (1991). It uses a classic Folkman formulation (Folkman 1971; Folkman and Hochberg 1973). The tumor is located at the top center of a box of finite volume and the capillary at the bottom. The evolution of molecular species is governed by discretized reaction-diffusion equations. A stochastic differential equation simulates the migration of endothelial tip cells employing a particle dynamics in TAF concentration field. This approach is the basis of many other hybrid models (e.g., Chaplain 2000; Godde and Kurz 2001; Preziozi 2003; Alarcon et al. 2005; Amyot et al. 2006; Milde et al. 2008; Dormann and Deutsch 2002; Moreira and Deutsch 2002; Topa 2008) which differ in:

- geometrical properties of the simulation such as: dimensionality (2-D, 3-D), type of discretization of space and time (on-grid, gridless), structure of vascular network (rigid, structured, unstructured) etc.
- modeling accuracy—defined e.g. by the number of angiogenic factors and other subprocesses included in the model,
- methodology of simulation of the process of vascularization and tumor growth (stochastic, deterministic, cellular automata, lattice-gas, DLA etc.),

The accuracy of the computational model depends on the proper choice of processes and multiple scales crucial for tumor growth. Because of the complexity of cancer evolution, many models to date focus on single key sub-processes, disregarding their interactions with others. Many attempts assume either a static tumor and concentrate on dynamic vascularization in the absence of tumor growth or a static network topology. Some of them use dynamic network describing hydrodynamics of blood flow while neglecting its interaction with concentration fields and tissue components (Godde and Kurz 2001). There are also many models which describe only avascular phase of tumor dynamics (Dormann and Deutsch 2002). The usefulness of these models is strongly constrained. Truly important are the models, which track coupled tumor growth and tumor-induced neovascularization using a discrete approach for both.

These hybridized models are the basis for the development of the multiple-scale approaches which—apart from the tissue scale evolution—include processes from cellular and molecular scales. They represent the most advanced simulation methodologies. Multiple scale models incorporate e.g. cellular heterogeneity, intercellular phenomena, more complex mechanical laws to describe the response of the tissue to external forces, blood hydrodynamics and vessel remodeling. Advanced multi-scale models of tumor progression are presented, e.g., in papers by Bellomo et al. (2003) and by Alarcon et al. (2005). Rigid geometric constrains which disable realistic visualization and are the sources of many serious artifacts belongs to the principal weaknesses of these models (Wcislo et al. 2009).

For challenging high computational demand of discrete approaches in modeling spatial scales exceeding 1 cm, coupled continuum–discrete descriptions of tumor cells are realized. It is the second type of hybridized models in which the tumor itself is described using both continuum and discrete components to reduce computational complexity of discrete approach. This hybrid modeling is very important as it affords the possibility of seamlessly up-scaling from the cell-scale to the tumor and tissue scales. For example, a greater part of the system can be simulated using continuum approach while some critical regions can be modeled using discrete approach, as in (Kim et al. 2007; Stolarska et al. 2009). They simulate the motion of separate (discrete) cells in the outer proliferating rim of an avascular tumor while they use continuum description of cell dynamics (i.e., density of cells in various states) in the inner quiescent and necrotic regions of the tumor. On the other hand, both the continuum and discrete representations of tumor cells can be employed simultaneously throughout space, subject to mass and momentum conservation laws which incorporate interactions among the discrete and continuous fields (Bearer et al. 2009). Very

detailed overview of the recent continuum, discrete, hybrid and multi-scale models of tumor proliferation, containing almost 600 references, are presented in Lowengrub et al. (2010).

In macroscopic (>1 cm) and mesoscopic (>1 mm) scales tumor growth is a purely mechanical phenomenon. Just mechanical interactions influences the most the structure of vasculature, blood flow, tumor shape and decides about its directional progression. Due to the lack of computational framework, existing computational paradigms are not able to reproduce adequately this basic process. This is a serious obstacle in creation of a truly multi-scale model of tumor dynamics, which enables not only addition of novel components representing the chemical and biochemical inter and intracellular processes but allows for realistic tumor visualization in macroscopic scale as well. Moreover, the lack of computational metaphor which enables realistic visualization of tumor dynamics disables the possibility of creation interactive tumor growth simulator which allows for its dynamics be observed and modified directly by oncologist. We propose here the complex automata paradigm (CxA) based on particle dynamics as a computational framework of the mechanistic tumor model.

In Fig. 38.2 we present the main processes from various spatio-temporal scales involved in tumor growth. The microscopic processes such as cell motility and cell cycle are discrete. The macroscopic scale refers to phenomena which are typical for continuum systems such as diffusion (of oxygen and TAF), overall tumor condensation and blood flow. In macroscopic models, microscopic phases can be approximated

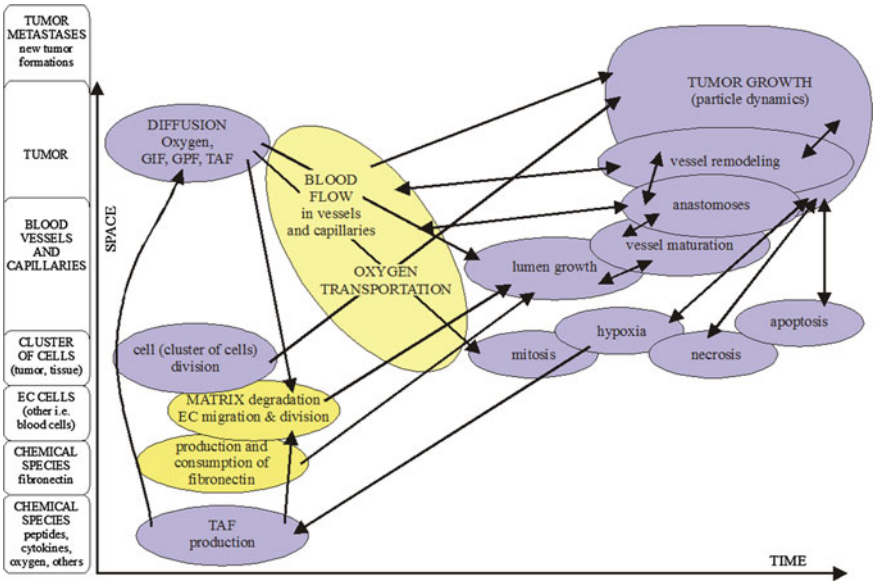


Fig. 38.2 Main processes from various spatio-temporal scales involved in tumor proliferation (from Wcislo et al. 2009)

by coarse grained models as long as the methodology of multi-scale simulation and adequate computational resources are lacking. The arrows in Fig. 38.2 show the relationships between these processes. The particle model presented in the following sections refers only to the processes shaded in blue.

Complex automata (CxA) are a generalization of cellular automata paradigm. They represent a scalable hierarchical aggregation of CA and agent-based models (Hoekstra et al. 2007). Globally, CxA can behave either as the classical CA nodes on a structural lattice or as interacting particles whose dynamics is described, e.g., by the Newtonian laws of motion or other stochastic laws.

In Wcislo and Dzwinel (2008, 2010), Wcislo et al. (2009, 2010) it was shown that the complex automata paradigm employing both particle dynamics and cellular automata rules can be used as a robust modeling framework, e.g., for developing realistic models of tumor growth as a result of emergent behavior of many interacting cells. This framework represents the spatio-temporal scales involving mechanical interactions between growing tumor, normal tissue and expanding vascular network. This framework can be easily extended by including both fine grained processes responsible for tumor creation/proliferation and be coupled with tissue scale processes modeled by continuum reaction-diffusion and blood hydrodynamics equations. In the following paragraph we present briefly the assumptions of our CxA particle model.

38.3 Particle Model as a Framework of the Interactive Visualization System

38.3.1 Model Description

As shown in Fig. 38.3, in our Complex Automata model a fragment of tissue and vasculature is made of particles. The particle is defined as $\Lambda_N = \{O_i : O(\mathbf{r}_i, \mathbf{v}_i, \mathbf{a}_i), i = 1, \ldots, N\}$ where: i is the particle index; N—the number of particles, $\mathbf{r}_i, \mathbf{v}_i, \mathbf{a}_i$—particle position, velocity and attributes, respectively. We assume additionally that:

- Each particle represents a single cell with a fragment of ECM (extracellular matrix).
- The vector of attributes $\mathbf{a}_i$ is defined by:
  - the particle type {*tumor cell* (*TC*), *normal cell* (*NC*), *endothelial cell* (*EC*)},
  - cell life-cycle phase shown in Fig. 38.4 {*newly formed, mature, in hypoxia, after hypoxia, apoptosis, necrosis*},
  - other parameters such as: cell size, cell age, *hypoxia* time, concentrations of k = TAF (tumor angiogenic factor), or O_2 (and other diffusive substances) and total pressure exerted on particle i from the rest of tumor body and tissue mass.
- The particle system is closed in 3-D computational box under a constant external pressure.

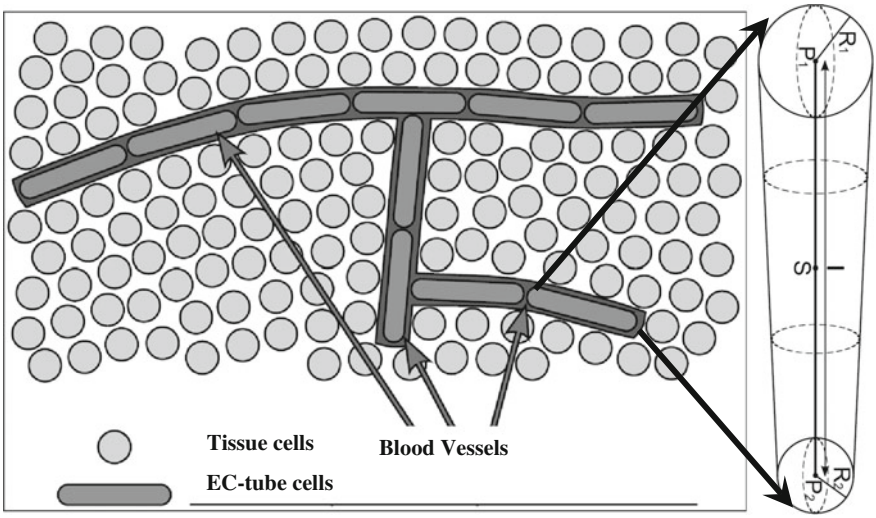


Fig. 38.3 Tissue particles and tube-like EC-tube particles made of two spherical "vessel particles"

- The vessel is constructed of tube-like "particles" (called EC-tubes) made of two particles connected by a spring (see Fig. 38.4).
- We define three types of interactions: particle–particle, particle–tube, and tube–tube.
- The forces between particles mimic both mechanical repulsion and attraction due to cell adhesiveness and depletion interactions cause by both ECM matrix and the cell itself.
- We postulate the particle–particle conservative interaction potential $\Omega(d_{ij})$ defined as follows:

$$\Omega(d_{ij}) = \begin{cases} a_1 d_{ij}^2, & \text{for } d_{ij} < 0 \\ a_2 d_{ij}^2, & \text{for } 0 < d_{ij} < d_{cut} \\ a_2 d_{cut}^2, & \text{for } d_{ij} \geq d_{cut} \end{cases} \quad \text{where } a_1 > a_2 \qquad d_{ij} = |\mathbf{r}_{ij}| - (r_i + r_j) \tag{38.1}$$

where $|\mathbf{r}_{ij}|$ is the distance between particles while r_i and r_j are their radiuses. Additional viscosity force, proportional to the particle velocity, simulates dissipative character of interactions. The cells of all kinds (tumor, normal and EC-tubes) evolve in discrete time according to Newtonian dynamics in the continuum diffusion fields of TAF and nutrients. The concentration fields are updated every time-step. We assume that both the concentrations and hydrodynamic quantities are in steady state in the time-scale defined by the time-step of numerical integration of equations of motion. This assumption is justified because diffusion of oxygen and TAFs through the tissue is many orders of magnitude faster than the process of tumor growth. On the other

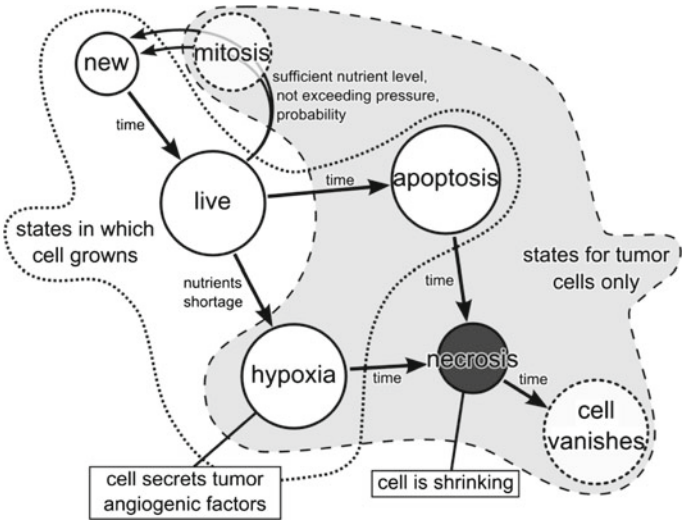


Fig. 38.4 Cell life cycle and various cell stages

hand, the blood circulation is slower than diffusion but still faster than cell-life cycle. Therefore we used fast approximation procedures for both calculation of blood flow rates in capillaries and solving reaction-diffusion equation.

As shown in Fig. 38.5 (see http://www.icsr.agh.edu.pl/~wcislo/Angiogeneza/index.html) we can observe and control both the phase of avascular tumor dynamics and its proliferation after angiogenic switch.

For example, the section of the tumor spheroid, shown in Fig. 38.5a, displays a layered structure. We can observe formation of a core zone composed mainly of necrotic material surrounded by a layer of quiescent tumor cells and an outer ring of proliferating tumor cells. It is crucial to understand the processes, which are responsible for the growth of a layered and saturating tumor.

Expanding tumor involves even more factors such as blood dynamics in newly formed vessels their regression/maturation and remodeling due to complex mutual interactions of TAF/nutrients/perycites and growing tissue/tumor pressure. Particle model allows for attacking many important problems influencing angiogenic switch e.g., if intravenous infusion of pericytes grown ex vivo will stabilize angiogenesis and slow down tumor growth (e.g., Darland et al. 2003). The snapshots from pilot simulations using the CxA particle model are presented in Fig. 38.5b.

The newly formed blood vessels due to the process of angiogenesis become functional when they form anastomoses. It allows for blood flow due to pressure gradient on its ends. Otherwise, we assume that the functional capillaries but without pericyte support dissolve. Moreover, nonfunctional and immature vessels without blood flow undergo the process of regression. The vessel maturation is controlled by the density of pericytes. The varying degrees of pericyte recruitment indicate differences in the functional status of the tumor vasculature. In simulation presented in

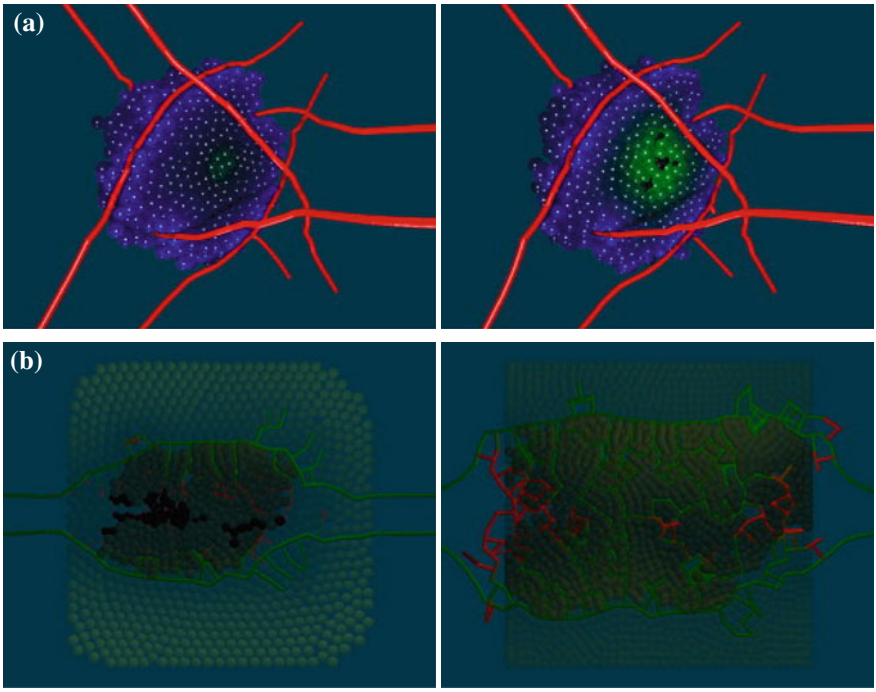


Fig. 38.5 Snapshots from particle based simulations of dynamics of **a** avascular tumor (necrotic center is displayed) **b** vascularization of tumor influenced by pericyties (functional vessels are in *green*, the *red* ones are capillaries under regression)

Fig. 38.5b we assume very simplistic model of vessel maturation. The regression time depends on the local density of EC tubes assuming constant concentration of pericytes. If the density is too high regression time is shorter. This model should be corrected calculating pericyte concentration using the continuum models mentioned earlier.

38.3.2 Directions of the Model Development

For characterization of e.g. melanoma cell lines, the model should simulate the migration of cancer cells in normoxic and hypoxic conditions. We show in Wcislo et al. (2009) that the inward motion of cells from the rim of 3-D tumor globule to the necrotic center is a purely mechanical effect caused by pressure drop from the surface towards this center. However, the mechanical effects allowing for cancer cell motility outward the tumor is more sophisticated phenomena (e.g. Matsumoto et al. 2008).

To find the mechanical and biological factors enabling tumor cell motility, the tumor mass dynamics should be modeled in a greater precision than it was done in Wcislo et al. (2009). In this model we simulate the tumor as a collection of soft spheres. However, for many types of tumor, the mass can be treated rather as a physical system with properties on the border between the solid and the liquid. Therefore, to simulate better the influence of adhesiveness of cells and viscosity for tumor fingering and compartmentalization we plan to use different model of cell interactions, similar to that in dissipative particle dynamics and fluid particle dynamics methods (Español 1998; Dzwinel et al. 2002).

The tumor, normal tissue cells and EC-tubes will be defined by its mass, moment of inertia, translational and angular momenta. Two particles i and j interact with one another by a collision operator $\mathbf{F}_{ij}$ defined as a sum of constituent forces, whose parameters are dependent on the type of interacting particles. The forces are central and non-central and consist of conservative $\mathbf{F}_{\mathrm{C}}$ (see Eq. (38.2)), dissipative $\mathbf{F}_{\mathrm{D}}$ and Brownian $\mathbf{F}_{\mathrm{B}}$ components and:

$$\begin{aligned} \mathbf{F}_{ij} &= \mathbf{F_C} + \mathbf{F_D} + \mathbf{F_B} \\ \mathbf{F}_{ij} &= -F(r_{ij}) \cdot \mathbf{e}_{ij} - \gamma \cdot [A(r_{ij})\mathbf{1} + B(r_{ij})\mathbf{e}_{ij}\mathbf{e}_{ij}] \\ &\quad \circ \left[\mathbf{v}_{ij} + \frac{1}{2}\mathbf{r}_{ij} \times (\vec{\omega}_i + \vec{\omega}_j) \right] + \tilde{\mathbf{F}}_B(r_{ij}) \end{aligned} \tag{38.2}$$

where: $F(r_{ij})$—is a central conservative force, whose formula depends on the types of interacting particles, γ—is a scaling factor of dissipative forces corresponding to viscosity, $A(r_{ij})$ and $B(r_{ij})$ are the weighting functions, $\mathbf{F_B}$ is a random component representing cell random motion. In original fluid particle method this factor is scaled by the temperature of the system and expressed in terms of the weighted Wiener increments. In our model it will define the dissipative properties of the tissue, i.e., its softness.

The value of $\mathbf{F}_{ij}$, is equal to 0 if the separation distance between two particles i and j, r_{ij}, exceeds a cut-off radius R_{cut}. The total force per each particle i is computed as a sum of forces interacting with particle i within the sphere of the radius equal to R_{cut}. The temporal evolution of the particle ensemble obeys the Newtonian equations of motion with rotation of the particles included,

$$\begin{gathered} \dot{\mathbf{P}}_i = \sum_{j;r_{ij}<r_{cut}} \mathbf{F}_{ij}(\mathbf{r}_i, \mathbf{v}_i, \omega_i), \quad \dot{\mathbf{r}}_i = \mathbf{v}_i, \quad \dot{\omega}_i = \frac{1}{I_i} \sum_{j;r_{ij}<r_{cut}} \mathbf{N}_{ij}(\mathbf{r}_i, \mathbf{v}_i, \omega_i), \\ \mathbf{N}_{ij} = -\frac{1}{2}\mathbf{r}_{ij} \times \mathbf{F}_{ij} \end{gathered} \tag{38.3}$$

where $\mathbf{r}_i$ is the position of particle i, $\mathbf{P}_i$ is its momentum and ω_i angular velocity. We assume that the interactions between spherical particles and EC-tube particles have a similar character.

To enable the simulations of tumor sizes above one centimeter the hybridization of two models—continuum approach and the CxA based framework—is the principal methodological goal for the future. It will consist of four steps:

1. Development of continuum mathematical model, which can be use for simulating tumor larger than 10 cm in diameter.
2. On the base of the wavelet solver (Vasilyev 2003)—extraction of the ROI (regions of interest) in which the discrete model will be used.
3. Elaboration of bridging procedures between continuum and discrete models.
4. Further development of the discrete particle based model towards, so called, coarse grained particle models where a particle does not represent a single cell but a fragment of tissue.

In the rest of this paper we limit our consideration to the particle based framework presented in Sect. 38.3.1. Our goal is to provide a flexible and fast simulation tool for interactive visualization which could be used on small but strong stand-alone workstations by clinicians for both educational and/or research purposes. In the following paragraph we present the implementation issue, which are crucial to achieve this goal.

38.4 Parallel Implementation of Particle Model

For modeling of tumors of realistic sizes, i.e., a few millimeters of diameter, the dynamics of 10^5–10^7 particles—normal, cancerous and EC-tube cells—have to be simulated by exploiting the power of nowadays multi-core CPUs, multi-processor systems and by using optimized N-body parallel codes. The box of the size 3.5 mm contains about 10^6 cancer cells—i.e. the approximate number of particles that can be simulated on a strong laptop in a reasonable computational time. To simulate larger tumors, the codes should be tuned to current parallel processors.

However, the CxA particle system is very different than standard particle ensembles such as in the short range molecular dynamics. Consequently, the process of code parallelization is more complicated (Wcislo and Dzwinel 2010; Wcislo et al. 2010). The cells can proliferate, change their size or annihilate. Moreover, they have additional attributes, which evolve according to the rules of CA and influence the cells' dynamics. The attributes, in turn, depend on concentration fields of O_2, TAF and other substances. This requires solving reaction-diffusion equations and calculating the intensity of blood flow in capillary vessels every time step.

38.4.1 Algorithms and Data Structures

Classical *N*-body codes, such as molecular dynamics (MD), simulate spatio-temporal evolution of a particle ensemble confined in a periodic cube by integrating numeri-

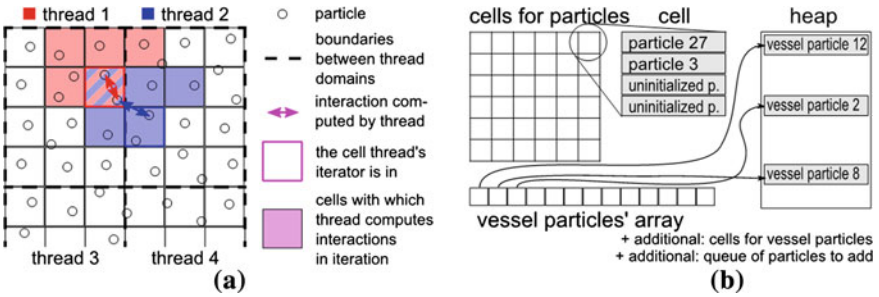


Fig. 38.6 **a** Domain decomposition used for forces calculation. **b** Data structures storing spherical and vessel particles (Wcislo and Dzwinel 2010)

cally Newtonian equations of motion (Haile 1992). Single time-step consists of two consecutive procedures: computation of forces acting on each particle and moving them according to the total momentum calculated.

For short-range interactions, the forces can be computed using fast $O(N)$ method exploiting alternately Hockney or Verlet algorithms [16]. However, both the calculation of forces and approximate procedure used for solving diffusion equation, require finding all the pairs of particles in the nearest neighborhood. As shown in Fig. 38.6a, the computational box is divided onto cubic sub-boxes with edges equal to the interaction range. The particle located in a given sub-box interacts with other particles located in this sub-box and in adjacent sub-boxes.

As shown in Figs. 38.7 and 38.8, the computation of EC-tube particles interactions is the most critical component influencing computational efficiency. The length of EC-tube is considerably greater than its width. It involves considerably larger sizes of sub-boxes (Fig. 38.6) than those used for spherical particles. Moreover, the tubes can grow exceeding the size of 5 sub-boxes used for forces calculation between spherical cells.

To solve this problem we propose using instead of one array of particle positions, two separate data structures **P** and **V**: **P** for storing spherical particles and **V** for EC-tubes, respectively (see Fig. 38.6b). The **P** data structure is represented by 3-D array of Hockney sub-boxes (Hockney cells) with tumor and normal particles. The **V** is a data structure consisting of the array of pointers to records representing EC-tubes and the additional 3-D array of sub-boxes used to compute particle-tube and tube-tube interactions. The sub-boxes in this array correspond to respective sub-boxes in **P**. Because vessel particle is long enough to cross several sub-boxes, it cannot be assigned to a single sub-box, as it is in Hockney algorithm. Instead, EC-tube is placed in a minimal cuboid composed of all the sub-boxes it crosses. This cuboid is enlarged then by one sub-box margin in each direction, covering the vessel particle together with its cut-off radius. We assume that the vessel particle belongs to all the sub-boxes forming this final cuboid (Fig. 38.9).

Calculation of forces between particles is realized by three separate algorithms: particle–particle, particle–vessel and vessel–vessel interactions calculation. Particle–

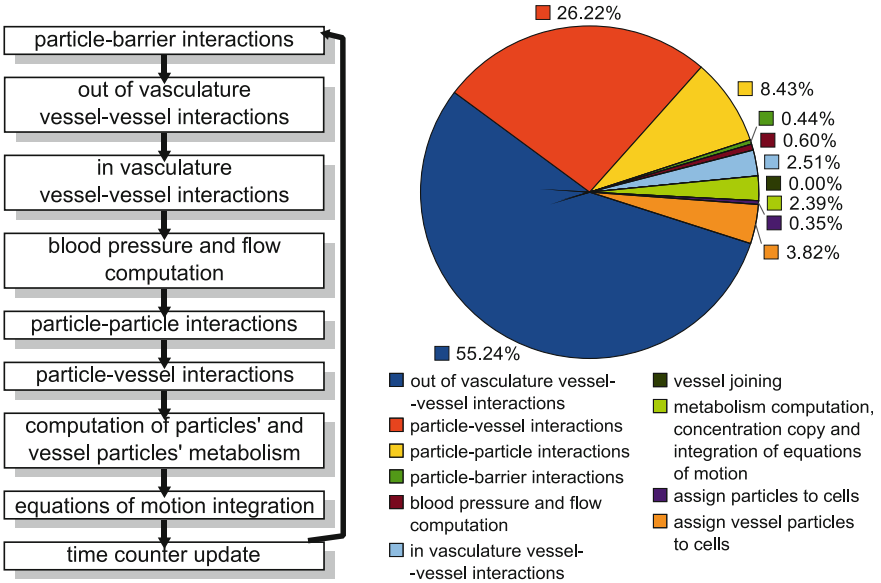


Fig. 38.7 The main procedures invoked in a single time-step and diagram showing the shares of computational time used by various procedures of the model (the evolution of 10^6 particles was simulated)

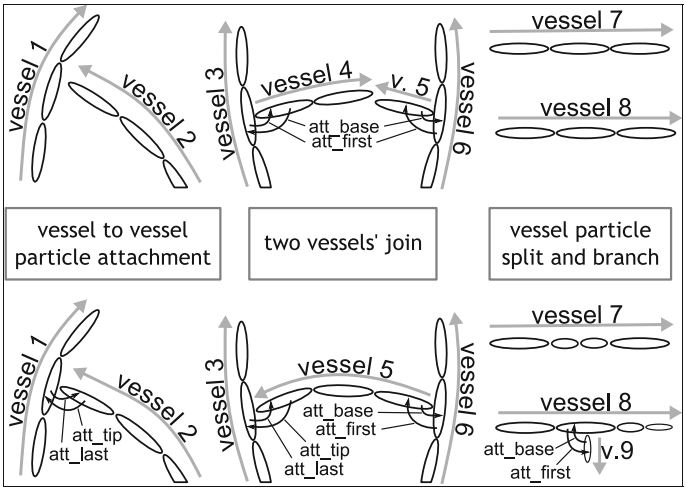


Fig. 38.8 Vessel–vessel interactions and vessel growth rule

particle forces are calculated using standard Hockney algorithm (Hockney and Eastwood 1981). In case of particle-vessel computation, for each corresponding pair of sub-boxes *cp*, *cv* from **P** and **V**, respectively, particles from *cp* are tested against

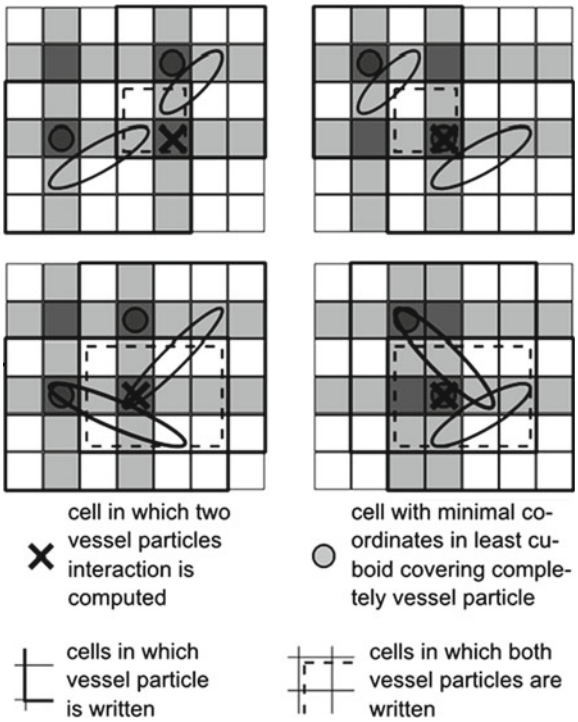


Fig. 38.9 Graphical interpretation of relation (4) (Wcislo and Dzwinel 2010)

vessel particles from *cv*. If the distance between the pair of particles is shorter then the cut-off radius, their mutual interaction is calculated.

The algorithm for vessel–vessel computations is constructed knowing that if two EC-tubes lie in a distance shorter than cut-off radius, there exists at least one sub-box in **V** containing both particles. Therefore, all interacting pairs can be found by iterating throughout all sub-boxes and testing all-to-all distances. The problem is that a pair of particles representing two interacting EC-tubes can be found in many sub-boxes while it should be taken only once. To solve it, we introduce ternary relation R (see Fig. 38.9), which eliminates redundant interactions:

$$Ec \times Ec \times C \supseteq R = \left\{ \begin{array}{r} (e_1, e_2, c) : e_1, e_2 \in Ec; c \in C; \\ \max(e_1.mx, e_2.mx) = c.x, \\ \max(e_1.my, e_2.my) = c.y, \\ \max(e_1.mz, e_2.mz) = c.z \end{array} \right\} \tag{38.4}$$

where: Ec is the set of EC-tubes, C is the set of sub-boxes in mesh, $c.x$, $c.y$, $c.z$ are coordinates of sub-box c in 3-D array and

$$\begin{aligned} e.mx &= \min(\text{sub-box}(e.p1).x, \text{sub-box}(e.p2).x), \\ e.my &= \min(\text{sub-box}(e.p1).y, \text{sub-box}(e.p2).y), \\ e.mz &= \min(\text{sub-box}(e.p1).z, \text{sub-box}(e.p2).z), \end{aligned} \tag{38.5}$$

where: *e.p1* and *e.p2* are two ends of EC-tube e, sub-box(p) is the sub-box to which point p belongs to.

To reduce the number of cache misses, the sub-boxes in **P** do not contain pointers to particle records but whole records instead. Each sub-box is represented then by an array of fixed number of objects (see Fig. 38.6b). All the tumor and normal particles are allocated directly inside corresponding sub-boxes. This guarantees that particles are always properly ordered in memory according to their positions. However, we pay the price of greater memory consumption. This is because the sub-boxes have various numbers of particles and many records are empty. As particles move, they change the sub-boxes they belong to. Therefore, the arrays **P** and **V** are updated after each time-step. In case of **V**, the sub-boxes are built from the beginning by using the array of EC-tube pointers located in **V**. Whereas for **P**, because particles are allocated inside the sub-boxes, changing location from one sub-box to the other means that the whole particle record must be moved to a different memory location.

This takes longer time in comparison to pointers operation in the standard approach. However, because of steady nature of particles dynamics in our model, such the situation does not occur too often. In fact, in our simulations the process of reordering particles requires less time than standard linked-list procedure. The reason is, that in the former, the particles which do not change their sub-boxes need only "read" operation of their coordinates from the memory, while in the latter, for all the particles there is an additional "write" operation.

During simulation, the number of spherical and EC-tube particles can both increase due to *mitosis* and decrease as the result of *apoptosis* and *necrosis*. Information of newly formed and dead particles must be added and removed from the data structures. As doing this directly could cause problems with synchronization, three intermediate data structures are employed: for newborn particles in **P**, for new vessel particles in **V** and for indexes of dying vessel particles in **V**. Removing objects from **P** is done directly as it is sub-box-local operation, which does not impair other threads operation and never cause data structures to be rebuilt. Moving object from intermediate structures to **P** and **V** and removing object from **V** is done sequentially between separate time-steps.

38.4.2 Speedups and Exemplary Results

Our parallel algorithm is constructed for a single shared memory node and is implemented in C++ with OpenMP interface. The timings were obtained for SGI Altix XE 1300 cluster consisting of 256 SGI Altix XE 300 nodes and SPARC Enterprise T5120. The single Altix node consists of two four-core processors Intel Xeon 2.66

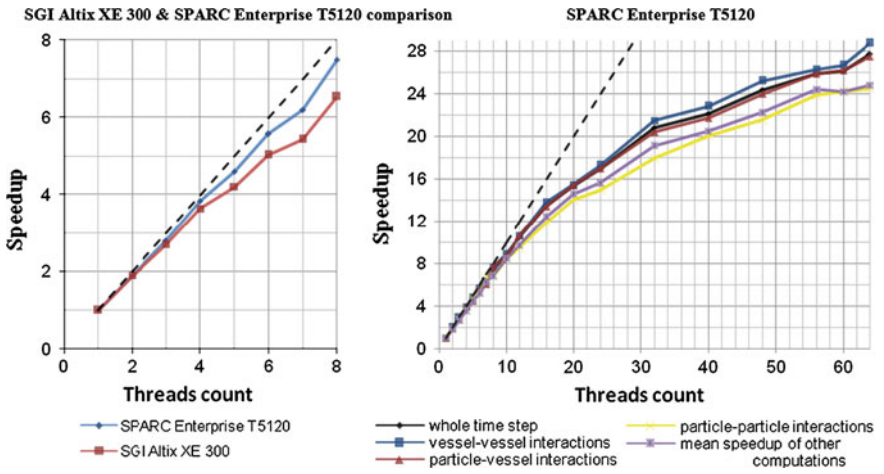


Fig. 38.10 The speed-ups obtained for the main procedures of the tumor model during simulation of 10^6 particle ensembles on two test machines (Wcislo and Dzwinel (2010))

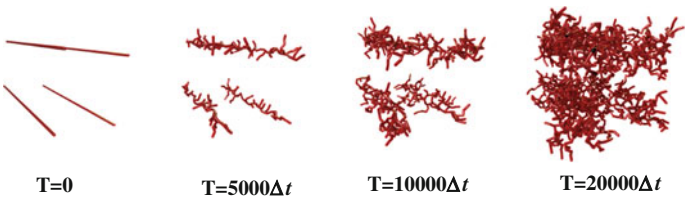


Fig. 38.11 The snapshots showing the time evolution of tumor vasculature. The tissue particles are invisible

with 16 GB of RAM allowing for maximum 8 threads executed in parallel. The SPARC computer consists of eight-core 1.2 GHz UltraSPARC T2 CPU capable of running in parallel eight threads per single core. It gives in total 64 threads per node executed concurrently on 32 GB of RAM.

We have employed domain decomposition both along one side of the computational box (each box slice was handled by one thread) and dividing the box onto sub-boxes of equal sizes (for 8 threads we have $2 \times 2 \times 2$ grid of sub-boxes, while for 64, $4 \times 4 \times 4$ grid of sub-boxes).

As shown in Fig. 38.10, the preliminary timings obtained for our parallel code are very encouraging. We got speed-up of about 7 on 8 threads CPU and about 30 on 64 threads CPU simulating 10^6 particles. The timings could be better for more realistic vessel densities much lower than those considered in the test runs.

The snapshots from simulations of tumor vasculature progression obtained for timing tests are shown in Fig. 38.11. The tests were performed for particle ensembles of various sizes. The initial scene consists of two straight parallel vessels, the cells representing normal tissue and a few cancerous cells located between the vessels.

Because of increasing TAFs concentration, secreted by the tumor cells in *hypoxia*, we can observe newborn capillaries sprouting out from the source vessels. The vasculature expands and is continually remodeled due to tumor growth dynamics. The sprouts can bifurcate and merge creating anastomoses. The blood flow is stimulated by pressure difference in anastomosing vessels. Only productive vessels have a chance to survive if the TAFs concentration is sufficiently high. Unproductive vessels disappear after some time. Well oxygenated cells are colored blue (dark gray) while the cells in hypoxia are marked by shades of green (light gray). Necrotic cells are black.

38.4.3 Possible Improvements

To exploit the full power of multiprocessor system, the second level parallelism could be introduced based on massage-passing MPI interface. However, it would make the code extremely complicated and rigid for improvements. This could also extend the time for implementation and tests. Moreover, running the code on the large number of CPUs is usually restricted by system administrators and consumes much time and money. So, having in mind the shift in modern chip technology towards production of multiple-core CPUs (empowered by GPU) we decided to meet this trend implementing the code open for both future improvements in the model and technological progress.

Tuning the code for GPU architecture seems to be much better idea. We can expect that the computations can be considerably accelerated. We have implemented on Nvidia GPUs in CUDA environment the procedure of calculation of particle–particle interactions, which exploits brute force algorithm combined with Hockney's one. Proposed algorithm was tested on two CUDA-enabled devices: GeForce GT 330M and Tesla C1060. Their technical parameters are summarized in Table 38.1.

In the algorithm used for forces calculations we divide the computational box on reference boxes in which we calculate interparticle distance arrays using Hockney algorithm. We apply data structure consisting of independent vectors representing particles (storing particle position, velocities, forces, and attributes) which number depends on the average number of particles placed inside a single reference box. We increase the size of reference boxes to check the algorithm stability, having in mind

Table 38.1 The parameters of GPU devices

Device	GeForce GT 330M	Tesla C1060
Compute capability	1.2	1.3
Number of multiprocessors (MP)	6	30
Number of CUDA cores (8/MP)	48	240
Global memory (MB)	1023.3	4095.8
Clock frequency (GHz)	1.26	1.296

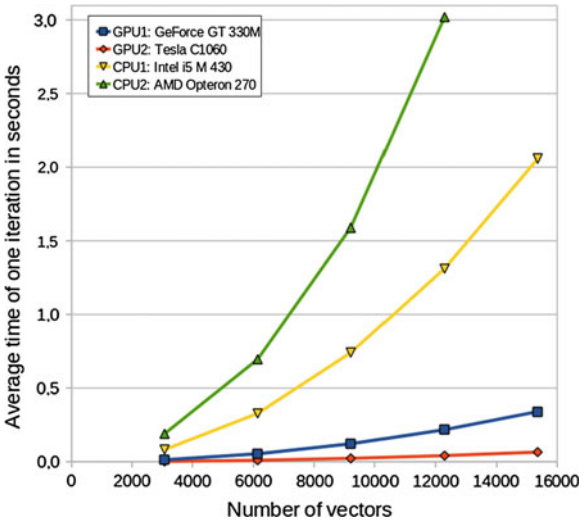


Fig. 38.12 Timings obtained for two versions of particle–particle procedure: one optimized in CUDA and the second in OpenMP environment

that the computation of the vessel–vessel interactions requires larger Hockney sub-boxes (see Fig. 38.9). The computational time increases with square of the number of vectors, however, in our model, the number of particles in reference boxes will be not larger than 200.

As shown in Fig. 38.12, GPU implementations of particle–particle procedure are order of magnitude faster than their multi-thread CPU counterparts implemented in OpenMP environment. In Table 38.2 we present the speed-ups obtained for the particle–particle procedure (in 3-D) implemented in CUDA implementation versus its MPI four-thread implementation on Intel i5 M 430. We can observe that the speed-up is very stable and does not depend on the number of the vectors processed, i.e., no degradation of computational efficiency is observed for increasing size of the cut-off radius. This result is very encouraging.

However, the comparisons are made for the same algorithm employing in both environments. The confrontation with the algorithm presented in Sect. 38.4 involves tuning its GPU version to the requirements of the whole model and real simulation conditions. We plan to implement soon this algorithm in our CxA model of tumor dynamics. We expect to obtain the GPU/CPU speed-up at least 10 comparing to the CPU algorithm presented in Sect. 38.4.1. Such decrease of computational time will enable to perform interactive simulations and visualizations of same types of tumors (e.g. melanoma) in angiogenic phase. That is, the tumors of sizes of up to 1 cm in diameter can be interactively visualized.

Table 38.2 Speed-up calculated against CPU algorithm employing four threads and running on Intel i5 M 430

Number of particles	GeForce GT 330M	Tesla C1060
128	6.42	31.37
3072	6.12	31.77
15360	6.04	31.21

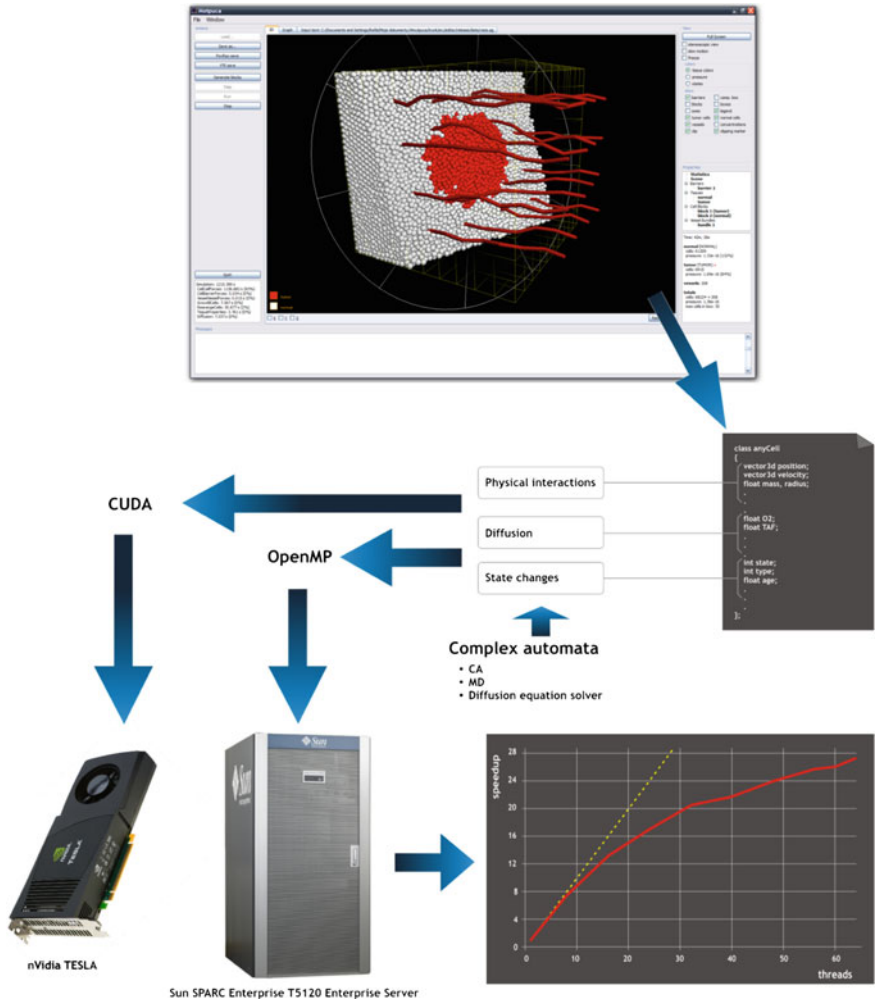


Fig. 38.13 The concept of the interactive tool for visualization of tumor dynamics

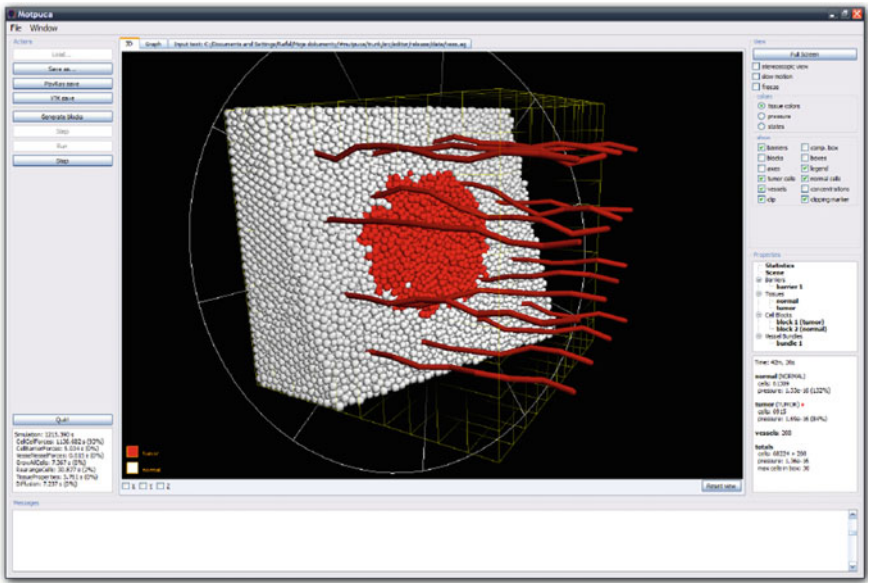


Fig. 38.14 The screenshot of the interactive system interface

38.5 The System Implementation

In Fig. 38.13 we present our vision of a stand-alone system for interactive visualization of tumor dynamics. The system consists of the parallel implementation—both on CPU and GPU—of the CxA based tumor model and a flexible, user-friendly, interface (see Fig. 38.14).

The simulation program allows several types of tissues to be simulated. Each type of tissue (soft tissue, bones etc.) can be described by the set of characteristic features such as hardness, density, average size of cells, the rate of diffusion of particular substances (oxygen, TAF), oxygen requirements, life span, resistance to oxygen deficiency, etc. In our system we use predefined sets of data for selected types of tissues, so it is not necessary to set all the parameters every time. These sets are prepared earlier by the specialists on the basis of biomedical data.

The interface, displayed in Figs. 38.14 and 38.15, enables to position the tissues as well as adjust their size, location and space orientation facilitating the creation of initial simulation scene. For example, we display in Fig. 38.14 a typical situation when a cancerous tissue is surrounded by healthy cells and the two are interwoven with the network of blood vessels. Such the operations are mostly performed with the help of a computer mouse and a few keyboard shortcuts and function keys. The interface allows to the user both rendering the scene and watching the simulation from various angles, zoom in and zoom out, rotate etc. Moreover, it makes possible

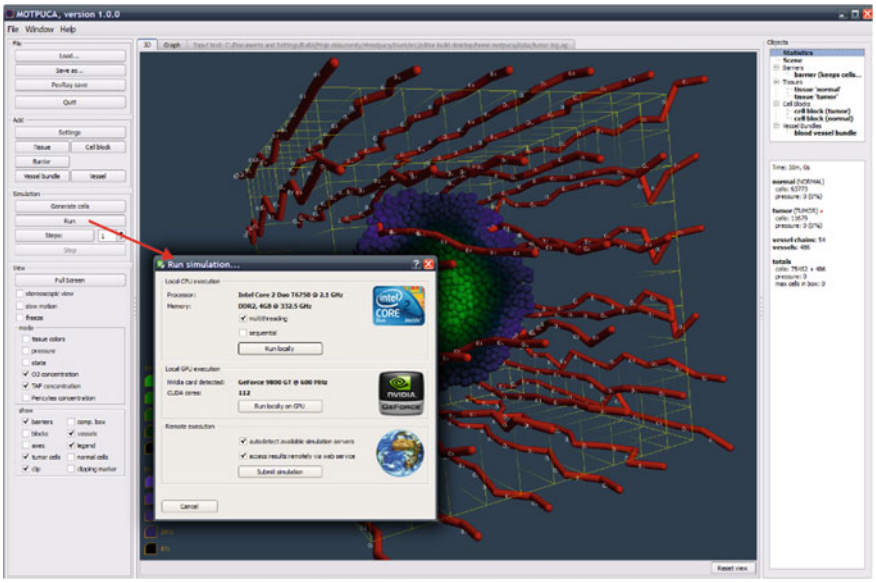


Fig. 38.15 Different modes of system run: CPU OpenMP or GPU CUDA

to observe an arbitrary intersection of the evolving tumor and surrounding tissue. This way the interface allows for observing tissue interior for 3-D simulations.

During both initiation stage and in the course of simulation, each model parameter can be modified interactively. Moreover, the simulation conditions can be changed anytime, e.g., some parts of tissue can be removed and/or added. This functionality is very important for examination the tumor dynamics after surgical intervention. The modified simulation scenario might also be saved as a template for the following simulations.

The modeling can be carried out in two or three dimensions. However, though 3-D simulations are more realistic, they are more demanding computationally. It can be easily estimated that assuming the same number of particles, 3-D simulation will be at least three times slower than its 2-D counterpart. As soon as the simulation scenario and initial conditions are defined, the system is ready to run. It might be run either on a multi-core CPU with the shared memory—then the simulation program uses the OpenMP libraries—and, in the nearest future, on GPU devices in CUDA environment. This warrants some degree of sustainability just in case when one technology appears to be superior over the other. As shown in Fig. 38.15, the proper version can be selected at the beginning of modeling experiment.

Thanks to parallel realization, the system allows for interactive, fluent visualization of tumor dynamics for particle ensembles consisting of hundred of thousands of particles. If the simulation is performed on-line (Figs. 38.14 and 38.15), the program displays information about particular tissues (e.g. the number of cells, the number

of cells in various states, pressure, O2 concentration). Larger models can be also simulated in batch mode, producing movies and diagrams.

38.6 Validation Tool

The microvascular density (MVD) is the parameter describing structural properties of blood network used from years as an important descriptor in cancer therapy (Eberhard et al. 2000). The value of MVD depends on space position, moreover it gives poor knowledge about the topology of a vasculature . Therefore, it cannot be used alone as a "fingerprint" describing current state of cancer dynamics. A quantitative understanding of tumor dynamics requires more sophisticated, universal, space and case invariant descriptors representing the most relevant topological features of tumor vasculature.

The knowledge about topology of biological networks can play a significant role in understanding processes taking place in tissue and organs (see e.g. Albert and Barabasi 2002; Girvan and Newman 2002). As shown in Newman (2003), the study of network topology using graph descriptors can help in understanding the behavior of the entire system and also in models validation by comparing results of in-silico experiments with in-vivo ones. The structure of a network encodes a number of global and local information, which can be extracted by dedicated measures and used in further analysis. Quantitative evaluation of such network properties as connectivity, symmetry, ability to transfer signals or to form node clusters can reveal qualitative features of underlying complex system. In addition, local information, e.g., vertex centrality or vertex clustering coefficient enables to find sub-networks playing the most significant role in the whole system and allows for grouping functionally similar nodes. The analysis of vertex or edge features distributions can also bring insight into system-level characteristics. Some general complex network descriptors are collected in Table 38.3.

In order to evaluate whether vascular networks generated in numerical simulations are really similar to the realistic ones, we plane to validate our model using complex network descriptors (Newman 2003). This validation can be conducted on the basis of the comparison between realistic images of tumor vascular networks from confocal microscopy and computer experiments. As shown in Figs. 38.16 and 38.17 the topology of vasculature can be extracted using skeletonization filter and consequently transformed to a graph. Subsequently, the graph can be described by the feature vectors with statistical and/or algebraic descriptors of complex networks (Newman 2003; Czech 2012) as the feature vector components. Finally, pattern recognition and machine learning methods can be used for the vector classification and hypotheses formulation. This situation is sketched in Fig. 38.18.

Promising application of graph matching algorithms arises as far as inter-regional variation of vasculature in tumor is considered. Quantifying relations between vascular networks in different types of tumors can also bring valuable conclusions.

Table 38.3 Selected graph descriptors available in *Graph Investigator* application

Descriptor	Remarks
Randić Connectivity Index	$\chi(G) = \sum_{(u,v)\in E} (k_u, k_v)^{\frac{1}{2}}$, where k_i denotes the degree of a vertex i. Connectivity measure derived from chemical graph theory
General Connectivity Index	${}^{l}\chi(G) = \sum_{P_l \subset G} \left(\prod_{w \in V_P} k_w \right)^{-1/2}$, where P_l denotes path of length l and V_P all vertices that belong to this path
Zagreb Index M_1	$M_1 = \sum_{v\in V} (k_v)^2$
Zagreb Index M_2	$M_2 = \sum_{s\in E} w_s$, where w_s is a weight of edge e, for unweighted graphs $w_s = 1$
Modified Zagreb Index (${}^{m}M_1$)	$({}^{m}M_1) = \sum_{u\in V} (k_v)^{-2}$
Modified Zagreb Index (${}^{m}M_2$)	$({}^{m}M_2) = \sum_{s\in E} (w_s)^{-1}$
Total Adjacency Matrix	$A(G) = \sum_{v\in V} (k_v) = 2\|E\|$
Modified Total Adjacency Matrix	$({}^{m}A(G)) = \sum_{v\in V} (k_v^{-1})$
B Index*	$B(G) = \sum_{v\in V} \frac{k_v}{d_v}$, where d_v is vertex distance for vertex v
Vertex distance	Sum of distances between v and all other vertices from a graph G. $d_u = \sum_{v\in V} d(u, v)$
Density of edges	$den(G) = \frac{2m}{n(n-1)}$, where $m = \|E\|$
Information of vertex degrees	$I_{vd} = \sum_{v\in V} (k_v \log_2 k_v)$ Reflects connectivity and topological complexity in terms of number of branches, cycles, cliques, etc.
Radius of graph*	$r(G) = min_{v\in V}\varepsilon_v$, where ε_v is eccentricity of vertex v
Vertex eccentricity*	Maximum distance between vertex v and any of the remaining graph vertices $\varepsilon_v = max_{u\in V} d(u, v)$
Graph diameter*	$diam(G) = max_{u,v\in V} d(u, v)$ Reflects density of graph connections, achieving its maximal value for paths and minimal for cliques
Total Walk Count	Counts all paths of all lengths in the graph and depends on the size, cyclicity, and branching of the graph quantifying property called *labyrinthicity*
Efficiency*	$E(G) = \frac{1}{n(n-1)} \sum_{u,v\in V, u\neq v} \frac{1}{d(u,v)}$ Measures the traffic capacity of a network and reflects its parallel-type transfer ability
Heat Content, Heat Content Coefficients*	Parameterized descriptors based on heat kernel matrix
Closeness*	Mean distance to each other vertex
Betweenness	Measures relative importance of a vertex in a shortest-path transfer through graph edges
B-Matrix descriptors*	Graph B-matrix encodes information about distribution of l -order degrees of graph vertices. As l -order degree of a vertex v we understand the size of the set of all vertices located at distance l from a vertex v

The descriptors denoted by asterisk can be computed efficiently using CUDA kernels

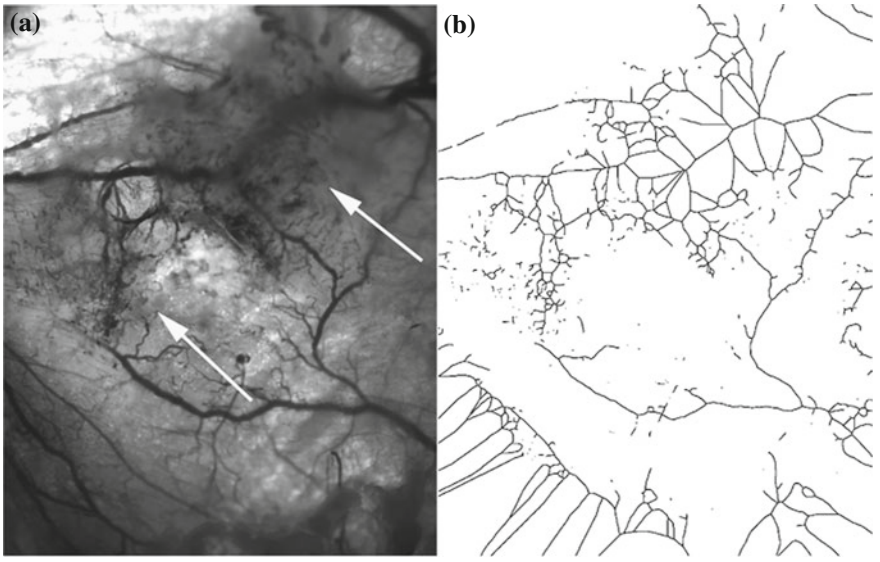


Fig. 38.16 The result of skeletonization of a picture displaying a blood network surrounding tumor mass

Fig. 38.17 The result of skeletonization of a picture displaying a vascular network in healthy adipose tissue (Parish 2003)

For example, by comparing topologies of blood vessels in different metastatic tumors of the same type we might test hypothesis about their similarity (Fig. 38.19).

We have recently developed Czech et al. (2011) *Graph Investigator*—a robust programming package—which is capable of capturing topological features of networks with the use of various descriptors derived from graph theory. The screenshot of user interface is shown in Fig. 38.20. The set of available graph descriptors includes over eighty statistical and algebraic measures. It allows for performing both inter-network comparisons and to analyze global and local structural properties of networks on the basis of diverse criteria. Furthermore, it allows for quantifying inter-graph similarity

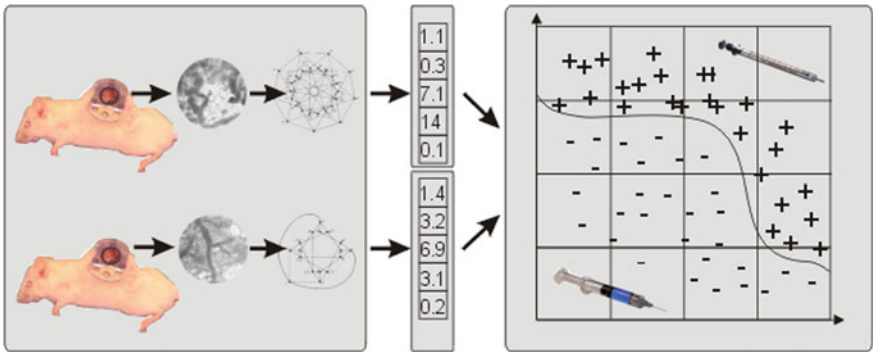


Fig. 38.18 The diagram showing the process of feature vector generation on the basis of graph descriptors of tissue vasculature and hypotheses generation using machine learning tools

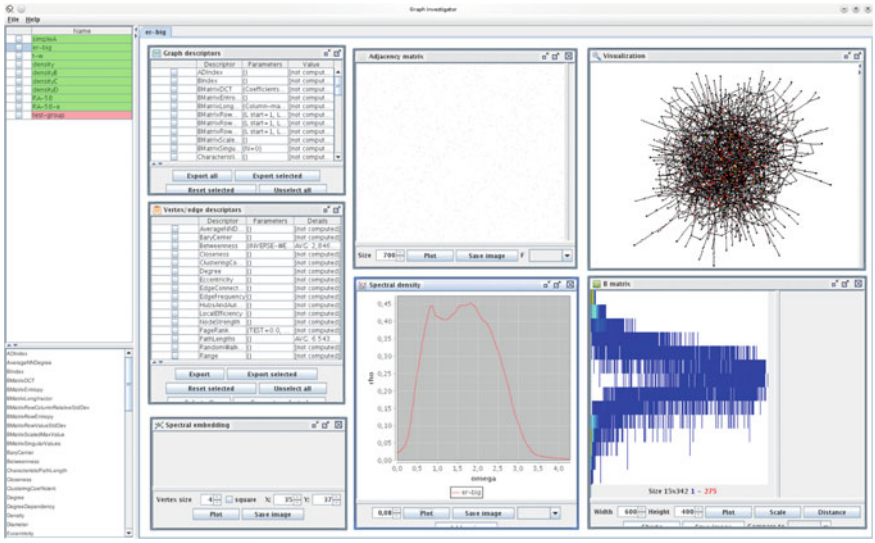


Fig. 38.19 The main panel of *Graph Investigator*. The internal windows show visualization and computational modules of the application

by embedding graph patterns into low-dimensional space or distance measurement based on feature vectors.

To supply the user with validation tool we plan to integrate *Graph Investigator* application with the tumor modeling simulator. The simulator is written in Java and much of descriptors are computed exploiting CUDA environment on GPGPU Nvidia devices. Recently, the efficient GPU implementations of BFS (Breadth-First Search) and all-shortest-paths algorithms were presented (Luo et al. 2010; Tran 2010). We use CUDA implementation of BFS algorithm to obtain distance matrix for a graph that form the basis for computation of several graph descriptors such as efficiency

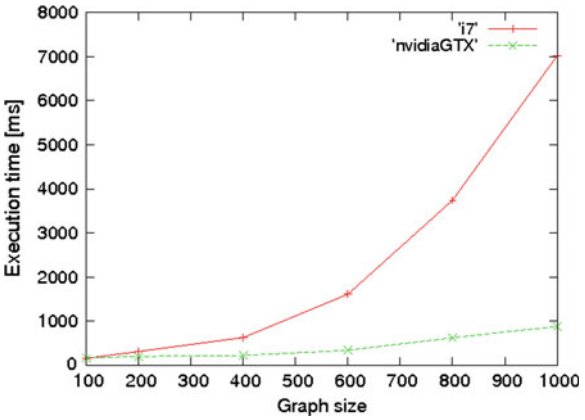


Fig. 38.20 The comparison of computation times of graph diameter descriptor for Erdős-Rényi graphs with edge existence probability p = 0.001 and sizes varying from 100 to 1000. The *red line* corresponds to CPU execution of BFS algorithm (Intel Core i7-920). The *green line*—CUDA implementation of BFS (Nvidia GTX 280)

or graph diameter. The CUDA kernels are invoked form *Graph Investigator* through *Java Native Interface*. The graph descriptors that can be computed with a help of GPU are marked in Table 38.3 with asterisk. Figure 38.20 displays comparison of computation times of graph diameter for CPU and GPU implementations. The test was performed on random Erdős-Rényi graphs with p = 0.001 (probability of edge existence). The graph density grows with a graph size. The execution times presented in Fig. 38.20 were averaged over 100 instances. For graphs of size 1000, the GPU implementation is 8 times faster.

Using both applications i.e., simulator and validation tool, it is be possible to observe, analyze and control on-line vasculature evolution using network descriptors. The multidimensional feature vector can be visualized using multidimensional scaling or principal component analysis tools. This way one can control qualitative changes in tumor dynamics (Topa and Dzwinel 2009) and match interactively modeling parameters to experimental results.

38.7 Summary and Conclusions

We have discussed a general vision of the system for interactive simulation and visualization of tumor dynamics and its ready-to-use components. It is intended as educational and research tool for oncologist and clinicians which could be deployed on a small but strong stand-alone workstations. In the future, such the tool could be used for planning cancer treatment.

The system is based on the concept of Complex Automata, which combines particle method and Cellular Automata modeling techniques. We show that the complex automata paradigm can be used as a framework for developing realistic models of tumor growth as a result of emergent behavior of many interacting cells. Successful and realistic reconstruction of mechanical interactions between proliferating tumor, healthy tissue, and evolving vascular network is the most important advantage of the CxA framework over other modeling paradigms.

The CxA model has many limitations. For example, it addresses a limited number of biological processes. Some of them, such the blood flow, received less attention than in other approaches. We did this intentionally to reduce the computational load. On the other hand, CxA model of tumor growth can be easily extended by implementing more precise sub-models of all the processes—known and unknown—responsible for tumor proliferation. However, the improvements by including more detailed processes and developing the particle model on its own (as it is proposed in Sect. 38.3.2) can be undertaken provided that the efficient parallel version of the model will be implemented allowing for interactive visualization and simulation. We have demonstrated that due to parallelization of the model using OpenMP environment on modern multiple core CPUs we can improve computational efficiency almost one order of magnitude. We show that similar factor is achievable using GPGPU devices. This way, relatively large tumors can be modeled.

We expect that just technological progress simultaneously with improvements in parallel implementation of the model will allow for increasing of both the size of tumors simulated and the accuracy of the results obtained. This is the reason that the system is constantly being tuned to nowadays computational and visualization facilities allowing for considerable increase of simulation efficiency.

In the paper we also describe the interface that will be employed for setting up initial conditions and to assure interactive visualization of tumor dynamics. Such the user-friendly interface is particularly important for preparing the initial simulation scenario of tumor growth, which requires handling with hundreds of parameters.

The system will be developed in accord to both ICT technology development and the increasing knowledge about reasons and processes involved to cancer growth. The following versions of the system will be replenished, on the one hand, with more and more detailed microscopic and macroscopic processes and, on the other, its better numerical realization and computer implementation.

To supply the user with validation tool, we will integrate the tumor growth simulator with *Graph Investigator*—ready-to-use data mining tool for interactive analysis of topological structure of tumor vasculature. Such an integrated simulation/data analysis tool would allow to do research on cancer development undoubtedly more effectively due to faster search of parameters domain and hypothesis verification.

Acknowledgments This research is financed by the Polish Ministry of Higher Education and Science, project N519 579338 and partially by AGH grant No. 11.11.120.777. We would like to thank Nvidia Company for support and donating Authors with Tesla C1060 GPU. The Authors are also very grateful to Professor Dr Arkadiusz Dudek MD PhD from University of Minnesota Medical School, Division of Hematology, Oncology, and Transplantation, Department of Medicine,

and Mr Piotr Pawliczek from AGH Institute of Computer Science for cooperation in designing algorithms and preparing timings on GPU processors.

References

Alarcon T, Byrne H, Maini PK (2005) A multiple scale model for tumor growth. Multiscale Model Simul 3:440–475

Albert R, Barabasi A (2002) Statistical mechanics of complex networks. Rev Modern Phys 74:47–97

Amyot F, Small A, Gandjbakhche AH (2006) Stochastic modeling of tumor induced angiogenesis in a heterogeneous medium, the extracellular matrix. In: Proceedings of 28th IEEE EMBS annual international conference New York City, USA, 30 Aug-3 Sept 2006

Bauer AL, Jackson TL, Jiang YA (2007) Cell-based model exhibiting branching and anastomosis during tumor-induced angiogenesis. Biophys J 92:3105–3121

Bearer EL, Lowengrub JS, Chuang YL, Frieboes HB, Jin F, Wise SM, Ferrari M, Agus DB, Cristini V (2009) Multiparameter computational modeling of tumor invasion. Cancer Res. 69:4493–4501

Bellomo N, de Angelis E, Preziosi L (2003) Multiscale modeling and mathematical problems related to tumor evolution and medical therapy. J Theor Med 5:111–136

Chaplain MAJ (2000) Mathematical modelling of angiogenesis. J. Neuro-Oncol 50:37–51

Czech W (2012) Invariants of distance k-graphs for graph embedding. Pattern Recogn Lett 33(15):1968–1979

Czech W, Dzwinel W, Goryczka S, Arodz T, Dudek AZ (2011) Exploring biological networks with Graph Investigator research application 30:1001–1030

Darland DC, Massingham LJ, Smith SR, Piek E, Saint-Geniez M, D'Amore PA (2003) Pericyte production of cell-associated VEGF is differentiation-dependent and is associated with endothelial survival. Dev Biol 264(1):275–288

Dormann S, Deutsch A (2002) Modeling of self-organized avascular tumor growth with a hybrid cellular automaton. In Silico Biol 2:393–406

Dzwinel W, Yuen DA, Boryczko K (2002) Mesoscopic dynamics of colloids simulated with dissipative particle dynamics and fluid particle model. J Mol Model 8:33–45

Eberhard A, Kahlert S, Goede V, Hemmerlein B, Plate KH, Augustin HG (2000) Heterogeneity of angiogenesis and blood vessel maturation in human tumors: implications for antiangiogenic tumor therapies. Cancer Res 60:1388–1393

Español P (1998) Fluid particle model. Phys Rev E 57:2930–2948

Folkman J (1971) Tumor angiogenesis: therapeutic implications. N Engl J Med 285:1182–1186

Folkman J, Hochberg M (1973) Self-regulation of growth in three dimensions. J Exp Med 138: 745–753

Girvan M, Newman MEJ (2002) Community structure in social and biological networks. Proc Natl Acad Sci 99:7821

Godde R, Kurz H (2001) Structural and biophysical simulation of angiogenesis and vascular remodeling. Dev Dyn 220:387–401

Haile PM (1992) Molecular dynamics simulation. Wiley, New York

Hockney RW, Eastwood JW (1981) Computer simulation using particles. McGraw-Hill, New York

Hoekstra AG, Lorenz E, Falcone LC, Chopard B (2007) Towards a complex automata framework for multi-scale modeling: formalism and the scale separation map. Lect Notes Comput Sc 4487:1611–3349

Jemal A, Siegel R, Jiaquan Xu, Ward E (2010) Cancer statistics 2010. CA Cancer J Clin 60:277–300

Kim Y, Stolarska MA, Othmer HG (2007) A hybrid model for tumor spheroid growth in vitro: I. Theoretical development and early results. Math Methods Appl Sci 17:1773–1798

Lee D-S, Rieger H, Bartha K (2006) Flow correlated percolation during vascular remodeling in growing tumors. Phys Rev Lett 6:058104–1-4

Lowengrub JS, Frieboes HB, Jin F, Chuang Y-L, Li X, Macklin P, Wise SM, Cristini V (2010) Nonlinear modelling of cancer: bridging the gap between cells and tumours. Nonlinearity 23: R1–R91

Luo L, Wong M, Hwu W (2010) An effective implementation of breadth-first search. In: Proceedings of the 47th design automation conference, 2010

Mansury Y, Kimura M, Lobo J, Deisboeck TS (2002) Emerging patterns in tumor systems: simulating the dynamics of multicellular clusters with an agent-based spatial agglomeration model. J Theor Biol 219:343–370

Matsumoto K, Nakamura T, Sakai K, Nakamura T (2008) Hepatocyte growth factor and Met in tumor biology and therapeutic approach with NK4. Proteomics 8:3360–3370

Milde F, Bergdorf M, Koumoutsakos PA (2008) Hybrid model of sprouting angiogenesis. Lect Notes Comput Sc 5102:167–176

Moreira J, Deutsch A (2002) Cellular automaton models of tumor development: a critical review. Adv Complex Syst 5:247–269

Newman MEJ (2003) The structure and function of complex networks. SIAM Rev 45(2):167–256

Parish CR (2003) Cancer immunotherapy: the past, the present and the future. Immunol Cell Biol 81:106–113

Preziozi L (ed) (2003) Cancer modelling and simulation. Chapman & Hall/ CRC Mathematical Biology & Medicine, London, New York, Washington DC, p 426

Stéphanou A, McDougall SR, Anderson ARA, Chaplain MAJ, Sherratt JA (2005) Mathematical modelling of flow in 2D and 3D vascular networks: applications to antiangiogenic and chemotherapeutic drug strategies. J Math Comput Model 41:1137–1156

Stokes CL, Lauffenburger DA (1991) Analysis of the roles of microvessel endothelial cell random motility and chemotaxis in angiogenesis. J Theor Biol 152:377–403

Stolarska MA, Kim Y, Othmer HG (2009) Multiscale models of cell and tissue dynamics. Phil Trans R Soc A 367:3525–3553

Szczerba D, Lloyd BA, Bajka M, Szekely GA (2008) Multiphysics model of Myoma growth. Lect Notes Comput Sc 5102:187–196

Topa P, Dzwinel W, Yuen DA (2006) A multiscale cellular automata model for simulating complex transportation systems. Int J Modern Phys C 17(10):1437–1460

Topa P (2008) Dynamically reorganising vascular networks modelled using cellular automata approach. Lect Notes Comput Sc 5191:494–499

Topa P, Dzwinel W (2009) Using network descriptors for comparison of vascular systems created by tumor-induced angiogenesis. Theor Appl Inf 21(2):83–94

Tran QN (2010) Designing Efficient many-core parallel algorithms for all-pairs shortest-paths using CUDA. In: Proceedings of 2010 Seventh international conference on information technology, 2010

Vasilyev OV (2003) Solving multi-dimensional evolution problems with localized structures using second generation wavelets. Int J Compt Fluid Dyn (Special issue on High-resolution methods in Computational Fluid Dynamics) 17(2):151–168

Wcislo R, Dzwinel W (2008) Particle based model of tumor progression stimulated by the process of angiogenesis. Lect Notes Comput Sc, ICCS 2008. LNCS vol 5102:177–186

Wcislo R, Dzwinel W, Yuen DA, Dudek AZ (2009) A new model of tumor progression based on the concept of complex automata driven by particle dynamics. J Mol Mod 15(12):1517–1539

Wcislo R, Dzwinel W (2010) Particle model of tumor growth and its parallel implementation. Lect Notes Comp Sc, PPAM, LNCS, Wrocław, pp 322–331, 13–16 Sept 2009

Wcislo R, Gosztyla P, Dzwinel W (2010) N-body parallel model of tumor proliferation. In: Proceedings of SCS summer computer simulation conference 2010, Ottawa, Canada, 11–14 July 2010

Chapter 39
High Throughput Heterogeneous Computing and Interactive Visualization on a Desktop Supercomputer

S. Zhang, R. Weiss, S. Wang, G. A. Barnett Jr. and D. A. Yuen

Abstract At a cost below $2500, a desktop supercomputer was built from scratch by assembling the basic parts including a Tesla C1060 card and a GeForce GTX 295 card. This commodity desktop runs a Linux operating system together with CUDA, MPI and other needed software. MPI is used not only for distributing and/or transferring the computing loads among the GPU devices, but also for controlling the process of visualization. Several applications of heterogeneous computing have been successfully run on this desktop. Calculation of long-ranged forces in the n-body problem with fast multi-pole method can consume more than 85 % of the cycles and generate 480 GFLOPS of throughput. Mixed programming of CUDA-based C and Matlab has facilitated interactive visualization during simulations. One such MIMD application is the simulation of an idealized Belousov-Zhabotinsky Reaction (BZR), which is distributed evenly on three GPU devices (two on GTX 295 and one on Tesla) through message passing interface (MPI) and visualized at a given frequency displaying the evolution of the simulated reaction. One additional MPI process is over-subscribed onto one GPU device for monitoring the thermal status and memory usage of all the GPU devices as the BZR simulation progresses, further enhancing the throughput. (Submitted as a part of the paper is a movie capturing the self-organization process of cellular spirals resembling the Belousov-Zhabotinsky Reaction.) Our test runs have shown that running multiple applications on one GPU device or running one application across multiple GPU devices can be done as conveniently as on traditional CPUs.

Keywords CUDA · SIMD · MIMD · Matlab · MPI · Heterogeneous computing

S. Zhang (✉) · D. A. Yuen
Minnesota Supercomputing Institute, University of Minnesota, Minneapolis, USA

R. Weiss · S. Wang · D. A. Yuen
Department of Geology and Geophysics, University of Minnesota, Minneapolis, USA

G. A. Barnett Jr.
Department of Applied Mathematics, University of Colorado, Boulder, USA

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 639
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_39,

39.1 Introduction

On the 15th of February 2007, nVIDIA Corporation released to the public the initial version of Compute United Device Architecture (CUDA) (Wikipedia—CUDA), starting the successful era of accelerating general-purpose computation by using the computational resources available on the Graphic Processing Unit (GPU). So far, many successful stories of superior performance accelerated by CUDA-based GPUs have been reported (Selected Publications by NVIDIA). At the annual Supercomputing conference, general scientific and engineering calculations on GPU or heterogeneous computing has become one of the key thrust areas (Conference overview of SC10). The GPU accelerator movement towards HPC has demonstrated the beginning of accelerator-based computing at a large scale. Many think accelerated HPC is the primary path to exascale computing for a fixed energy budget (SC10 focuses on a heterogeneous future).

One common feature of these successful GPU accelerations is that they are being developed on nearly turn-key machines built by professional vendors. Many of the publications focus on how to accelerate the calculation of a particular computing (Takashi et al. 2010). Few papers, however, discuss from the operational perspective what is needed to enable GPU computing or how to make it work at little additional cost, or to use an already CUDA-compatible GPU for computing. Individual pieces of information can be found easily here and there on the internet, for example, about CUDA GPU computing, but it is difficult to find one that has put the pieces of information together to help beginners to start quickly, which motivate us to write this paper.

Our experience with GPU computing was started by a undergraduate student S. Wang in May of 2009. Subject to a given budget of $2500, he ordered the basic parts, assembled them into a desktop, installed the needed software and got the desktop ready for GPU computing. So far we have built three such desktops made of different generations of the GPU cards. They are named MARK1, MARK2 and MARK3, together forming a small GPU cluster. Multiple GPU systems, such as the three Marks, which we have constructed in the past 18 months are regarded as being the high performance standard because it is cost-efficient and practical. GPU-based computers allow for high performance computing to be taken out of the data center. With the GPU-based approach to computing, the performance of 1999 era cluster systems, such as the SGI 2000 Origin, can be achieved at the deskside as a personal supercomputer. This allows the immense power and performance capabilities of a high-performance cluster to be merged with the portability and usability of a workstation to enable a new era of personal innovation in strategic science, research, development and visualization.

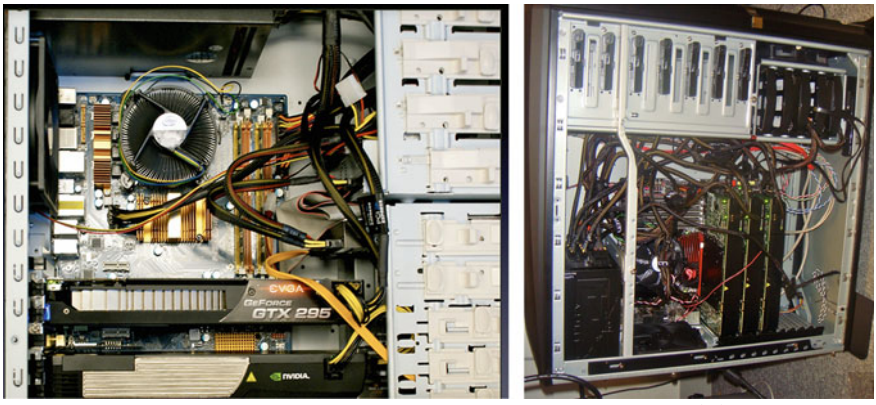
Our work distinguishes from most of the published ones by the fact that multiple GPU devices on one desktop are used by multiple users for different kinds of applications concurrently. The GPU acceleration has gone beyond the single program multiple data (SPMD)—the dominant paradigm of GPU computing (Takashi et al. 2010). We have extended it to the applications of multiple program multiple data (MPMD), requiring several pieces of software running together on the GPU devices

(Wang et al. 2009). One objective of our work is to build a more—user friendly GPU computing environment on a single desktop and enrich and/or enhance the capabilities of GPU so that users can develop a variety of applications that can benefit from the GPU acceleration.

In this paper, we would like to share our experience from an operational perspective regarding how to build a user friendly GPU computing environment, starting from scratch by assembling hardware parts and integrating individual pieces of supporting software and how to efficiently use the available resources on the devices and maximize its throughput. Certainly, the technical tactics described in this paper may not be directly applicable to general desktops because of MARK1's specific GPU settings, but the concepts and strategies embedded in the technical tactics, we believe, are generic and useful for people to enhance the usability of his/her already existing workstation or even laptop. The tools and the programs we developed can be ported easily to different GPU settings.

39.2 Hardware

The major pieces of hardware on MARK1 (Fig. 39.1a) consist of one quad-core CPU (Intel Core 2 Quad Q6600 Kentsfield 2.4 GHz), a motherboard with 2 PCIe slots (2.0×16 expansion), two GPU cards (nVidia's GeForce GTX 295 and Tesla C1060), and a 1200 W power supply. Not much difference in assembling MARK1 was found from building a normal PC desktop except the end product has more GPU devices and requires a bigger power supply (Fig. 39.1). Certainly, the assembling is a very rewarding experience because it helps understand the parts, customizes the machine and saves some money. Table 39.1 gives a breakdown of the costs of individual parts.



(a) **(b)**

Fig. 39.1 **a** Photo of MARK1 assembled from individual parts. **b** Photo of MARK3 made of 3 Fermi GPUs C-2070 and C-2050 in addition to a 8 core CPU

Table 39.1 Cost breakdown to individual parts

Part description	Cost(US $)
CPU—Intel C2Q Q6600 2.4 GHz	194.99
MB—ASROCK P45XE-WiFiN 775	124.99
CASE—COOLERMASTER	49.99
RAM—Kingston 2×1 Gb & 2×512 Mb	97.99
PSU—BFG 1200W	249.99
CASE FAN—Generic 120 mm	14.99
CPU FAN—Generic Socket 775	14.99
GPU—GTX 295 Card (1.9 GB Ram)	509.60
GPU—Tesla C1060 Card (4 GB Ram)	1219.99
Total	$2477.51

MARK3 shown in Fig. 39.1b is a high end desktop at a cost around $8200. This cost does not include the 40 Gbits/s Infiniband fabric which is used to connect the three individual desktops. MARK3 is assembled from the following parts: Asus Rampage III Extreme motherboard, Intel i7 950 CPU @ 3.07 GHz, and 12 GB DDR3 RAM, and 3TB of RAID storage. The GPUs in this computer are two Tesla C2070's and one Tesla C2050. MARK2 is in between MARK1 and MARK3, or a middle-end desktop, made of a CPU (Intel i7 920 @ 2.66 GHz processor with 4 cores) and two GPU devices (Tesla 2050 and GTX 480). MARK2 and MARK3 will be connected by the Infiniband with 40 Gbit/s capability.

39.3 Supporting Software

MARK1 runs the openSUSE Linux operating. We have used NVIDIA's CUDA (Wikipedia—CUDA) technology to enable the GPU computing. In Table 39.2 we list the major pieces of software installed on MARK1 for a variety of applications.

39.4 Technical Tactics

39.4.1 Selection of GPU Devices

CUDA provides device management functions (see Table 39.3) that can be used for checking how many GPU devices are available, what the ID number is for each of the devices, what physical properties are associated with each of the device IDs, and for selecting a device subject to a certain criterion.

These functions provide the programmers with the useful means to flexibly select one of the GPU devices to meet a particular need of different applications in the multi-

Table 39.2 Major piece of software installed for CPU-GPU hybrid computing

Software	Version	Description
OpenSUSE	11.1	Linux operation system
GNU compiler	4.3.2	The compiler for CUDA and MPI application
CUDA toolkit	2.2	Parallel computing architecture that
CUBLASCUF	2.3	leverages the parallel compute engine in
FT	3.0	NVIDIA graphics processing units (GPUs)
computeprof	3.2	to solve computational problems
MPICH	2-1.1	Message Passing Interface (MPI) for the use of multi-GPUs and MIMD applications.
MAGMA	0.2s	Matrix Algebra on GPU and Multicore Architectures.
Lapack & BLAS	3.2.1	Matrix algebra library on CPU used for benchmarking GPU performance with MAGMA
MATLAB	R2009a	For interactive visualization
nvidia-smi	1.1	System Management Interface program

Table 39.3 Device management functions with CUDA

Function name	Functionality
cudaGetDeviceCount(&device_count);	To tell how many GPU devices are available
0 ≤ ID < device_count	The IDs of the available GPU devices
cudaGetDeviceProperties(&deviceProp, ID);	To get the device properties for a given ID
cudaSetDevice(ID).	To use the device associated with ID
cudaChooseDevice **(ID, &deviceProp)**	To chose a device having the device properties

GPU environment. That is, one can run multiple applications on the same device (e.g., Tesla C1060) and one can also let one application run across multiple devices. The flexibility of selecting GPU devices will be better demonstrated in Sect. 39.4.3 with MPI through an MPMD application.

39.4.2 Memory Management

The typical CPU-GPU heterogeneous computing involves running code on two different platforms concurrently: a *host* system with one or more CPUs and one or more *devices* with CUDA-enabled NVIDIA GPUS. CUDA C best practices guide (Cuda, C Best Practices Guide) recommends the amount of memory on the *host* system matches that of the GPU devices to maximize the performance and throughput. But there is a shortcoming on MARK1 due to the budget constraint. The main memory on the CPU part is 3 GB whereas 5.8 GB of memory is available on the GPU side (4 GB on Tesla C1060 and 1.8 GB on GTX 295).

```
  …
 cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice) );
  cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice) )
  …
// setup execution parameters
   dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
   dim3 grid(WC / threads.x, HC / threads.y);

// free the memory on CPU
   free(h_A);
   free(h_B);

// execute the kernel
   for (res=0; res <=1; res++) {
   matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
   }

// allocate memory on CPU for the output from GPU
   float* h_C = (float*) malloc(mem_size_C);

// copy result from device to host
   cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);
…
```

Fig. 39.2 Typical procedure for efficient use of the available memory on the host system

We have used the strategy in the code development that frees the CPU memory allocated for the array initialization after the data are copied onto device, but before the GPU kernel executes and keeps the data on the GPUs as long as possible. The memory needed for receiving the output from GPU computing will not be allocated on CPU until the GPU kernel execution finishes. This strategy works well in reducing the impact of CPU memory shortage on the performance and maximizing the throughput of the desktop. The following chart (Fig. 39.2) shows a piece of the code to demonstrate how to efficiently use the available memory on the host system so that all the GPUs can be fully used.

39.4.3 Use of nvclock for Monitoring the Thermal Status of GPU Devices

In the multi-GPU and multi-user environment, more often it is necessary to know which GPU has been used and/or how much of its memory has been used before launching a new job to balance the computing loads. Also, it is necessary to check whether a job is actually running on the selected GPU device. However, there were no direct ways to get this kind of information on Linux systems before November of 2010 when CUDA toolkit 3.2 was released. We found that the utility ***nvclock*** provides a means to check the temperature of all GPU devices and that their thermal status is sensitive to the relative intensiveness of individual device usage.

Then we developed a tool that can keep track the thermal status of all GPU devices and report the temperature variations in graphic format indicating which device is idle or busy and/or show how the thermal status of a particular device corresponds to the job running status. This tool has two independent graphic interfaces. One interface is written in java-script and uses html to report thermal status of GPU devices to a web browser allowing remote users to check the current GPU's usages. One example of the online reporting is shown in the left top panel in Fig. 39.5. The other interface is written in Matlab, which can be coupled into a real application code for the local users to check the thermal status in the real simulation time. The temperature plots generated by Matlab are shown in the left lower panel in Fig. 39.5.

It should be noted that as of the release of CUDA 3.2 the **nvidia-smi** utility is better used to gain access to relevant GPU statistics. By calling the nvidia-smi executable with –q –a switches, many valuable GPU statistics can be accessed. This includes temperature, GPU and Memory utilization percentages, and ECC memory statistics.

39.4.4 Use of All GPU Devices for One MPMD Application with MPI

As described in previous sections, MARK1 is equipped with two different GPU cards, whose memory is not the same as the CPU memory. These heterogeneous characteristics make MARK1 well suitable for MPMD applications that can benefit from the use of all GPU devices. In this section, we would like to describe the implementation of a Belousov-Zhabotinsky Reaction (BZR) model on MARK1 and show how we maximize the throughput.

BZR is a now well-known chemical reaction that was discovered in the 1950s by Boris Belousov and is a demonstration of an oscillating chemical reaction (Winfree 1984). While the details of the chemistry behind this reaction are most certainly beyond the scope of this paper, an unusual and interesting feature of the reaction is that as it progresses on a two dimensional plate, self-organizing spirals are formed. Many computer models have been constructed to simulate the evolution of these spirals. In 1988, Alexander Dewdney proposed a transition rule for a cellular automaton, which is summarized in Fig. 39.3. The resultant patterns of this cellular automaton closely match the observed ones from actual BZ reactions (Turner 2009).

Robin Weiss implemented Dewdney's transition rule in CUDA C as BZR.cu. The reaction region is divided into three subdomains, each of which is distributed onto one GPU device for calculating the local cellular automaton. Correspondingly, three MPI processes have been used. Each of them controls one of the GPU devices by attaching one MPI rank to one GPU device ID. MPI_Send and MPI_Recv functions are used to move the necessary data among the GPUs and CPUs. Two critical issues must be resolved before claiming the success of parallel cellular automaton:

```
All cells begin in random state between 0 and maxState
1. let k1, k2, and g be constants
2. if currState == maxState
3. nextState = 0
4. else
5. a = # of neighbors where: (0 < neighbor.value < maxState)
6. b = # of neighbors where: (neighbor.value == maxState)
7. if currState == 0
8. nextState = floor(a/k1) + floor(a/k2)
9. else
10. s = sum of all neighbor states
11. nextState = floor(s / (a + b + 1)) + g
```

Fig. 39.3 Pseudocode proposed by Dewdney

(1) The spatial continuity of the cellular structures across the edge connecting two subdomains. A local-grid (or cell on the edge) needs to access the current state of abutting cells that are located on a different MPI process.
(2) The temporal continuity of the cellular patterns and evolution. A careful design is needed to handle the race condition that can occur as multiple (over 560,000) threads attempt to update the local grid with the nextState values of particular cells. If one of the threads terminates the update for one cell, the computation of neighboring cells will be invalid.

To maintain the spatial continuity of the cellular structures, at the beginning of every iteration, each MPI process retrieves the top- and bottom-most row of cells from their local grid. These two rows are then sent via MPI Send/Recv functions to the immediate neighbor "above" and "below" the process. Upon receipt of these rows, the MPI process loads the data onto the GPU so that the GPU kernel function can access the values.

A buffer is used to handle the race condition. Essentially, each MPI process allocates twice the amount of memory needed for the local grid (or cells) on the GPU. One of these locations contains the current state of the local grid, call it currGrid, and the other is used to collect results, the buffer. When the kernel function (BZR.cu) is called to compute the next state of the cells in the local grid, the threads computing these values write their results to the buffer. Then, at the beginning of next iteration, the memory location holding the buffer is considered the currGrid, and the memory location that was holding currGrid is considered the buffer, having its values overwritten in this iteration.

The overall procedure of performing the idealized BZR simulation shown in Fig. 39.4 manifests well the suitability of MARK1 for the MPMD application, in which three GPU devices are used for computing and one of them picks up additional work of visualization. This overloaded MPI process is assigned onto the Tesla C1060 device because it has much more memory than the other two. Furthermore, one additional MPI process in the BZR simulation is over-subscribed onto one GPU device (the Tesla C1060), which couples the tool described in Sect. 39.4.3 for monitoring the thermal status of all the GPU devices during the BZR simulation.

Each MPI Process:

1. Allocate space on GPU for currGrid and buffer
2. Initialize buffer with random values
3. Repeat for temporal evolution
    1). Set currGrid = buffer
    2). Copy top and bottom rows of currGird off of the GPU and send to adjacent process
    3). Receive above- and below-rows and load them onto GPU device
    4). Call the BZR.cu to compute the next state of localGrid, writing results into the buffer
    5). (Optional) at a given frequency, one MPI process collects the cellular state and displays the results over the whole region.

Fig. 39.4 High-level pseudocode for distributed cellular automaton

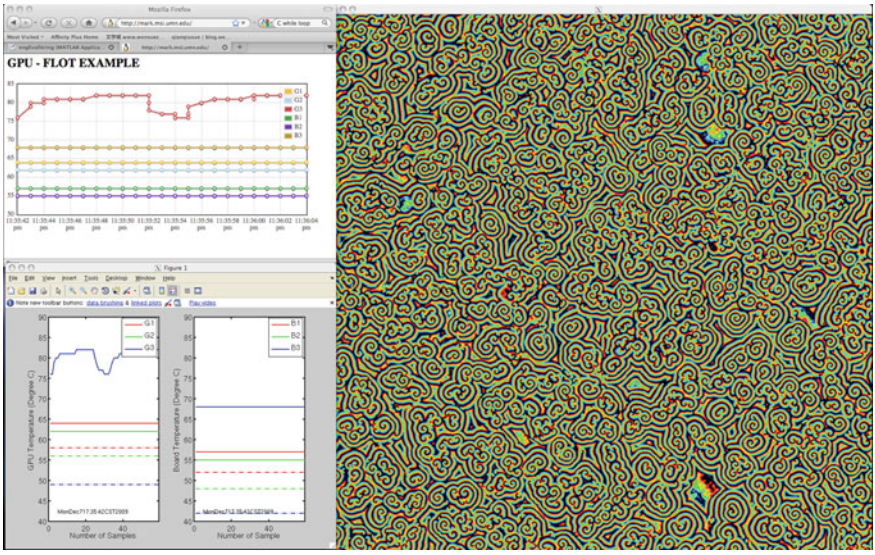


Fig. 39.5 A snapshot of graphics appearing on MARK1's monitor

Figure 39.5 is one snapshot of the graphics on MARK1's monitor when the BZR simulation was on. The right panel shows a state of cellular automata over the whole reaction region. The left upper panel shows the temperature plots generated on the web browser. The left lower panel shows the temperature plots generated by Matlab. The legend G1, G2 and G3 stand for GPU devices 1, 2 and 3 respectively. The legend B1, B2 and B3 stand for boards on which GPU devices 1, 2 and 3 are respectively attached. The solid curves are the instant temperature reflecting the relative working load on individual devices. The dashed curves show the temperature when the devices are idle.

One can observe that GPU1 and GPU2 (the two devices on GTX 295) remain in more stable and cooler condition than that of GPU3. That is due to the fact that GPU3 or the Tesla card is assigned more work—showing the pattern evolution of the spirals and displaying the thermal status of all devices. It is an interesting observation that when the machine is idle, the temperature of Tesla card is 7 °C lower than that of GTX 295 although exploring the deep causes is beyond the scope of this paper.

39.5 Performance and Throughput

We have focused our attention towards maximizing the throughput of MARK1 through balancing the working loads among the CPU and GPUs. That requires a good understanding on the capabilities of each GPU device. We have got several applications running on this commodity desktop. In this section we will describe what we have learned and the performance achieved with different applications.

One experiment was done to compare the performance of individual GPU device for conducting a single precision matrix multiplication of $A \times B$ with A being a 9600×9600 array and B being a 9600×4800 array. As a reference we also conducted the same calculation with different OpenMP threads or CPU cores on a Intel(R) Xeon(R) E5345 workstation. The timing results summarized in Table 39.4 clearly show the superior acceleration from the GPU devices. When one OpenMP thread is used for running the matrix application takes 98 s of wall clock time. It shortens to 25 s with four OpenMP threads. The same calculation takes over 14 s on one GPU device with 150×300 thread blocks and 32×32 threads per block.

But GPU3 (or Tesla C1060) yields only a slight better performance than GPU1 or GPU2. That was a little disappointing at first as we expected much better performance from Tesla C1060 as it has much bigger memory. A careful comparison of the device properties depicted by the CUDA driver (Table 39.5) shows that Tesla C1060 has almost identical subsystems as GTX 295 except C1060 has a slightly higher clock rate, which account for their minor performance difference shown in Table 39.4.

Another experiment conducted was to concurrently run multiple copies of the same job as shown in Table 39.4 on Tesla C1060. The interesting finding is that the time taken to run the 3 copies of the same job concurrently is over 20 % shorter

Table 39.4 Performance comparison of matrix multiplication

	GPU1	GPU2	GPU3	CPU
Name	G295	G295	C1060	L13[a]
Clockrate (MHz)	1242	1242	1296	2327
Available memory (MB)	900	900	4096	16439
Used memory (MB)	737	737	737	737
Time on GPU (s)	9.92	9.93	9.49	0
System time (s)	2.48	2.44	2.48	0.48
Wall clock time (s)	14.48	14.48	14.04	25.42

[a] *Note* L13 is an Intel(R) Xeon(R) E5345 workstation. 4 OpenMP threads are used for this test

Table 39.5 Comparison of the subsystems between Tesla 1060 and GTX 295

Device properties	GPU1 (GTX 295)	GPU3 (Tesla 1060)
Number of multiprocessors	30	30
Number of cores:	240	240
Total amount of constant memory (B)	65536	65536
Total amount of shared memory per block (B)	16384	16384
Total number of registers available per block	16384	16384
Warp size:	32	32
Maximum number of threads per block	512	512
Maximum sizes of each dimension of a block	512 × 512 × 64	512 × 512 × 64
Maximum sizes of each dimension of a grid	65535 × 65535 × 1	65535 × 65535 × 1
Maximum memory pitch (B)	262144	262144
Texture alignment (B)	256	256
Clock rate (GHz)	1.24	1.30
Concurrent copy and execution	Yes	Yes
Support host page-locked memory mapping	Yes	Yes

than that taken to run the job three times one after another. What so? Very likely the superior performance is from the contribution of the big number of cores combined with the big amount of memory that Tesla C1060 has, which play the role toward the throughput in a similar way as the cache on CPU does. Although the actual mechanism is not clear, this test indicates that running more applications on the GPU device subject to its available memory may improve the overall throughput.

Matthew Knepley has got his code running on MARK1, which does the calculation of long-ranged forces in the n-body problem with fast multipole method (FMM). The FMM simulation consumed 85 % of the cycles of Tesla C1060 or yielded 480 GFLOPS of throughput (Knepley 2009).

Scientific computation more often relies on other basic capabilities including supportive libraries. We have ported MAGMA—the LPACK library developed at the University of Tennessee for GPU and Multicore Architectures (LAPACK for GPUs and Multicore architectures) and made it working on MARK1. The following chart (Fig. 39.6) shows three GPU devices compute a QR factorization of a matrix of 6010 in double precision.

```
./testing_dgeqrf -N 6016
device 0: GeForce GTX 295, 1242.0 MHz clock, 895.8 MB memory
device 1: GeForce GTX 295, 1242.0 MHz clock, 895.7 MB memory
device 2: Tesla C1060, 1296.0 MHz clock, 4095.8 MB memory

N          CPU GFlop/s          GPU GFlop/s          ||R||_F  /  ||A||_F
=========================================================================
6016               1.14                44.70          3.674079e-15
```

Fig. 39.6 Performance of QR factorization in double precision for a matrix of 6010 × 6010 achieved with MAGMA

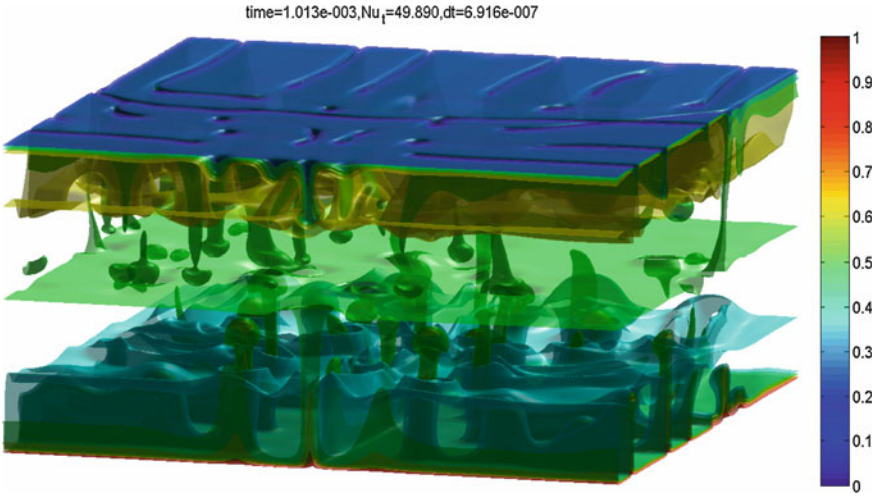


Fig. 39.7 Boss transition with $600 \times 600 \times 300$ points, 2nd order method, Ra =10**7, 5 s/time-step using the Tesla C1060 GPU

G. Barnett Jr. developed a Matlab mantle convection code which implements a three-dimensional, second-order finite-difference model of Rayleigh-Benard thermal convection. We have modified this code and accelerated it with a GPU through Jacket (Jacket—The GPU Acceleration Engine for Matlab). On the Telsa C1060 we ran the simulation with sufficient resolution ($600 \times 600 \times 300$ gridpoints). $5\times$ speedup was achieved. One snapshot of the simulation results is shown in Fig. 39.7.

39.6 Concluding Remarks and Future Work

Although we have built up two more workstations (MARK2 and MARK3) with newer NVIDIA GPU cards such as the Fermi C-2070, these machines start to increase in price to over eight thousand dollars, which could not be easily classified as cost-effective supercomputing. In this paper we have focused our presentation on GPU computing with MARK1 because of the originality of building MARK1. The lessons learned from this pioneering effort and the accomplishment carried out on the first desktop have made the work of assembling the following two much easier. Certainly, our interest of building MARK2 and MARK3 is not just simply to repeat MARK1, but to meet the new challenges of using multiple GPUs and connecting Mark-2 and Mark-3 with a 40 Gb/s Infiniband. They will allow us to get on to the newer NSF centers in Tennessee, the Keeneland project (Keeneland Success at SC10), which will house soon 500 Fermis and will have a peak speed close to a Petaflop.[1]

[1] What does this sentence mean? We are going to be part of the Keeneland project? Or are we saying that we have a competitive system?

Our work itself manifests a great success with multi-GPU computing starting from scratch by assembling parts. We have enabled the simulation of mantle convection for Rayleigh number of 10**7 on Tesla C1060, which shows a 5× speed up compared the performance on CPU with Matlab. We have achieved a throughput of 480 Gflops with the FMM code in solving the N-Body problem. In the implementation of BZR simulation, we have used MPI for evenly distributing computing work and exchanging the data among three GPU devices. The BZR application clearly shows the tremendous potential embedded in MARK1—the commodity desktop. It can not only yield high throughput with GPU computing, but also visualize its results during the simulation. In addition it also keeps track of and displays the thermal status of all GPU devices corresponding to their usage by the simulation.

Our work also showcases the usefulness of CUDA in managing and connecting GPUs for various applications. With CUDA one can selectively select one of the GPU devices for computing and visualization. This functionality is especially useful for matlab-based visualization to run on Tesla, which has more memory, because Matlab itself can start only on the GPU card located on the first slot (GTX 295). The interesting finding is that multiple graphic applications can run simultaneously on one GPU device (we put them on Tesla because it has 4 GB memory). Our experience has shown that running multiple (including graphic) applications on one GPU device (Tesla C1060) or running one application across multiple GPU devices can be done as conveniently as on CPUs, but their performance can be 5–10 times or even better.

Future work would involve integration with Intel many-core processors such as the Sandybridge from Intel and Knights Ferry and Knights Corner. A hybrid setup of Nvidia GPUs and new Intel chips would make a powerful combination. A lot of this integration would depend on the costs of these new many-core chips, capable of using traditional compilers, such as Fortran and C.

Acknowledgments We want to thank National Science Foundation for the CMG grant and the Vlab grant.

References

Conference overview of SC10. http://sc10.supercomputing.org/?pg=conference.html

Cuda C best practices guide. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

Jacket-The GPU acceleration engine for Matlab. http://www.omatrix.com/jacket.html

Keeneland success at SC10. http://keeneland.gatech.edu

Knepley M (2009) Understanding the performance of the fast multipole method (FMM) on a GPU, SC09 presentation at MSI's booth. http://static.msi.umn.edu/curtain/docs/MSISC09PresentationSchedule.pdf

LAPACK for GPUs and multicore architectures. http://icl.cs.utk.edu/magma/

SC10 focuses on a heterogeneous future. http://insidehpc.com/2010/11/05/sc10-focuses-on-a-heterogeneous-future

Selected Publications by NVIDIA. http://research.nvidia.com/publications

Shimokawabe T, Auki T, Muroi C, Ishida J, Kawano K, Endo T, Nukada A, Maruyama N, Matsuoka S (2010) An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In: Proceedings of the 2010 ACM/IEEE conference on supercomputing (SC'10), New Orleans.

Turner A (2009) A simple model of the Belousov-Zhabotinsky reaction from first principles. ScientificCmmons, http://en.scientificcommons.org/50894615

Wang S, Zhang S, Weiss RM, Barnett GA, Yuen DA (2009) Commodity CPU-GPU system for low-cost. High Perform Comput 90:52

Wikipedia-CUDA. http://en.wikipedia.org/wiki/CUDA

Winfree AT (1984) The prehistory of the Belousov-Zhabotinsky oscillator. J Chem Educ 61:661–663

Chapter 40
Applications of Microtomography to Multiscale System Dynamics: Visualisation, Characterisation and High Performance Computation

Jie Liu, Klaus Regenauer-Lieb, Chris Hines, Shuxia Zhang, Paul Bourke, Florian Fusseis and David A. Yuen

Abstract We characterise microstructure over multiple spatial scales for different samples utilising a workflow that combines microtomography with computational analysis. High-resolution microtomographic data are acquired by desktop and synchrotron X-ray tomography. In some recent 4-dimensional experiments,

J. Liu (✉) · K. Regenauer-Lieb · F. Fusseis
Multi-scale Earth System Dynamics Group, School of Earth and Environment,
University of Western Australia, 35 Stirling Hwy, Crawley,
WA 6009, Australia
e-mail: liujieigcea@gmail.com

K. Regenauer-Lieb
Computational Geoscience, CSIRO ESRE, PO Box 1130, Bentley,
WA 6102, Australia

K. Regenauer-Lieb · F. Fusseis
Western Australian Geothermal Center of Excellence, PO Box 1130, Bentley,
WA 6102, Australia

C. Hines
iVEC, 'The hub of advanced computing in WA', PO Box 1130, Bentley,
WA 6102, Australia

S. Zhang
Supercomputing Institute, University of Minnesota, Minneapolis, MN 55455, USA

P. Bourke
iVEC @ University of Western Australia, 35 Stirling Hwy, Crawley,
WA 6009, Australia

D. A. Yuen
Department of Geology and Geophysics, University of Minnesota,
Minneapolis, MN 55455, USA

D. A. Yuen
Department of Earth Sciences and Minnesota Supercomputing Institute,
University of Minnesota, Minneapolis, MN 55455, USA

D. A. Yuen
School of Environmental Sciences, China University of Geosciences, Wuhan, China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 653
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_40,

microstructures that are evolving with time are produced and documented in situ.The microstructures in our materials are characterised by a numerical routine based on percolation theory. In a pre-processing step, the material of interest is segmented from the tomographic data. The analytical approach can be applied to any feature that can be segmented. We characterise a microstructure by its volume fraction, the specific surface area, the connectivity (percolation) and the anisotropy of the microstructure. Furthermore, properties such as permeability and elastic parameters can be calculated. By using the moving window method, scale-dependent properties are obtained and the size of representative volume element (RVE) is determined. The fractal dimension of particular microstructural configurations is estimated by relating the number of particular features to their normalized size. The critical exponent of correlation length can be derived from the probability of percolation of the microstructure. With these two independent parameters, all other critical exponents are determined leading to scaling laws for the specific microstructure. These are used to upscale the microstructural model and properties. Visualisation is one of the essential tools when performing characterisation. The high performance computations behind these characterisations include: (1) the Hoshen-Kopelman algorithm for labelling materials in large datasets; (2) the OpenMP parallelisation of the moving window method and the performance of stochastic analysis (up to 640^3 voxels); (3) the MPI parallelisation of the moving window method and the performance of stochastic analysis, which enables the computation to be run on distributed memory machines and employ massive parallelism; (4) the parallelised MPI version of the Hoshen-Kopelman algorithm and the moving window method, which allows datasets of theoretically unlimited size to be analysed.

Keywords Microtomography · Percolation theory · Hoshen-Kopelman algorithm · Multi-scale system · Quantitative analysis · OpenMP parallelisation · MPI parallelisation

40.1 Introduction

Micro computed tomography (micro-CT or microtomography) is a technique developed from medical CT scans for detecting structures in the interior of objects with submicron resolution. It has three important characteristics: (1) providing digital information on 3-D geometries; (2) no destruction of the samples; and (3) offering high resolution. These advantages motivate a wide range of applications of microtomography for different materials, including rock, bone, ceramic, metal and soft tissue. When the resolution of microtomography reaches the nano-meter scale, it is also known as nano-CT or nanotomography.

3-D digital microtomographic data are comprised of a stack of greyscale images which can be loaded into volume visualisation software packages. Although micro-CT and visualisation have already provided much qualitative information of the interior of the object, quantitative analysis can provide additional insights. For example,

it can give answers to questions such as how large individual entities are and what are the relationships between different entities. In addition to the characterisation of individual entities, we are also interested in the statistical characteristics of the sample. From statistical analysis of the sample a representative volume element (RVE) can be determined. RVE's are statistically representative volumes containing a sufficiently large set of microstructure elements such that their influence on the average macroscopic property (porosity, elasticity, permeability, etc.) has converged when a larger volume element is taken. It is important to specify the size of RVE when estimating the physical properties based on a structured volume. Furthermore, some scaling parameters can be extracted from the microstructures and this makes it possible to upscale the properties that were computed at micro-scale. Thus, the quantitative analysis of microtomography is performed in three levels: characterisation of individual entities, statistical characteristics and determination of RVE size, and extraction of scaling parameters for upscaling.

A quantitative analysis requires high performance computing for three reasons. The first reason is the size of datasets. At present a general micro-CT dataset is 2048×2048 pixels per slice, and there are around 2048 slices, this corresponds to 8GB if every pixel is represented by 1 byte. Moving to higher resolutions, 4096^3 volumes are now possible and it is expected that 8192^3 volumes will be generated in the near future. Without high performance computing techniques it is difficult to deal with datasets of this scale simply because of the memory required to represent the data. The second issue is that the computation of statistical properties often requires significant processing power. Thirdly, CT scan experiments in a time series increase both the volume of data and the computational requirements linearly with the number of time steps.

Software does exist to perform quantitative microstructural analysis (Lindquist 2005; Nakashima and Kamiya 2007; Ketcham 2005). However, concerning the stochastic analysis, high performance computing, and extraction of scaling parameters, our strategy (Liu et al. 2009; Liu and Regenauer-Lieb 2010) is unique, more comprehensive, and more powerful. The aims of this research are (1) characterisation of microstructures by visualisation and quantitative analysis; and (2) upscaling of micro-scale properties by stochastic analysis and percolation theory. In this paper we emphasise the technical details and the computational methods employed.

40.2 Microtomographic Data

40.2.1 X-Ray CT Theory and Synchrotron CT

In micro-CT experiments, X-rays are created from a high-power source and directed at a sample. A detector on the opposite side of the sample measures the intensity of the transmitted X-rays. The sample is rotated step by step, generally in 1° steps, and the detector records an image for every step. The intensity of the X-rays reaching

the detector depends on the sample path length and the X-ray attenuation coefficient of the material that it passes though. The longer the path length and the greater the attenuation coefficient of the material, the greater the number of diffraction and scattering events, thereby weakening the X-ray signal reaching the detector. Using reconstruction algorithms (i.e. Fourier transform), the 2-D images are converted to a complete 3-D map of the sample.

The digital 3-D map of the sample is generally represented by a stack of slices, each a greyscale image. The brightness of pixels within the images is related to the atomic number, regions with higher atomic numbers will be brighter than regions with lower atomic numbers. Pores in the sample are always shown as the darkest regions and are the easiest to identify. Different minerals in rock samples are recognisable as long as there is a significant difference in their atomic number.

The principle of synchrotron CT is similar to the general X-ray CT, the difference lies in the energy source employed. The synchrotron uses the energy from a particle accelerator in which the magnetic field and the electric field are carefully synchronised with the travelling particle beam. Generally, synchrotron CT results in better image quality and fewer artifacts compared to an ordinary X-ray CT. The other feature of synchrotron CT is that it results in images with a higher dynamic range. For example, pixels in X-ray CT image are generally 8-bit quantities, whereas pixels in synchrotron CT image are 32-bit values. Higher dynamic range means regions with smaller differences in atomic number can be identified.

40.2.2 Pre-Processing of Data

Before quantitative analysis, the 3-D microtomographic data needs pre-processing to select and label the phase of interest. This target phase can be pores, solid grains or any kind of material, which can be identified within the volume by its intensity value. We use the Avizo® software package to load the images. By choosing a suitable threshold value, the greyscale images are converted to binary images where the black and white pixels represent different materials, e.g., pores and solids in porous media. This is referred to as the "segmentation procedure". The Avizo® software enables us to build up the 3-D binary model, to display the structures in pores or in solids, and to export the label-data. The label-data is a binary volumetric data file in which each voxel that has a value lower than the threshold is labelled as 0 and those greater than or equal to the threshold are labelled as 1. This label-data volume is used for all subsequent analysis.

Very often the sample is cylindrical but sometimes they are a more irregular shape. Our quantitative analysis requires a parallelepiped, thus it is necessary to crop a parallelepiped volume from the whole 3-D dataset. This additionally removes regions that contain non-material voxels. This procedure is also implemented within Avizo®.

It is also possible to execute the segmentation of the target phase from CT scan data independent of Avizo® software by parsing and processing the image slices

independently using software tools developed inhouse. We have designed an alternative way to pre-process the microtomographic data by (1) reading the greyscale values directly from image files; (2) cropping the proper parallelepiped by specifying the boundary position in 3-directions; and then (3) exporting the similar label-data by comparing the value of each voxel with a user selected threshold value. This pre-processing stage has the advantage of being able to be run on high performance computers. Thus it can deal with larger datasets than commercial products and is not constrained by license requirements.

40.2.3 Examples

In the following we present some examples of microtomography.

Figure 40.1 shows pictures of a strongly deformed rock (mylonite) sample. Figure 40.1a is the hand specimen and the position of one sub-sample is marked (a total of 54 sub-samples have been scanned). Figure 40.1b is one slice of the synchrotron CT scan from the marked sub-sample. Each slice has 2048×2048 pixels at a resolution of $1.3\,\mu\text{m}$. The shape is a square in this slice of the sub-sample and the corners are cut off by a circle. Non-material pixels are those that extend beyond the overlap of the square and the circle, these should not be included in the exported label-data that will subsequently be used for quantitative analysis. Using the naked eye, we can recognise tiny pores, light-grey grains, a medium-grey material, and a white mineral. These can be identified as the target phase and analysed quantitatively. Figure 40.1c is the 3-D binary model of pores in the volume consisting of $1300 \times 1300 \times 1550$ voxels. The visualisation of the pore-structure illustrates how the pores are distributed in space, they are irregularly distributed and small pores tend to be are grouped together.

Figure 40.2 is a gypsum sample that is heated from room temperature to 235 °C. The description of the experiment is given in Fusseis et al. (2010b). As it is a laboratory made gypsum sample, it does not contain inner structure such as grains and pores but it does have some impurities that appear as white specks in the images. Being heated, the dehydration of gypsum leads to the creation of cracks, and the cracks propagate from the perimeter to the center of the sample. Figure 40.3 shows the cracks of the final step in a volume of $400 \times 800 \times 300$ voxels. Separated cracks are shown in different colours. We can see that the structure of the cracks is very complex, with thin, curved, and discontinuous features. Through quantitative analysis we can parameterise the size, shape, and orientation of each crack.

Figure 40.4 is an example of a tree branch. A portion of the CT scan is shown in Fig. 40.4a. The CT-scan slice is 2048×2048 pixels with a resolution of $2.88\,\mu\text{m}$. As can be seen in Fig. 40.4b there are a large number of pores, these are conduits of the tree branch. Figure 40.4c is a $288\,\mu\text{m}^3$ volume (100^3 voxels) and shows some conduits that are connected by sideways-connected cells (pit cells). Connected conduits are denoted by the same colour.

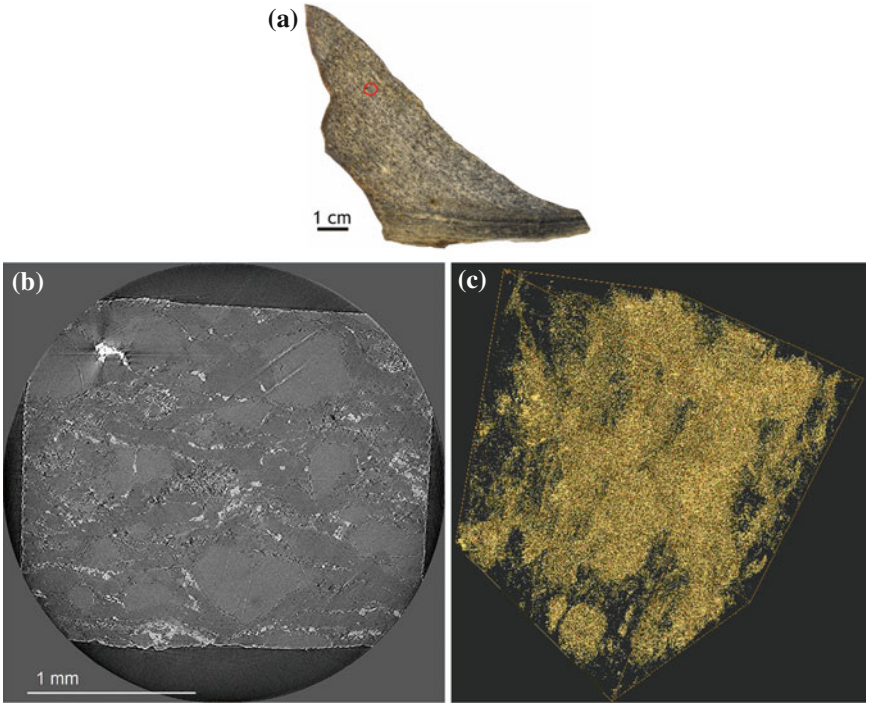


Fig. 40.1 Pictures of a strongly deformed rock (mylonite) sample, hand specimen and the position of a sub-sample (**a**), a slice of the sub-sample (**b**), and the 3-D pore-structure of $1300 \times 1300 \times 1550$ volume presented as an iso-surface using Avizo® (**c**)

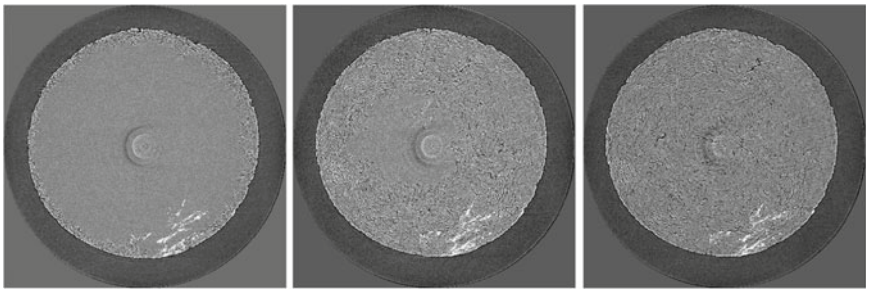


Fig. 40.2 A gypsum sample being heated, the same slice is shown at three time steps. Resolution is the same as in Fig. 40.1

40.3 Methodology

We use percolation theory for the analysis of microtomography. The mathematical definition of percolation refers to the nature of the connectivity in lattice models. Percolation theory was developed to analyse the connectivity of sites/bonds of

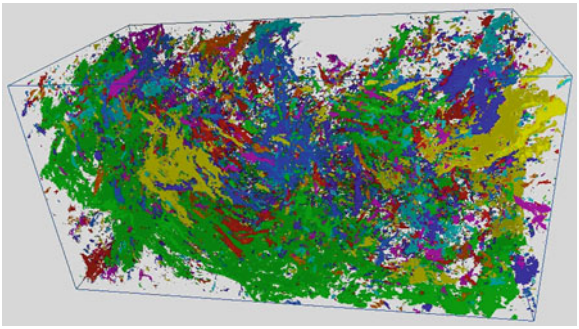


Fig. 40.3 3-D visualisation of dehydrated cracks of the gypsum sample in the final step of a volume of 400 × 800 × 300 voxels. Separated cracks are shown in different colours

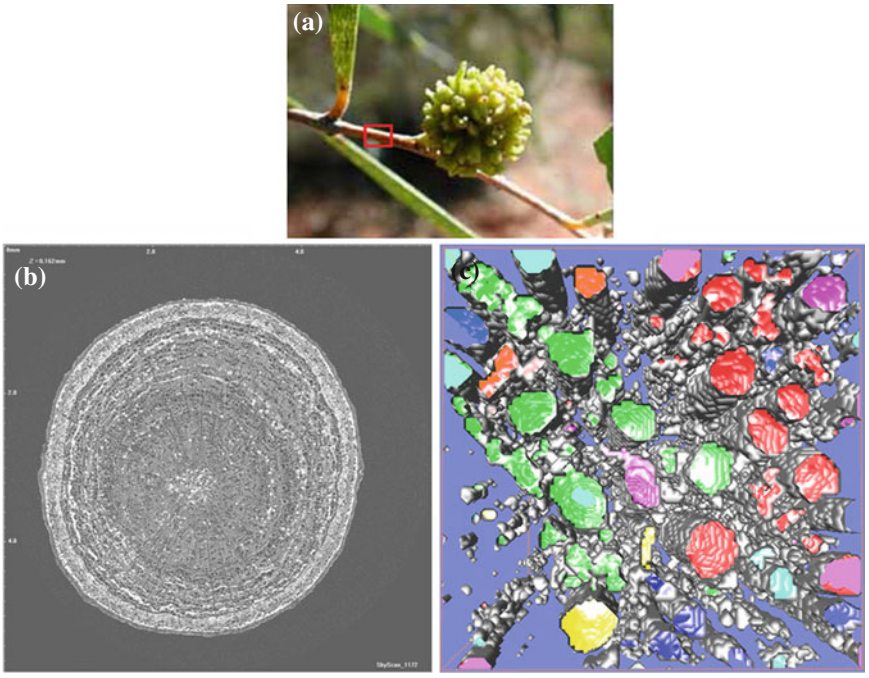


Fig. 40.4 A tree branch sample, tree specimen (**a**), a slice of CT-scan image in inverted colour (**b**), and a small volume showing conduits (**c**)

different lattice models, critical phenomena, and their related quantities (Stauffer and Aharony 1994). The 3-D binary model after segmentation can be considered as a simple cubic lattice model in which every cubic cell (or site) is equivalent to a voxel in a volumetric dataset.

40.3.1 Basic Output Parameters

The principal parameter of our quantitative analysis is the volume fraction of the target phase, which is, for example, porosity φ in porous media. It is defined as the ratio of the volume of target phase and the total (or bulk) volume. Both volumes can be determined from the label-data of the 3-D model. The porosity of a rock is, for instance, an important quantity for the evaluation of the potential volume of water or hydrocarbons it may contain.

The specific surface area (SSA) s_p is defined as,

$$s_p = S_p / V, \tag{40.1}$$

where S_p represents the surface area of target phase and V is the total volume. Surface area is equivalent to the iso-surface of volumetric data processing that separates the target phase from the matrix. As a voxel in cubic lattice model is recognized as a hexahedron, the surface area can be directly derived from the label-data. Note here s_p has the unit of a^{-1} (or $1/a$), where a is the lattice constant, i.e. the resolution of the images. The SSA is a derived quantity that can be used to determine the type and properties of a material. It has a particular importance in case of adsorption, heterogeneous catalysis, and reactions on surfaces.

For a granular material the distribution of grain size is called the particle size distribution (PSD). The PSD of a material can be important in understanding its physical and chemical properties. It affects the strength and load-bearing properties of rocks and soils, and the reactivity of solids participating in chemical reactions. PSD can also be referred to as pore size distribution when the pores are the target phase. They are obtained on the basis of stereology by choosing arbitrary planes parallel to x, y and z axis and counting the particles (or pores). For a plane parallel to x axis, for example, we count the target phase in different sizes in the x direction on the plane. After counting other planes parallel to x axis, the summations of different sizes of the target phase and their percentages in the x direction are calculated. A similar procedure is performed in the y and z directions.

40.3.2 Clusters and Orientations

In the simple cubic lattice model, the nearest-neighbours are voxels with one common plane. A cluster is a group of nearest-neighbours of the same material that are connected to each other. Labelling clusters is a process of giving all cells within the same cluster the same label. The Hoshen-Kopelman algorithm (Hoshen and Kopelman 1976) is used to reduce the computing time in this procedure. After the cluster-labelling, each voxel is assigned a value representing the cluster number it belongs to. The largest cluster (with maximum voxel number) is cluster 1, the second largest one is cluster 2, and so on. The dataset is known as cluster-labelled

data to distinguish it from the binary material-labelled data resulting from the segmentation.

Percolation of a model can be determined after cluster labelling. Since all clusters in the model now have their specific label, it is possible to know which labels appear in the outer boundaries. We define a direction as percolating when one boundary of the parallelepiped in this direction has the same cluster label with its opposite boundary. This means the two end boundaries of the direction are connected by the same cluster and the model is permeable.

The shape and orientation of a cluster can be described by a tensor. The tensor is obtained by using the star volume distribution method following Ketcham (2005). For the cluster that consists of a set of n sites (voxels), each site i is considered as a vector $\boldsymbol{a}_i = (a_{xi}, a_{yi}, a_{zi})^{\mathrm{T}}$ relative to the center of the cluster. The shape and orientation of the cluster can be defined by the following matrix

$$\boldsymbol{T} = \sum_{i=1}^{n} \boldsymbol{a}_i \boldsymbol{a}_i^{\mathrm{T}}. \tag{40.2}$$

The matrix $\boldsymbol{T}$ has 3 eigenvalues $\tau_1 < \tau_2 < \tau_3$ and corresponding eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3$. The direction cosines of the eigenvectors describe the orientation of the cluster. The matrix can be visually represented by an ellipsoidal glyph. Isotropy index $I = \tau_1/\tau_3$ and elongation index $E = 1 - \tau_2/\tau_3$ describe the shape of the cluster in a simplified way. For porous media, if we assume that permeability anisotropy is relevant to the character of the pore structure, then the matrix $\boldsymbol{T}$ denotes the anisotropy of permeability.

40.3.3 Stochastic Analysis

Stochastic analysis is employed to detect the effect of scale and is computed by using the extended local porosity theory (Liu et al. 2009; Hilfer 1992, 2002). To implement the local porosity theory a cubic sub-volume with side-length Lacts as a moving window (Williams and Baxter 2006) scanning through the whole model. Each placement of the moving window is denoted as $\mathrm{K}(\boldsymbol{x}, L)$, where $\boldsymbol{x}$ is the central position of the sub-volume. The window offset can be equal to the lattice constant or an integer multiple of it. For every placement of the sub-volume, porosity (volume fraction), percolation, and anisotropy of the percolating cluster are calculated. Statistical results can then be drawn from the calculations of all placements of the sub-volume.

The statistical results are given by three probabilities.

(1) The local porosity distribution $\mu(\varphi, L)$ is defined as

$$\mu(\varphi, L) = \frac{1}{m} \sum\nolimits_{\boldsymbol{x}} \delta(\varphi - \varphi(\boldsymbol{x}, L)), \tag{40.3}$$

where δ denotes the Dirac δ-function, $\varphi(\boldsymbol{x}, L)$ is the local porosity in the sub-volume $\mathrm{K}(\boldsymbol{x}, L)$, m is the total number of placements of the moving window. $\mu(\varphi, L)$ is the proportion of the number of placements with the porosity φ and of the sub-volume-size L.

(2) The local percolation probability in the α-direction is defined through

$$\lambda_\alpha(\varphi, L) = \frac{\sum_{\boldsymbol{x}} \Lambda_\alpha(\boldsymbol{x}, L)\delta_{\varphi,\varphi(\boldsymbol{x},L)}}{\sum_{\boldsymbol{x}} \delta_{\varphi,\varphi(\boldsymbol{x},L)}}, \tag{40.4}$$

where

$$\delta_{\varphi,\varphi(\boldsymbol{x},L)} = \begin{cases} 1 & \text{if } \varphi = \varphi(\boldsymbol{x}, L) \\ 0 & \text{otherwise} \end{cases},$$

$$\text{and} \quad \Lambda_\alpha = \begin{cases} 1 & \text{if percolating in } \alpha\text{-direction} \\ 0 & \text{otherwise} \end{cases},$$

$\alpha = x, y,$ or z denotes the direction along every axis, and $\alpha = 3$ represents the percolation along all three directions. It is the proportion of the percolating placements of the sub-volume-size L and porosity φ.

(3) The local anisotropy distributions are defined as

$$\chi(I, L) = \frac{1}{m'} \sum_{\boldsymbol{x}} \delta(I - I(\boldsymbol{x}, L)), \tag{40.5}$$

where m' is the total number of percolating sub-volumes, I represents two indices of anisotropy, the isotropy index $I_1 = I = \frac{\tau_1(\boldsymbol{x},L)}{\tau_3(\boldsymbol{x},L)}$ and the elongation index $I_2 = E = 1 - \frac{\tau_2(\boldsymbol{x},L)}{\tau_3(\boldsymbol{x},L)}$. $\chi(I, L)$ gives the proportion of the placements with the index value I_i and the sub-volume-size L. Here $\tau_i(\boldsymbol{x}, L)$ are three normalised eigenvalues of the orientation matrix of the percolating cluster in sub-volume $\mathrm{K}(\boldsymbol{x}, L)$ that satisfy $\sum_{i=1}^{3} \tau_i(\boldsymbol{x}, L) = 1$. When more than one cluster is percolating in a sub-volume, only the largest percolating cluster is considered for the statistical analysis.

The stochastic analysis has two important goals. Firstly, the three probabilities are scale-dependent, that is, for different scales or sub-volume-sizes L, there are different distribution functions. When all these probabilities converge at a given scale the size of RVE is determined. Secondly, combining the probabilities of porosity, percolation and anisotropy of microtomography with the statistical moment method allows the generation of derivative models that retain the original material structure. Thus, we can create digital samples that represent the characteristics of the microstructure. Digital samples provide another way of upscaling, but it will not be discussed in this paper.

40.3.4 Scaling Parameters and Upscaling Scheme

In this section the definition and computation of scaling parameters are given, such as percolation threshold, critical exponent of correlation length, fractal dimension. The strategy of upscaling is also explained briefly. A more detailed description and applications can be found in Liu and Regenauer-Lieb (2010).

40.3.4.1 Percolation Threshold

In percolation theory (Stauffer and Aharony 1994), concentration p is the probability of a site belonging to the target phase; it is equivalent to volume fraction for the whole model. The (critical) percolation threshold $p_c$is the minimum concentration at which percolation would occur in the model. To determine the percolation threshold it is essential to have models with similar structures but different volume fractions. From static microtomographic data the percolation threshold is difficult to find for the specific structure.

We have developed a morphological technique to determine the percolation threshold of a microstructure by deflating (shrinking) and inflating (expanding) the target phase. As in 3-D binary models, the target phase is labelled as 0 and the matrix is labelled as 1. A deflation/inflation operation involves modifying the labels of voxels in the model, i.e., deflation changes a label from 0 to 1 for any target-phase-voxel that has one or more nearest-neighbours belonging to the matrix, inflation changes label from 1 to 0 for any matrix-voxel when it has one or more nearest-neighbours belonging to the target phase. Only nearest-neighbours are considered when modifying labels. This is consistent with the definition of clusters in the percolation analysis. The operations of inflation and deflation are manipulated iteratively from the original model to create inflated and deflated models, respectively. In this way a series of derivative models are created. These models have different volume fractions but retain structures similar to those in the original sample. By analysing the percolation of these models the critical model is identified as the percolating model closest to the non-percolating model. The volume fraction of the critical model is recognized as percolation threshold (Liu and Regenauer-Lieb 2010).

40.3.4.2 Critical Exponent of Correlation Length

There are several different critical exponents in percolation theory that describe some quantities that are divergent when concentration p is close to the percolation threshold p_c. The significance of critical exponents is that critical exponents are related to scaling laws and scaling laws enable up-scaling/down-scaling of the characteristics identified at a particular scale. Only two critical exponents are independent and in most cases the critical exponent of correlation length is the easiest to calculate.

The correlation length ξ defines the average distance between any two sites belonging to the same cluster (Stauffer and Aharony 1994). It is defined as

$$\xi^2 = \frac{2\sum_s R_s^2 s^2 n_s}{\sum_s s^2 n_s}, \tag{40.6}$$

where s is the number of sites of a cluster, n_s is the number of such s-site clusters per lattice site, R_s defined as

$$R_s^2 = \sum_{i=1}^{s} \left(|x_i - x_0|^2 / s \right) \tag{40.7}$$

is the radius of a complex cluster of s-site; where $x_0 = \sum_{i=1}^{s} (x_i/s)$ is the position of the centre of mass of the cluster, and x_i is the position of the ith site in the cluster. The correlation length ξ diverges when volume fraction p approaches the percolation threshold p_c, as

$$\xi \propto |p - p_c|^{-\nu}. \tag{40.8}$$

Here ν is the critical exponent of correlation length (or correlation length exponent). It can be extracted by a finite-size scaling scheme from the local percolation probabilities $\lambda(p, L)$ of different sub-volume-size L and different volume fraction p (Liu et al. 2009; Hilfer 1992, 2002).

40.3.4.3 Fractal Dimension

The fractal dimension D is derived from a power law describing a characteristic versus a statistical variable. In percolation theory the fractal dimension is included in scaling laws with critical exponents. The fractal dimension can be found from the relative size and the number of clusters defined as

$$D = \frac{\log_{10} N\left(l \geq \dfrac{R}{R_{\max}}\right)}{\log_{10} l\left(= \dfrac{R}{R_{\max}}\right)}, \tag{40.9}$$

where the relative size of cluster l is the radius of Eq. (40.7) normalised by the largest cluster radius, the number of cluster N includes clusters that are equal to or larger than the normalised relative radius.

40.3.4.4 Strategy of Upscaling

With all above listed parameters defined and extracted from the micro-CT we can move on to the upscaling scheme.

The stochastic analysis provides probabilities of porosity, percolation and anisotropy at different scales (limited to the 3-D microstructure model), and can be used to determine the size of the RVE. Different numerical simulation methods, such as finite element, finite difference and lattice-Boltzmann method, can be used to compute material properties from microstructures. Simulations should be based on the RVE, since only the RVE can represent the general characteristics. The upscaling workflow involves (1) cluster analysis using Eq. (40.6) to derive the fractal dimension D; (2) finite-size scaling scheme using the probability of percolation to derive the critical exponent of correlation length ν; (3) inflation/deflation of original structure to determine percolation threshold from a series of derivative models. The critical model with percolation threshold provides independent way to extract fractal dimension D and the crossover length ξ. The crossover length is a special correlation length separating the critical behaviour and non-critical behaviour. When we obtain the critical exponent of correlation length ν and fractal dimension D from individual samples, the final derived scaling laws are exact for the specific structures.

40.4 High Performance Computing

Computationally, the percolation analysis described by Eqs. (40.3)–(40.10) is implemented in Fortran90. Test and validation of the implementation has been reported in Liu et al. (2009), Liu and Regenauer-Lieb (2010). We have developed two main codes. One is "PROP" for stochastic analysis. In PROP only the largest percolating cluster in a sub-volume is analysed concerning the anisotropy in statistics as described in Sect. 40.3.3. The other code "CTSTA" accomplishes the analysis and output of all clusters in an entire model. Statistics of individual clusters in a model provides detailed information of the inner structure and can also be used to calculate the fractal dimension of the microstructure. Both codes are written for high performance computers.

40.4.1 Parallelisation of Stochastic Analysis

The stochastic analysis of the moving window method in PROP is the most time consuming computation. For a single volume, the analysis of porosity, percolation and anisotropy can be calculated on a single CPU in a short amount of time, e.g., 0.1–0.2 s for a volume of 1 million voxels. However, the total number of sub-volumes m in a stochastic analysis is given by

$$m = \prod_{i=1}^{3} \left(\mathrm{INT}\left(\frac{M_i - L}{n} \right) + 1 \right), \tag{40.10}$$

where M_i is the side-length of the model in units of the lattice constant a, i.e. the resolution of the images, n represents the moving step in units of the lattice constant. When $n = L$, there is a non-overlapping placement of sub-volumes that can lead to fluctuation in the results. For $n < L$, there is an overlapping placement and there is a higher weight given to the central region of the model. For a large data set and a small n the computing time on a single processor becomes increasingly problematic. To alleviate this problem, it is necessary to parallelise the calculation.

40.4.1.1 OpenMP Version (PROP_omp)

The probabilities defined in Eqs. (40.3)–(40.5) are dependent only on the summation of data within a moving window. By using the OpenMP programming techniques it was possible to quickly and easily augment the DO loops that move this window to move a series of windows, one for each available CPU. This parallelisation distributes the most time consuming part of the calculation cross multiple CPUs. As shown in Liu et al. (2009), the speedup is quite good when fewer than 32 CPUs are used.

The usability of PROP_omp is limited to shared memory processing (SMP) systems. Subject to the number of cores and physical memory available, PROP_omp can perform reasonably well only for medium size of sub-volume. Suppose each voxel in the sub-volume is represented by a 4 byte integer, the maximum theoretical sub-volume size is around 812^3 voxels for 2GB of RAM. In practice it is less than this as storage is also required for additional data structures, e.g., sub-volume-size can be from 500^3 to 640^3 depending on the volume fraction of the target phase. In addition, large-scale SMP systems consisting of 1000+ cores are becoming available for high performance computing. However due to limitations of the OpenMP architecture it is very difficult for PROP_omp to scale to hundreds of cores.

40.4.1.2 MPI Version (PROP_mpi_1)

To address these issues we then have parallelized PROP with message passing interface (MPI) in order to perform the stochastic analysis on larger sub-volumes (in the order of 1000^3 voxels) on a distributed memory machine and ensure the analysis runs in a massively parallel manner. The parallelisation consists of three major tasks: (1) flexibly assign or distribute the sub-volumes described by Eq. (40.10) onto different computing cores, (2) efficiently deal with input and output and (3) efficiently collect the statistics of the sub-volumes.

The second and third tasks are relatively straightforward, i.e, the parallel I/O is achieved through linking the I/O channels to MPI rank ID. The collection of the statistics over the sub-volumes is achieved with the MPI_Reduce function. The first task requires a careful design of a mapping topology between the order of sub-volumes and the number of computing cores. The mapping topology is chosen such that one can run PROP_mpi_1 with the number of cores being a factor of N_3 or $N_3 \times N_2$, where $N_3 = 1 + (M_3 - L)/n$ and $N_2 = 1 + (M_2 - L)/n$, assuming there

are a sufficient number of cores on a massively parallel system. One can also flexibly run PROP_mpi_1 with any number of MPI processes on a small Linux clusters with load imbalance minimised if sufficient number of cores are not available.

The results of PROP_mpi_1 have been compared with those generated by PROP_omp for the same set of data ($469 \times 657 \times 626$). Identical results are obtained (up to 5th digit), indicating the cumulative error associated with the use of MPI_Reduce is negligible. It is worth mentioning that the use of the MPI_Reduce function is a global operation and requires all MPI processes to send the data to the master. The scalability of PROP_mpi_1 with a large number of cores can be compromised if the statistical variables are not packed optimally or too many calls of MPI_reduce are used. Scalability tests of PROP have been performed on up to 540 cores, showing a linear speedup.

40.4.2 Parallelisation of Stochastic Analysis and Hoshen-Kopelman Algorithm (PROP_mpi_2)

The two versions of PROP code discussed above are based on the idea of analysing one sub-volume per CPU core. The calculation thus requires large physical memory when the size of sub-volume increases. Processing even larger volumes requires overcoming the physical memory constraint, we must distribute our analysis of a single sub-volume across multiple CPU cores. While this is more efficient use of RAM it is expected to be less efficient from a pure speed point of view as data dependencies will be introduced necessitating that the CPU cores coordinate their analysis.

Decomposition of sub-volume data proceeds with a technique commonly used when dealing with 3D regular grids. That is, if we have two CPUs we split the data in half along the x axis, 4 CPUs splits both the x and y axis, and 8 CPUs splits the x y and z axis. A similar process can be applied if more CPUs are available. We also allocate a "halo" of voxels, so that each CPU can receive information about voxels on the other side of the split.

The analysis proceeds by applying the Hoshen-Kopelman algorithm to each fragment of the sub-volume individually. Each CPU begins numbering clusters in its fragment of the sub-volume at "1". At this point it is not possible to identify if a cluster on one CPU is the same as a cluster on a neighbouring CPU. Nor is it possible to identify if two clusters on one CPU are actually joined through a cluster on an adjacent CPU.

Once the Hoshen-Kopelman algorithm has completed on each CPU, the CPUs communicate their number of clusters found and clusters are re-labelled to give them unique cluster labels. That is, if CPU 1 found 10 clusters then CPU 2 will add 10 to each of its cluster labels. If CPU 1 found 10 clusters and CPU 2 found 8 clusters then CPU 3 will add 18 to each of its cluster labels.

Each CPU will now send to its neighbours (defined by the x y and z axes) the cluster labels associated with voxels on its boundary. In this way each CPU can calculate which of its clusters are in direct contact with clusters on a neighbouring CPU. Contact between two clusters is defined by what amounts to an entry in something similar to a Compressed Sparse Row (CSR) representation of the adjacency matrix of the graph. That is, an array is allocated of m rows by (number of clusters) columns, where m is the maximum of number of clusters that are likely to be touching. Each CPU fills in the columns associated with the clusters it stores locally. The information in the adjacency matrix is shared between all CPUs by a call to MPI_Allreduce. We note that the CSR representation is inefficient in RAM when the correlation length is much less than the side length of the sub-volume fragment. If the correlation length is short, most clusters will not touch a side of their sub-volume fragment and will be guaranteed not to have neighbours. In this case, the CSR representation will allocate space for column with no rows.

Using the computational infrastructure presently available to us we have not run a calculation large enough to find the upper limit of the code. We predict that as the sub-volume includes more voxels the CSR representation will the limiting factor in the calculation size.

Finally we can use a standard Depth First Traversal (DFT) of the graph to create a mapping between old cluster labels and new cluster labels. A depth first traversal means that each cluster has a list of neighbours. We pick a cluster and relabel its first neighbour to point to this cluster. We then relabel the first neighbour's first neighbour. If this cluster has no subsequent neighbours we can move up to the first neighbour's second neighbour. Eventually we will move onto the subsequent neighbours of our original node. The same depth first traversal is performed on all CPUs used in the calculation thus avoiding the cost of communicating the results of the DFT between all CPUs. Further work on the parallel Hoshen-Kopelman implementation would involve replacing the CSR representation with a more RAM efficient representation and performing the graph traversal in parallel.

40.4.3 Calculation of Individual Clusters

To calculate the position, shape and orientation of every cluster might be troublesome when there are a great number of clusters in a volume. The problem arises from the large number of clusters and the large differences in cluster sizes. As defined in Eq. (40.2), to calculate $\boldsymbol{T}$ we firstly need to define arrays to load all the vectors of $\boldsymbol{a}_i = (a_{xi}, a_{yi}, a_{zi})^{\mathrm{T}}$ of clusters.

There are two options, either one array loads all clusters or there is one array per cluster. For the former option no matter how many dimensions of the array we define the size of the dimension must be sufficient to load the largest cluster. That means that when the smallest array (1 voxel) is being calculated it is using the same size of array and the same memory as the largest one. For the later option, as it is impossible to define millions of arrays in the code loading an uncertain numbers of clusters, we

can only define a dynamic array to load $\boldsymbol{a}_i = (a_{xi}, a_{yi}, a_{zi})^{\mathrm{T}}$ for only one cluster at a time. That means we need to read the 3-D cluster-labelled data each time for every cluster, which is unacceptably inefficient.

We cope with this problem by grouping the clusters, which is similar to data decomposition. A certain amount of memory is allocated to load the vectors $\boldsymbol{a}_i$. Since the dimension-size of the array always fits the largest cluster in the group, when large clusters are being calculated the group has fewer clusters; then when small clusters are being processed, the group can include a great number of clusters. This idea balances memory usage and computing time and results in a high performance solution.

40.5 Applications

The capabilities of our method have been verified and presented as specific case studies in Liu et al. (2009), Liu and Regenauer-Lieb (2010). Here we give some more unpublished examples and expand on the capabilities. Some post-processing techniques and visualisation are also described in the examples.

The following presents the results of the sample introduced in Fig. 40.1c. The porosity is 1.23 %; the model is not percolating; the SSA is 0.0327; the pore size distributions in three directions are shown in Fig. 40.5. It shows that more than 90 % pores are smaller than 9 μm (7 voxels) in all x, y and z directions. The largest pore is 70 μm (54 voxels) in z direction and it is the only one (out of the range of x axis of Fig. 40.5).

There are in total 1,895,341 (∼1.9 million) clusters in the volume. Nearly 1 million clusters are 1 or 2 voxel-clusters. The distribution of cluster size and the percentage of each cluster size over all pores are shown in Fig. 40.6. The number of clusters linearly decreases with increasing cluster size on a double logarithmic plot. Large clusters are unique and also exhibit a linear distribution along the x axis of Fig. 40.6. Interestingly, the large number of small clusters and the few large clusters occupy

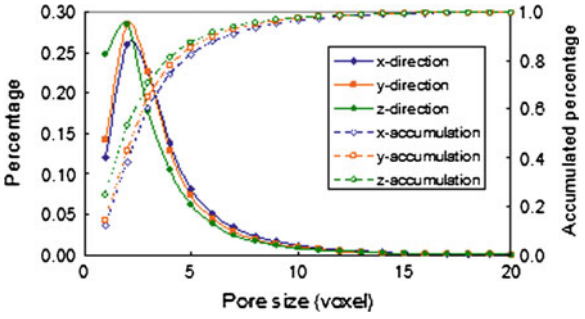


Fig. 40.5 Pore size distribution of the strongly deformed rock sample

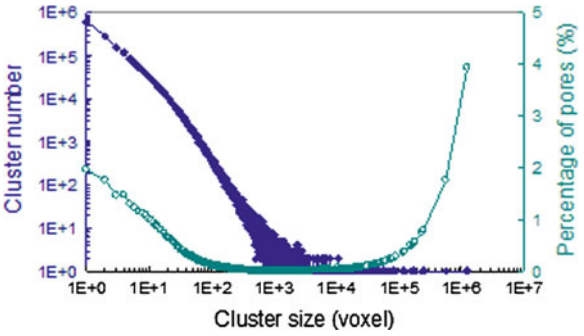


Fig. 40.6 Histogram of cluster size of the sample in Fig. 40.1, and the volume percentage of different cluster-sizes

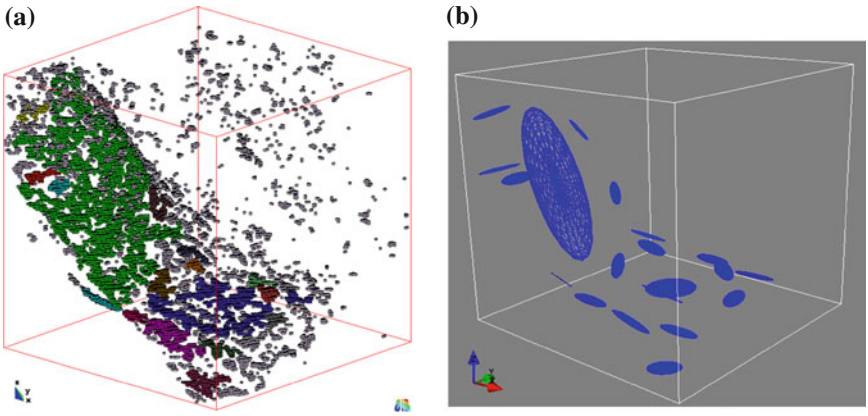


Fig. 40.7 Display of clusters in a 100 cube volume of the sample in Fig. 40.1

a high percentage of the volume of all pores; in contrast, middle-size clusters (with 100–10000 voxels) occupy a low proportion of the pores in the model. The largest cluster occupies less than 4 % space of the porosity.

Visualisation of clusters illustrates the characteristics of each cluster of the target phase in a model, which provides more information than visualisation of the target phase itself (as shown in Fig. 40.1c). Figure 40.7a shows the structures of clusters in a small volume of the same sample using GID®, in which each voxel is presented by a hexahedron. Different clusters are differentiated by using different colours but all tiny clusters are presented in one colour. Figure 40.7b illustrates the corresponding orientation tensors of the 20 largest clusters presented by ellipsoidal glyphs using Mayavi 2. Glyphs are located in the center of the cluster, and the radii are proportional to the sizes of the clusters. In this small volume pores are apparently mostly distributed along a crack but the connectivity of pores is poor and the volume is not percolating.

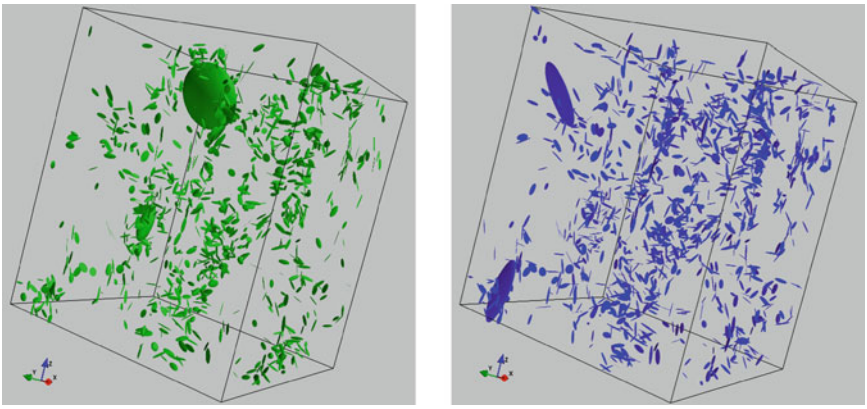


Fig. 40.8 Selected clusters of the sample in Fig. 40.1, the 1000 largest clusters (*left*) and large strongly anisotropic clusters that index $\leq$0.1 and cluster size $\geq$1000 voxels (*right*)

Although our code can deal with very large datasets the visualisation of large datasets can be limited by software. Some small post-processing codes were written to help to maximize the ability to meaningfully visualize the data. For instance, the orientation of very small clusters, especially 1- or 2-voxel-clusters, is generally not important, thus we can choose to only include the large clusters. We also can select which clusters to display according to their positions, size, isotropy index, elongation index, or combinations of these quantities. Figure 40.8 shows the selected clusters of the sample in Fig. 40.1, the 1000 largest clusters (left) and large strongly anisotropic clusters with isotropy index $\leq$0.1 and cluster size $\geq$1000 voxels (right). Both of these two groups of clusters illustrate the general character of the distribution of pores of the sample.

The histograms of the isotropy index and elongation index of the same sample are shown in Fig. 40.9. We can see that the isotropy index of most clusters is less than 0.2 and are more concentrated between 0.03–0.13. The elongation index of most clusters is larger than 0.5 and peaks at around 0.8. Both these two indices suggest that most pores in the sample are strongly anisotropic. Note that the statistics in Fig. 40.9 do not include clusters $\leq$20 voxels. The reason is that the orientation tensor is calculated by summing the dyadic product of the site vector, see Eq. (40.2). When the cluster is small each site vector has a major influence on the orientation tensor. The sensitivity of the site vector to orientation tensor places a lower limit on the cluster size for the meaningful statistic analysis. For example, there are high counts of 1, 2 and 3 voxel-clusters that generate an extremely high fraction of isotropy index of 1, 0.5, and 0.33, which should be ignored in the statistics.

For the applications of stochastic analysis the following examples and applications have been published. (1) Liu et al. (2009) gave a synthetic sandstone sample as the benchmark. The probabilities of porosity, percolation and anisotropy all asymptotically converge when the sub-volume-size approaches to 400 voxels (1 mm),

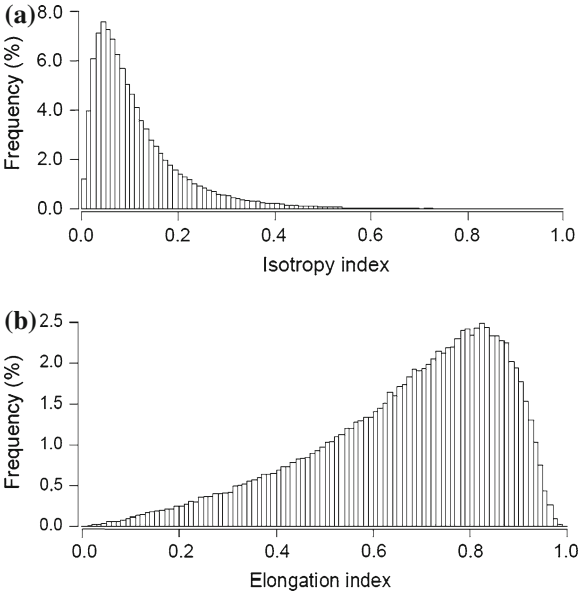


Fig. 40.9 Histogram of isotropy index (**a**) and elongation index (**b**) of the sample of Fig. 40.1

thus the size of RVE of this sample was determined. The RVE was later used to calculate the permeability and the result was comparable with experimental laboratory result (Liu and Regenauer-Lieb 2010). (2) Liu et al. (2010) gave two sub-samples from the same hand specimen of Fig. 40.1. These two sub-samples showed different characteristics from the synthetic sandstone sample and they are different from each other. As the probabilities of anisotropy showed fluctuation the size of RVE cannot be determined. (3) Liu and Regenauer-Lieb (2010) also gave some results of a tree branch sample. This application showed that percolation theory can be used in analysis of this type of strongly anisotropic model, but the extraction of correlation length and critical exponent of correlation length are unattainable.

The scaling parameters and upscaling scheme is our latest progress and mostly presented in Liu and Regenauer-Lieb (2010). It uses the results of PORP and CTSTA computations and combines some new approaches such as inflation/deflation of images, calculation of fractal dimension, and supplementary analysis. It is based on the visualisation, characterisation and high performance computing of microtomography and makes it possible to extend the scale from micro-scale to macro-scale. The relevant discussion is beyond the scope of the present contribution. For a more detailed description please refer to the aforementioned paper.

40.6 Conclusion and Discussion

Our method and computer algorithms have achieved (1) computation of major parameters, including volume fraction, specific surface area, particle (or pore) size distribution; (2) stochastic analysis and determination of RVE, probabilities of porosity, percolation, and anisotropy for different scales are computed, when these probabilities are convergent at a scale the size of RVE is determined; (3) extraction of critical exponent of correlation length, fractal dimension, and percolation threshold, thus parameters of scaling laws are determined making it possible to scale up properties from micro-scale to macro-scale. With these three capabilities, our method is unique, more comprehensive, and more powerful than other commercial/published codes for the analysis of microtomography.

We are now applying this methodology to routinely derive elastic percolation, electrical percolation, thermal conductivity and many other properties from CT-scans. Our analyses showed that for samples with strongly anisotropic structure the critical exponent of correlation length is not attainable (Liu and Regenauer-Lieb 2010). In this situation other methods for achieving the upscaling must be used. Our long-term goal is to establish an integrated methodological system with applications to multi-scale system dynamics. The progress in this paper is a milestone that not only provides a tool for detailed analysis of microtomography, but also links the micro-scale to macro-scale.

Considering the size of microtomographic data and the degree of computation required, the main code has been progressively parallelised. The first stage of the parallelisation is based upon OpenMP, which performs very well for most situations. Limited by the OpenMP architecture the performance does not continue to scale as more CPUs are used. The development of the second stage of parallelism was based on MPI. This version removes the limitation of 32 CPUs and the performance improvement is linear up to at least hundreds of CPUs. However, this version does not consider the decomposition of data, thus the size of datasets is still a barrier. To improve this aspect of the code a new version, also based upon MPI, has been developed which includes data decomposition. By distributing the whole dataset and sub-volumes into CPUs the size of datasets is no longer a barrier. This third version has additionally achieved a significant breakthrough by parallelising the Hoshen-Kopelman algorithm.

Although all the progress and capabilities have been very successful there is still significant scope for further improvements. We believe the performance of the latest MPI version can be improved further. It is also acknowledged that GPU computing has been maturing over the last few years and is showing great advantages in computing speed and energy efficiency. We expect the algorithms presented here to be suited to GPU computing and thus significant performance gains should be achievable.

Acknowledgments This project was funded through the Premier's Fellowship Program of the Western Australian Government, CSIRO OCE Postdoctoral Program, and the Geothermal Centre of Excellence of Western Australia. We are grateful to iVEC for technical support and access to high performance computing resources.

References

Fusseis F, Schrank C, Liu J, Karrech A, Llana-Fúnez S, Xiao X, and Regenauer-Lieb K (2012) Pore formation during dehydration of polycrystalline gypsum observed and quantified in a time-series synchrotron x-ray tomography experiment. Solid Earth 3: 71-86

Hilfer R (1992) Local porosity theory for flow in porous media. Phys Rev B 45:7115–7121

Hilfer R (2002) Review on scale dependent characterization of the microstructure of porous media. Transp Porous Med 46:373–390

Hoshen J, Kopelman R (1976) Percolation and cluster distribution 1. Cluster multiple labelling technique and critical concentration algorithm. Phys Rev B 14:3438–3445

Ketcham RA (2005) Three-dimensional grain fabric measurements using high-resolution X-ray computed tomography. J Struct Geol 27:1217–1228

Lindquist WB (2005) 3DMA-Rock. http://www.ams.sunysb.edu/~lindquis/3dma/3d_imaging.html. Accessed 13 Dec 2010

Liu J, Regenauer-Lieb K (2010) Application of percolation theory to microtomography of structured media: percolation threshold, critical exponents and upscaling. Phys Rev E (in press)

Liu J, Regenauer-Lieb K, Fusseis F (2010) Stochastic analysis of percolation and anisotropic permeability from microtomographic images and an application to mylonites. In: Satake K (ed) Advances in geosciences, vol 20. World Scientific Publishing Co., Singapore, pp 229–245

Liu J, Regenauer-Lieb K, Hines C, Liu K, Gaede O, Squelch A (2009) Improved estimates of percolation and anisotropic permeability from 3-D X-ray microtomographic model using stochastic analyses and visualization. Geochem Geophys Geosyst 10:Q05010. doi:10.1029/2008GC002358

Nakashima Y, Kamiya S (2007) Mathematica programs for the analysis of three-dimensional pore connectivity and anisotropic tortuosity of porous rocks using X-ray computed tomography image data. J Nucl Sci Technol 44:1233–1247

Stauffer D, Aharony A (1994) Introduction to percolation theory, 2nd edn. Taylor & Francis Ltd., London

Williams TO, Baxter SC (2006) A framework for stochastic mechanics. Probab Eng Mech 21:247–255

Chapter 41
Three-Dimensional Reconstruction of Electron Tomography Using Graphic Processing Units (GPUs)

Xiaohua Wan, Fa Zhang, Qi Chu and Zhiyong Liu

Abstract Three-dimensional (3D) reconstruction of electron tomography (ET) has emerged as a leading technique to elucidate the molecular structures of complex biological specimens. Iterative methods using blob basis functions are advantageous reconstruction methods due to their good performance especially under noisy and limited-angle conditions. However, iterative reconstruction algorithms for ET pose tremendous computational challenges. Graphic processing units (GPUs) offer an affordable platform to meet these demands. Nevertheless, due to the limited available memory of GPUs, the weighted matrix involved by iterative methods cannot be located into GPUs especially for the large images. To meet high computational demands, we propose a multilevel parallel scheme to perform iterative algorithm reconstruction using blob on GPUs. In order to address the large memory requirements of the weighted matrix, we also present a matrix storage technique, called blobELL-R, suitable for GPUs. In the storage technique, several geometric related symmetry relationships have been exploited to significantly reduce the storage space. Experimental results indicate that the multilevel parallel reconstruction scheme on GPUs can achieve high and stable speedups. The blobELL-R data structure only needs nearly 1/16 of the storage space in comparison with ELLPACK-R (ELL-R) storage structure and yields significant acceleration compared to the standard and matrix with CRS implementations on CPU.

X. Wan (✉) · F. Zhang · Q. Chu · Z. Liu
Institute of Computing Technology, Beijing, China
e-mail: wanxiaohua@ict.ac.cn

F. Zhang
e-mail: zf@ncic.ac.cn

Q. Chu
e-mail: chuqi@ict.ac.cn

Z. Liu
e-mail: zyliu@ict.ac.cn

X. Wan · Q. Chu
Chinese Academy of Sciences, Graduate University, Beijing, China

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, 675
Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7_41,

Keywords Electron tomography · Three-dimensional reconstruction · Iterative methods · Blob · GPUs

41.1 Introduction

In biosciences, three-dimensional (3D) structural information is critical to understanding biological specimens. Electron tomography (ET) uniquely enables the study of complex cellular structures, such as cytoskeletons, organelles, viruses and chromosomes (Frank 2006). From a set of projection images taken from a single individual specimen, 3D structure can be obtained by means of tomographic reconstruction algorithms. Tomographic reconstruction can be modeled as a least square problem to be solved by matrix algorithms, where a large sparse weighted matrix is involved (Herman 2009). These matrix algorithms consist of weighted backprojection (WBP) (Frank 2006) and iterative schemes (e.g. Simultaneous Algebraic Reconstruction Technique (SART) and Simultaneous Iterative Reconstruction Technique (SIRT)) (Andersen and Kak 1984; Gilbert 1972a). WBP, as a current standard algorithm, is firstly formulated as a set of sparse matrix-vector products (SpMV) (Vazquez et al. 2010). Iterative methods have an advantage over WBP due to their good performance especially under limited-angle and noisy conditions (Frank 2006).

However, the convergence process of iterative methods is generally time-consuming and the data volume of images in ET increases constantly to fulfill the resolution needs. These huge computational demands have prevented iterative methods from being used extensively. The traditional solution to cope with the high computational cost has been the use of supercomputers and large computer clusters (Fernandez et al. 2004; Fernandez 2008), but such hardware is expensive and can also be difficult to use. Graphics processing units (GPUs) offer an attractive alternative platform in terms of price and performance. The GPU method proposed by Castano-Diez et al. can be viewed as a first step towards achieving high-performance ET (Castano-Diez et al. 2007). Recently, an advanced GPU acceleration framework has been proposed to allow 3D ET reconstruction to be performed on the order of minutes (Xu et al. 2010). These methods adopt traditional voxel basis functions. In the field of 3D reconstruction, the choice of basis functions greatly influences results to be reconstructed. Spherically symmetric volume elements (blobs) with smooth transition to zero have been thoroughly investigated as alternatives to voxels due to their overlapping nature (Lewitt 1992). Accordingly, we present a multilevel parallel strategy of 3D reconstruction of ET using blob basis functions on GPUs. The parallel strategy yields smoother reconstructions under noisy conditions.

Furthermore, owing to the large memory requirements and the limited available memory of GPUs, the weighted matrix is usually computed on the fly in previously proposed GPU-based ET implementations (Bilbao-Castro et al. 2006). But it could bring the redundant computations since the weighted matrix has to be computed twice at one iteration. Although the memory available in GPUs today allows storage of large sparse matrices, sparse matrix data structures must be carefully

devised to optimize the access to the memory hierarchy. Compressed row storage (CRS) is the most extended format to store the sparse matrix on CPUs (Bisseling 2004). ELLPACK can be considered as an approach to outperform CRS (John and Ronald 1985). Vazquez et al. has proved that a variant of the ELLPACK format called ELLPACK-R (ELL-R) is more suited for the sparse matrix data structure on GPUs (Vazquez et al. 2009). Nevertheless, as data collection strategies and electron detectors improve, the memory demand of the weighted matrix rapidly increases. Storing such a large sparse matrix is extremely difficult for most GPUs and accessing the matrix data substantially reduces the efficiency of 3D reconstruction of ET on GPUs. Facing this problem, we adopt a data structure named blobELL-R with several symmetry optimization techniques to significantly reduce the storage space of the weighted matrix. It only needs nearly 1/16 of the storage space in comparison with the other methods. Furthermore, the blobELL-R matrix storage scheme is adopted to ensure optimal coalesced global memory access. Thus, the blobELL-R format allows efficient implementations of 3D-ET reconstruction algorithms on GPUs.

In this work, we present a multilevel parallel strategy for iterative reconstruction using blob basis functions on GPUs and an optimal data structure blobELL-R, and we have implemented them on a NVIDIA GeForce GTX 295. Experimental results show that the parallel strategy using blobELL-R greatly reduces memory requirements and exhibits a significant acceleration factor compared with the standard implementation.

The rest of the paper is organized as follows: Sect. 41.2 reviews relevant background both on reconstruction algorithms and on GPU hardware. Section 41.3 presents the multilevel parallel strategy of 3D reconstruction of ET and its parallelization on GPUs. Section 41.4 describes blobELL-R data structure and several symmetry optimization techniques. Section 41.5 shows and analyzes the experimental validation in detail. Then we summarize the paper and present future work in Sect. 41.6.

41.2 Background

In this section, we give a brief overview of iterative reconstruction algorithms using blob, describe a kind of iterative method called SIRT, present GPU computational model and highlight some of the features critical to our algorithm.

41.2.1 Iterative Reconstruction Methods Using Blob

In ET, projection images are acquired from a specimen through the so-called single-axis tilt geometry. The specimen is tilted over a range, typically from $-60°$ (or $-70°$) to $+60°$ (or $+70°$) at small tilt increments ($1°$ or $2°$). An image of the same object area is recorded at each tilt angle and then the 3D reconstruction of the specimen is obtained from a set of projection images with iterative methods.

Iterative methods represent 3D volume f as a linear combination of a limited set of known and fixed basis functions b_j, with appropriate coefficients x_j, as shown in Eq. 41.1.

$$f(r) \approx \sum_{j=1}^{J} x_j b_j(r). \tag{41.1}$$

During the 1990s, spherically symmetric volume elements (blobs) have been thoroughly investigated. As a consequence, the use of blob basis functions provides iterative methods with better resolution-noise performance than traditional voxel basis functions due to their overlapping nature (Lewitt 1992). Thus, we consider blob, instead of the traditional voxel, as a basis function. The blob basis function is constructed with generalized Kaiser-Bessel (KB) window functions:

$$b(r) = \begin{cases} \frac{\left(\sqrt{1-(r/a)^2}\right)^m I_m\left(\alpha\sqrt{1-(r/a)^2}\right)}{I_m(\alpha)}, & 0 \le r \le a \\ 0, & \text{otherwise,} \end{cases}$$

where $I_m(\cdot)$ denotes the modified Bessel function of the first kind of order m, a is the radius of the blob, and α is a non-negative real number controlling the shape of the blob. The choice of the parameters m, a, and α will influence the quality of the blob-based reconstructions. Since the principles involved by Matej and Lewitt in selecting particular types of blobs as good basis functions were quite general, we adopted the basis functions advocated by them in this paper (i.e., $a = 2$, $m = 2$ and $\alpha = 3.6$) (Matej and Lewitt 1995).

In 3D-ET, the data collection method assumes the model of the image formation process as the following linear system:

$$p_i \approx \sum_{j=1}^{J} w_{ij} x_j,$$

where p_i denotes the ith measured image of f and w_{ij} the value of the ith projection of the jth basis function. Under such a model, the element w_{ij} can be calculated (or, at least, estimated) according to the projecting procedure as follows:

$$w_{ij} = 1 - (rf_{ij} - \text{int}(rf_{ij})), \quad rf_{ij} = projected(x_j, \theta_i),$$

where rf_{ij} is the value of the pixel x_j projected with an angle θ_i. It is found that the weighted matrix involved in every slice is the same since the projections have the same geometry for all slices. Here, W is defined as a sparse matrix with M rows and N columns. In general, the demand of the weighted matrix W rapidly increases as the size of the images increases. It is hard to store the weighted matrix W in GPUs especially when the number of views and the size of the projections increase.

41.2.2 Simultaneous Iterative Reconstruction Technique

Considering the fact that the simultaneous iterative reconstruction technique (SIRT) shows the best performance under the extremely noisy condition, we adopt SIRT to perform 3D reconstruction of ET. SIRT, as a classical iterative method, begins with an arbitrary $X^{(0)}$ and repeats the iterative processes (Gilbert 1972b). Our previous work has proven that the initial solution $X^{(0)}$ obtained via the back projection technique (BPT) is much closer to the true value than the arbitrary initial solution (Xiaohua 2009). In iterations, the residuals, i.e. the differences between the actual projections P and the computed projections P' of the current approximation $X^{(k)}$ (k is the iterative number, $k = 0, 1, \ldots$), are computed and then $X^{(k)}$ is updated by the backprojection of these discrepancies. Thus, the algorithm produces a sequence of N-dimensional column vectors $X^{(k)}$. The SIRT algorithm is typically written by the following expression:

$$x_j^{k+1} = x_j^k + \frac{1}{\sum_{i=1}^{M} w_{ij}} \sum_{i-1}^{M} \frac{w_{ij}(p_i - \sum_{h=1}^{N} w_{ih} x_h^{(k)})}{\sum_{h=1}^{N} w_{ih}}.$$

SIRT adopts a global strategy: an approximation is updated simultaneously by all the projections P. SIRT updates each x_j only once per iteration, which means its updating strategy is pixel-by-pixel. Thus, SIRT is very suitable for parallel computing on GPUs.

41.2.3 GPU Computation Model

Our algorithm is based on NIVIDIA GPU architecture and compute unified device architecture (CUDA) programming model. GPUs are a massively multi-threaded data-parallel architecture, which contains hundreds of processing cores. Eight cores are grouped into a Streaming Multiprocessor (SM), and cores in the same SM execute instructions in a Single Instruction Multiple Thread (SIMT) fashion (NVIDIA 2008). During execution, 32 threads from a continuous section are grouped into a warp, which is the scheduling unit on each SM.

NVIDIA provides the programming model and software environment of CUDA. CUDA is an extension to the C programming language. A CUDA program consists of a host program that runs on CPU and a kernel program that executes on GPU itself. The host program typically sets up the data and transfers it to and from the GPU, while the kernel program processes that data. Kernel, as a program on GPUs, consists of thousands of threads. Threads have a three-level hierarchy: grid, block, thread. A grid is a set of blocks that execute a kernel, and each block consists of hundreds of threads. For the execution, each block is assigned to one SM composed by eight

cores called scalar processors (SPs). CUDA devices use several memory spaces including global, local, shared, texture, and registers. Of these different memory spaces, global and texture memory are the most plentiful. Perhaps the single most important performance consideration in programming for the CUDA architecture is coalescing global memory accesses. Global memory loads and stores by threads of a half warp (16 threads) are coalesced by the device in as few as one transaction (or two transactions in the case of 128-bit words) when certain access requirements are met. Coalesced memory accesses deliver a much higher efficient bandwidth than non-coalesced accesses, thus greatly affecting the performance of bandwidth-bound programs.

NVIDIA GeForce GTX 295 GPU card consists of two GeForce GTX 200 GPUs on a single card. Each GTX 200 GPU comprises a set of 30 SMs, where each SM consists of eight SPs clocked at 1242 MHz, meaning 480 SPs in total on GTX 295 GPU. The on-chip memory for each SM contains 16,384 registers and 16 KB shared memory. The off-chip memory (or device memory) contains 1792 MB GDDR3 (896 MB per GPU) global memory.

41.3 Multilevel Parallel Strategy for Iterative Reconstruction Using Blob

The processing time of 3D reconstruction with iterative methods remains a major challenge due to large reconstructed data volume in ET. So parallel computing is becoming paramount to cope with the computational requirement. The use of blob basis functions makes the single-tilt axis reconstruction relatively more straightforward to implement on GPUs. We present a multilevel parallel strategy for iterative reconstruction using blobs and implement it on the OpenMP-CUDA architecture.

41.3.1 Coarse-Grained Parallel Scheme Using OpenMP

In the first level of the multilevel parallel scheme, a coarse-grained parallelization is straightforward in line with the properties of ET reconstruction that single-tilt axis geometries allows a data decomposition into slabs equals the number of GPUs, and each GPU reconstructs its own slab. Consequently, the 3D reconstruction problem can be decomposed into a set of 3D slabs reconstruction subproblems. However, due to their overlapping nature, the use of blob basis functions makes the slabs interdependent. Therefore, each GPU have to communicate with other GPUs through the memory of CPU.

We have implemented the 3D ET reconstruction based on GeForce GTX 295 which consists of two GeForce GTX 200 GPUs on a single card. Thus the first-level parallel strategy makes use of two GPUs to perform the coarse-grained parallelization

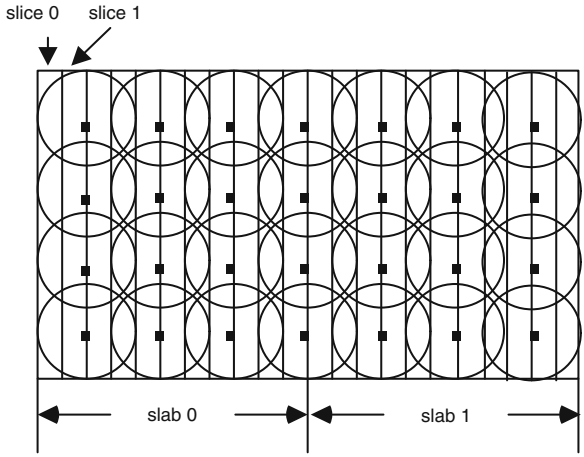


Fig. 41.1 Coarse-grained parallel scheme using blob

of the reconstruction. The 3D reconstruction parallel scheme allows the 3D volume data to be halved into two slabs, and a slab contains many slices, as shown in Fig. 41.1. According to the shape of the blob adopted, four slices are involved in one blob. Each slab is assigned to and reconstructed on each individual GTX 200 on GTX 295 in parallel. In contrast, all slices in a slab are serially reconstructed. Certainly, the parallel strategy can be applied on GPU clusters (e.g. Tesla-based cluster). In a GPU cluster, the number of slabs equals the number of GPUs for the decomposition described above.

A shared-memory parallel programming scheme (OpenMP) is employed to fork two threads to control the separated GPU. Each slab is reconstructed by each parallel thread on each individual GPU. Consequently, the partial results attained by GPUs are combined to complete the final result of the 3D reconstruction.

41.3.2 Fine-Grained Parallel Scheme Using CUDA

In the second level, 3D reconstruction of one slab, as a fine-grained parallelization, is implemented with CUDA on each GPU. In the 3D reconstruction of the slab, the generic iterative process is described as follows:

- Initialization: compute the matrix W and make a initial value for $X^{(0)}$ by BPT;
- Reprojection: estimate the computed projection data P' based on the current approximation X;
- Backprojection: backproject the discrepancy ΔP between the experimental and calculated projections, and refine the current approximation X by incorporating the weighted backprojection ΔX.

```
Decidemap<<< >>> ;
(1)For i slice in the reconstructed volume
    BPT<<<>>> for every pixel of the ith slice;
  (2)For t in N iterations of the iTh slice
        Reprojection<<<>>> for each projection;
        Backprojection<<<>>> for every pixel of the ith slice;
```

Fig. 41.2 Pseduo code for one slab reconstruction based on SIRT

All the stages are compute-intensive tasks and suitable to highly parallel many-core architecture.

Figure 41.2 shows the pseudo code for the 3D reconstruction by means of the aforementioned stages. All the slices in the 3D volume are reconstructed sequentially in loop (1). Before loop (1), the kernel called Decidemap performs the initialization to compute the weighted matrix W. And at the beginning of loop (2), $X^{(0)}$ is approximated by BPT in the pixel-level parallel kernel called BPT. In each iteration, two kernels (i.e. Reprojection and Backprojection) perform the next stages including reprojection and backprojection on GPUs. In Reprojection, P' of each projection is computed by each thread. Thus, reprojection is a projection-level parallel process. Finally, we invoke a backprojection kernel on GPUs. Each thread of the kernel performs the pixel-based backprojection.

41.4 BlobELL-R Format with Symmetric Optimization Techniques

On the basis of the voxel basis functions, Vazque et al. presented an optimal data structure, called ELL-R, for the weighted matrix W on GPUs (Vazquez et?al. 2009, 2010). ELL-R consists of two arrays, A[] and I[] of dimension N $\times$ MaxEntriesbyRows, and an additional integer array called rl[] of dimension N is included in order to store the actual number of nonzeroes in each row. With the size and number of projection images increasing, the memory demands rapidly increase accordingly in general implementation. Thus the weighted matrix involved is too large to load into most of GPUs due to the limited available memory, even with the ELL-R data structure.

In our work, we present a data structure named blobELL-R with several geometric related symmetry relationships. The blobELL-R data structure decreases the memory requirement and then accelerates the speed of ET reconstruction on GPUs.

41.4.1 General Principle

As shown in Fig. 41.3, the maximum number of the rays related to each pixel is four on account of the radius of the blob (i.e., a = 2). First, the blobELL-R format stores

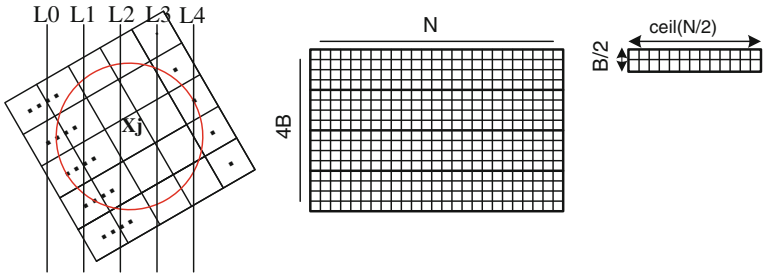


Fig. 41.3 BlobELL-R format. In the blob (the radius a=2), a projected pixel contributes to four neighbor projection rays using only one view (*left*). The nonzeroes of matrix W are stored in a 2D array A of dimension (4B) × N in blobELL-R (*middle*). The symmetric optimization technique is exploited to reduce the storage space of A to almost 1/16 of original size (*right*)

the sparse matrix W on two 2D arrays, one float A[], to save the entries, and one integer I[], to save the columns of every entry (see Fig. 41.3 middle). Both arrays are of dimension (4B) × N, where N is the number of columns of W and 4B is the maximum number of nonzeroes in the columns (B is the number of the projection angles). Because the percentage of zeros is low in the blobELL-R data structure, it is not necessary to store the actual number of nonzeroes in each column.

Although blobELL-R can reduce the storage of the sparse matrix W, the number of (4B) × N is rather large especially when the number of N increases rapidly. The optimization takes advantage of the symmetry relationships as follows:

41.4.1.1 Symmetry 1

Assume that the *j*th column elements of the matrix *W* in each view are w_{0j}, w_{1j}, w_{2j} and w_{3j}. The relationship among the adjacent column elements is:

$$w_{0j} = 1 + w_{1j}; w_{2j} = 1 - w_{1j}; w_{3j} = 2 - w_{1j}.$$

So, only w_{1j} is stored in the blobELL-R structure, whereas the others are easily computed on the fly. This scheme can reduce the storage spaces of A and I to 25 %.

41.4.1.2 Symmetry 2

Assume that a point (x, y) of a slice is projected to a point r ($r = project(x, y, \theta)$) in the projection corresponding to the tilt angle θ and $project(x, y, \theta)$ is shown as follows:

$$project(x, y, \theta) = x \cos\theta + y \sin\theta.$$

It is easy to see that the point (−x, −y) of a slice is then projected to a point $r1$ ($r1 = -r$) in the same tilt angle. Therefore, there is no need to store the weighted value for almost a half of the points (x, y) in the slice. As a result, the space requirements for A and I are further reduced by nearly 50 %.

41.4.1.3 Symmetry 3

In general, the tilt angles used in ET are halved by 0°. Under the condition, a point (−x, y) with the tilt angle $-\theta$ is projected to a point r2 (r2 = −r). Therefore, the projection coefficients are shared with the projection of the point (x, y) with the tilt angle θ. This further reduces the storage spaces of A and I by nearly 50 % again.

With the symmetric optimizations mentioned above, the size of the storage for two arrays in the blobELL-R format is almost $(B/2) \times (N/2)$ reducing to nearly 1/16 of original size.

41.5 Result

In order to demonstrate the applicability of the blobELL-R format to real data, 3D reconstructions of the sample of the caveolaes from the porcine aorta endothelial (PAE) cell (Shufeng et al. 2009) have been performed on a CPU based on Intel Core 2 Q8200 at 2.33 GHz, 4 GB RAM and 4MB L2 cache, under Linux, and a NVIDIA GeForce GTX 295 card including two GPUs, one GPU with 30 SMs of 8 SPs (i.e. 240 SPs) at 1.2 GHz, 896 MB of memory and compute capability 1.3, respectively. The standard SIRT and the matrix SIRT with CRS are used for the serial program on the CPU. The standard SIRT denotes that the weighted matrix W is not stored but computed on the fly in the process of ET reconstruction. However, the weighted matrix W is pre-computed and stored with a sparse matrix data structure (e.g. CRS, ELL-R, blobELL-R) in the matrix SIRT. The standard SIRT, the matrix SIRT with ELL-R and the matrix SIRT with blobELL-R are adopted in the parallel program on the GPU. And three different experimental datasets are involved (denoted by small, medium, large) with 56 images of 512×512 pixels, 112 images of 1024×1024 pixels and 119 images of 2048×2048 pixels to reconstruct tomograms of $512 \times 512 \times 190$, $1024 \times 1024 \times 310$ and $2048 \times 2048 \times 430$ respectively.

Figure 41.4 shows the memory demanded by the sparse data structure (i.e. CRS on the CPU, ELL-R and blobELL-R on the GPU respectively). In general, the requirements rapidly increase with the dataset size, approaching 3.5 G in the large dataset. This amount turns out to be not a problem in modern computers, but a limit owing to the upper boundary imposed by the memory available in most GPUs. However, in the blobELL-R matrix structure, three symmetry relationships can greatly decrease the memory demands and make them affordable on GPUs.

Three sets of experimental data have been subjected to tomographic reconstruction with the standard and matrix SIRT on the CPU and GPU respectively.

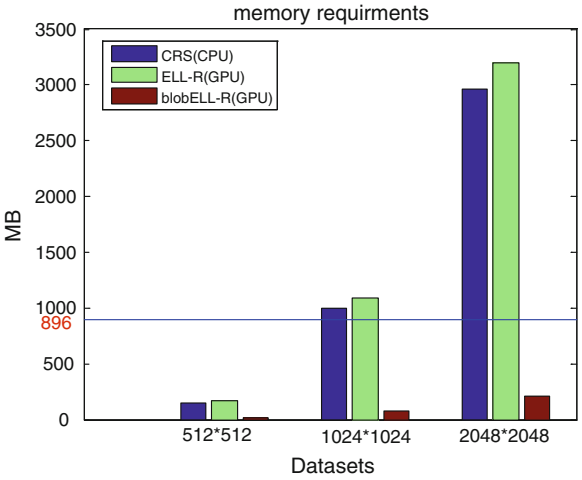


Fig. 41.4 Memory requirements of the different implementations for the datasets. The limit of 896 MB is imposed by the memory available in the GPU used in the work. The implementations needing higher amounts of memory than the value cannot run on this GPU. However, the use of the blobELL-R data structure reduces the demands, making most of the problems affordable

Table 41.1 Running times (s)

Datasets	Kernels	CPU		GPU		
		Standard	Matrix (CRS)	Standard	Matrix (ELL-R)	Matrix (blobELL-R)
512 × 512	Decidemap	–	0.96	–	0.00141	0.00017
	BPT	0.90	0.13	0.00081	0.00069	0.000591
	Reprojection	0.95	0.17	0.00399	0.00312	0.00284
	Backprojection	0.91	0.14	0.00133	0.00112	0.00101
	Reconstruction	1413.12	235.52	16.13	8.96	6.45
1024 × 1024	Decidemap	–	6.26	–	0.03225	0.00128
	BPT	5.91	0.90	0.00464	0.00381	0.00301
	Reprojection	6.27	1.21	0.02111	0.01851	0.01714
	Backprojection	6.08	1.02	0.00937	0.00767	0.00621
	Reconstruction	18708.48	3225.60	185.34	132.11	87.86
2048 × 2048	Decidemap	–	18.46	–	–	0.00369
	BPT	17.63	2.74	0.01332	–	0.00984
	Reprojection	18.56	3.62	0.06292	–	0.05123
	Backprojection	18.02	3.07	0.02364	–	0.01898
	Reconstruction	111288.32	19476.48	980.99	–	538.63

The results marked with '–' represent unaffordable cases

Table 41.1 shows the computation times of different kernels (i.e. Decidemap, BPT, Reprojection, Backprojection) with different methods on the GPU versus CPU for three datasets. As shown in Table 41.1, the kernel Decidemap is not involved in the standard SIRT owing to the recomputation of the coefficients. And all the computa-

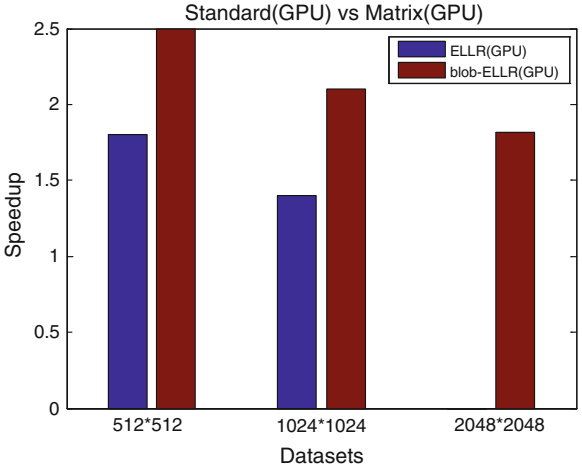


Fig. 41.5 Speed-up factors showed by the different matrix SIRT implementations over the standard SIRT based on recomputation of the coefficients. These factors were obtained on the GPU

tion times of 3D reconstruction are given in the last items of three datasets. For a fair comparison, the time required for generating the matrix is taken into account though it is negligible. Furthermore, the matrix SIRT with ELL-R is not implemented on the GPU for the largest dataset because the memory requirements exceed the upper boundary imposed by the memory available on GTX 295.

And the acceleration factor is approaching to 2.5× by means of matrix approach on the GPU in the Fig. 41.5. These results demonstrate that matrix SIRT succeeds in reducing the computing time required for tomography reconstruction.

Figure 41.6 compares the speed-up of matrix SIRT on the GPU versus CPU. The GPU exhibits excellent acceleration factors compared with the CPU. In the matrix SIRT with ELL-R, the speed-up is almost 25× for two datasets. And in the case of the matrix SIRT with blobELL-R, the acceleration factors increase and reach up to 35× for three experiments.

Finally, in order to estimate the performance of matrix SIRT and GPU computing, the speed-up factors against the standard SIRT on the CPU are showed in Fig. 41.7. For comparison, the acceleration factors of the GPU over the CPU on the standard SIRT are presented. The general rule is that the higher speedup is obtained with the larger dataset due to multithreaded computation on the GPU. However, the cases corresponding to the matrix SIRT with ELL-R and blobELL-R cannot fulfill the rule owing to the memory access delay. And it is clearly seen that the matrix SIRT with blobELL-R on the GPU yields excellent speed-ups approaching to 220×.

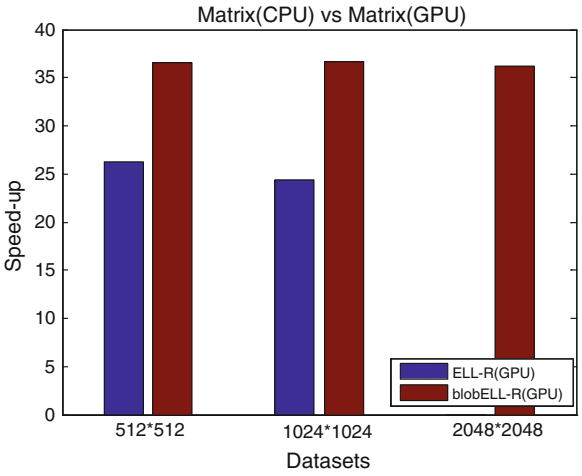


Fig. 41.6 Speed-up factors of matrix SIRT implementation on the GPU versus CPU

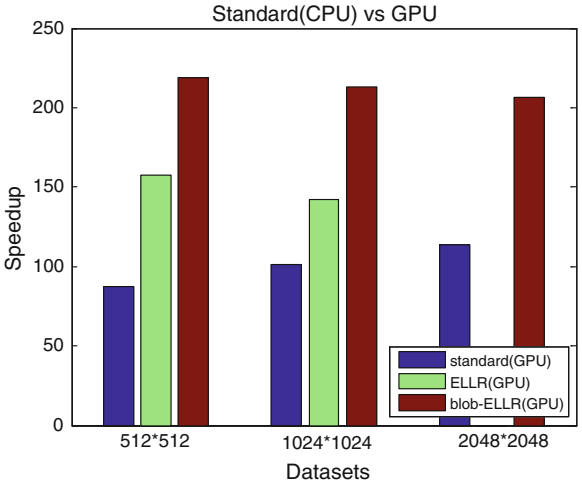


Fig. 41.7 Speed-up factors derived from the different approaches (the standard based on recomputation and the two matrix approaches) on the GPU compared to the standard approach on the CPU

41.6 Conclusion

ET allows elucidation of the molecular architecture of complex biological specimens. Iterative methods using blob basis functions yield better results than other methods extremely under limited-angle and noisy conditions. However, they have not been

used extensively in ET because of their huge computational demands. GPUs have emerged as powerful platforms to cope with the computational requirements.

In this work, we present a multilevel parallel iterative reconstruction scheme to perform the parallelization of ET reconstruction on GPUs. SIRT is adopted as a kind of effective iterative methods since it shows the best performance under the extremely noisy condition. The experimental results on NVIDIA GeForce GTX 295 show that the parallel scheme can achieve high and stable speedups approaching to 220×. The weighted matrix involved in SIRT, nevertheless, is too large for the memory of most GPUs especially for the large images. So we also propose a blob-ELLR data storage format and exploit several geometry related symmetry relationships to reduce the memory demands. In the storage technique, three geometric related symmetry relationships have been exploited to significantly reduce the storage space. It only needs nearly 1/16 of the storage space in comparison with the ELL-R storage structure. The conjunction of the blobELL-R data structure and GPU computing yields significant acceleration compared to the standard and matrix with CRS implementations on CPU.

As for future work, we will further investigate and implement the multilevel parallel scheme with blobELL-R on cluster of GPUs. We would also like to perform more experiments and analyze features of different experimental data to enhance the performance of the method proposed in the paper.

Acknowledgments We would like to thank Prof. Fei Sun and Dr. Ka Zhang in Institute of biophysics for providing the experimental datasets. Work supported by grants National Natural Science Foundation for China (61003164, 61103139, 61202210, 61202059).

References

Andersen AH, Kak AC (1984) Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm. Ultrason Imaging 6:81–94

Bilbao-Castro JR, Carazo JM, Garcia I, Fernandze JJ (2006) Parallelization of reconstruction algorithms in three-dimensional electron microscopy. Appl Math Model 30:688–701

Bisseling RH (2004) Parallel scientific computation. Oxford University Press, Oxford

Castano-Diez D, Mueller H, Frangakis AS (2007) Implementation and performance evaluation of reconstruction algorithms on graphics processors. J Struct Biol 157:288–295

Fernandez JJ (2008) High performance computing in structural determination by electron cryomicroscopy. J Struct Biol 164:1–6

Fernandez JJ, Garcia I, Garazo JM (2004) Three-dimensional reconstruction of cellular structures by electron microscope tomography and parallel computing. J Parallel Distrib Comput 64:285–300

Frank J (2006) Electron tomography: methods for three-dimensional visualization of structures in the cell, 2nd edn. Springer, New York

Gilbert P (1972a) Iterative methods for the 3D reconstruction of an object from projections. J Theor Biol 76:105–117

Gilbert P (1972b) Iterative methods for the 3D reconstruction of an object from projections. J Theor Biol 36:105–117

Herman GT (2009) Image reconstruction from projections: the fundamentals of computerized tomography, 2nd edn. Springer, London

John RR, Ronald FB (1985) Solving elliptic problems using ELLPACK. Springer, New York
Lewitt RM (1992) Alternatives to voxels for image representation in iterative reconstruction algorithms. Phys Med Biol 37:705–716
Matej S, Lewitt RM (1995) Efficient 3D grids for image-reconstruction using spherically-symmetrical volume elements. IEEE Trans Nucl Sci 42:1361–1370
NVIDIA (2008) CUDA Programming Guide. http://www.nvidia.com/cuda
Shufeng S et al (2009) 3D structural investigation of caveolae from porcine aorta endothelial cell by electron tomography. Prog Biochem Biophy 36(6):729–735
Vazquez F, Garzon EM, Fernandez JJ (2009) Accelerating sparse matrix-vector product with GPUs. In: Proceedings of CMMSE09', pp 1081–1092
Vazquez F, Garzon EM, Fernandez JJ (2010) A matrix approach to tomographic reconstruction and its implementation on GPUs. J Struct Biol 170:146–151
Xiaohua W (2009) Modified simultaneous algebraic reconstruction technique and its parallelization in cryo-electron tomography. In: Proceedings of ICPADS09', 2009
Xu W et al (2010) High-performance iterative electron tomography reconstruction wigh long-object compensation using graphics processing units (GPUs). J Struct Biol 171:142–153

Index

D. A. Yuen et al. (eds.), *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, DOI: 10.1007/978-3-642-16405-7,