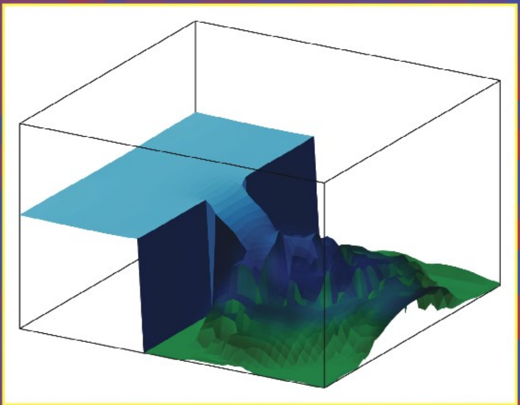
State-of-the-Art Survey

LNCS 6310

Rainer Keller
David Kramer
Jan-Philipp Weiss (Eds.)

Facing the Multicore-Challenge

Aspects of New Paradigms and Technologies in Parallel Computing



Springer

Lecture Notes in Computer Science 6310

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Rainer Keller David Kramer
Jan-Philipp Weiss (Eds.)

Facing the Multicore-Challenge

Aspects of New Paradigms and Technologies in Parallel Computing

Springer

Volume Editors

Rainer Keller
High Performance Computing Center Stuttgart (HLRS)
Universität Stuttgart
Nobelstr. 19
70569 Stuttgart, Germany
E-mail: keller@hlrs.de

David Kramer
Institute of Computer Science and Engineering
Karlsruhe Institute of Technology, Germany
Haid-und-Neu-Str. 7
76131 Karlsruhe, Germany
E-mail: kramer@kit.edu

Jan-Philipp Weiss
Engineering Mathematics and Computing Lab (EMCL)
& Institute for Applied and Numerical Mathematics 4
Karlsruhe Institute of Technology, Germany
Fritz-Erler-Str. 23
76133 Karlsruhe, Germany
E-mail: jan-philipp.weiss@kit.edu

Library of Congress Control Number: 2010935359

CR Subject Classification (1998): D.1-3, C.1.4, C.4, I.3.1, F.2.1, G.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-16232-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16232-9 Springer Berlin Heidelberg New York

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

The proceedings at hand are the outcome of the conference for young scientists titled *Facing the Multicore-Challenge* held at the Heidelberger Akademie der Wissenschaften, March 17–19, 2010. The conference focused on topics related to the impact of multicore and coprocessor technologies in science and for large-scale applications in an interdisciplinary environment. The conference was funded by the Heidelberger Akademie der Wissenschaften and placed emphasis on the support and advancement of young scientists.

The aim of the conference was to bring together leading experts as well as motivated young researchers in order to discuss, recent developments, the present status of the field, and its future prospects the exchange of ideas, in a pleasant atmosphere that stimulates. It was the designated goal to address current issues including mathematical modeling, design of parallel algorithms, aspects of microprocessor architecture, parallel programming languages, compilers, hardware-aware computing, heterogeneous platforms, emerging architectures, tools, performance tuning, and requirements for large-scale applications. This broad range of issues is reflected by the present conference proceedings. The results of the presented research papers clearly show the potential of emerging technologies in the area of multicore and manycore processors that are paving the way towards personal supercomputing. However, many issues related to parallel programming environments, development of portable and future-proof concepts, and the design of scalable and manycore-ready algorithms still need to be addressed in future research. Some of these points are the subject of the presented papers.

These proceedings include diverse and interdisciplinary research work. An assessment of parallel programming environments like the RapidMind platform and the perspective of GPGPU computing in large data centers is presented. The proceedings further address issues of hardware architecture by exploring way-adaptable caches. The management of parallel units is considered in papers on thread affinities and on thread creation. Application aspects on modern processor technologies are investigated for the Cell Broadband Engine by means of the G-means application for data mining and a numerical study on 3D multigrid methods. A complex fluid dynamic application modeled by the lattice Boltzmann equations is considered on multi- and manycore processors like the multicore CPUs, GPUs, and Cell. The potential of FPGA and GPU technology is outlined for a sorting problem. Application studies on GPUs include image segmentation and parallel volume rendering. Furthermore, fault tolerance of pipeline workflows is the subject of presented research work.

The conference organizers and editors would like to thank the Heidelberger Akademie der Wissenschaften for giving us the opportunity to organize this conference at this inspiring venue. Without the funding of the Heidelberger

Akademie der Wissenschaften and the comprehensive support for this fruitful event this conference would not have been possible. In particular, we would like to thank all the friendly people at the Heidelberger Akademie der Wissenschaften for making this conference happen. Last but not least, thank you very much to all the contributors submitting exciting, novel work and providing multi-facetted input to the discussions.

March 2010

Rainer Keller
David Kramer
Jan-Philipp Weiss

Preface from the Heidelberg Academy of Sciences and Humanities

The focus of this publication is: How are innovative computer systems going to have a crucial impact on all branches of science and technology? Multicore systems are opening up new perspectives to cope with challenges which seemed to have been out of range to be mastered up to now. However, they are also posing new challenges in adapting all domains, ranging from mathematical modeling, numerical methods and algorithms to software and hardware design and development. The contributions presented in this volume offer a survey on the state of the art, the concepts and perspectives for future developments. They are an outcome of an inspiring conference conceived and organized by the editors within the junior scientist program of Heidelberg Academy for Sciences and Humanities. The Academy is happy to promote junior scientists getting involved in innovative research and daring to break new ground. Springer deserves high recognition for handling the publication efficiently and thus helping to face the multicore challenges.

Willi Jäger

Acknowledgements

The conference *Facing the Multicore-Challenge* has been kindly funded and supported by the Heidelberger Akademie der Wissenschaften, Karlstr. 4, 69117 Heidelberg. The Shared Research Group 16-1 of Jan-Philipp Weiss has received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and the industrial collaboration partner Hewlett-Packard.

Organization

General Chair

Jan-Philipp Weiss	Karlsruhe Institute of Technology, Germany
Rainer Keller	Oak Ridge National Laboratory, USA
David Kramer	Karlsruhe Institute of Technology, Germany

Mentorship

Willi Jäger	University of Heidelberg, Germany

Program Committee

David A. Bader	Georgia Tech, Atlanta, USA
Michael Bader	University of Stuttgart, Germany
Rosa Badia	Barcelona Supercomputing Center, Spain
Richard Barrett	Oak Ridge National Labs, USA
Mladen Berekovic	TU Braunschweig, Germany
Arndt Bode	TU Munich, Germany
George Bosilca	University of Tennessee Knoxville, USA
Jim Bovay	Hewlett-Packard, USA
Rainer Buchty	Karlsruhe Institute of Technology, Germany
Mark Bull	EPCC, Edinburgh, UK
Hans-Joachim Bungartz	TU Munich, Germany
Franck Cappello	LRI, Université Paris Sud, France
Claudia Fohry	Kassel University, Germany
Richard Graham	Oak Ridge National Labs, USA
Thomas Herault	Université Paris Sud, France
Hans Herrmann	ETH, Zürich, Switzerland
Vincent Heuveline	Karlsruhe Institute of Technology, Germany
Michael Hübner	Karlsruhe Institute of Technology, Germany
Ben Juurlink	TU Berlin, Germany
Wolfgang Karl	Karlsruhe Institute of Technology, Germany
Rainer Keller	Oak Ridge National Labs, USA
Hiroaki Kobayashi	Tohoku University, Japan
Manfred Krafczyk	TU Braunschweig, Germany
Hsin-Ying Lin	Intel, USA
Anton Lokhmotov	Imperial College, London, UK
Dieter an Mey	RWTH Aachen, Germany
Bernd Mohr	FZ Jülich, Germany
Claus-Dieter Munz	Stuttgart University, Germany

Norihiro Nakajima	JAEA and CCSE, Japan
Wolfgang Nagel	TU Dresden, Germany
Christian Perez	INRIA, France
Franz-Josef Pfreundt	ITWM Kaiserslautern, Germany
Rolf Rabenseifner	HLRS, Stuttgart, Germany
Thomas Rauber	Bayreuth University, Germany
Michael Resch	HLRS, Stuttgart, Germany
Gudula Rünger	Chemnitz Technical University, Germany
Olaf Schenk	Basel University, Basel, Switzerland
Martin Schulz	Lawrence Livermore National Labs, USA
Masha Sosonkina	Ames Lab, USA
Thomas Steinke	ZIB, Berlin, Germany
Carsten Trinitis	TUM, Munich, Germany
Stefan Turek	Dortmund University, Germany
Wolfgang Wall	TUM, Munich, Germany
Gerhard Wellein	RRZE, Erlangen, Germany
Josef Weidendorfer	TUM, Munich, Germany
Jan-Philipp Weiss	Karlsruhe Institute of Technology, Germany
Felix Wolf	FZ Jülich, Germany
Stephan Wong	TUD, Delft, The Netherlands

Table of Contents

Invited Talks

Computer Architecture and Parallel Programming

Applications on Multicore

GPGPU Computing

Analyzing Massive Social Networks Using Multicore and Multithreaded Architectures

David Bader

Georgia Institute of Technology, USA

Abstract. Emerging real-world graph problems include detecting community structure in large social networks, improving the resilience of the electric power grid, and detecting and preventing disease in human populations. Unlike traditional applications in computational science and engineering, solving these problems at scale often raises new challenges because of sparsity and the lack of locality in the data, the need for additional research on scalable algorithms and development of frameworks for solving these problems on high performance computers, and the need for improved models that also capture the noise and bias inherent in the torrential data streams. The explosion of real-world graph data poses a substantial challenge: How can we analyze constantly changing graphs with billions of vertices? Our approach leverages the Cray XMT's fine-grained parallelism and flat memory model to scale to massive graphs. On the Cray XMT, our static graph characterization package GraphCT summarizes such massive graphs, and our ongoing STINGER streaming work updates clustering coefficients on massive graphs at a rate of tens of thousands updates per second.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, p. 1, 2010.

MareIncognito: A Perspective towards Exascale

Jesus Labarta

Barcelona Supercomputing Centre, Spain

Abstract. MareIncognito is a cooperative project between IBM and the Barcelona Supercomputing Center (BSC) targeting the design of relevant technologies on the way towards exascale. The initial challenge of the project was to study the potential design of a system based on a next generation of Cell processors. Even so, the approaches pursued are general purpose, applicable to a wide range of accelerator and homogeneous multicores and holistically addressing a large number of components relevant in the design of such systems.

The programming model is probably the most important issue when facing the multicore era. We need to offer support for asynchronous data flow execution and decouple the way source code looks like and the way the program is executed and its operations (tasks) scheduled. In order to ensure a reasonable migration path for programmers the execution model should be exposed to them through a syntactical and semantic structure that is not very far away from current practice. We are developing the StarSs programming model which we think addresses some the challenges of targeting the future heterogeneous / hierarchical multicore systems at the node level. It also integrates nicely into coarser level programming models such as MPI and what is more important in ways that propagate the asynchronous dataflow execution to the whole application. We are also investigating how some of the features of StarSs can be integrated in OpenMP.

At the architecture level, interconnect and memory subsystem are two key components. We are studying in detail the behavior of current interconnect systems and in particular contention issues. The question is to investigate better ways to use the raw bandwidth that we already have in our systems and can expect to grow in the future. Better understanding of the interactions between the raw transport mechanisms, the communication protocols and synchronization behavior of applications should lead to avoid an exploding need for bandwidth that is often claimed. The use of the asynchronous execution model that StarSs offers can help in this direction as a very high overlap between communication and computation should be possible. A similar effect or reducing sensitivity to latency as well as the actual off chip bandwidth required should be supported by the StarSs model.

The talk will present how we target the above issues, with special details on the StarSs programming model and the underlying idea of the project of how tight cooperation between architecture, run time, programming model, resource management and application are needed in order to achieve in the future the exascale performance.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, p. 2, 2010.

The Natural Parallelism

Robert Strzodka

Max Planck Institut Informatik, Saarbruecken, Germany

Abstract. With the advent of multi-core processors a new unwanted way of parallel programming is required which is seen as a major challenge. This talk will argue in exactly the opposite direction that our accustomed programming paradigm has been unwanted for years and parallel processing is the natural scheduling and execution model on all levels of hardware.

Sequential processing is a long outdated illusionary software concept and we will expose its artificiality and absurdity with appropriate analogies of everyday life. Multi-core appears as a challenge only when looking at it from the crooked illusion of sequential processing. There are other important aspects such as specialization or data movement, and admittedly large scale parallelism has also some issues which we will discuss. But the main problem is changing our mindset and helping others to do so with better education so that parallelism comes to us as a friend and not enemy.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, p. 3, 2010.

RapidMind: Portability across Architectures and Its Limitations

Iris Christadler and Volker Weinberg

Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften,
D-85748 Garching bei München, Germany

Abstract. Recently, hybrid architectures using accelerators like GPGPUs or the Cell processor have gained much interest in the HPC community. The "RapidMind Multi-Core Development Platform" is a programming environment that allows generating code which is able to seamlessly run on hardware accelerators like GPUs or the Cell processor and multi-core CPUs both from AMD and Intel. This paper describes the ports of three mathematical kernels to RapidMind which have been chosen as synthetic benchmarks and representatives of scientific codes. Performance of these kernels has been measured on various RapidMind backends (cuda, cell and x86) and compared to other hardware-specific implementations (using CUDA, Cell SDK and Intel MKL). The results give an insight into the degree of portability of RapidMind code and code performance across different architectures.

1 Introduction

The vast computing horsepower which is offered by hardware accelerators and their usually good power efficiency has aroused interest of the high performance computing community in these devices. The first hybrid system which entered the Top500 list [1] was the TSUBAME cluster at Tokyo Institute of Technology in Japan. Several hundred Clearspeed cards were used to accelerate an Opteron based cluster; the system was ranked No. 9 in the Top500 list in November 2006. Already in June 2006, a sustained Petaflop/s application performance was firstly reached with the RIKEN MD-GRAPE 3 system in Japan, a special purpose system dedicated for molecular dynamics simulations. In 2008, the first system ever to reach a sustained High Performance LINPACK (HPL) performance of more than one Petaflop/s was "Roadrunner", the No. 1 system on the lists in July 2008 and November 2008. Roadrunner is a hybrid system based on Opteron processors and accelerated with PowerXCell8i processors, a variant of the Cell B.E. (Broadband Engine) with increased double-precision capability.

However, applicability of hardware accelerators for general-purpose HPC systems is still a source of debate. In 2008, the landscape was quite diverse; many different hardware solutions existed (Cell, Nvidia and AMD/ATI GPUs, ClearSpeed accelerator boards, FPGA based systems) and every system had its own programming language and paradigm. At the same time, the x86 processors started to become multi-core processors and first HPC systems were based on

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 4–15, 2010.

hundred thousands of cores. Improving the scalability of HPC codes to be able to utilize the increased core counts was already difficult for the scientific communities; trying to add support for one of the new accelerators was a huge porting effort with a high risk: what if either the hardware or the software would not be supported on the long run? Solutions which offered support for different hardware architectures became appealing.

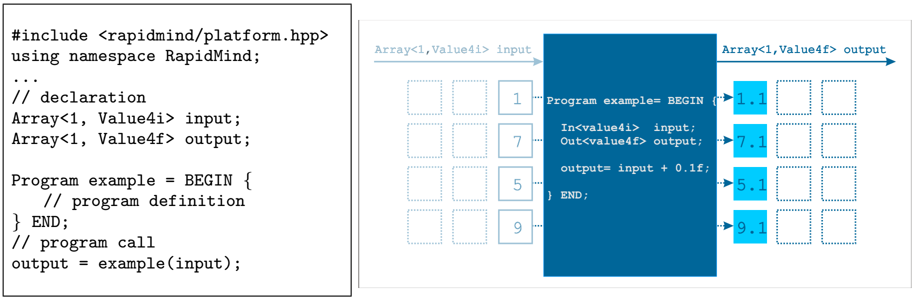
While in the meantime several solutions (e.g. OpenCL [2], PGI accelerator compiler [3], CAPS hmpp [4], StarSs [5]) exist which provide an abstraction of the underlying hardware characteristics from the programmer, the situation was different two years ago: RapidMind Inc. was one of the first companies providing support for general purpose computing on graphic processing units, nowadays known as GPGPUs. RapidMind started in 2004 based on the academic research related to the Sh project [6] at the University of Waterloo. Their work was started at a time when the first "programmable" GPUs were just released and the only way to program these devices was by using "shading languages". Already at that time people tried porting simulation codes to GPUs [7]. Since then, RapidMind has subsequently added the Cell processor backend (2007) and the x86 multi-core processor backend with the rise of multi-core processor CPUs for the consumer market (2008). In 2009, version 4.0 was released which introduced the cuda backend, necessary to support double-precision arithmetic on GPUs. Even today, RapidMind is still the only product that fully supports Cell, GPUs and multi-core CPUs. All other solutions are either limited by the hardware which they support or require an adaptation of the code.

At SC06 a paper was published which showed impressive performance gains by using RapidMind for porting three algorithms (SGEMM, FFT and Black-Scholes) to the GPU [8]. This is a follow-on work assessing the state-of-the-art three years later. However, the main reason for choosing RapidMind for a deeper investigation has been its programming paradigm which differs from serial programming languages and abstracts the massive parallelism of the underlying hardware more than any other language concept currently discussed for HPC.

2 Overview

2.1 Software

The "RapidMind Multi-Core Development Platform" promises easy and portable access not only to multi-core chips from Intel and AMD but also to hardware accelerators like GPUs and Cell. The basic concept of the RapidMind language is called "data-stream processing"; a powerful technology to express data parallelism. A simple example of a RapidMind program is given in Fig. 1 (a). Figure 1 (b) represents a schematic view of the executed stream program. A call to a RapidMind program can be inserted in any valid C++ program and needs to include the RapidMind library in the header of the file and during linkage. Unless specified explicitly, the RapidMind runtime environment will automatically search for available accelerator hardware and compile the program at runtime using the RapidMind backend for the detected hardware.



```
#include <rapidmind/platform.hpp>
using namespace RapidMind;
...
// declaration
Array<1, Value4i> input;
Array<1, Value4f> output;

Program example = BEGIN {
   // program definition
} END;
// program call
output = example(input);
```

(a)

(b)

Fig. 1. RapidMind programming scheme

RapidMind adds special types and functions to C++ which allow the programmer to define sequences of operations (RapidMind programs) on streams of data (special arrays). With these, data dependencies and data workflows can be easily expressed and will naturally contain all necessary information for an efficient (data-) parallelization. The compiler and the runtime environment then have maximum information to decide how to auto-parallelize the code.

The structure of RapidMind code forces the programmer to decide early in the development process which operations could be performed in parallel without any side-effects. This usually results in many small code snippets that can run in parallel which is optimal to fill the pipeline of a GPU or other massively parallel devices.

2.2 Hardware

Three different platforms are used for the performance measurements. An Nvidia Tesla based system is used to measure the cuda backend from RapidMind against implementations based on CUDA and the CUDA libraries cuBLAS and cuFFT. Tesla is Nvidia's first dedicated general purpose GPU with enhanced double-precision capability. A C1060 supports partly IEEE-754, consists of 240 thread processors with an overall performance of 78 GFlop/s in double-precision and 933 GFlop/s in single-precision. One Nvidia Tesla S1070 1U rack consists of four

Table 1. Hardware overview

Hardware	SP peak perf.	DP peak perf.
1 C1060 GPU	933 GFlop/s	78 GFlop/s
1 Tesla S1070	4140 GFlop/s	345 GFlop/s
Nehalem-EP (2.53 GHz, 1 core)	20 GFlop/s	10 GFlop/s
Nehalem-EP (2.53 GHz, 8 cores)	162 GFlop/s	81 GFlop/s
1 PowerXCell8i (8 SPUs)	205 GFlop/s	102 GFlop/s
1 QS22-blade 2 PowerXCell8i (16 SPUs)	410 GFlop/s	205 GFlop/s

C1060 computing processors with a total single-precision performance of around 4 TFlop/s.

An IBM QS22-blade based system is used to compare RapidMind's cell backend with code using Cell intrinsics which is taken from the SDK. Each QS22-blade hosts two PowerXCell8i, the processors used to accelerate Roadrunner [9]. Each PowerXCell8i is running at 3.2 GHz, is partly IEEE-754 conform and has a single-precision peak performance of 204.8 GFlop/s and a double-precision peak performance of 102.4 GFlop/s. A QS22-blade has therefore a total of slightly more than 400 GFlop/s single-precision performance. The main difference between the Cell processor and GPUs or current multi-core CPUs is its inhomogeneity; eight synergistic processor units (SPUs) are added to one PowerPC processor unit (PPU). The Cell processor has a very good performance per Watt ratio and the 6 most energy efficient supercomputers, as ranked by Green500 [10] in November 2009, are based on PowerXCell8i technology.

RapidMind's x86 backend is benchmarked against code using Intel's Math Kernel Library (MKL) on one of the latest Intel processors, a Xeon E5540 known as "Nehalem-EP". A Nehalem-EP core running at 2.53 GHz has a single-precision peak performance slightly above 20 GFlop/s and a double-precision peak performance of around 10 GFlop/s. One Nehalem-EP node consists of 2 sockets with four cores per socket. A Nehalem-EP node with 8 cores reaches 162 GFlop/s single and 81 GFlop/s double-precision performance.

The performance figures of all three architectures are summarized in Table 1. Since the double-precision peak performance of one Nehalem-EP node (8 cores, 81 GFlop/s) is quite comparable with the double-precision performance of 1 Nvidia C1060 GPU (78 GFlop/s) and 1 PowerXCell8i (102 GFlop/s) we tried to compare these platforms directly where possible.

3 The RapidMind Ports and Their Performance

To judge the suitability of recent accelerator hardware for scientific computing and high-performance computing, three mathematical kernels from the Euroben benchmark suite [11] have been chosen:

- mod2am: a dense matrix-matrix multiplication,
- mod2as: a sparse matrix-vector multiplication,
- mod2f: a one-dimensional Fast Fourier Transformation (FFT).

The kernels have been selected to show both the advantages and the pitfalls of current accelerators. They are representatives of three (dense linear algebra, sparse linear algebra and spectral methods) of the "seven dwarfs", an ontology for scientific codes introduced by [12]. According to Fig. 11 in [13] these three dwarfs account for approximately one third of the workload of current European HPC Tier-1 centers. The selection of kernels was performed by the EU FP7-project PRACE, published in [14] and should be extended to cover all important dwarfs in the future.

3.1 Dense Matrix-Matrix Multiplication (mod2am)

The dense matrix-matrix multiplication ($C = A \times B$) is one of the most basic algorithms used in scientific computing. It is the basis of the High Performance LINPACK code, which determines the Top500 rank of a system. The schoolbook version of the algorithm is composed of three nested for-loops. Many sophisticated optimization strategies exist, and one of the fastest implementations is the MKL version. Making use of the MKL functions is straightforward and basically needs a call to `cblas_dgemm` (double-precision arithmetic) or `cblas_sgemm` (single-precision arithmetic).

A first implementation in RapidMind is straightforward. In a first step, the RapidMind data types must be used to express the matrices A (of size $m \times l$), B ($l \times n$) and C ($m \times n$). All matrices can be represented by two-dimensional arrays of floating point data:

```
Array<2,Value1f> A(m,l);
Array<2,Value1f> B(l,n);
Array<2,Value1f> C(m,n);
```

In a second step the RapidMind program `mxm` needs to be declared. Since there are no clear data streams which could be fed into the program a helper index array is used. This index array ensures that the defined program `mxm` can sum up the corresponding entries of the input matrices A and B. All matrices are automatically transferred to the GPU memory at execution time. The RapidMind control flow construct `RM_FOR` is used to allow manipulation of the streamed data.

```
Program mxm = BEGIN {
    In<Value2i> ind;
    Out<Value1f> c = Value1f(0.);

    Value1i k;
    // Computation of C(i,j)
    RM_FOR (k=0, k < Value1i(l), k++) {
        c += A[Value2i(ind(0),k)]*B[Value2i(k,ind(1))];
    } RM_ENDFOR;
} END;
```

The call to the RapidMind program then looks as follows:

```
C= mxm(grid(m,n));
```

The call to the RapidMind function `grid(m,n)` generates a virtual helper array of size $m \times n$ which does not require additional storage. The whole helper array is automatically initialized with integers from $(0, 0)$ to (m, n) and is directly passed to the program and used for the index computation.

After this first naive approach a second, *GPU-optimized version* of the matrix multiplication has been produced. This version is based on code available at the

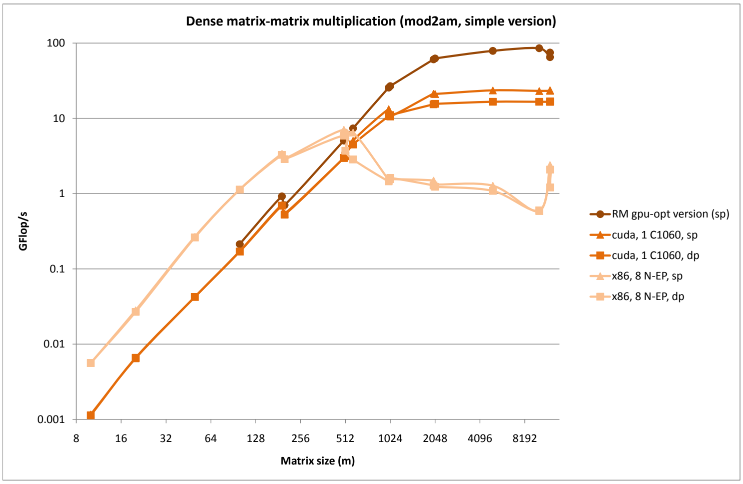


Fig. 2. Performance comparison of the simple mod2am version on various RapidMind backends and associated hardware. The simple version is also compared to the GPU-optimized version running on 1 C1060 GPU in single-precision.

RapidMind developer portal [15]. The basic difference between both versions is the fact, that the GPU-optimized version operates on arrays of `Value4f`, to optimal use the GPU vector registers; 4 × 4 submatrices are multiplied and accumulated.

Figure 2 [1] shows the performance of the simple version using the cuda and x86 RapidMind backends and compares the single-precision cuda backend performance with the GPU-optimized version. It can be seen that the GPU-optimized version is indeed four times faster than the naive approach for single-precision arithmetic. This means that the use of `Value4f` instead of `Value1f` really improves performance. It is important to note, that neither the language nor the program definition of the simple approach should prevent the compiler from doing this optimization by itself.

Measurements for double-precision reveal that the simple approach is actually faster than the GPU-optimized version. This is counterintuitive and only becomes understandable if one takes into account, that the cuda backend is the latest RapidMind backend and was introduced with version 4.0 in July 2009. The target of this version was to enable RapidMind to support Nvidia Tesla cards; a RapidMind version with improved performance of the cuda backend was scheduled for version 4.1.

Figure 3 shows the performance of the GPU-optimized version on various backends and compares it with hardware-specific languages (CUDA and MKL). It shows that the performance of the RapidMind implementation is more than an

[1] Time is always measured for the execution of the whole kernel. This includes the time to transfer data between host and accelerator for GPU and Cell results. The y-axis uses log-scale to better cover the whole performance range for all matrix sizes.

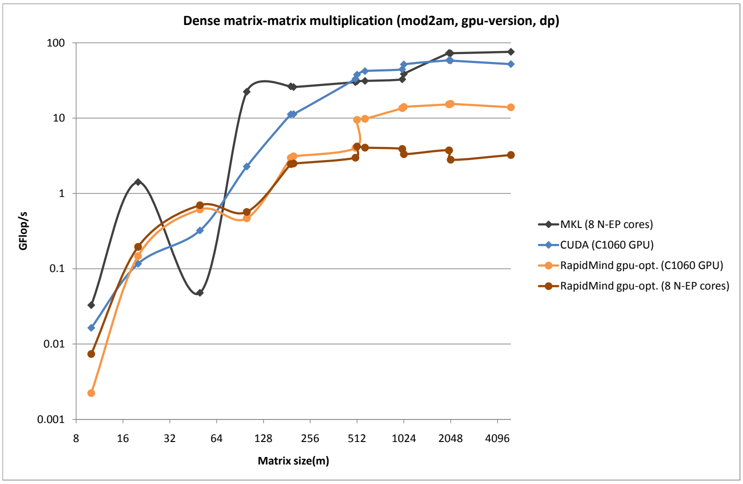


Fig. 3. Performance comparision of the GPU-optimized version on various backends. Performance measurements have been performed both on an Nvidia GPU and a Nehalem-EP socket with eight cores. The RapidMind version is compared to a CUDA version based on cuBLAS and an MKL implementation of the dense matrix-matrix multiplication. Performance measurements are based on double-precision arithmetic.

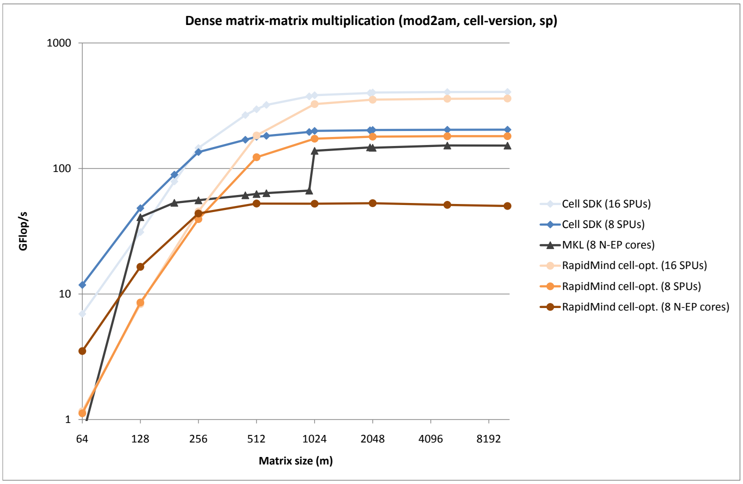


Fig. 4. Performance comparison of the Cell-optimized version on various RapidMind backends. Performance measurements have been performed on 1 PowerXCell8i (8 SPUs), 1 QS22-blade (16 SPUs) and an 8-core Nehalem-EP node. The RapidMind version is compared to a dense matrix-matrix multiplication example from the Cell SDK and the MKL implementation. All performance results are based on single-precision arithmetic.

order of magnitude slower than the MKL implementation, while the difference between RapidMind and CUDA performance is only a factor of 3. Comparing Fig. 2 with Fig. 3 reveals that the performance difference between the two RapidMind implementations varies extremely for certain matrix sizes, although the implementations vary only slightly.

The performance of both the simple version and the GPU-optimized version are not able to deliver decent performance on the Cell platform. A third version *optimized for the Cell processor* is based on another code available through the RapidMind developer portal. This time, computation is performed using a block partitioning of 64 by 64 blocks. All matrices are in a "block swizzled" format so that these blocks are contiguous in memory. The computations and memory transfers are overlapped using double buffering and are partly based on the matrix-matrix multiplication example from the IBM Cell SDK (`/opt/cell/sdk/src/demos/matrix_mul/`). The Cell SDK version is also used for performance comparison.

Figure 4 gives an insight into the performance of the Cell-optimized version. Again the RapidMind figures have been compared with implementations in other languages. Since the Cell SDK version is based on single-precision arithmetic, it has been compared to single-precision results obtained with the RapidMind cell and x86 backends and an SGEMM implementation using MKL on 8 Nehalem-EP cores. This time, the RapidMind version is able to nearly meet the performance of the hardware-specific and highly optimized Cell SDK version; it reaches 88% of the SDK version. However, this comes at the price of a hardware-specific RapidMind implementation and contradicts the idea of seamlessly portable code.

In conclusion, the three different implementations illustrate the current limitations of code and performance portability. Hardly any problems were experienced when moving the code to other platforms, but in many cases the performance was not predictable. Tuning the code to better exploit certain characteristics of the employed hardware normally yields better performance but requires to stick with this hardware. The idea behind RapidMind is that the language and program definitions are generic enough to allow the compiler to do hardware-specific optimizations itself.

3.2 Sparse Matrix-Vector Multiplication (mod2as)

Sparse linear algebra is another building block of many scientific algorithms. The sparse matrix-vector multiplication exposes a low computational intensity and is usually memory bound. It is a good example for code that will not perform well on recent hardware accelerators on which the transfer between the x86 host memory and the accelerator memory is a severe bottleneck. Even x86 hardware will only run at a small percentage of its theoretical peak performance. While mod2am reaches more than 90% of peak, mod2as runs at rates less than 5% of peak on Nehalem-EP. Since this algorithm is not well suited for accelerators, we provided only one RapidMind mod2as implementation and put a special focus on the performance achieved with the x86 backend on Nehalem-EP (shown in Fig. 5).

The implementation of mod2as is based on [16]. The input matrix A of mod2as is stored in a 3-array variation of the CSR (compressed sparse row) format which can be easily transferred to RapidMind. The array `matvals` contains the non-zero elements of A, the element `i` of the integer array `indx` is the number of the column in A that contains the `i`-th value in the `matvals` array and element `j` of the integer array `rowp` gives the index of the element in the `matvals` array that is the first non-zero element in row `j` of A. The input and output vectors are declared as:

```
Array<1,Value1i> indx(nelmts);
Array<1,Value1i> rowp(nrows+1);
Array<1,Value1f> matvals(nelmts);

Array<1,Value1f> invec(ncols);
Array<1,Value1f> outvec(nrows);
```

Once again a helper array based on a call to `grid(nrows)` is created and used as input vector to allow the correct index computation. The RapidMind program is very clean: using RapidMind's `RM_FOR()` control structure, the program loops over one row of the input matrix and computes the matrix-vector product.

```
Program spMXV = BEGIN {
    In<Value1i> i;
    Out<Value1f> c;

    c = Value1f(0.);
    Value1i j;

    RM_FOR(j=rowp[i], j < rowp[i+1] , j++) {
        c += matvals[j] * invec[indx[j]];
    } RM_ENDFOR;

} END;
```

3.3 One-Dimensional Fast Fourier Transformation (mod2f)

The Fast Fourier Transformation (FFT) is widely used in many scientific programs. Its computational intensity is not as high as for mod2am, but is already in a range where accelerators should be beneficial. The RapidMind version of mod2f computes the FFT using a split-stream algorithm as described in [17]. The implementation is a straightforward conversion of a one butterfly Cooley-Tukey radix-2 FFT; the butterfly kernels are defined as RapidMind programs. Figure 6 gives the achieved performance for different platforms and shows that one implementation is able to deliver performance on at least two backends.

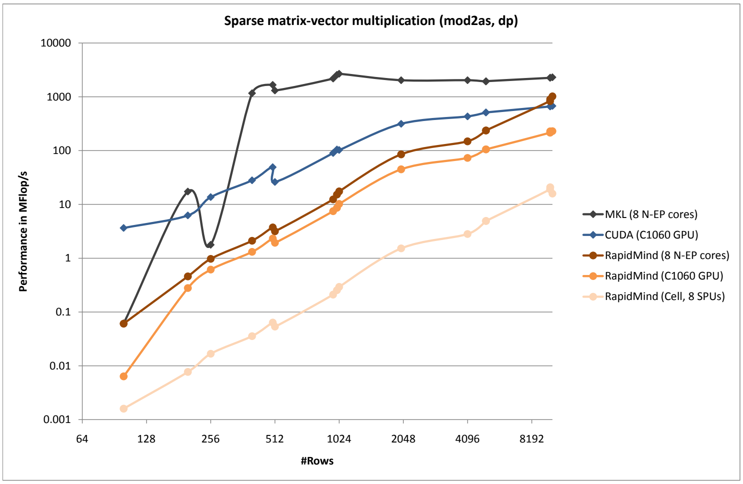


Fig. 5. Performance comparison of the sparse matrix-vector multiplication. Performance results for the RapidMind implementation on various backends are given and compared with implementations in CUDA and based on MKL. The difference between the MKL and the RapidMind x86-results is less than a factor of 3 for big matrix sizes.

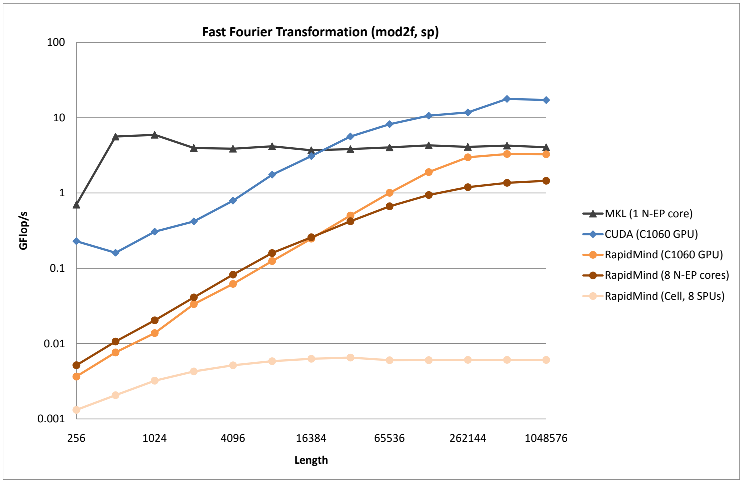


Fig. 6. Performance comparison of the one-dimensional Fast Fourier Transformation. The RapidMind implementation is compared to a CUDA version of mod2f based on cuFFT and the corresponding MKL implementation. The gap between the RapidMind cuda-results and the highly-optimized cuFFT version is a factor of 5, the difference between the x86-results and the MKL version is again less than a factor of 3.

4 Conclusions and Future Work

The work presented in this paper has shown that RapidMind really offers code portability across various architectures, both multi-core x86 CPUs and accelerators like GPUs or the Cell processor. Using RapidMind for the Euroben kernels has been straightforward: the code development of the first naive implementation took only a few days for each. Adapting the original version to new backends comes practically for free and is a matter of hours and of getting used to the new environments.

However, performance portability differs: code written naturally without a deep understanding of the hardware and RapidMind's internal mode of operation will not deliver optimal performance in most circumstances and hardly exploit the potential of the hardware. For mod2am, the highly optimized cell-version is able to reach 88% of the SDK implementation but will deliver poor performance when used on GPUs. The fastest mod2am implementation using CUDA is three times faster than any RapidMind code. For none of the used benchmarks, RapidMind code was able to fully reach the performance of hardware-specific implementations. This is not a big surprise, since it is one of the drawbacks of the achieved code portability. But it is important to state, that the language design optimally supports the compiler. To efficiently use this information to full capacity requires that many people constantly improve all backends, adapting them to the latest hardware and its accompanying language features.

Recently, RapidMind Inc. has been acquired by Intel. Their product will dissolve in Intel's new language Ct (C for throughput computing) [18]. The basic concepts of both languages have always been very similar. The acquisition has pros and cons: on one hand, it is up to speculations if – or when – Ct will support non-Intel architectures. On the other hand, Intel has much experience with mantaining high-performance compilers and analyzing tools.

Future work will focus on Intel's Ct and other approaches that are able to deliver support for multiple accelerators. This might include OpenCL, the PGI accelerator compiler, hmpp from CAPS and the StarSs concept. Our work will focus on the question of portability, both in terms of source code and in terms of achievable performance. The number of kernels will be increased to get a better coverage of the "Berkeley dwarfs".

Acknowledgements

This work was financially supported by the KONWIHR-II project "OMI4papps" and by the PRACE project funded in part by the EU's 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528. We specially thank our colleague Hans Hacker for providing the CUDA ports and performance figures, and JSC and CEA for access to their accelerator systems and support.

References

1. The Top500 supercomputing sites, http://www.top500.org/
2. OpenCL, http://www.khronos.org/opencl/
3. PGI Accelerator Compiler, http://www.pgroup.com/resources/accel.htm
4. CAPS hmpp workbench, http://www.caps-entreprise.com/hmpp.html
5. Planas, J., Badia, R.M., Ayguade, E., Labarta, J.: Hierarchical Task-Based Programming with StarSs. The International Journal of High Performance Computing Applications 23(3), 284–299 (2009)
6. Sh project, http://libsh.org/
7. Ernst, M., Vogelgsang, C., Greiner, G.: Stack Implementation on Programmable Graphics Hardware. In: VMV 2004, pp. 255–262 (2004)
8. McCool, M., Wadleigh, K., Henderson, B., Lin, H.-Y.: Performance evaluation of GPUs using the RapidMind development platform. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (2006)
9. Los Alamos Lab: Roadrunner, http://www.lanl.gov/roadrunner/
10. The Green500 list of energy efficient supercomputers, http://www.green500.org/
11. The Euroben benchmark home page, http://www.euroben.nl/
12. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley (2006), http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf
13. Simpson, A., Bull, M., Hill, J.: PRACE Deliverable D6.1 Identification and Categorisation of Applications and Initial Benchmarks Suite, http://www.prace-project.eu/documents/Identification_and_Categorisation_of_Applications_and_Initial_Benchmark_Suite_final.pdf
14. Cavazzoni, C., Christadler, I., Erbacci, G., Spiga, F.: PRACE Deliverable D6.6 Report on petascale software libraries and programming models (to appear), http://www.prace-project.eu/documents/public-deliverables-1/
15. RapidMind developer site, https://developer.rapidmind.com/sample-code/matrix-multiplication-samples/rm-sgemm-gpu-5938.zip
16. Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA, http://www.nvidia.com/object/nvidia_research_pub_001.html
17. Jansen, T., von Rymon-Lipinski, B., Hanssen, N., Keeve, E.: Fourier Volume Rendering on the GPU Using a Split-Stream-FFT. In: VMV 2004, pp. 395–403 (2004)
18. Intel Ct Technology, http://software.intel.com/en-us/data-parallel/

A Majority-Based Control Scheme for Way-Adaptable Caches

Masayuki Sato[1], Ryusuke Egawa[2,3],
Hiroyuki Takizawa[1,3], and Hiroaki Kobayashi[2,3]

[1] Graduate School of Information Sciences, Tohoku University
[2] Cyberscience Center, Tohoku University
[3] JST CREST
{masayuki@sc.,egawa@,tacky@,koba@}isc.tohoku.ac.jp

Abstract. Considering the trade-off between performance and power consumption has become significantly important in microprocessor design. For this purpose, one promising approach is to employ way-adaptable caches, which adjust the number of cache ways available to a running application based on assessment of its working set size. However, in a very short period, the estimated working set size by cache access locality assessment may become different from that of the overall trend in a long period. Such a locality assessment result will cause excessive adaptation to allocate too many cache ways to a thread and, as a result, deteriorate the energy efficiency of way-adaptable caches. To avoid the excessive adaptation, this paper proposes a majority-based control scheme, in which the number of activated ways is adjusted based on majority voting of locality assessment results of several short sampling periods. By using majority voting, the proposed scheme can make way-adaptable caches less sensitive to the results of the periods including exceptional behaviors. The experimental results indicate that the proposed scheme can reduce the number of activated ways by up to 37% and on average by 9.4%, while maintaining performance compared with a conventional scheme, resulting in reduction of power consumption.

1 Introduction

In the last four decades, a continuous increase in the number of transistors on a chip realizes high-performance microprocessors. However, this advance also induces high energy consumption. Considering the trade-off between performance and energy has become significantly important in modern microprocessor design. Especially, power management of on-chip caches has been attracting attention, because a large on-chip cache is essential to achieve high performance but also increases power consumption.

Among a lot of studies on energy-efficient cache managements, one promising approach is a *way-partitioned cache* [1], in which each way of a set-associative cache is independently managed to find a good trade-off between performance and power consumption. One of such mechanisms called *dynamic cache partitioning mechanisms* for multi-core processors [2,3,4] can exclusively allocate ways to

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 16–28, 2010.

each thread and avoid *inter-thread kickouts* [5], which cause performance degradation on multi-core processors. Albonesi et al. have discussed the effects of changing the number of ways on the performance and power consumption [6]. *The way-adaptable cache* [7] dynamically allocates an appropriate number of cache ways to each thread, and also inactivates unused cache ways for energy saving. *Power-aware dynamic cache partitioning* [8] was also proposed to achieve both resource partitioning among threads and power management.

In these energy-efficient cache mechanisms, it is important to correctly estimate the amount of the cache resource required by individual threads. Almost all the mechanisms estimate the working set size of a thread by sampling cache accesses in a certain period, and estimate the minimum number of cache ways to keep the performance. As a result, the number of cache ways allocated to each thread is decided so as to provide the cache capacity larger than the working set size. If there are unused cache ways not allocated to any thread, power supply to those ways is cut off to reduce the energy consumption

In this paper, we first discuss estimation accuracy of the cache capacity necessary for a thread. Here, we assume the way-adaptable cache [7] as the cache power management mechanism. Observing the cache access behavior during a certain period, we point out that irregular accesses with a little impact on the overall performance temporarily happen. In addition, if such accesses are included in the statistic data used to estimate the working set size, the required cache capacity is inaccurately estimated, resulting in degradation of energy efficiency.

To solve the problems caused by the sensitiveness to these unexpected and irregular accesses, this paper proposes a new control scheme for the way-adaptable cache. In the scheme, the number of activated ways is decided by the majority voting of working set assessment results of several short sampling periods to eliminate the effect of exceptional cache behavior occurring in a short period. Using the proposed scheme in the way-adaptable cache, the number of ways required by each thread is reduced without performance degradation. Consequently, this will lead to a reduction in power consumption and improvement of energy efficiency of caches.

The rest of this paper is organized as follows. In Section 2, we show the way-adaptable cache mechanism and discuss the effect of exceptional and temporary cache access behavior. In Section 3, we propose a new control scheme for way-adaptable caches based on the discussions in Section 2. In Section 4, the proposed scheme is evaluated with the number of activated ways on the way-adaptable cache. Finally, Section 5 concludes this paper.

2 The Way-Adaptable Cache Mechanism

2.1 Mechanism Overview

This paper assumes the situation where each thread is executed on the way-adaptable cache mechanism [7] under the single-thread execution environment. Figure 1 shows the basic concept of the way-adaptable cache. The mechanism is designed for a set-associative cache and manages each way individually. The

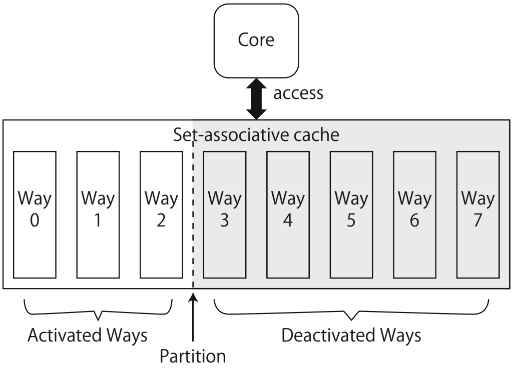
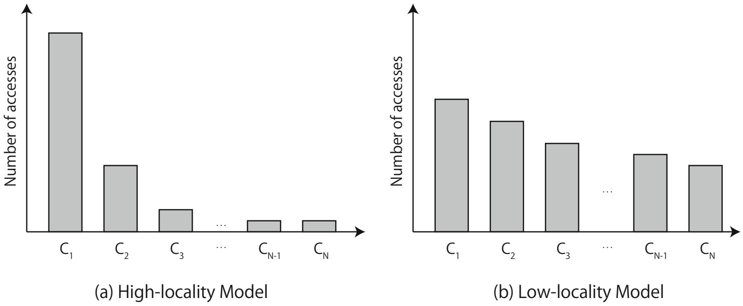


Fig. 1. Basic concept of the way-adaptable cache (in the case of eight ways)



(a) High-locality Model

(b) Low-locality Model

Fig. 2. Stack Distance Profiling

mechanism estimates the minimum number of required ways to run a thread, allocates them to keep the performance, and inactivates unused ways. When inactivating these ways, the mechanism writes back dirty lines in the ways to the lower-level memory hierarchy. After that, power supply to the ways are disabled by power-gating to reduce power consumption.

Metric for Locality Assessment. To estimate the number of ways required by a thread, a metric is required to judge whether the thread requires more ways or not. The way-adaptable cache mechanism uses *stack distance profiling* [9] for this purpose. From the profiling result, metric D for locality assessment is calculated.

Figure 2 shows two examples of the results of stack distance profiling. Let $C_1, C_2, ..., C_n$ be N counters for an N-way set-associative cache with the LRU replacement policy. C_i counts the number of accesses to the i-th line in the LRU stack. Therefore, counters C_1 and C_n are used to count the numbers of accesses to MRU lines and LRU lines, respectively. If a thread has a high locality, cache accesses are concentrated on the MRU lines as shown in Figure 2(a). In this case, the ratio of LRU accesses to MRU accesses becomes lower. However, if

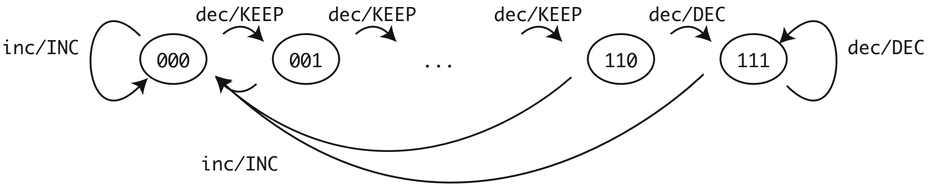


Fig. 3. The 3-bit state machine to decide the way-adaptation

a thread has a low locality, cache accesses are widely distributed from MRU to LRU as shown in Figure 2(b); the ratio of LRU accesses to MRU accesses becomes higher. From the above observations, this ratio, which is defined as D in the following equation, represents the cache access locality of a thread.

$$D = \frac{LRU_{count}}{MRU_{count}}. \tag{1}$$

A Control Mechanism of the Way-Adaptable Cache. To adjust the number of activated ways to dynamic phase changes of an application, the mechanism samples cache accesses in a certain period, called a *sampling period*. After the sampling period, the mechanism has an opportunity to judge a demand of cache resizing, i.e., changing the number of activated ways. If the mechanism actually judges that resizing is needed, the number of activated ways is changed one by one. We call this opportunity an *adaptation opportunity*. The sampling period and the adaptation opportunity are alternately repeated as the execution proceeds.

At the adaptation opportunity, the mechanism uses a cache-resizing control signal of a state machine to judge whether the number of activated ways should be changed or not. The state machine records the locality assessment results of the past sampling periods as its state. The locality assessment result of each sampling period is input to the state machine at the following adaptation opportunity. These past results are used for stabilizing the adaptation control.

In the locality assessment of each sampling period, D is compared with thresholds t_1, t_2 ($t_1 < t_2$). If $D < t_1$, the mechanism sends a result *dec* to the state machine to suggest decreasing the number of ways. On the other hand, if $t_2 < D$, the mechanism sends a result *inc* to suggest increasing the number of ways. If $t_1 < D < t_2$, the mechanism sends a result *keep* to the state machine to suggest keeping the current number of ways.

Figure 3 shows the state transition diagram of the 3-bit state machine used in this mechanism. The state machine changes its state when signal *inc* or *dec* is input. When *inc* is given to the state machine, it outputs the cache up-sizing control signal *INC* and then always transits to State 000 from any state. However, in the case of *dec* given, the machine works conservatively to generate the down-sizing signal *DEC*. Before moving into State 111 for generating the *DEC* signal, the machine transits to intermediate states from 001 to 110 to judge the continuity of the down-sizing requests. During these states, it outputs *KEEP* to keep the current activated ways. After continuing *dec* requests, the machine eventually outputs the *DEC* signal for down-sizing and then transits to State 111. That is, *dec* does not immediately change the number of ways but *inc* does. According to this behavior of the state machine, the mechanism can avoid performance degradation caused by insensitivity to *inc* and frequent changes of the number of activated ways.

2.2 Exceptional Disturbances of Cache Accesses

We define temporal and exceptional cache behaviors that generate a lot of LRU accesses in a short period as *exceptional disturbances*, which have little impact on the overall performance in thread execution. When exceptional disturbances happen, D becomes large and hence the cache access locality will be judged excessively low, even if the cache access locality is almost always high except the short period of exceptional disturbances, and thus additional ways are not required. Figure 4 shows the examples of two sampling periods. In the figure, the sampling period A does not include exceptional disturbances. On the other hand, the sampling period B includes exceptional disturbance. Comparing average locality assessment metric D in each period, average D of the period B is obviously larger than that of the period A. In this case, the way-adaptable cache will increase the number of activated ways, even though this does not increase the thread performance but the power consumption.

To reduce the effect of exceptional disturbances, one solution is to increase the interval, because average D of the sampling period B in Figure 4 decreases as the sampling period B becomes longer. However, there are drawbacks of a long sampling period. If the sampling period becomes long, the probability of including multiple exceptional disturbances in one sampling period increases. If multiple exceptional disturbances are in a long period, the effect of reducing average D of a long period gets balanced out, and average D in a long period is eventually comparable to average D in a short period. As a result, the effect of exceptional disturbances in locality assessment at an adaptation opportunity is not reduced even if using a long period, and the number of activated ways cannot be reduced. Moreover, once the number of ways is increased by exceptional disturbances, a long time is required to reduce the number of activated ways because the interval between adaptation opportunities is long. As a result, the number of activated ways is increased because excessive activated ways are maintained for a long time. Consequently, a new technique is required to alleviate the effect of exceptional disturbances, resulting in a reduction in the number of activated ways, while the interval between adaptation opportunities is unchanged.

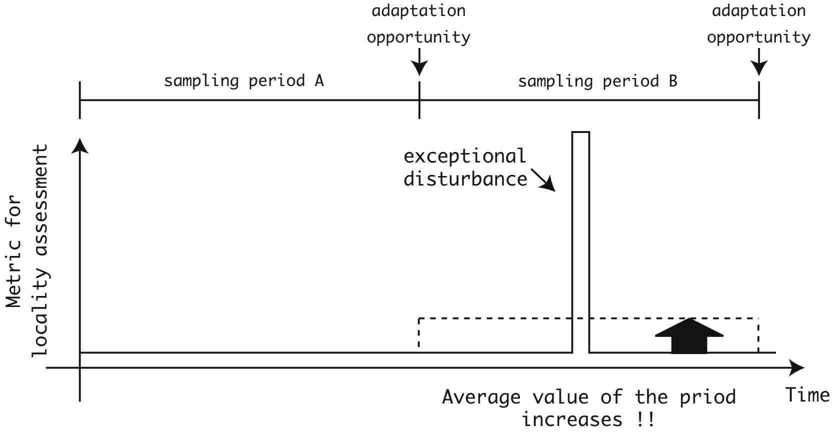


Fig. 4. Exceptional disturbances and the effect to locality assessment

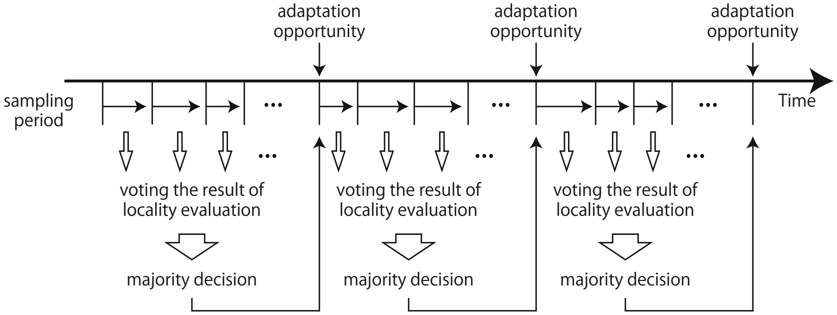


Fig. 5. Timing chart of the proposed mechanism

3 A Majority-Based Control Scheme for Way-Adaptable Caches

In this paper, we focus on the problem that the way-adaptable cache mechanism excessively adapts to exceptional disturbances. To reduce the effect of exceptional disturbances, a majority-based cache-resizing control scheme is proposed in this section. The proposed scheme uses locality assessment results of several short sampling periods in locality assessment between two consecutive adaptation opportunities, in order to reduce the effect of the locality assessment results of sampling periods with exceptional disturbances on the assessment. Using this scheme in the way-adaptable cache, the reduction in the number of activated ways and, as a result, the reduction in power consumption of the cache can be expected without performance loss.

Figure 5 illustrates the relationship between adaptation opportunities and sampling periods in the proposed scheme. The end of each sampling period does not always mean an adaptation opportunity; a sampling period ends at a fixed interval, and then a new sampling period may immediately begin without an adaptation opportunity. At the end of every sampling period, the result of locality assessment of an application in the period, i.e. *inc*, *dec*, or *keep* is decided with Eq. 1 and two thresholds (t_1, t_2). If an adaptation opportunity comes during a sampling period, the period is immediately terminated and the locality assessment in the period is performed using the statistic information collected until then.

At an adaptation opportunity, the number of activated ways should be considered based on the series of locality assessment results of several sampling periods. However, to relax the negative effects of exceptional disturbances, the mechanism needs to take account of the results by the sampling periods including exceptional disturbances. If there are fewer periods that produce different results against the other periods, the scheme decides that these periods are affected by exceptional disturbances. On the other hand, if almost all the periods since the last adaptation opportunity produce the same result, the scheme decides that the cache access behavior is stable and consistent, and hence adjust the number of activated ways based on the result.

To realize such a control mechanism, the proposed scheme uses majority voting by the locality assessment results of multiple periods between consecutive adaptation opportunities. In the proposed scheme, each sampling period votes to *inc*, *dec*, or *keep*. The proposed scheme adjusts the number of activated ways based on the majority of votes. If the number of votes to *dec* is equal to that to *inc*, or if that to *keep* is the largest, the mechanism does not change the number of activated ways. On the other hand, if the number of votes to *dec* is larger than those to the others, the mechanism decreases the number of activated ways. Similarly, if the number of votes to *inc* is larger than those to the others, the mechanism increases the number of ways.

In the proposed scheme, the length of sampling period and the interval between adaptation opportunities are important parameters to appropriately estimate the number of ways. The number of votes available at an adaptation opportunity should be as large as possible to allow the mechanism to correctly identify sampling periods including exceptional disturbances. Hence, the length of sampling period should be as short as possible in terms of the identification accuracy. However, a certain number of accesses are required to obtain the statistically-reliable results to evaluate the locality in a period. In the proposed scheme, *access-based interval* [8] is used for deciding sampling periods, in which the end of a period comes after a certain number of cache accesses. By using this interval, the proposed scheme can ensure that a certain number of sampled accesses are included in one sampling period. In addition, an adaptation opportunity comes at a fixed *time-based interval* [3] to make the adaptation frequency moderate.

Table 1. Processor and memory model

Parameter	Value
fetch, decode, issue, and commit width	8 instructions
Working frequency	1GHz
L1 I-Cache	32kB, 4-way, 64B-line, 1 cycle latency
L1 D-Cache	32kB, 4-way, 64B-line, 1 cycle latency
L2 Cache	1MB, 32-way, 64B-line, 14 cycle latency
Main Memory	100 cycle latency

4 Evaluations

4.1 Experimental Setup

In this paper, the proposed mechanism is examined in single-thread execution with the way-adaptable cache. We have developed a simulator including the way-adaptable cache based on the M5 simulator [10]. Table 1 shows the simulation parameters of a modeled processor and memory hierarchy. The way-adaptable cache mechanism with the proposed control scheme is applied to the L2 cache. We assume a 32-way set-associative cache in our evaluation, because caches with more than 16 ways are employed in some latest industrial microprocessors. For example, AMD Phenom II processor has a 64-way maximum set-associative cache [11]. As demonstrated in [12], the effect of the way-adaptable cache increases with the number of ways.

In the proposed scheme, thresholds (t_1, t_2) are required as mentioned in Section 2.1. According to the previous work [8], $(t_1, t_2) = (0.001, 0.005)$ is a fine-tuned parameter set to maintain performance, and we use these values as the thresholds. Benchmarks examined on the simulator are selected from the SPEC CPU2006 benchmark suite [13]. Each simulation is done by executing first one billion cycles of the simulated processor.

4.2 Deciding the Length of the Periods

As mentioned in Section 3, the number of votes available at one adaptation opportunity should be as large as possible. This means that the sampling period should be as short as possible, because the interval between adaptation opportunities is fixed. However, in terms of the statistic reliability of a vote, the number of cache accesses in a sampling period should be as large as possible. Therefore, we first investigate an appropriate length of a sampling period.

In the preliminary evaluation, we assume that the end of a sampling period and an adaptation opportunity come simultaneously at a fixed number of cache accesses, and hence we use an n-bit saturating counter for counting the number of cache accesses. A sampling period ends when the counter overflows. The purpose of the preliminary evaluation is to find the minimum number of cache accesses that is large enough for locality assessment, and hence to obtain reliable votes as many as possible.

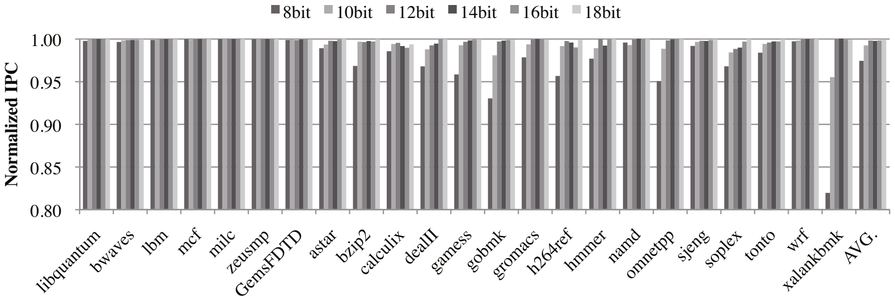


Fig. 6. Effect of sampling period length on performance

Figure 6 shows the evaluation results. In this figure, "8-bit" means that an 8-bit counter is used; the end of a sampling period and an adaptation opportunity come at every 256 accesses. These results clearly indicate that the use of 8-bit and 10-bit counters often leads to severe performance degradation. On the other hand, if the number of bits in the counter is 12 or more, the performance does not change significantly. From these observations, 4096 accesses saturating a 12-bit counter are at least required to properly assess the cache access locality. Since a shorter sampling is better to increase the number of votes at an adaptation opportunity, a 12-bit counter is used in the proposed scheme in the following evaluation.

In addition, the interval between adaptation opportunities is set to 5 ms in the following evaluation as with [3]. Specially, an adaptation opportunity comes at every 5 million cycles, because the clock rate of the simulated microprocessor is 1GHz.

4.3 Evaluation Results of the Proposed Scheme

For each benchmark, we evaluated the proposed scheme in terms of performance and the average number of activated ways. In our evaluation, there is no significant difference in performance between the conventional scheme and the proposed scheme. In the following, therefore, this paper discusses only the average number of activated ways of each scheme.

Figure 7 shows the average number of activated ways for each benchmark. In the figure, the "conventional scheme (5 ms)" indicates a original scheme, in which the end of a sampling period and an adaptation opportunity simultaneously come at every 5 ms. The "majority-based scheme (12 bit-5 ms)" indicates our proposed scheme, in which the length of a sampling period is decided with a 12-bit counter, and the interval between consecutive adaptation opportunity is 5 ms. According to Figure 7, the average number of activated ways in the majority-based scheme is 9.4% less than that in the conventional scheme. This indicates that the majority-based scheme can decrease the number of activated ways, and thereby is more effective to reduce the energy consumption with keeping the same performance.

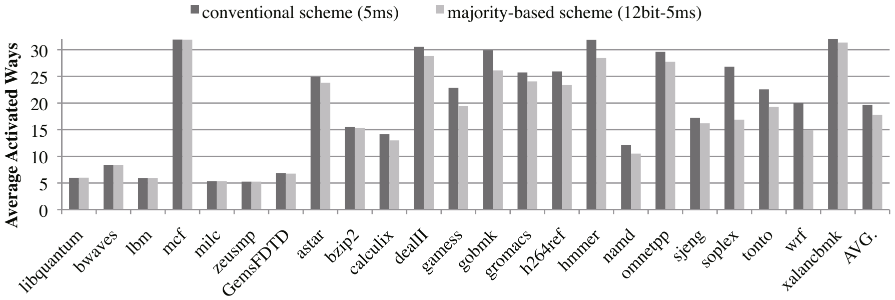


Fig. 7. Average number of activated ways

From the evaluation results, the benchmarks can be classified as follows.

Category I: `libquantum`. For this benchmark, the average number of activated ways of the majority-based scheme is slightly increased compared to that of the conventional scheme. This benchmark causes redundant activations of the ways in the proposed scheme. In this case, when the number of votes to *keep* in a period is equal to that to *inc*, the locality assessment by the conventional scheme results in *keep* at the following adaptation opportunity. On the other hand, the locality assessment by the majority-based scheme results in *inc*. However, such a case rarely happens and has little effect in evaluated benchmarks through all categories.

Category II: `bwaves`, `lbm`, `mcf`, `milc`, and `zeusmp`. For each benchmark in this category, the average number of activated ways of the majority-based scheme is almost the same as that of the conventional scheme. These benchmarks in this category do not include exceptional disturbances. This is because the number of L2 accesses is very small in these benchmarks. However, in `mcf`, all the ways are used and hence the average numbers of activated ways are larger than those of the others in this category. Even in `mcf`, cache access patterns are relatively regular and predictable.

Category III: `GemsFDTD`, `astar`, `bzip2`, `calculix`, `dealII`, `gamess`, `gobmk`, `h264ref`, `hmmer`, `namd`, `omnetpp`, `sjeng`, `soplex`, `tonto`, `wrf`, and `xalancbmk`. For each benchmark in this category, the average number of activated ways of the majority-based scheme is smaller than that of the conventional scheme. Especially for `soplex`, the average number of activated ways is reduced by up to 37%. These benchmarks often cause exceptional disturbances because they frequently access the cache. The difference in the number of activated ways indicates that the proposed scheme can properly eliminate the negative effect of exceptional disturbances. Figures 8 and 9 show the cache access distributions of `wrf` with the conventional scheme and the majority-based scheme, respectively. These distributions are observed by stack distance profiling in the same period in cycles of the simulated processor. In this period, exceptional disturbances are clearly observed. In Figure 8, the number of accesses with high LRU states

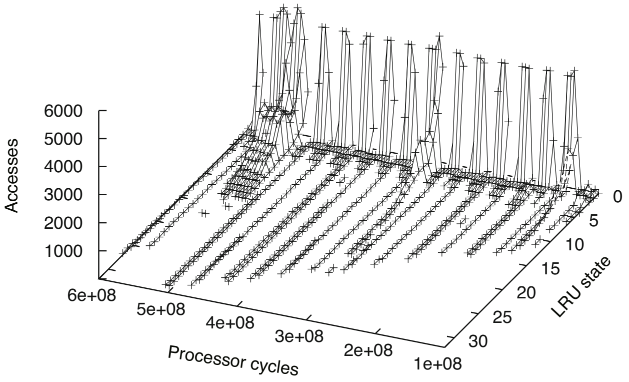


Fig. 8. Cache access distribution observed by stack distance profiling (the conventional scheme)

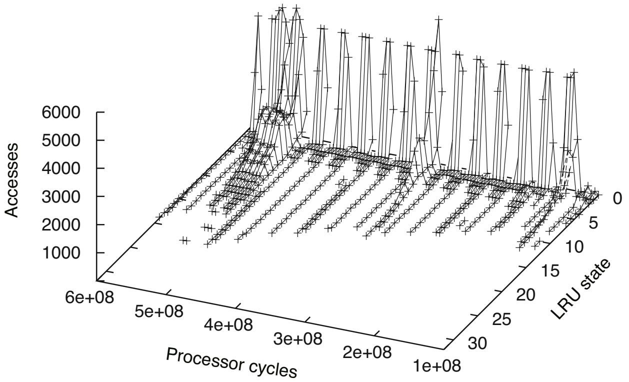


Fig. 9. Cache access distribution observed by stack distance profiling (the majority-based scheme)

gradually increases as the number of cycles proceeds. This means that the number of activated ways in the conventional scheme also increases due to the exceptional disturbances. However, the majority-based scheme can successfully inhibit the increase as shown in Figure 9. From the above results, it is obvious that the majority-based scheme is robust to an unstable situation frequently causing exceptional disturbances. The superiority of the majority-based scheme against the conventional one becomes more remarkable in the benchmarks with frequent cache accesses.

Consequently, these results clearly demonstrate that the majority-based scheme can reduce the number of activated ways to achieve the same performance, compared to the conventional scheme with time-based interval. As the number of activated ways is strongly correlated with the static leakage power of the cache, the majority-based scheme will be effective to save the energy consumption. In the future work, we will thoroughly evaluate how much power consumption of the way-adaptable cache is reduced by the majority-based scheme.

5 Conclusions

Way-partitioned caches are promising approaches to realize energy-efficient computing on multi-core processors. In these caches, it is important to accurately estimate the number of ways required by a thread, because inaccurate partitioning and adaptation degrade energy efficiency. This paper has discussed the exceptional disturbances of cache accesses and their effects on the locality assessment results used to estimate the number of required ways. To reduce the effects, this paper also proposed a scheme that decides the number of activated ways based on majority voting of the results in several short sampling periods. By using the proposed scheme for the way-adaptable cache, the average number of activated ways is decreased by up to 37%, and 9.4% on an average without performance degradation. From this observation, the way-adaptable cache mechanism with the proposed scheme enables the lower-power and higher-performance execution than that with the conventional scheme. This also indicates that the proposed scheme can estimate the number of ways required for maintaining performance more accurately than the conventional scheme.

In our future work, we will evaluate the power consumption of the way-adaptable cache with the proposed scheme. We will also apply the proposed scheme to estimation the number of ways in other way-partitioned cache mechanisms, and cache-aware thread scheduling [14].

Acknowledgement

The authors would like to thank the anonymous reviewers for constructive comments, and also thank all members of our laboratory for valuable discussions. This research was partially supported by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency (JST).

References

1. Ravindran, R., Chu, M., Mahlke, S.: Compiler-Managed Partitioned Data Caches for Low Power. In: Proc. the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 237–247 (2007)
2. Suh, G., Rudolph, L., Devadas, S.: Dynamic Partitioning of Shared Cache Memory. Journal of Supercomputing 28(1), 7–26 (2004)
3. Qureshi, M.K., Patt, Y.N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In: Proceedings of 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 423–432 (2006)

4. Settle, A., Connors, D., Gibert, E., Gonzáles, A.: A Dynamically Reconfigurable Cache for Multithreaded Processors. Journal of Embedded Computing 2(2), 221–233 (2006)
5. Kihm, J., Settle, A., Janiszewski, A., Connors, D.: Understanding the Impact of Inter-Thread Cache Interference on ILP in Modern SMT Processors. The Journal of Instruction-Level Parallelism 7 (2005)
6. Albonesi, D.H.: Selective Cache Ways: On-Demand Cache Resource Allocation. In: Proceedings of 32nd Annual International Symposium on Microarchitecture, pp. 248–259 (1999)
7. Kobayashi, H., Kotera, I., Takizawa, H.: Locality Analysis to Control Dynamically Way-Adaptable Caches. ACM SIGARCH Computer Architecture News 33(3), 25–32 (2005)
8. Kotera, I., Abe, K., Egawa, R., Takizawa, H., Kobayashi, H.: Power-Aware Dynamic Cache Partitioning for CMPs. Transaction on High-Performance Embedded Architectures and Compilers 3(2), 149–167 (2008)
9. Chandra, D., Guo, F., Kim, S., Solihin, Y.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 340–351 (2005)
10. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K.: The M5 Simulator: Modeling Networked Systems. IEEE Micro 26(4), 52–60 (2006)
11. AMD: Family 10h AMD Phenom II Processor Product Data Sheet. Technical Documents of Advanced Micro Devices (June 2009)
12. Yang, S.H., Powell, M.D., Falsafi, B., Vijaykumar, T.N.: Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In: Proceedings of The Eighth International Symposium on High-Performance Computer Architecture (2002)
13. Henning, J.L.: SPEC CPU2006 Benchmark Descriptions. ACM SIGARCH Computer Architecture News 34(4), 1–17 (2006)
14. Sato, M., Kotera, I., Egawa, R., Takizawa, H., Kobayashi, H.: A Cache-Aware Thread Scheduling Policy for Multi-Core Processors. In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, pp. 109–114 (2009)

Improved Scalability by Using Hardware-Aware Thread Affinities

Sven Mallach[1] and Carsten Gutwenger[2]

[1] Universität zu Köln, Germany
[2] Technische Universität Dortmund, Germany

Abstract. The complexity of an efficient thread management steadily rises with the number of processor cores and heterogeneities in the design of system architectures, e.g., the topologies of execution units and the memory architecture. In this paper, we show that using information about the system topology combined with a hardware-aware thread management is worthwhile. We present such a hardware-aware approach that utilizes thread affinity to automatically steer the mapping of threads to cores and experimentally analyze its performance. Our experiments show that we can achieve significantly better scalability and runtime stability compared to the ordinary dispatching of threads provided by the operating system.

1 Introduction

Today, multicore processors have become an ultimate commodity in private and scientific computing allowing multiple levels of parallelism. Namely one can achieve parallelism by superscalarity, SIMD instructions, multiple cores, and simultaneous multithreading using only a single processor. Under these conditions cache/memory considerations become more complex and, when combining two or more processors within one system, complexity rises extremely if an implementation shall scale on different architectures.

However, even within the omnipresent products of the x86 processor market, systems have significant differences concerning cache and memory subsystem design, which has a high impact on the overall performance of an implementation. This heterogeneity will considerably increase when multicore processors will offer 64, 128 or even more cores on a single chip. In such a scenario not all cores can have uniform access to all caches (or cache levels) and memory in terms of latencies. Therefore the choice of the "right" cores to share data will be of great importance. At the same time, other heterogeneous designs arise. E.g., the Cell processor [3] shipped with every Playstation 3 has lead to a community using the offered parallel floating point computation power for scientific purposes. Again, exploiting this potential requires new and non-standard programming techniques as well as knowledge about hardware issues.

Taking a look at current designs in the x86 processor market, we consider NUMA to become the dominant type of multiprocessor systems. Despite the

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 29–41, 2010.

never ending discussion whether explicit (hand-tuned) or implicit (compiler-driven) parallelism is the better strategy, hardware details have to be taken into account in order to achieve good scalability. This fact makes it hard for parallelized implementations to perform reasonably in general.

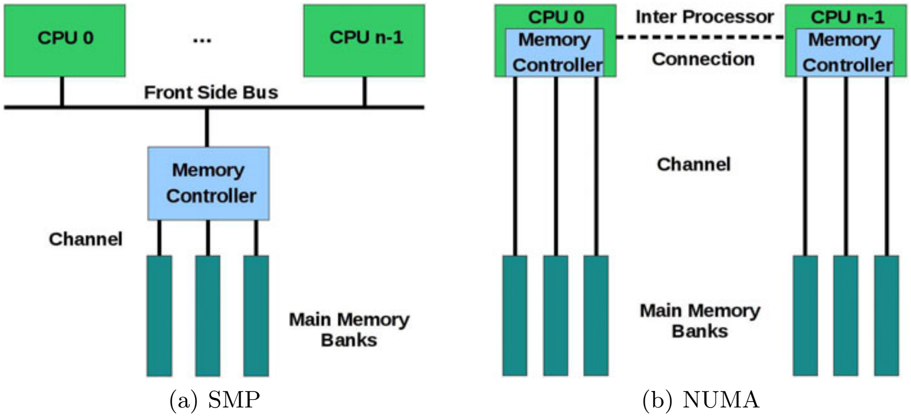
In this paper, we present a *hardware-aware* thread management using *thread affinity* as a key concept and analyze its performance. Our goal is to show that the effort to investigate the underlying system topology and properties in combination with a steered mapping of threads to execution units is worthwhile. When discussing parallel implementations, it is often argued that they lead to less "determinism" in terms of running times. In contrast to that we will show in our experimental analysis that the controlled dispatch of threads results in much more stable and reliable scalability as well as improved speedups.

Thread affinity is not an entirely new idea. Studies which propose its use within OpenMP-driven programs can be found, e.g., in [10]. The authors in [9] use affinities to prevent interrupt handling routines to be executed by different cores. Further, autopin [5] is an application that already binds threads to execution units in order to optimize performance. However, autopin asks the user to specify possible bindings which are then successively applied to find out the superior one. This requires detailed knowledge about the underlying hardware. Moreover, the implementation of the hardware performance counters used for evaluation is not yet standardized. Reading them out needs modified kernels and their values can be influenced by other processes. Additionally, the time needed for the optimization process is high compared to the runtime of typical small algorithmic tasks on relevant input sizes as we will show here. In contrast to that, the presented approach tries to exploit operating system (OS) calls as well as processor instructions to retrieve topology information automatically.

The remainder of this paper is organized as follows. In Sect. 2, we describe the differences in common memory subsystem and processor designs in detail. Sect. 3 presents our hardware-aware thread management. We explain how the respective strategies are implemented using thread affinity. In Sect. 4, we evaluate the sustained performance when applying the proposed strategies and compare them to the performance achieved when leaving the full responsibility to schedule threads to the OS. We will conclude on our approach in the last section.

2 Hardware Preliminaries

The central difference between todays multiprocessing designs is their way of accessing the memory. Besides uniform systems, like SMP (*symmetric multiprocessing*), the number of *non-uniform memory architectures* (NUMA) steadily increases. In SMP systems (see Fig. 1(a)), we have a single amount of physical memory shared equally by all execution units, and one memory controller that (sequentially) arbitrates all incoming requests. Not necessarily, but in general, this implies that all processors are connected to the controller via a single bus system called *front side bus* (*FSB*). The main advantage of this design is that data can be easily shared between processors at equal cost. The disadvantage is that it does not scale for arbitrary numbers of processors, since memory



(a) SMP (b) NUMA

Fig. 1. Typical memory system topologies

bandwidth quickly becomes a bottleneck if multiple processors require access simultaneously. In contrast to that, scalability is the most important design target of NUMA systems; see Fig. 1(b). Each processor has its own memory banks, its own memory controller, and its own connection. The drawback shows up if one processor needs access to data stored in a memory bank controlled by a different processor. In this case considerably slower connections between the processors have to be used to transfer the data. This can lead to a significant performance loss if an algorithm or thread management is not able or not designed to keep data local to processors.

3 Hardware-Aware Thread Management

3.1 Hardware Awareness

We consider now SMP and NUMA systems with multicore processors, i.e., we have not only parallelism between processors but also *within* each processor. For simplicity of presentation, we use the terms *core* and *execution unit* as synonyms, even though in case of simultaneous multithreading hardware-threads would represent multiple execution units on one core. Due to the different extent of locality we have to treat SMP and NUMA in different ways if we are to share data and to synchronize threads. Therefore we are interested in controlling which thread runs on which core, and—at any time—to be able to pick the best processor core for the current job that is to be processed. For this purpose, our hardware-aware thread management tries to gain as much topology information as possible, e.g., how many cores exist and which of them are placed on the same processor chip—possibly sharing caches. For x86 processors the corresponding mapping is obtained by determining the APIC IDs of their cores [2]. Similar functionality is extracted from the *numactl*-API [4] to investigate the node structure of the underlying multiprocessing system.

In this paper, we focus on the typical use case where a sequence of memory (e.g., a container data structure) has been initialized sequentially (e.g., by reading some input) and will now be worked on in parallel. Due to the *first touch strategy* applied by many operating systems, the data or part of it is only stored within the cache(s) belonging to the core which executed the main thread, while all other caches in the system remain *cold*. Since the end of the data structure was initialized last, we can also assume that the probability of remaining cache data increases towards the end of it.

A thread running on the same core (or cores with access to the same cache-level) will perform well and threads running on cores located on the same chip with access to a coherent higher-level cache or at least local memory will incur only small delays. But threads on cores of other processors that cannot exploit any locality will experience high latencies. Therefore, especially for small tasks or only once traversed data, page migration on NUMA systems is often not worthwhile compared to the reuse of a subset of the local cores while keeping remote ones idle.

Even though there exist routines in multithreading libraries that look like they start a bunch of threads at once, behind the scenes, the respective number of threads has to be started one by one. This means every thread t_i has a start time T_i^{start} and an end time T_i^{end} that varies with the choice of the executing core. Due to synchronization needs, the overall performance of the parallel computation depends on the "slowest" thread, i.e., for t threads in the parallel section the running time ΔT is $max_{i=1}^{t} T_i^{\text{end}} - \min_{j=1}^{t} T_j^{\text{start}}$. Therefore our goal is to minimize ΔT by starting potentially slow threads as early as possible and finding an optimal mapping of threads (e.g., processing certain array intervals) to cores.

Under the assumptions made, it is reasonable to start threads on "distant" cores processing the front-end of the memory sequence first, since they will be the "slowest" ones. There is only a small chance that the data resided in a cache, and on a NUMA system there is a memory transfer needed to get the data to the executing processor. Coming closer to the end of the sequence, the probability that it can be obtained from a cache of the processor executing the main thread rises. It is therefore promising to use cores as local as possible to the core executing the main thread for processing these intervals. The last interval might then even be processed by the main thread itself.

3.2 Thread Affinity

While the concept of *thread pooling*, i.e., keeping once used threads alive for fast re-use without creation overhead, is a well known and widely applied concept (even in compiler libraries like OpenMP), *thread affinity* has not yet gained the focus it deserves. This is especially surprising since thread affinity can easily be combined with pooling leading to much more reliable parallel execution.

If we consider a multiprocessing system with k cores, our usual assumption is that, if we create k threads, these will be scheduled one-by-one on the k available units. In many cases this assumption is wrong and operating systems

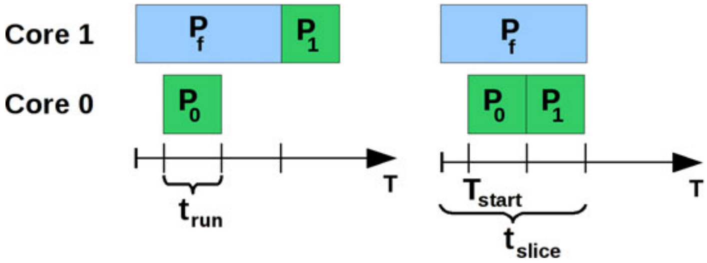


Fig. 2. Worst case scenario: using thread affinity (left); with flexible scheduling (right)

tend to schedule more than one thread on one core, or need some time slices to move threads to a less loaded one. The concept of *thread affinity* allows us to take control over the spreading of threads by explicitly configuring which thread shall run on which execution unit. Together with knowledge about the topology and memory design of the system, we use it as a powerful instrument to exploit algorithmic properties and significantly improve runtime latencies. Nearly all strategies that derive from the observations in Sect. 3 depend on the possibility to control the mapping of threads to execution units for exploiting data localities and available memory bandwidth.

The usual way thread affinity is implemented is by using OS functions. The Unix scheduler offers the function `sched_setaffinity()` and when using POSIX threads [1], the function `pthread_setaffinity()` can also be applied. Windows offers the API function `SetThreadAffinityMask()` for this purpose. Solaris also knows ways to implement thread affinity, but with the drawback that they all need privileged execution rights.

Despite all advantages, there exists a pathological example where the application of thread affinity may have negative impacts if other compute-intensive processes load one or more of the cores where threads are bound to. As a simple example for a dual-core processor, imagine some program P that can be trivially partitioned into two parts P_0 and P_1. Suppose its parallel execution shall start at some point of time T_{start} and time t_{run} is needed to execute P_0 or P_1. Now assume that at T_{start}, there is already another process P_{f} scheduled on core 1, as shown in the left part of Fig. 2. Since the thread that shall execute part P_1 is bound to core 1, the scheduler is forced to wait until the end of the current time slice. If we abstract from the costs of context switches and assume that P_0 can be shortly re-scheduled for synchronization, the running time of the program will be $t_{\mathrm{slice}} + t_{\mathrm{run}} - T_{\mathrm{start}}$. If the binding could be released, as is indicated in the right part of Fig. 2, a running time of $2 \cdot t_{\mathrm{run}}$ would suffice. In contrast to the left scenario, this would mean no loss compared to sequential execution of P and with more cores speedup could still be achieved. To enable the release of a binding we could relax the restriction of a thread's affinity mask to only one core, but this would lead to non-optimal scheduling in other cases. One solution could be to augment the mask only if necessary—at the cost of additional overhead for testing the respective conditions. Experiments how to realize such a strategy efficiently are just in their infancy.

3.3 General Pitfalls

When leaving the responsibility completely to the OS, it often happens that threads are dispatched on the same core that also executes the main thread. In the worst case, if the assigned tasks have running times in the order of a few time slices, every thread performs its tasks one-by-one instead of being moved to another core. Hence, a more or less sequential execution takes place. Even if this is not the case, a considerable performance penalty due to synchronization overhead results. Consider a thread that is to be awakened. No matter which implementation of threads is used, some time before it will have acquired a mutex lock and called the `wait()` function of a condition variable, which releases the lock again. If now the main thread wants to wake up the thread, it locks the mutex and calls `signal()` for the corresponding condition variable. Afterwards it would unlock the mutex. Experiments [6] have shown that the signaled thread may receive a time slice before the main thread has reached the atomic part of the unlock, like shown in Fig. 3. As returning from `wait()` results in a try to re-lock the mutex, the time slice will be spent with waiting, until, in the next time slice assigned to the main thread, the unlock completes.

In Sect. 4 we will see that it can even be profitable to use less than the maximum available execution units for a given task. As shown in [6], the execution time needed to perform a task on a given input size by a single thread taking part in the parallel execution can increase when the overall number of participating threads rises. This is the case if the memory controller is saturated or if inter-processor communication has to be used in a NUMA system. Even the assumption that the sustainable memory bandwidth for one processor is sufficient to serve all its cores is wrong as our and other [5] results demonstrate. Especially if we decide to use only a subset of the available threads, it is important not to start with the most "distant" ones, but to use those that are most local to the core which initialized the data. Employing this strategy we can be considerably faster than if we would have spread the work randomly.

3.4 Implementation of Dispatch Strategies

In order to facilitate flexible strategies for different subsystem designs and scenarios by means of thread affinity, we implemented a thread-pool that keeps

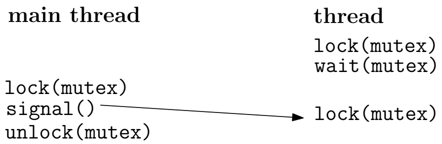
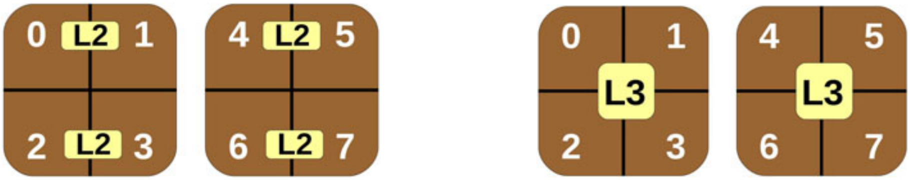


Fig. 3. Synchronization process between the main thread and a computation thread

threads sorted according to their topology in the system. To avoid the issues described above, we also control which core executes the main thread and bind no other thread to it. Hence, now every sequential initialization will be performed on that known core that we now refer to as c_{main}. We implement dispatch strategies optimized for different memory subsystem designs, tasks, and numbers of participating threads by deciding to group threads on, e.g., the processor comprising c_{main}, or to alternate between the available processors in order to balance load and optimize bandwidth. For this purpose, the functions that realize the selection of particular threads mainly operate on the sorted array mentioned above.

In case of a *compute-bound* operation with dynamic load balancing, two major properties have to be taken into account, namely the memory architecture and the number of threads to use. Due to our assumption to work on sequentially initialized data, we emphasize on exploiting locality on NUMA systems. That is, the dispatching function will not alternate between the nodes, but try to use as many cores from the processor comprising c_{main} as possible. In other scenarios with threads allocating a lot of memory themselves, a bandwidth-optimizing distribution strategy would be more appropriate. On an SMP architecture, incorporating cores of another processor does not incur higher delays. Additionally, if the processors have their own bus connections to the memory controller (like in our Penryn test system; see Sect. 4) it is worthwhile to use cores on distinct ones even if only two threads have to be dispatched, since using only one of them would not fully exploit the sustainable bandwidth of the controller.

In case of a *memory-bound* operation things become more complicated because, for small inputs, cache considerations decide whether a dispatch strategy is a good one or not. If the input entirely fits into the cache, it is generally difficult to be faster than sequential execution, since other cores have cold caches. To react on this, our dispatching function takes into account the amount of memory that will be worked on by a single thread as a third parameter. If we have small inputs and a coherent cache level for some or all cores on one processor, we try to dispatch *all* threads on the processor with core c_{main}. This leads to a trade off: Using less cores than available means a weaker parallelization, but these fewer threads now work on hot caches and receive much faster memory access. On SMP systems we can expect only little effects resulting from that, but on NUMA this strategy avoids the high latencies that would be incurred by a page migration to other processors. If the input size is bigger than the cache, local and remote cores will equally have to read from main memory. The cost for inter-processor communication can therefore be amortized by a stronger parallelization, so it is worthwhile to use cores on other processors, too. Similarly, if available, we can get more bandwidth by using multiple bus connections of SMP systems. The dispatching function starts the threads on cores of the other processors first as they will have the highest delays and then moves on to the processor with the main thread. If only a subset of the available cores is used, only as many "distant" cores as necessary are involved. Fig. 4 shows the



(a) A 2-processor system with 4 cores and two cores sharing an L2-cache

(b) A 2-processor system with 4 cores and a shared L3-cache each

scenario / core	0	1	2	3	4	5	6	7
NUMA (a) and (b), 4 threads, out-of-cache					0	1	2	3
SMP (a) and (b), 4 threads, out-of-cache	0	1					2	3
NUMA and SMP (a), 4 threads, in-cache							0/2	1/3
NUMA and SMP (b), 4 threads, in-cache					0	1	2	3

(c) Example dispatchs for the above system architectures

Fig. 4. Treatment of different architectures by our thread management

resulting thread dispatch strategies for some example scenarios that can be mapped one-to-one to the test systems used in the evaluation.

4 Experimental Evaluation

A good indicator for the performance of a thread management is its memory throughput. Due to the constantly growing gap between computational and memory throughput, most algorithms will not be able to scale linearly as long as they do not comprise intensive computations on a relatively small amount of data. Otherwise speedup can only increase to a certain factor well known as *memory wall* [8]. This factor cannot be superseded even by use of arbitrary numbers of cores, since every execution unit spends its time on waiting for memory transfers and additional units may even worsen the pollution of the memory bus. Another important measure is the speedup for small inputs. The advantages of efficient dispatching strategies, synchronization and small latencies achieved by a thin management layer can be visualized here. For our purposes, this focus is straight-forward, as we want to demonstrate the benefits of a hardware-aware thread management rather than present a competition between algorithms implemented on top of different backends which have their own impact on the measured performance.

We consider two scenarios to demonstrate the advantages when using thread affinity. We begin with a shallow copy (the C function `memcpy()`) as an examination of a *memory bound* directive which can be trivially parallelized and therefore theoretically achieve a linear speedup only bounded by the sustainable memory throughput. With this example we gain insight about the current state of the memory gap mentioned above, sensitize for the dependencies between speedup and memory throughput, and demonstrate the effects of our cache-aware thread management. We intentionally do not use established benchmarks like, e.g., STREAM [7] for comparison because they initialize data in parallel. As a second experiment, we consider a more computationally intensive operation which is not memory bound. For this purpose, we choose the function `partition()` from the Standard Template Library, which is part of the C++ standard. It has been parallelized using dynamic load balancing [6,11] and is now executed on top of our thread-pool. Being a basis for many sorting and selection problem algorithms, it is an adequate example to demonstrate the improved scalability and stability achieved.

4.1 System Setup

The test systems used in our experimental evaluation are summarized in Table 1. For our benchmarks we use the C function `clock_gettime()`, measuring *wall clock time* with a resolution of 1 ns on all test systems, and the `g++-4.4.1` compiler with optimization flag `-O2`. We compare the arithmetic average of the running times (or throughputs) of 1000 calls to the sequential and parallel function on *different* input data, thus wiping out cache effects between successive function calls. This simulates real world applications, which usually have cold caches before the start of a parallel computation. The pseudo-random inputs are equal for every number of used threads (by using seeds) and stored in containers of type `std::vector`. For the throughput benchmarks these are 64-bit floating point numbers, and for `partition()` we use 32-bit floating point numbers in the range $[0, MAX\_RAND]$ with $MAX\_RAND/2$ as pivot element. For our test systems MAX_RAND is $2^{32} - 1$.

4.2 Results

Fig. 5 shows the results for the `memcpy()` function. We first notice the typical progression of the line representing the sequential memory throughput. On all platforms we have a very high throughput at the beginning stemming from full in-cache data, which steadily decreases with the proportion of data that fits into the cache. When no data from the initialization is reusable, it becomes stagnant.

Table 1. Systems used for benchmarks

	Nehalem	**Shanghai**	**Penryn**
CPU	Intel Core i7 940	AMD Opteron 2378	Intel Xeon E5430
# CPUs / Frequency	1 / 2.93 GHz	2 / 2.4 GHz	2 / 2.66 GHz
Cores per CPU	4 (8 Threads)	4	4
Memory Architecture	Single	NUMA	UMA / SMP
L1 Data / Instr. (Core)	32 KB / 32 KB	64 KB / 64 KB	32 KB / 32 KB
L2	256 KB per Core	512 KB per Core	2×6 MB per CPU
L3 (CPU)	8 MB	6 MB	-
Main memory	12 GB	16 GB	8 GB
Linux (x86-64) Kernel	2.6.28-14-generic		2.6.24-25-generic

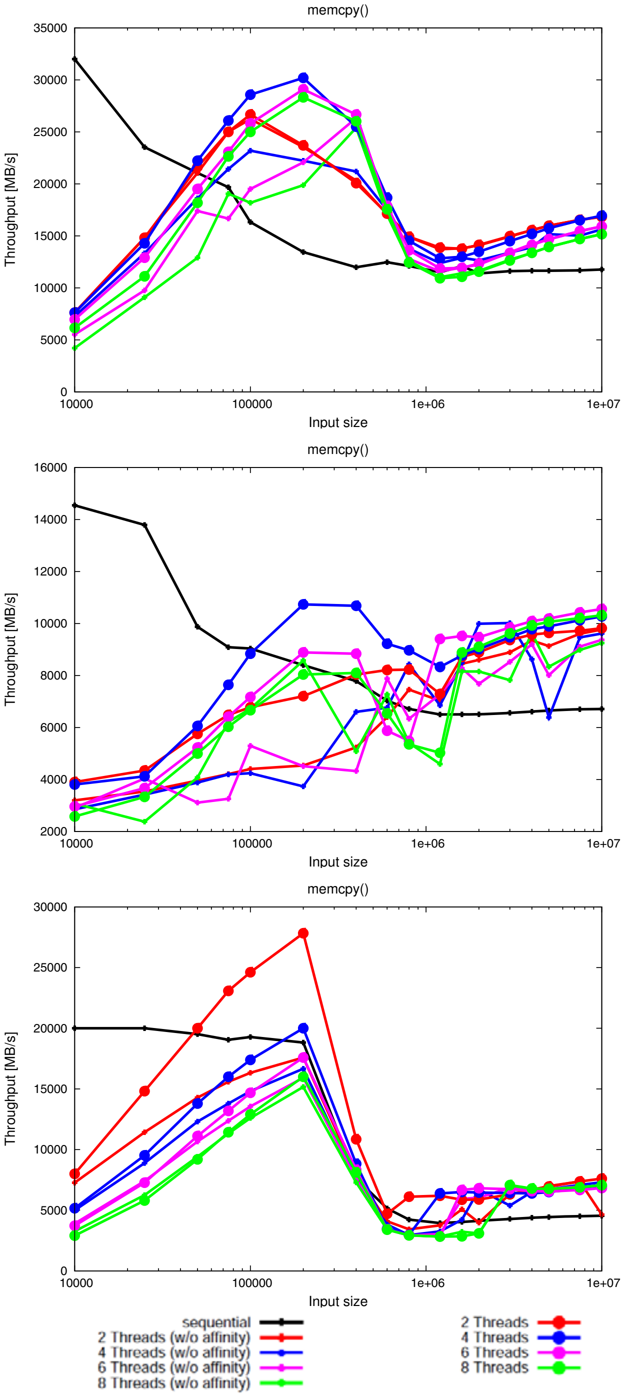


Fig. 5. Throughput for `memcpy()`: Nehalem (top), Shanghai (middle), Penryn (bottom)

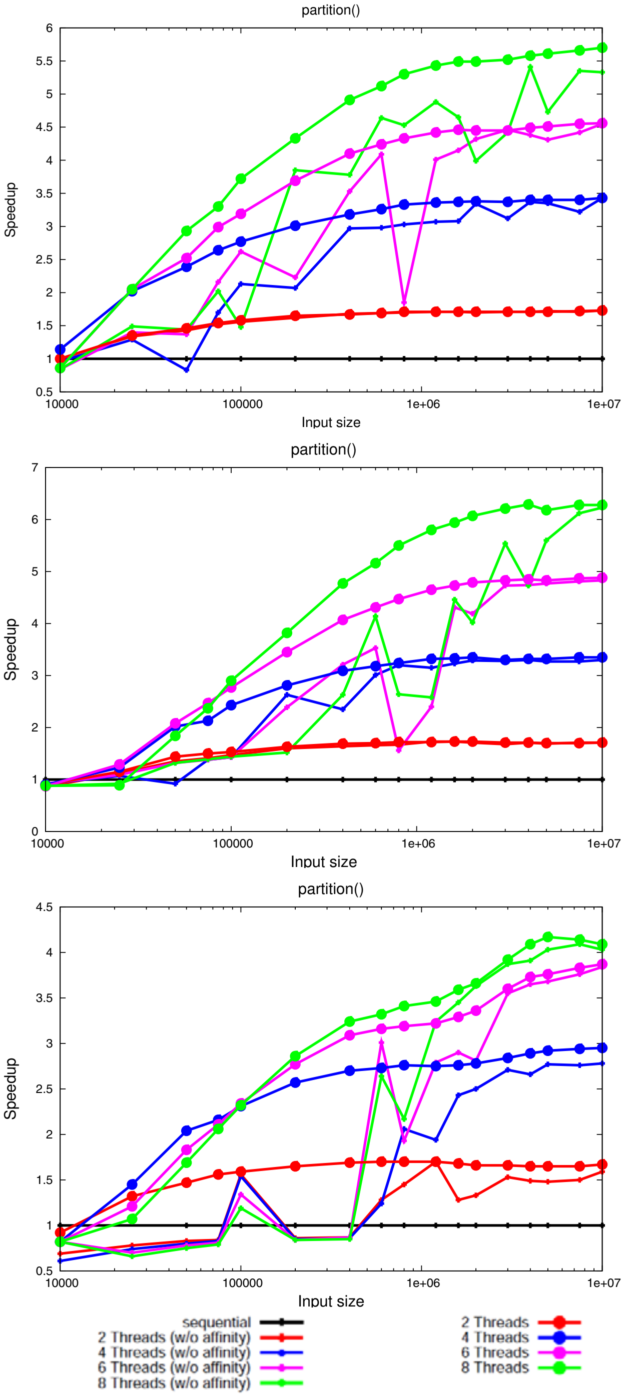


Fig. 6. Speedup for `partition()`: Nehalem (top), Shanghai (middle), Penryn (bottom)

For the parallel execution, we firstly observe a very poor performance due to the sequential initialization of the data, which is therefore only available in coherent or shared on-chip caches. Therefore and due to less overhead, execution with 2 (Penryn) or 4 threads (Nehalem and Shanghai) performs slightly better than with more threads. This effect dominates and leads to peak throughputs on all test systems until the input size runs ahead the cache size, e.g., at 10^6 (8 MB) for the Nehalem system. Using the presented hardware-aware approach we generally achieve a higher throughput than without thread affinity in this interval. Afterwards, the throughput becomes solely dependent on the bandwidth to the memory where the input was stored after initialization. As claimed before, all platforms cannot even serve their execution units with twice the sequential throughput. When the input size reaches a certain threshold, the gap between our multithreading strategy and the performance of the OS becomes smaller, since tasks now run long enough to be successively spread over all cores and smaller latencies gained by a smart dispatch order do not further dominate the overall execution time.

Fig. 6 gives the results for the `partition()` function. We observe very similar results on the Nehalem and Shanghai systems with excellent scaling for the highest input sizes measured. The performance of the Penryn system is reasonable, despite that the additional speedup using 6 and 8 threads is not comparable to the other systems. While the proposed hardware-aware implementation begins to scale already for small inputs, the lines representing execution without affinity have a less steady and sometimes even chaotic progression. Again and due to the same reasons, for the highest input sizes applied the respective lines tend to close up to each other.

5 Conclusion

We have presented a hardware-aware approach for efficient thread management. We pointed out how thread affinity can be used to improve speedup as well as stability of parallel executions and showed the effect of these improvements using two practical examples, namely a memory bound copy scenario and a function for partitioning a range of numbers. We can conclude for both scenarios that the use of thread affinity in connection with knowledge about the underlying processor topology and memory subsystem has lead to better and more reliable memory throughput and scalability. As claimed in the introduction, we confirmed that affinity-based execution is especially worthwhile for small inputs and showed that the interval where improvements make up a considerable difference is small. As a consequence, there is no scope to try out different strategies at runtime and the presented thread management is designed to automatically obtain the information needed to find a good setup.

We also sensitized that already today, and even in NUMA systems, generally not all cores can be simultaneously served with transferred data. As the number of cores steadily increases, this ratio will even worsen in the near future. Consequently, the effort of thread management paying respect to this fact will increase at the same time. Whereas in the past even small and simple data structures were stored in memory for reuse, it may soon make sense to recompute them instead of waiting for satisfied load requests. Though memory has become cheaper, a change in algorithm design could lead to applications being faster and considered superior as previous implementations by consuming less memory.

References

1. Drepper, U., Molnar, I.: The native POSIX thread library for linux. Technical report, Red Hat, Inc. (February 2005)
2. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual (April 2008)
3. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. IEEE Micro 26(3), 10–23 (2006)
4. Kleen, A.: A NUMA API for linux. Technical report, Novell Inc., Suse Linux Products GmbH (April 2005)
5. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: `autopin` — Automated optimization of thread-to-core pinning on multicore systems. In: Transactions on High-Performance Embedded Architectures and Compilers. Springer, Heidelberg (2008)
6. Mallach, S.: Beschleunigung paralleler Standard Template Library Algorithmen. Master's thesis, Technische Universität Dortmund (2008), http://www.informatik.uni-koeln.de/ls_juenger/people/mallach/pubs/diplomarbeit.pdf
7. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, Dezember (1995)
8. McKee, S.A.: Reflections on the memory wall. In: Proceedings of the 1st Conference on Computing Frontiers (CF), p. 162. ACM Press, New York (2004)
9. Scogland, T., Balaji, P., Feng, W., Narayanaswamy, G.: Asymmetric interactions in symmetric multi-core systems: analysis, enhancements and evaluation. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 2008), pp. 1–12. IEEE Press, Los Alamitos (2008)
10. Terboven, C., Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: Proceedings of the Workshop on Memory Access on Future Processors (MAW), pp. 377–384. ACM Press, New York (2008)
11. Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In: Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP), pp. 372–381. IEEE Press, Los Alamitos (2003)

Thread Creation for Self-aware Parallel Systems

Martin Schindewolf, Oliver Mattes, and Wolfgang Karl

Institute of Computer Science & Engineering
KIT – Karlsruhe Institute of Technology
Haid-und-Neu-Straße 7
76131 Karlsruhe, Germany
{schindewolf,mattes,karl}@kit.edu

Abstract. Upcoming computer architectures will be built out of hundreds of heterogeneous components, posing an obstacle for efficient central management of system resources. Thus, distributed management schemes, such as Self-aware Memory, gain importance. The goal of this work is to establish a POSIX-like thread model in a distributed system, to enable a smooth upgrade path for legacy software. Throughout this paper, design and implementation of protocol enhancements of the SaM protocol are outlined. This paper studies the overhead of thread creation and presents first performance numbers.

1 Introduction

The rising integration level in chip manufacturing enables to combine more logic on a single chip every year. By now building multiprocessor systems on chips (MPSoCs) or manycore processors is feasible, as demonstrated by the Intel Polaris or its successor called Intel Singlechip Cloud Computer (SCC) which contains 48 x86-cores in one processor. In the future processor architectures are expected to change from homogeneous to heterogeneous processor designs, consisting of different types of cores on one chip.

Efficient usage of these complex processors largely depends on the availability of thread management, memory management and synchronization mechanisms. Managing resources – such as the memory or the CPU – efficiently, gets more and more complicated with every new processor generation. Up to now the management tasks are commonly handled by the operating system (OS). The OS handles the access of all different user programs to system-level entities. Thus, a centralized management scheme results in the bottleneck of the system. So a universally applicable and scalable method for system management with direct support of heterogeneous parallel systems is required.

In addition to the complex management tasks, the reliability gains importance as a major topic in future systems. Due to the increasing integration level and the complex structures, the probability of hardware failures, during the execution of a program, rises. Executing the operation system on the failing component leads to a breakdown of the whole system although unaffected components could continue to run.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 42–53, 2010.

Therefore we propose a decentral system in which several independent operating system instances are executed on a pool of different processor cores. If one operating system instance fails, other parts of the system proceed. A new way for managing this kind of decentral system is required. Our approach relies on self-managing components by integrating self-x capabilities, especially self-awareness. As an example for such a decentralistic system we approach Self-aware Memory (SaM). Coming along with that, memory accesses are processed by self-managing system components. Especially, one hardware component called SaM-Requester manages its assigned resource – the main processing unit – and handles the logical to physical memory mapping. A more detailed introduction to Self-aware Memory is given in Section 2.

However, disruptive hardware changes which break backwards compatibility are unwanted, as programmers must be able to readapt their software without rewriting. Therefore, we propose a smooth upgrade path for hardware and software. In the proposed design software and hardware operate interactively enabling a legacy code base to be executed. Thus, parallel programs written using the POSIX-thread model – which is a well established parallel programming model for shared memory architectures – continues to run. The goal of our work is to keep the POSIX-compatibility so that only minimal software changes are required. In this paper we present a way of enabling the spawning of multiple threads without a central instance.

In the section 2 we present the background and related work. Section 3 holds the design and implementation, followed by the results and the conclusion in section 4 and 5.

2 Background and Related Work

2.1 Background

Self-aware Memory (SaM) [3], [7] represents a memory architecture, which enables the self-management of system components to build up a loosely coupled decentral system architecture without a central instance. Traditionally, the memory management tasks are handled software-based such as operating system and program libraries assisted by dedicated hardware components e.g. the memory management unit or a cache controller. The main goal of SaM is the development of an autonomous memory subsystem, which increases the reliability and flexibility of the whole system which is crucial in upcoming computer architectures.

First SaM controls memory allocation, access rights and ownership. Hence, a central memory management unit is obsolete and the operating system is effectively relieved from the task of managing memory. The individual memory modules act as independent units, and are no longer directly assigned to a specific processor. Figure 1 depicts the structure of SaM [3].

Due to this concept SaM acts as a client-server architecture in which the memory modules offer their services (i.e., store and retrieve data) to client processors. The memory is divided into several autonomous self-managing memory modules, each consisting of a component called SaM-Memory and a part of the

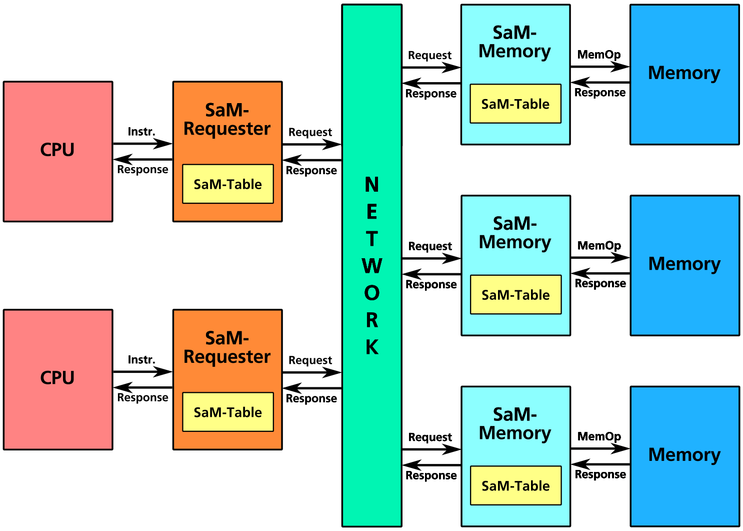


Fig. 1. Structure of SaM

physical memory. The SaM-Memory is responsible for handling the access to its attached physical memory, administration of free and reserved space as well as mapping to the physical address space. As a counterpart of the SaM-Memory, the so called SaM-Requester augments the processor with self-management functionality. Dedicated instructions for memory allocation and management (e.g., free memory, modify access rights) enhance the processor. The SaM-Requester is responsible for handling memory requests, performing access rights checks and mapping of virtual address space of the connected processor into the distributed SaM memory space. However, the SaM simulation prototype has been extended with support for basic synchronization constructs. These, also known as atomic instructions, enable lock-based synchronization of parallel threads. However, until now the hardware prototype only ran single-threaded applications, using the SaM protocol mechanisms to allocate/deallocate and access memory. Additionally, the CPU node provides local memory to store the program which is executed from the local memory. Dynamically allocated memory is handled by the SaM mechanisms without operating system support. Until now, the hardware prototype has no operating system at all.

2.2 Related Work

Previous related work proposes Organic Computing as a paradigm for the design of complex systems. Cited from [9]: “The objective of Organic Computing is the technical usage of principles observed in natural systems”. These technical systems interact and are capable of adapting autonomously to a changing environment [10]. The key properties to realize complex systems are the so called self-x properties [6]. Originally intended to realize autonomic computing systems, these properties, namely self-configuration, self-optimization, self-healing, and self-protection, are also researched in the context of Organic Computing for embedded systems. To achieve self-organization, previous related work addresses task allocation in organic middleware [2]. Task allocation is bio-inspired and relies on artificial hormones to find the best processing element. DodOrg is a practical example for a grid of processing elements organized by organic middleware. In contrast, our approach targets the programmability of distributed systems at the thread level, (instead of the task level) and favors a simplistic selection scheme over a rather complex one.

Rthreads (remote threads) is a framework to allow the distribution of shared variables across a clusters of computers employing software distributed shared memory. Synchronization primitives are derived from the POSIX thread model. The Rthreads implementation utilizes PVM, MPI, or DCE and introduces little overhead [4]. DSM-Threads supports distributed threads on top of POSIX threads employing distributed virtual shared memory [8]. DSM-Threads feature various consistency models and largely rely on a POSIX-compliant operating system. Related work combining Pthreads and distributed systems relies on operating system support combined with a library implementation. However, our proposed design relies on neither of those as an operating system is currently not used. The SaM concept greatly simplifies to build a distributed shared memory system and for the prototype, operating system support is not needed either.

3 Design and Implementation

This section presents design and implementation of the envisioned SaM protocol enhancements. The protocol enhancements over the single-threaded case are three-fold: first, the communication between CPU and SaM-Requester is presented, communication between multiple SaM-Requesters is highlighted and finally, a use case spawning an additional thread involving two SaM-Requesters and one SaM-Memory node is shown.

3.1 Protocol Design

To allow the SaM-Requester to manage the processor as a resource, it needs additional information about the state of the processor. Further, information instruction set architecture, frequency, bus width, caches, special operational modes (e.g. barrel shifter, floating point unit, etc.) of the processor are of importance. In order to reliably answer requests of other SaM-Requesters, CPU and dedicated SaM-Requester have to obey the following rules:

1. The CPU signals the SaM-Requester once it finished its work (idle)
2. The SaM-Requester tracks the current state of the CPU (idle or working)

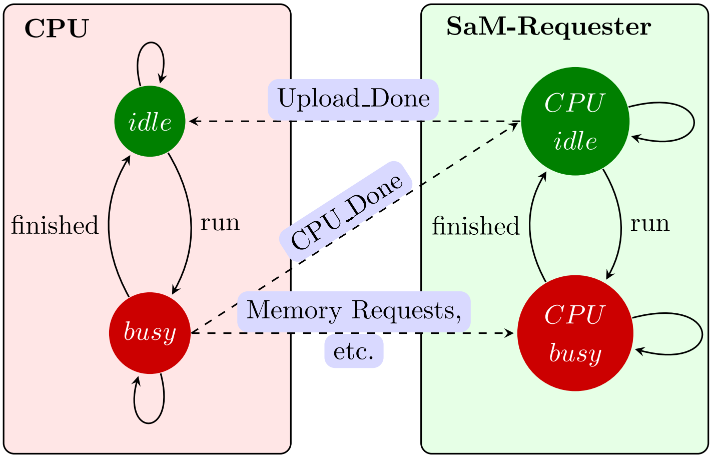


Fig. 2. Schematic protocol overview between SaM-Requester and CPU

3. If the CPU is idle its assigned SaM-Requester competes with other SaM-Requesters for work
4. If the SaM-Requester has been chosen to work, it retrieves the program and signals the CPU to start.

The protocol states and transitions are shown in Figure 2 – messages are marked with dashed arrows. The CPU may either be in an idle state, waiting for a program to arrive or currently executing (busy state). Once the CPU finishes executing a program, a CPU_Done message is sent to the SaM-Requester. By this means the SaM-Requester can track the state of the CPU. Subsequently, the SaM-Requester starts answering to SaM-Requesters demanding CPU resources. If a program is transferred to the SaM-Requester, it sends an Upload_Done message to the CPU, which immediately starts executing. During program execution the SaM-Requester serves memory requests from the CPU as illustrated in [3].

Figure 3 illustrates the negotiation of SaM-Req0 and SaM-Req1 designed to spawn a new thread running in a shared address space. First, SaM-Req0 broadcasts a request for a processing unit into the SaM space (CPU Req). If no answer is received within a predefined time span, an error is returned to the CPU. In case a SaM-Requester answers with a CPU Ack message if the corresponding CPU is idle and fulfills the requirements. Now SaM-Req0 collects answers for a predefined time span t_1 and selects the processor which fits best. CPU Grant messages are exchanged in turn between the Requesters to acknowledge the match. This second round of messages enables the SaM-Req1 to answer subsequent CPU requests while not having granted the CPU. Then, SaM-Req0 sends the CPU state and all SaM-Table entries. As already mentioned the CPU state is needed to start the thread. The SaM-Table entries define the memory space of the first thread. Until now this memory was uniquely assigned to the first thread – creating a new thread requires to share this memory. Thus, sending the SaM

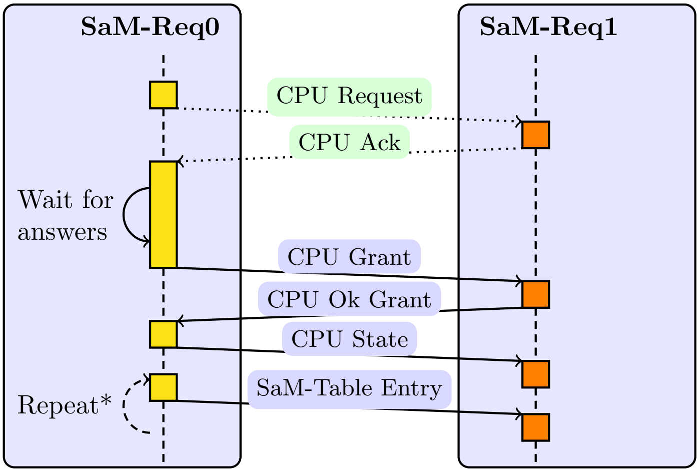


Fig. 3. SaM-Requester: protocol to spawn thread (on CPU1)

Table entries makes them shared. Since memory modules are scattered throughout the system and the first thread may have reserved memory in many of these modules, a distributed shared memory space results from creating the second thread. For additional threads, the memory space already exists and copying the SaM-Table entries suffices.

The example protocol to create a thread on CPU1 is shown in Figure 4: initiated by a call to thread_init, CPU0 sends a Thread_Init message to its SaM-Req0. On behalf of that message, SaM-Req0 finds a SaM-Memory node, which is capable of storing the program. After copying the program from the local memory to the SaM-Memory node, the SaM-Req0 returns. CPU0 continues executing and reaches the point to create a new thread (via Thread Create). SaM-Req0 reacts by finding a suitable CPU and setting up a shared memory space (for more details refer to the section above). After SaM-Req1 received and inserted the SaM-Table entries, it requests the program from the SaM-Memory node. SaM-Mem now sends the program to the SaM-Req1, which on successful completion signal Create_Ok to the SaM-Req0. Further, the newly created thread is executed on CPU1. SaM-Req0 forwards the successful creation of the thread to CPU0 which continues to execute.

3.2 Implementation

A schematic overview of the implementation of SaM-Requester and CPU node is shown in Figure 5. The picture is an excerpt obtained from Xilinx Platform Studio [1]. On the left hand side the microblaze processor with 64 KByte of local block random access memory (BRAM) connected through a local memory bus, divided into separate paths for instruction and data, is shown. The right hand side shows the identical design for the SaM-Requester, which is implemented as

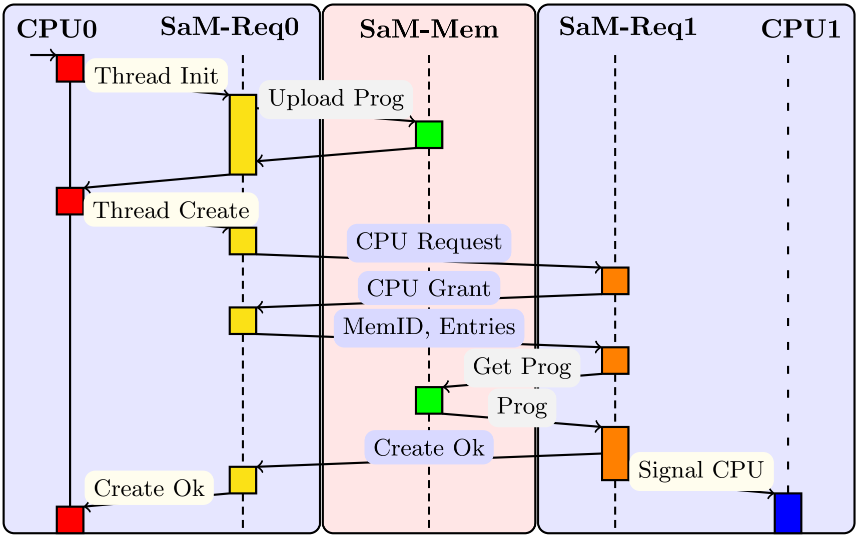


Fig. 4. Protocol overview creating a thread on CPU1

a program running on the microblaze processor. Both microblazes – representing CPU and SaM-Requester – communicate by sending messages over a fast simplex link (FSL) bus. The processor local bus (PLB) connects peripherals as well as the DRAM to the processors.

The implementation of the *idle* CPU state works as follows: the BRAM of the CPU microblaze contains a small program called *tiny bootloop*. This program polls the FSL and waits for the Upload_Done message from the SaM-Requester. The message contains the following information: instruction pointer, pointer to thread local data, return address, and stack pointer. Assigning these information to the architectural registers is done inside the tiny bootloop before branching to the new instruction pointer and executing the newly created thread.

While communication between CPU and Requester is bus based, Requester and Memory nodes communicate over ethernet. As reliable communication is of key importance, we utilize lwip4, a light weight TCP/IP stack for embedded systems [5]. The implementation of lwip4 comprises IP, TCP, ICMP, and UDP protocols. The hardware needs to be interrupt driven and deliver timer as well as ethernet interrupts to the processor. The lwip library takes care of the interrupt handling, exposing callback functions to the programmer (as low level API). By Registering and customizing these callbacks, the communication between SaM-Memory and SaM-Requester is implemented. However, utilizing only one of the available transport layer protocols (TCP or UDP) is not sufficient. While TCP does not support broadcast messages, UDP is connectionless and does not provide reliable communication. Hence, a combination of UDP (for mulitcasts and broadcasts) and TCP (for reliable communication) is employed. Figure 3

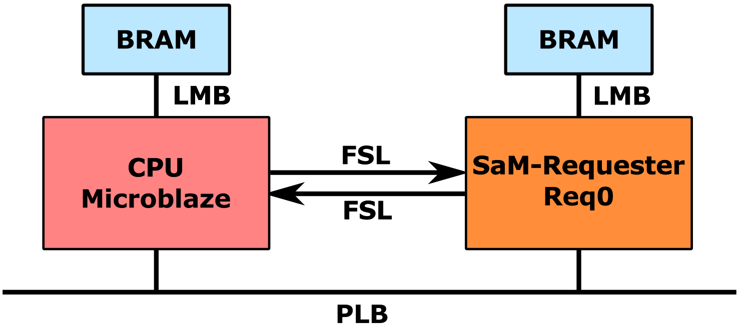


Fig. 5. Implementation of SaM prototype on Xilinx FPGA boards

illustrates this interaction: the messages with dotted lines (CPU Request and CPU Ack) are sent using the UDP protocol. The CPU Request is broadcasted whereas the answer (e.g. CPU Ack) is sent directly to the requesting node. Then, SaM-Req0 initiates a hand-over from UDP to TCP and establishes a connection to SaM-Req1. In order to establish a connection, SaM-Req0 to listen on a specific port and accept the incoming request. Once established, the connection serves as bidirectional communication channel. Now, the connection-oriented TCP protocol enables the reliable transmission of subsequent packets (CPU Grant, etc.).

To show the big picture of the SaM prototype, we would like to draw your attention to Figure 5. The components shown in this figure, CPU and SaM-Requester are implemented on one FPGA. The fast simplex link (FSL) enables message passing between these two components. The design of the SaM-Requester comprises a dedicated interrupt controller, prioritizing and delivering requests to the microblaze. Since FSL messages are of key importance, the priority of these messages is high. In order to service an FSL interrupt request a dedicated FSL interrupt handler was written: it copies messages to a buffer and defers processing of the packets. Thus, the interrupt handler occupies the processor only for a small amount of time. This is a critical aspect as interrupt handlers execute in a different context than user programs. If the interrupt controller raises an interrupt, masks all interrupts are masked (that is held back) until the interrupt handler acknowledges the current one. Thus, processing data packets in an interrupt handler not only degrades reactivity of the whole system but also may lead to a complete system lock up (e.g. because interrupts are masked while trying to send a TCP/IP packet).

4 Results

This section presents first results obtained from the SaM prototype introduced in this paper. Three Spartan-3A DSP 1800A FPGA boards from Xilinx where connected through an 8 port 10/100 Mbps ethernet switch. Each FPGA board holds one SaM component – in total two SaM-Requesters and one SaM-Memory.

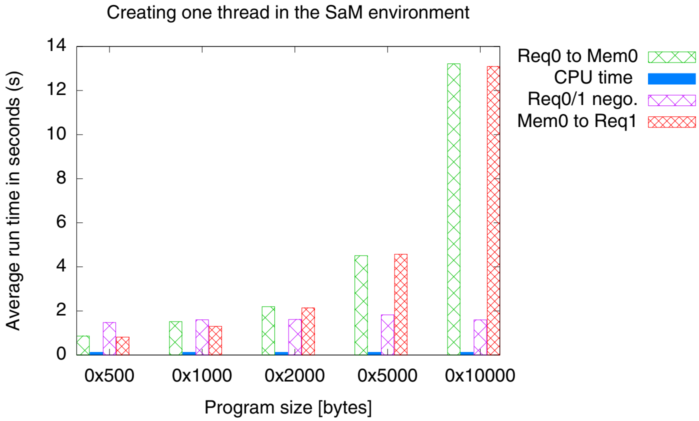


Fig. 6. Average time for spawning one thread while varying program size from 0x500 to 0x10000 Bytes

In our test case the CPU bundled with the first SaM-Requester (denoted Req0) starts the program, indirectly initiates a program transfer to the SaM-Memory (called Mem0) and later calls thread create. During the create call the second SaM-Requester (Req1) positively acknowledges a CPU request from Req0. This section studies the overhead of the process and improvements of the prototype. All times reported in this section are the average of 5 runs. By repeating the process, the effect of outliers on the reported numbers is mitigated.

Figure 6 depicts the various phases of the thread creation process (also confer to figure 4): the first bar shows how long a transfer of the program to a SaM-Memory takes, second the CPU time between `Thread_init` and `Thread_create`, third the negotiation process of the two SaM-Requesters is timed and last transferring the program from Memory to Req1 is shown. Figure 6 shows that only the duration of a program transfer is influenced by the program size. Further, as the program size rises (above 0x1000 Bytes) the time for the program transfer is the most prominent in the whole process. In addition selecting a particular CPU, takes a constant amount of time.

The second scenario simulates the case where a second CPU is not available immediately. Thus, an artificial delay was introduced before Req1 answers the request. The delay time is varied between 0 and 10000 milliseconds as shown in Figure 7. From the reported times we conclude that delays between 0 and 500 milliseconds will go unnoticed by the user, whereas larger delays (5 seconds and above) let the delay contribute the largest individual time to the overall process. The program size used in this scenario is fixed (0x5000 Bytes).

From the first two overhead studies presented before, we concluded that the implementation of the SaM-Memory is crucial for the overall performance. Especially, the program transfer time needs to be reduced. As a program has to be written to DRAM by the SaM-Memory, we decided to speed up the process

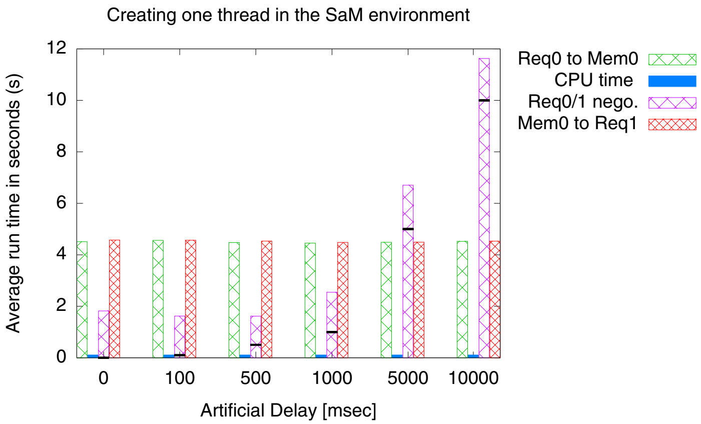


Fig. 7. Average time for spawning one thread while varying an artificial time delay from 0 to 10000 milliseconds

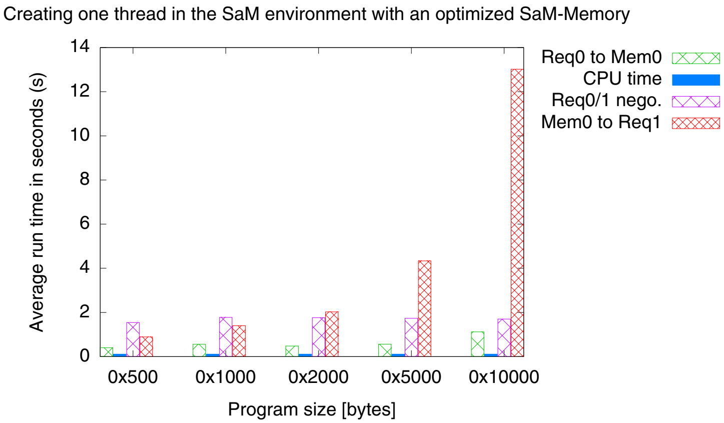


Fig. 8. Average time for spawning one thread while varying program size from 0x500 to 0x10000 Bytes employing an optimized SaM-Memory

by adding Cache Links for instruction and data to the microblaze CPU. The impact of these cache links is studied in the following.

Figure 8 shows a significantly reduced program transfer time from Req0 to Mem0. Thus, copying a program to DRAM memory from the ethernet interface is sped up significantly by adding the cache links.

The same becomes apparent in Figure 9. As writing the program to memory seems to be a crucial factor, equipping the SaM-Requester with cache links could lead to a reduced transfer time from Mem0 to Req1, further reducing the impact of the program transfer time.

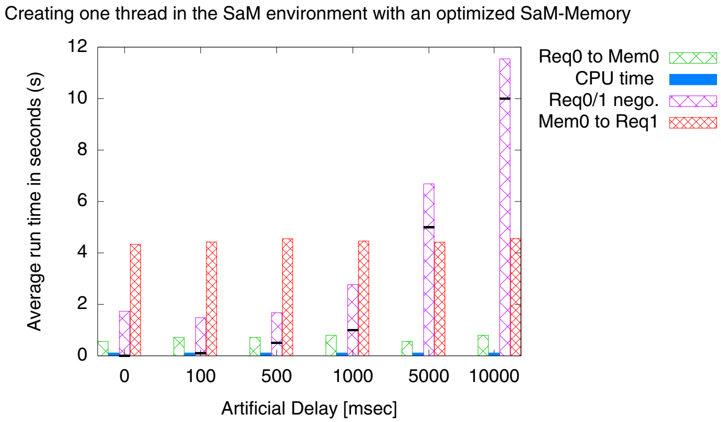


Fig. 9. Average time for spawning one thread while varying an artificial time delay from 0 to 10000 milliseconds with an optimized SaM-Memory

5 Conclusion

In this paper we present an approach towards the flexible use and management of computing resources in a distributed environment. Besides design and implementation of the POSIX-like thread concept, we also showed first performance numbers measured on a SaM prototype utilizing several FPGA boards.

Concluding from the results presented in Section 4, it became apparent that with larger programs (size $> 0x2000$ Bytes) the transfer time of the program becomes the most prominent factor in the protocol. This effect can be mitigated by adding cache links to respective components, as demonstrated with the optimized SaM-Memory design. Hence, future work will consider the optimization of the SaM-Requester.

Further, the response time of the SaM-Requester should not exceed 1 second. Otherwise, the relation between response time and transfer time is disproportional. However, with a optimized design this might change in the future - reducing the tolerable response time.

These are important insights and fundamentals that will help us to advance our work and experiment with real applications and more complex scenarios. In particular we would like to take the next step and design and implement the *join* operation to complement the creation of threads.

References

1. Asokan, V.: Designing multiprocessor systems in platform studio. In: White Paper: Xilinx Platform Studio (XPS), pp. 1–18 (November 2007)
2. Brinkschulte, U., Pacher, M., von Renteln, A.: An artificial hormone system for self-organizing real-time task allocation in organic middleware. In: Würtz, R.P. (ed.) Organic Computing, pp. 261–284. Springer, Heidelberg (March 2008)

3. Buchty, R., Mattes, O., Karl, W.: Self-aware Memory: Managing Distributed Memory in an Autonomous Multi-master Environment. In: Brinkschulte, U., Ungerer, T., Hochberger, C., Spallek, R.G. (eds.) ARCS 2008. LNCS, vol. 4934, pp. 98–113. Springer, Heidelberg (2008)
4. Dreier, B., Zahn, M., Ungerer, T.: The rthreads distributed shared memory system. In: Proc. 3rd Int. Conf. on Massively Parallel Computing Systems (1998)
5. Dunkels, A.: Full tcp/ip for 8-bit architectures. In: MobiSys 2003: Proceedings of the 1st International Conference on Mobile Systems, Applications and Services, pp. 85–98. ACM, New York (2003)
6. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
7. Mattes, O.: Entwicklung einer dezentralen Speicherverwaltung für verteilte Systeme. University of Karlsruhe (TH), Diploma thesis (May 2007)
8. Mueller, F.: Distributed shared-memory threads: Dsm-threads. In: Workshop on Run-Time Systems for Parallel Programming, pp. 31–40 (1997)
9. Müller-Schloer, C.: Organic computing: on the feasibility of controlled emergence. In: CODES+ISSS 2004: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 2–5. ACM, New York (2004)
10. Schmeck, H.: Organic computing - a new vision for distributed embedded systems. In: ISORC 2005: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, USA, pp. 201–203. IEEE Computer Society, Los Alamitos (2005)

G-Means Improved for Cell BE Environment

Aislan G. Foina[1,2], Rosa M. Badia[2], and Javier Ramirez-Fernandez[1]

[1] Universidade de São Paulo, Av. Prof. Luciano Gualberto, Travessa 3,
158, 05508-970, São Paulo, Brazil
[2] Barcelona Supercomputing Center and Artificial Intelligence Research Institute (IIIA) –
Spanish National Research Council (CSIC), Jordi Girona, 31, 08034, Barcelona, Spain
{Aislan.Foina,Rosa.M.Badia}@bsc.es, Jramirez@lme.usp.br

Abstract. The performance gain obtained by the adaptation of the G-means algorithm for a Cell BE environment using the CellSs framework is described. G-means is a clustering algorithm based on k-means, used to find the number of Gaussian distributions and their centers inside a multi-dimensional dataset. It is normally used for data mining applications, and its execution can be divided into 6 execution steps. This paper analyzes each step to select which of them could be improved. In the implementation, the algorithm was modified to use the specific SIMD instructions of the Cell processor and to introduce parallel computing using the CellSs framework to handle the SPU tasks. The hardware used was an IBM BladeCenter QS22 containing two PowerXCell processors. The results show the execution of the algorithm 60% faster as compared with the non-improved code.

Keywords: Data mining, Cell BE, parallel programming, software performance.

1 Introduction

The number of transistors in the processor has increased every year, according to Moore's Law. However, the processor speed is not increasing at the same rate. This is caused due to the difficulty in dissipating the great amount of heat generated inside the processor and the electric problems when a very high frequency is applied to the components, such as parasite capacitance. For these reasons, the processor industry decided to increase the number of cores inside a single chip instead of increasing its internal speed. Thus, a few years ago, some multi-core processors started to appear in the market, and now chips with more than 4 cores available can be found.

Following the modification in the hardware architecture, new programming models have to be developed to maximize the use of these new multi-core processors. Consequently to new programming models, applications have to be modified or adapted to use these programming models in order to exploit the computational resources available in an optimal way.

One area that has demanded more and more processing power is data mining, since the data generated by society doubles every year [1]. Data mining is the process of extracting patterns from an amount of data. One of the most used data mining

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 54 – 65, 2010

techniques is clustering, which consists in searching for groups of affinity between each point of the given dataset. There is a popular iterative algorithm called k-means commonly used to conduct this search [2].

Given a dataset of X points of d dimensions, k-means divides the points in k clusters, where each point is assigned to the cluster center with the nearest mean distance. It starts by defining some random centers and then assigns each point x that belongs to X to the nearest center c_j, c_j belonging to the set of centers C and j, a value between 1 and k. After that, it recalculates each c_j based on the mean distance of the points belonging to it, and runs the algorithm all over again. Although k-means is a commonly used algorithm for clustering, it has some problems. Firstly, the application has to know the number of clusters k to be passed as a parameter to the algorithm. Secondly, it is known by its worst case performance in terms of execution time of the algorithm.

So as to solve the first problem, some modifications to the k-means were developed by the scientific community; one of them being the G-means algorithm [3]. This algorithm, after running the k-means and finding the centers, verifies each points cluster in order to check if the cluster data follow a Gaussian distribution. If the data does not look like a Gaussian distribution, the G-means replaces the cluster center by two new centers. If any of the centers has been split, it runs the k-means again with the new value of k. The G-means does it until all the centers have a Gaussian data distribution, and its algorithm can be seen in Algorithm 1.

Algorithm 1. G-means(X, α)

```
1: Considering $C$ the set of two centers $c_1$ and $c_2$ randomly chosen inside $X$.
2: $C \leftarrow$ k-means ($C$, $X$)
3: Let $X_j$ be the set of data points assigned to center $c_j$.
4: Use a statistic test for each $X_j$ to check if a Gaussian distribution (at a confidence level $\alpha$) follows.
5: When $X_j$ passes the test, keep $c_j$. Otherwise, replace $c_j$ with two new centers, updating $C$.
6: Go back to step 2 with the new $C$ until no more centers are added.
```

The statistic test used by G-means contains the Anderson-Darling – A&D – normality test. Considering a dataset X_j, the data points of which are assigned to the center c_j, it executes the algorithm described in Algorithm 2.

Algorithm 2

```
1: Considering $C$ the set of two centers $c_1$ and $c_2$ randomly chosen inside $X_j$.
2: $C \leftarrow$ k-means ($C$, $x_j$)
3: Let $v= c_1 - c_2$ be a vector of $d$ dimensions connecting these two centers.
4: Project each point of $X_j$ onto $v$, being $X_j$' the 1-dimension vector containing the result of the projection.
5: Normalize $X_j$' to set it with mean 0 and variance 1.
6: Perform the Anderson-Darling normality test. If $A^2(X_j')$ is in the range of non-critical values at confidence level $\alpha$, keep $c_j$, and discard $c_1$ and $c_2$. Otherwise, replace $c_j$ by $c_1$ and $c_2$.
```

In brief, the G-means algorithm will have 6 execution phases. The first one is the *big k-means*; it will run the k-means for the whole dataset *X* starting with two centers. At the end of this phase, each point of the dataset will be assigned to one of the centers. At this point, the datasets will have to be reorganized to group the records that belong to each center. Based on that, the second phase is the *data group*, which executes this grouping. From the third stage to the last one, the phases are executed for each center data X_j. The third phase is the first step of the statistic test, the *two-center k-means*, where the k-means is executed using the data assigned to one specific center and two arbitrary centers. At the end of the k-means execution, the result vector *v* has to be calculated and the data projected on it. Hence, the next phase is the *projection* phase. At this stage, the mean and the standard deviation are calculated simultaneously with the projection, using the simplified standard deviation formula. After the *projection*, it is the *sort* phase turn, in which the projection vector is sorted in ascendant order. Finally, with the sorted projection vector, the *A&D calculation* phase proceeds. At this stage, the data is normalized and submitted to the statistical formulas of the A&D test, as the standard normal cumulative distribution function. At the end of the execution, if all clusters pass the A&D normality test, the algorithm is terminated; otherwise, another iteration of the 6 phases is made with the new number of centers.

Concerning the second problem about the k-means performance, since it is intrinsic to the algorithm, there are two approaches to increase its performance: 1) Modifying the algorithm to run in a different computer architecture, such as processors that support SIMD instructions or GPUs; 2) Modifying it to explore the parallelism in order to be executed in a multi-core processor and/or a multiprocessor environment, such as a cluster.

Having this in mind, the proposal is to modify the G-means algorithm to be executed in a Cell Broadband Engine Architecture – Cell BE – in order to verify the real speedup of the algorithm in this environment, using all Cell SPUs to explore the SIMD and the parallelism.

This paper is organized in 7 sections. This first one is the introduction, followed by the related work in section 2. Later, a description of the CellSs framework is provided in section 3. Section 4 presents the implementation, and section 5 shows the methodology followed. Next, section 6 presents the results obtained and its discussion. Finally, in section 7, the conclusions are presented and future work is proposed.

2 Related Work

Since the introduction of the Cell BE processor in the market in 2005, many papers have been published based on its architecture. One of them describes the implementation of the k-means in Cell processor [4]. Another paper written by Burhrer et al. published his implementation of k-means together with other data mining algorithms implemented for Cell processor as well, with some interesting benchmarks [5]. Other works focused on the implementation of the k-means in other architectures. One of these works implemented the k-means to run in a GPU using CUDA language [6]. Another author described the implementation of the k-means to a multi-processor architecture [7]. The difference of this research is the fact that the parallel k-means is part of a larger algorithm that calls the k-means frequently, but has many other calculations to automatically find the number of clusters existing in the dataset.

Regarding the number of clusters in the k-means, Polleg and Moore developed the X-means to find the number of clusters existing in the data set automatically, using the Bayesian Information Criterion – BIC – to evaluate the cluster distribution [8]. Nevertheless, this algorithm presented some problems with some kinds of distributions. Consequently, Hamerly and Elkan developed a k-means modified algorithm using the Anderson-Darling normality test to check the proximity of the cluster distribution to a Gaussian distribution, and called this algorithm as G-means [3]. Yet none of the authors explored the parallelism or the SIMD instructions in their algorithms. The study and implementation of such issues are the main point here.

3 StarSs Framework

Nowadays platforms are becoming more and more powerful, increasing the number of cores and offering different kind of processors and accelerators. But programmers' tasks are becoming more and more complex, as dealing with such architectures and making the most of them is not easy. That is why intuitive and more human-centric programming interfaces are necessary to exploit the power of the underlying architecture without exposing it to the programmer. The StarSs (Star Superscalar) model helps programmers with developing parallel applications. There are three versions: SMPSs, CellSs and NestedSs [9].

CellSs is a task-based programming model for the Cell BE. From a sequential code properly marked (using *#pragma* directives), it can exploit applications parallelism by dividing the execution into several small portions, called tasks. It is based on a source to source compiler and a runtime library. The runtime keeps and constantly updates a data dependence graph to choose which tasks can be executed satisfying data dependencies, or which ones must wait until the data they need is calculated.

Tasks are executed in the SPUs of the Cell BE and cannot call other tasks inside them. Therefore, a task must be a piece of code (a void function) and cannot access global data or return some value. In case of need, global data must be passed as an input or input/output parameter to the function and the returning value must be an output parameter.

4 Implementation

The implementation of the G-means was divided into several steps, and each step had its result compared with the previous step result. The first step to compare results is the definition of the reference code in order to evaluate the speedup generated by every new modification. In this way, it was defined that the reference execution time will be the one taken by the G-means implementation using the actual k-means speedup defined in Burhrer and Parthasarathy [10]. However, to reach this first milestone, the standard sequential k-means had to be implemented prior to the G-means.

The k-means implementation followed the Burhrer and Parthasarathy algorithms structure in order to support the SIMD instructions and the parallelization, with the difference that it was conducted using the CellSs framework to handle the parallel tasks. After the k-means implemented and tested with a known dataset of few thousand points and 2 dimensions, the improvements phase started. The first modification was in the k-means code, changing the calculation functions to use the SIMD instructions of the SPU processor. The distance formula used is a simplification of the Euclidean Distance, seen in (1).

$$d_1(p,q) = \sum_{i=1}^{n} | p_i - q_i |^2 \ . \tag{1}$$

This distance formula was used since the k-means only needs to find the nearest center, and the distance itself is not important. Thus, the square root calculation used in the standard Euclidean Distance is not necessary. This formula was modified to use the SIMD instructions *spu_sub* and *spu_madd*, seen in Algorithm 3.

Algorithm 3. Code showing the SIMD implementation of the distance algorithm.

```
vector float acc = {0,0,0,0};

for (i = 0; i < dimensions/4+1; i++) {
  result = spu_sub(recVec[i], centerVec[i]);
  acc = spu_madd(result,result,acc);
}
distance = spu_extract(acc, 0) + spu_extract(acc, 1)
      + spu_extract(acc, 2) + spu_extract(acc, 3);
```

The second modification was the organization of the data in consecutive memory position to introduce memory independence between the functions execution in order to define these functions as CellSs tasks to allow each SPU to work in parallel. With this, it was possible to define the CellSs tasks with the *#pragmas*. Hence, the k-means code could be compiled by the CellSs and executed using the Cell SPUs. The distance calculation and centers assignment were defined as a SPU task, and the SIMD instructions were used to handle the data.

In this way, the k-means Cell BE improvements already reported by other authors were implemented by using the CellSs framework. The k-means implementation being finished, a main code was created around the k-means so that the G-means could be implemented following the Hamerly and Elkan work [3]. This main code calls the k-means in two different ways: The first execution sends to the k-means all the dataset to find which centers each point belongs to: *big k-means*; the second one is called for each center, sending only the points that belonged to that center to check if the distribution is Gaussian, in order to split the center or not: *two-center k-means*. In the second k-means call, the test calculation had to be implemented to verify the distribution, and to decide whether the center splitting will occur. These implementations were made following the Hamerly work, and then finalizing the first functional version of the code. Now, G-means is using an improved k-means designed to use the SIMD instructions inside CellSs SPU tasks, and standard sequential calculation for the rest of the algorithm. This version of the code was used in the results section to measure the reference execution time of the G-means as compared with the improvements in the codes.

So as to define an optimization strategy, the algorithm was analyzed in order to find the parts that can be modified to be executed in parallel and/or to use SIMD instructions. As already mentioned in the introduction, the G-Means algorithm can be divided into six execution phases: First, the *big k-means* which needs all the dataset; second, the *data group* in order to group the records belonging to each center together; third, the *two-center k-means* execution considering the center splitting to find the result vector; next, the *projection* of the points in the result vector; later, the *sort* of this projection vector; and finally, the *A&D calculation*. Since the k-means algorithm is already improved, the two phases that use this algorithm, the *big k-means* and the *two-center k-means*, was discarded as an improvement candidate. Likewise, the phases in which the access to the whole memory is necessary can not be parallelized. Consequently, the *data group* and the *sort* can not be improved. The only remaining phases to be improved are the *projection* and the *A&D calculation*. Accordingly, the next phase of the implementation focused on the improvement of these G-means algorithm peculiarities.

4.1 First Improvement: The Projection in the Results Vector

The projection was modified to explore the SIMD instructions and to split the calculation between the SPUs. Its formula, presented in (2), consists of the scalar product between each point *x* and the reference vector *v* found, divided by the module of *v*.

$$|C| = \frac{P \bullet V}{\|V\|^2} \ . \tag{2}$$

First, the vector of *d*-dimension *v* is calculated using the result of the two-center k-means. With *v,* its module *powv* is calculated. Then, the dataset is divided into chunks of the same size used by the k-means. After that, one chunk of records, vector *v* and *powv* are sent to the SPUs in order to find the projection of each point in the chunk. At the same time, the SPU accumulates the value of the projection and its power of two in other variables to aid in the mean and the standard deviation calculation hereafter in the A&D test. Based on the projection formula, the SPU task implementation was created according to Algorithm 4.

Algorithm 4. SPU task code of the projection calculation

```
for (j = 0; j < chunk_number_of_records; j++) {
  vector float result = {0,0,0,0};
  for (i = 0; i < dimension/4+1; i++) {
    result = spu_madd(record[i],v[i], result);
  }
  float pdotv = spu_extract(result,0) +
                spu_extract(result,1) +
                spu_extract(result,2) +
                spu_extract(result,3);
  float c = pdotv / powv;
  projectionVec[j] = c;
  meanSum += c;
  stdSum += c*c;
}
```

4.2 Second Improvement: Anderson-Darling Test

The next implementation was in the Anderson-Darling calculation. This calculation is divided into steps, being the first one to sort the projection vector *Xj'*. After the sorting, (3) is applied to all values in the vector, one by one.

$$S=\sum_{k=1}^{n}\frac{2k-1}{n}\left[\ln F(Y_k)+\ln(1-F(Y_{n+1-k}))\right] . \tag{3}$$

Where *F(x)* is the standard normal cumulative distribution function and values of *Y* are the points of the projection vector normalized. After calculating the *S* value, (4) is applied.

$$A^2=-n-S \ . \tag{4}$$

Thus, the strategy to parallelize this calculation was to create a task to first normalize and calculate all the *F(x)* values in parallel, creating a vector *F* of size *n* with these values. Later, another task was created that calculates, in parallel, every value of *S* for each point in vector *F*, generating a new *S* vector with the results. After all tasks are executed, the values of *S* are summed at the end and the A^2 is calculated.

4.3 Third Improvement: CellSs Configuration and Code Parameters

The third and last improvement of the code is related to the parameter tuning of the G-means algorithm and the CellSs framework. For each task, the size of the chunk to be sent to each SPU has to be defined. If this value is too big, the SPU will spend a long time waiting for the DMA transference, increasing its idle time. Conversely, if this value is too small, the overhead generated by the CellSs and the cost of the data transference will reduce the performance, as well. Hence, these chunk size parameters were analyzed for each task in order to find the best values. Since the k-means and projection phases are dependent on the dimension of the data, the chunk size was calculated using the *GetChunksize* algorithm described in Buehrer paper. For the Anderson-Darling algorithm, since the vector size will always contain 1-dimension values independently of the number of dimensions of the dataset, the chunk size was defined as 1024 records per SPU. This value was chosen since the dataset normally used has 20,000 points for each cluster; thus this value will create 20 tasks to be sent to the 16 SPUs.

Similarly, the CellSs has its own real-time execution parameters. For instance, one of them is how many tasks will be send to each SPU at the same time. Two parameters were studied specifically, the *min_tasks* and *max_strand_size*. The *min_tasks* specifies the minimum number of ready tasks before they are scheduled, and the *max_strand_size* defines the maximum number of tasks that are simultaneously sent to an SPU. These parameters were analyzed, as well, to find the best configuration that reduces the maximum of the execution time of the G-means.

5 Test Methodology

A starting point to compare the execution of different implementations was the creation of a dataset to be used by the algorithm. Thus, to measure the execution time of the G-means, a data file was created, using the function *gen_gaussian_datafile()* provided by the k-means code developed by Pelleg and Moore [8]. A dataset was created, containing 600,000 points of 60 dimensions each, divided into 30 Gaussians distributions and it will be referred to as the 600k60d dataset in this paper. Each cluster has exactly 20,000 points. This size and dimensions were selected due to the application in which the G-means algorithm will be used to work with these numbers.

With the dataset ready, a measurement of the total execution time of each part of the G-means algorithm was made, showing the percentage of time spend in each part. Later, several different versions of the G-means was generated, each one with a new improvement compared to the previous version. Then, each of these versions was executed and all the results were compared between themselves.

To conclude, the code was written in C and compiled and executed using the CellSs version 2.2. The α confident level constant used during the test was α = 0.00001. For this confidence level, the critical value for the Anderson-Darling test is 2.357. The environment was a Linux running a kernel version 2.8.16-128.e15 SMP. The server used is an IBM BladeCenter QS22 containing two PowerXCell processors and totalizing 16 SPUs available.

6 Results and Discussion

In order to start the analysis of the results, a probe function was added to the unimproved source code to measure the time of each part of the G-means algorithm. An execution of the G-means using the 600k60d dataset had its results presented in Table 1, which describes the time spent in each part of the code.

Table 1. G-means execution time for each part of the algorithm

G-means Phases	Time	Percentage
Improved big k-means	2.00s	7%
Data group	3.35s	11%
Two-centers improved k-means	8.06s	27%
Projection calculation	3.42s	12%
Vector sort for the A&D	3.79s	13%
Anderson-Darling calculation	8.69s	30%
Total	**29.31s**	**100%**

As seen in Table 1, the *projection* and the *A&D calculation* account for a total of 42% of the execution time. Since the k-means is already improved and the sorting and grouping parts do not handle enough points to be worth the sort parallelization, these phases are the only parts of the algorithm that can be improved.

The execution of the unimproved version of the G-means with the 600k60d dataset using only one SPU resulted in an execution time of 78.35 seconds. This execution and its time will be addressed here as *base execution* and *base time*. The best execution time was 29.12 seconds when using 16 SPUs and it will be addressed here as *base parallel time*. This time means a speedup of 2.7 times. After the improvement in the projection phase, this best time reduced to 26.81 seconds using the same number of SPUs, resulting in a total speedup of 2.9 times when compared to the *base time* and 8% faster then the *base parallel time*. This small speedup was already expected, since the *projection* phase only accounts for 12% of the execution time.

Regarding the second improvement, the *A&D calculation*, the results are more expressive. The best execution time was 19.27s with all SPUs, presenting a speedup of 4.0 times compared to the *base execution*. Until now, the code modifications has reduced the execution time by 34% when compared to the *base parallel time*.

To finalize, all the variables about the chunk size and the compiler were tuned to find their best value. About the chunk sizes of the k-means and the projection, the 4kB value described in Buehrer's paper does not apply, since the CellSs handles the memory transfers and has the double buffer feature implemented. Hence, the execution time was measured with this parameter varying between 4kB, 8kB, 16kB and 32kB. For values greater than 32kB, there will be return fail due to the limited memory of the SPU. The results are presented in Table 2.

Table 2. Execution time of the G-means for different chunk sizes using the 600k60d dataset

Chunk Size (bytes)	4k	8k	16k	32k
Time (s)	40.04s	23.29s	18.12s	19.71s

As can be seen, the best chunk size measured was 16kB. Since the CellSs handles the memory transfers, the 16kB maximum DMA transfer limit of the Cell BE does not apply. But values bigger than this physical limit need two DMA transfers, which means two times the cost to start the transfers.

For the A&D calculation parameters, since it does not depend on the dataset dimension and it can have different parameters for each of its two tasks, the values used were numbers of records instead of kilobytes. The range of number of records measured was 64, 128, 256, 512, 1024, 2048 and 4096 for both tasks. The best value for the *F(x)* parameters calculation and *S* calculation are 256 and 512, respectively, but other values between 256 and 1024 showed close execution times, as well. When the number of records gets smaller than 256, the time increases significantly due to the large number of very small tasks, generating a processing overhead of the CellSs. Nevertheless, when it gets bigger than 1024, the number of records of each cluster divided by the chunk size is smaller than the number of SPUs, since the clusters have 20,000 points each.

After the software parameters are tuned, the two runtime CellSs parameters, *min_tasks* and *max_strand_size*, were analyzed. The default values for each of these parameters are 32 and 8, respectively. Since the tasks are created using the software algorithms to explore the limits of the SPUs, each single task is optimized to use all the resources available. As a consequence, when the *max_strand_size* and *min_tasks* were both set to 1, the G-means has the best execution time.

With all the parameters tuned, the best G-means execution time was 12.51 seconds, using 16 SPUs. It represents a speedup of 6.3 times the unimproved single SPU G-means execution, and 57% faster than the *base parallel time*.

To conclude the results part, the phase measurement performed in Table 1 will be carried out again, but with the execution phases of the improved version of the G-means. The time necessary for each phase of the improved code are presented in table 3. Comparing both tables, it is easy to notice the reduction of the participation of the two improvements in the overall execution time of the G-means. The *projection* phase had its time reduced from 3.42s to 0.50s (6.8x) and the *A&D calculation* was reduced from 8.69s to 0.33s (26.3x). The speedup was owed to the parallelization of the execution between the 16 SPUs, the use of SIMD instructions and the better scheduling of the tasks of the CellSs due to tuning the parameters. Figure 1 shows the speedup of each implementation varying the number of SPUs, as compared to the *base execution*.

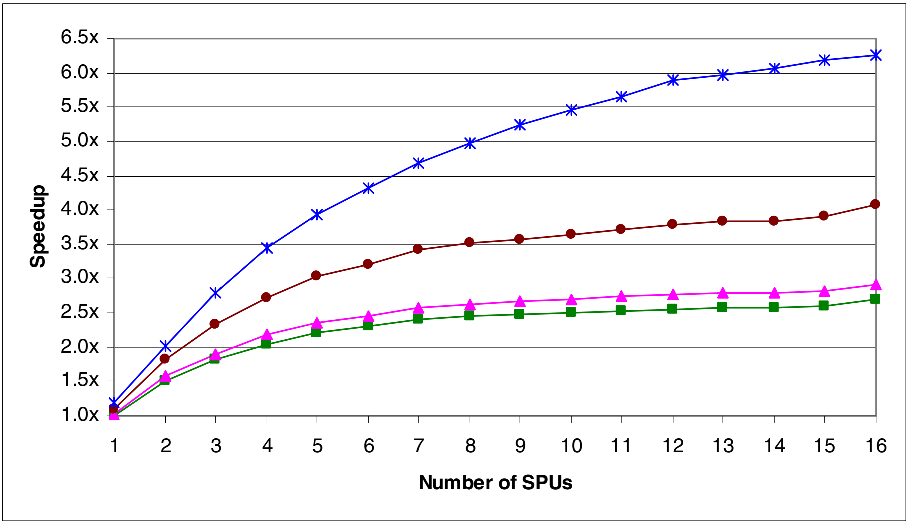


Fig. 1. Graph comparing the four different executions of the G-means. The bottom line (*square*) is the speedup of the code containing improvements only in the k-means function. The second one (*triangle*) presents the speedup measured for the G-means with the extra improvement in the projection function. The following (*circle*), is the speedup for the code containing improvement in the Anderson-Darling calculation, as well. The last one (*star*) represents the execution of the G-means with the final improved code after the tuning of the parameters.

Table 3. G-means execution time for each part of the improved algorithm

G-means Phases	Time	Percentage
Improved big k-means	1.16s	9%
Data group	3.34s	27%
Two-centers improved k-means	3.38s	27%
Projection calculation	0.50s	4%
Vector sort for the A&D	3.80s	30%
Anderson-Darling calculation	0.33s	3%
Total	**12.52s**	**100%**

7 Conclusions and Future Work

Modern applications need modification to explore the features of the new multi-core and SIMD processors. This research chose a common clustering algorithm to demonstrate some results of these modifications. The code parallelization can be divided into independent tasks significantly reducing the execution time. However, more than the implementation, the tuning of the parameters can easily increase the performance without too much effort.

The G-means development used the CellSs framework, the objective of which is to help the programmer to develop applications using the Cell BE. The use of the CellSs allowed an easy and quick implementation of the parallelism in the application, only the definition of the void functions, the correct organization of the memory and the insertion of the *#pragmas* in the code being necessary.

Another good point of the CellSs is its backward compatibility with other frameworks of its family, such as the SMPSs for multi-processors environment and the Hybrid StarSs, for heterogeneous and/or hierarchical platforms. There is one implementation directed to Cell BE, creating two task layers, one to be executed at the PPU level and the other at the SPU level. Another recent implementation of the Hybrid StarSs allows the use of the GPUs. Hence, as a future plan, small modifications in the *#pragmas* will be made to port the G-means code to different supercomputers architectures, such as SMP CPUs and GPUs.

Acknowledgments. This work was partly supported by the National Council for Scientific and Technological Development (CNPq) under Grants 142048/2008-9 and 201667/2008-8. Also, the researchers at BSC are supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050), the European Commission in the context of the HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement.

References

1. Lyman, P., Varian, H.R.: How Much Information (2003), `http://www.sims.berkeley.edu/how-much-info-2003` (retrieved from December 2009)

2. Macqueen, J.: Some Methods of Classification and Analysis of Multivariate Observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, pp. 281–297 (1967)
3. Hamerly, G., Elkan, C.: Learning the K in K-Means. Neural Information Processing Systems 16, 281–288 (2003)
4. Simek, F.: Implementation of K-means Algorithm on the Cell Processor. BSc. Thesis. Czech Technical University in Prague (2007)
5. Buehrer, G., Parthasarathy, S., Goyder, M.: Data mining on the cell broadband engine. In: Proceedings of the 22nd Annual International Conference on Supercomputing, Island of Kos, Greece, pp. 26–35. ACM, New York (2008)
6. Hong-tao, B., Li-li, H., Dan-tong, O., Zhan-shan, L., He, L.: K-Means on Commodity GPUs with CUDA. In: Computer Science and Information Engineering, WRI World Congress, pp. 651–655 (2009)
7. Tian, J., Zhu, L., Zhang, S., Liu, L.: Improvement and Parallelism of k-Means Clustering Algorithm. Tsinghua Science &Technology 10, 277–281 (2005)
8. Pelleg, D., Moore, A.: X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In: Proceedings of the 17th International Conf. on Machine Learning, pp. 727–734 (2000)
9. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Programming the Cell/B.E. made easier. IBM Journal of R&D 51(5), 593–604 (2007)
10. Buehrer, G., Parthasarathy, S.: The Potential of the Cell Broadband Engine for Data Mining. Ohio State University Technical Report OSU-CISRC-3/07–TR22 (2007)

Parallel 3D Multigrid Methods on the STI Cell BE Architecture

Fabian Oboril[1,3,⋆], Jan-Philipp Weiss[1,3], and Vincent Heuveline[2,3]

[1] SRG New Frontiers in High Performance Computing
[2] RG Numerical Simulation, Optimization, and High Performance Computing
[3] Engineering Mathematics and Computing Lab
Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
fabian.oboril@student.kit.edu,
{vincent.heuveline,jan-philipp.weiss}@kit.edu

Abstract. The STI Cell Broadband Engine (BE) is a highly capable heterogeneous multicore processor with large bandwidth and computing power perfectly suited for numerical simulation. However, all performance benefits come at the price of productivity since more responsibility is put to the programmer. In particular, programming with the IBM Cell SDK is hampered by not only taking care of a parallel decomposition of the problem but also of managing all data transfers and organizing all computations in a performance-beneficial manner. While raising complexity of program development, this approach enables efficient utilization of available resources.

In the present work we investigate the potential and the performance behavior of Cell's parallel cores for a resource-demanding and bandwidth-bound multigrid solver for a three-dimensional Poisson problem. The chosen multigrid method based on a parallel Gauß-Seidel and Jacobi smoothers combines mathematical optimality with a high degree of inherent parallelism. We investigate dedicated code optimization strategies on the Cell platform and evaluate associated performance benefits by a comprehensive analysis. Our results show that the Cell BE platform can give tremendous benefits for numerical simulation based on well-structured data. However, it is inescapable that isolated, vendor-specific, but performance-optimal programming approaches need to be replaced by portable and generic concepts like OpenCL – maybe at the price of performance loss.

Keywords: Cell BE, IBM Cell SDK, multigrid, Gauß-Seidel smoother, stencil kernel, performance analysis.

1 Introduction

Numerical simulation of complex physical processes in science and engineering relies on methods of high performance computing that produce reliable results in

[⋆] Corresponding Author.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 66–78, 2010.

a moderate amount of time. Since the grinding halt of frequency scaling, processor manufacturers are focusing on multicore technologies that have brought up a multitude of different concepts – including graphics processing units (GPUs) and heterogeneous concepts like the Cell BE. In this context, cache-based homogeneous multicore processors are dominating the scene due to their general applicability and the compatibility with classical parallel programming approaches like OpenMP and MPI. These processors are very flexible because of their general purpose instruction set architecture and their complex setup of the cores, but their floating point performance and prospect of scalability to larger core counts is limited [7]. GPUs from NVIDIA and AMD with up to 1600 cores [14,1] overcome the issue of scalability and offer unrivaled floating point performance. But the associated stream processing paradigm is based on data parallelism with a necessary high degree of locality, low level of couplings, uniformity of operations, and high arithmetic intensities. Moreover, the constraints of host interaction via the narrow PCIe datapath are degrading device-local speedups.

Within the next few years we expect to see two basic processor concepts. We will see complex cache-based processor architectures with moderate core counts, huge local caches, and shared memory that favor general applicability and ease of programmability. And we will see massively parallel and lightweighted cores with little local memory in a cluster-like design or network design (like the Cell BE and Intel Rock Creek [12]), or based on SIMD technology (like NVIDIA Fermi [13]). This approach takes into account all scalability issues by means of stream processing or software-managed local memories. However, the final answer still needs to be found whether a broad range of applications can be mapped beneficially to these types of platforms – providing further increasing performance and user-friendly programming environments.

The gap between scalability and huge floating point performance on the one hand and high flexibility on the other hand is bridged by the concept of the Cell Broadband Engine (BE) designed and developed by Sony, Toshiba and IBM (STI). It offers a considerable memory bandwidth which is crucial for a large class of numerical algorithms. The Cell BE architecture is a heterogeneous architecture consisting of two kinds of cores: the Power Processor Element (PPE) – a general purpose core for data management and orchestration – and eight Synergistic Processor Elements (SPEs) – highly optimized vector computing cores with software-controlled local stores instead of classical caches. The reduced instruction set of the SPEs is reflecting simplicity of the core design. But consequently, Cell's original programming approach by means of the IBM Cell SDK is assigning complex tasks to the programmer. Data transfers between the main memory and the local stores have to be user-controlled and overlapped with computation. Data alignment in different memory levels needs to be arranged manually, and computations need to be organized by vectorization intrinsics. However, Cell's low-level programming methodologies may result in efficient utilization of available resources – unleashing Cell's power. Beside the hardware capability and programming environment the choice of numerical solution schemes has a large impact on the quality of the results, the total amount of work to be done, and parallel performance. Multigrid methods for the discrete solution of problems of elliptic type have an optimal work complexity and a high degree of parallelism in the associated vector and stencil operations.

In this work, we have implemented and optimized a multigrid solver on the Cell BE architecture for a 3D Poisson model problem by means of the IBM Cell SDK. We have chosen a straightforward but realistic multigrid setting [17] in order to be able to assess all capabilities and programmability issues for the Cell BE and to provide a thorough performance analysis for all parts of the multigrid solution cycle. Major parts of this work are based on the study in [15].

In Section 2 we give a brief introduction to multigrid methods and we define our model scenario. In Section 3 we summarize hardware details of the Cell processor and provide some details on the IBM Cell SDK. Implementation aspects are described in Section 4. A thorough performance analysis is presented in Section 5. Our investigation shows that the comprehensive programming effort leads to convincing results and proves Cell's competitiveness. Section 6 provides some links to related work. In Section 7 we conclude with a short future perspective on hardware-aware and heterogeneous computing.

2 A Full Multigrid Scheme for a 3D Poisson Problem

Multigrid methods are very fast schemes for iterative solution of grid-based linear systems of equations. By utilizing a hierarchy of nested grids, error components can be damped efficiently on coarse grids with reduced complexity while the accuracy of the problem is still reflected by the solution on the finest grid. Apart from alternative approaches, the number of necessary iterations does typically not depend on the mesh resolution. Our chosen multigrid method uses a full multigrid cycle (FMG) for generating a fine grid solution starting from a coarse grid approximation. Then, consecutive V-cycles are applied until the desired accuracy is achieved. As a coarse grid solver we rely on a conjugate gradient (CG) method without preconditioning. Due to performance optimization on the Cell BE (see Section 4.3) and parallel efficiency, the coarse grid problems are solved on meshes of size 31^3 where the CG method provides a fast, parallel and reliable iterative solver without affecting solution accuracy. Furthermore, a full weighting restriction and a trilinear interpolation are used for grid transformations. For some interpolation steps during the FMG we use tricubic interpolation to ensure better accuracy (see Figure 1). For the relaxation on each subgrid either a Jacobi (with optimal relaxation parameter $\omega = 6/7$) or a Gauß-Seidel smoother ($\omega = 1.15$) is applied. The latter one uses a red-black numbering for parallelization. For more details on the described multigrid components we refer to [17].

The model problem under consideration is a three-dimensional Poisson problem $-\Delta u = f$ for an unknown function u in the 3D unit cube with Dirichlet boundary conditions set to zero and a given right hand side f. A typical discretization by means of finite difference or finite element methods on equidistant grids with grid size $h = 1/(n+1)$ for a large integer n results in a linear system of equations (LSE) that is characterized by the classical 7-point Laplacian stencil.

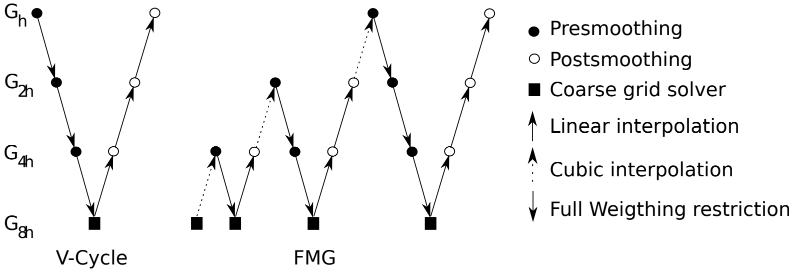


Fig. 1. Multigrid methods: V-Cycle (left) and Full Multigrid (right)

On all coarse grids 7-point stencils are used for the Laplacian operator instead of larger Galerkin-type stencils. For this type of problem, multigrid methods offer a solution scheme with optimal complexity.

3 The Cell Broadband Engine Architecture

The Cell Broadband Engine (BE) is a multicore architecture relying on innovative concepts for heterogeneous processing and memory management. The main unit of the Cell BE is the Power Processor Element (PPE) running at 3.2 GHz. Since it has only limited computing capabilities, the PPE mainly acts as controller for the eight SIMD processors called Synergistic Processor Elements (SPEs) which usually handle the computational workload. The SPEs mainly perform SIMD instructions on 128-bit wide 128 entry register files. In its latest release PowerXCell 8i, double precision (DP) performance reaches 12.8 GFlop/s by executing two FMAs per cycle – giving an aggregated DP peak performance of 102.4 GFlop/s. The eight SPEs are connected via the Element Interconnect Bus delivering aggregate peak bandwidth of 204 GByte/s. However, the data has to be accessed from main memory connected via a Memory Interface Controller with theoretical bandwidth of 25.6 GFlop/s. In the present work, we use an IBM BladeCenter QS22 with two PowerXCell 8i processors. Each Cell processor has its own 16 GByte of DDR2-800-DRAM main memory but it can by default access the main memory of the other processor resulting in NUMA effects. Both Cell processors are connected via the Flex I/O interface running a fully coherent protocol called Broadband Engine Interface (BIF). Theoretical throughput of the Flex I/O interface is 25.6 GByte/s, but due to an overhead of the BIF protocol speed is reduced.

Cell follows the approach of simplified cores with reduced instruction sets and less hardware features. Furthermore, the 256 KByte software-controlled Local Store (LS) memory decreases complexity while allowing maximal flexibility and efficiency. The LS does not operate like a conventional CPU cache since it does not contain hardware structures or protocols that store data which may possibly be reused or loaded. Data transfers are managed by Direct Memory Access (DMA) with full control and responsibility by the programmer or the software environment. The same strategy applies to performance-critical data alignment. Manual double buffering for overlapping computations with communication is supported by means of asynchronous DMAs. Moreover, all computations need to be vectorized for full utilization of the SPE's SIMD units. Due to the absence of branch predictors, branches should be avoided or hinted by intrinsics, and loops should be unrolled. Using the IBM Cell SDK [11], the programmer has to handle all mentioned aspects. Furthermore, the programmer has to load programs onto the SPE and start and stop them explicitly. He also has to take care of all synchronizations and communication with the PPE and other SPE programs. The chosen approach gives full flexibility but has severe drawbacks on coding complexity. An easier programming approach is exemplified by the RapidMind stream processing model [8], where the compiler and the runtime system are handling data transfers and parallelization. However, it has been observed that coding simplicity is accompanied with some performance loss [9]. For this reason, we are focusing on IBM's development platform. As of now, OpenCL [6] brings up a further standardization and generalization of the stream processing model and promises at least formal portability of implementations. This approach has to be investigated and compared to our results in future work.

4 Implementation and Optimization

In the following we provide some information on our implementation of the considered multigrid method. Our multigrid approach on equidistant grids relies on several types of stencil operators. For parallelization, the computational domain is divided into subgrids with minimal interaction across the interfaces. Our choice for data decomposition and minimizing data traffic is based on the circular queue method described in [4]. The domain is divided into blocks for the SPEs where each SPE-block is further divided into subblocks such that selected layers fit into the LS memory. Stencils are applied to whole layers where neighboring layers are preloaded via double buffering techniques. Ghost layers are required across SPE-blocks and corresponding subblocks. Due to the limited local store size, double buffering, optimal transfer size, and minimal vector length for BLAS operations, typical block size is 16 KByte for a vector update, and 6 layers (4 input, 2 output) of size 30-35 KByte each (computed dynamically) for the 7-point stencil.

4.1 Data Allocation and NUMA Effects

For each 3D grid in the multigrid hierarchy two 1D arrays are allocated in main memory. The first array contains information on the right hand side source vector, the second one keeps information on the target vector of the LSE. Both arrays are aligned at 128-Byte boundaries which is crucial for attaining maximal bandwidth. Data management is controlled by the PPE by means of the `_malloc_align` function. Data buffers in the LS are aligned manually. Our decomposition strategy explicitly specifies which SPE will update a particular grid point. To that end, we use a parallel routine where each SPE initializes the data on which it is working from now on. This "first touch"-policy coordinates correct mapping of data to sockets. Another possibility to ensure a correct placement of data is given by the `numactl` OS-tool which can bind data to specific memory banks or processors. Without this NUMA-aware allocation (SPEs may operate on both main memory locations), performance would be worse.

4.2 Stencil Computations

All subroutines based on matrix-vector-multiplication – including restriction, interpolation, smoothing step, CG-substep, and calculation of the residual – use fixed stencils instead of applying assembled sparse matrices. Due to the chosen equidistant grid each stencil is constant in space and only performs nearest neighbor computations on the input vector. Each point in the 3D grid is updated with weighted contributions from a subset of its neighbors where the weights are provided by specific stencil routines. In Figure 2 the procedure is exemplified by means of a 7-point stencil (e.g. Laplacian or Jacobian stencil).

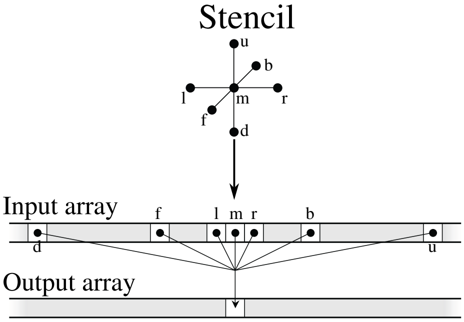


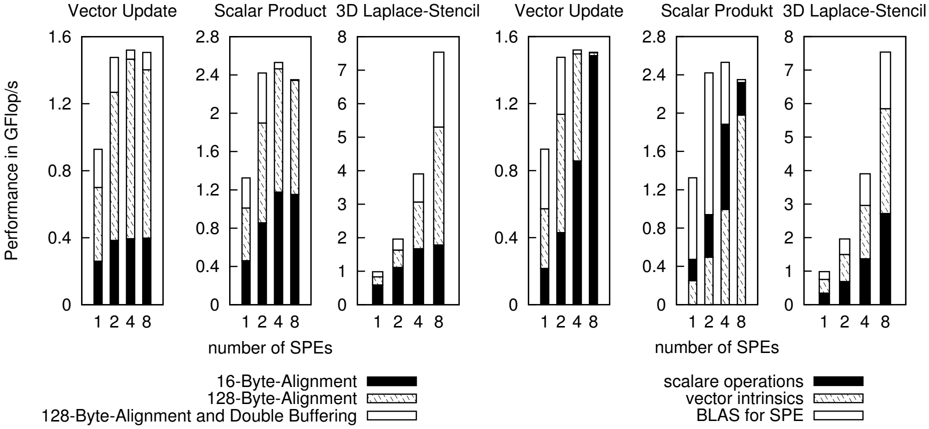
Fig. 2. 7-point stencil computation on the input vector

4.3 Optimizations for the Computations

We use a highly optimized double buffering scheme for hiding computations behind memory latencies and data transfers – and vice versa. Furthermore, we use only SIMD vector operations by means of vector intrinsics, and SPE-based BLAS 1 routines. Whenever possible, BLAS routines are preferred due to better performance characteristics and code readability. E.g. a stencil operation with 128-bit stride memory access can be built by BLAS 1 routines acting on several vector components at the same time. However, the array length has to be a multiple of 32 when using BLAS routines. Hence, the number of unknowns per subgrid is chosen by $(2^k - 1)^3$ for $k \geq 5$ with an additional array padding element. As a benefit, this constraint yields data aligned to 128 Byte boundaries. The main disadvantage of the BLAS routines for the SPEs is the fact that the input and output data have to be 16 Byte aligned. Hence, BLAS routines cannot be applied for the nearest neighbor interaction in unit stride direction for 3D stencils because of the 8 Byte offsets between neighboring vector components. Thus, for these operations vector intrinsics have to be used. In our code loops are maximally unrolled and branches are avoided.

5 Performance Results

To evaluate our implementation and the impact of the applied optimization strategies, we start with examining some basic linear algebra kernels for a problem size of 255^3 unknowns. Then, we discuss the results of the complete multigrid method and compare them with those of a CG method. All our computations are performed in double precision arithmetic. All tests have been run with the GNU 4.1.2 C/C++-compiler and the -O3 optimization flag. On the QS22, we only run a single Cell processor with its own local main memory.



(a) Alignment and double buffering (b) Optimizations for the calculation

Fig. 3. Performance of a vector update, scalar product and a 3D Laplacian stencil for 255^3 unknowns with different optimization techniques

5.1 Influence of Applied Optimization Techniques

In Figure 3 (a) the influence of data alignment and double buffering is depicted for the 3D Laplacian stencil, a vector update of type $x = x + \alpha y$ and a scalar product $< x, y >$. Figure 3 (b) shows the differences in performance when using scalar operations, the SPE's vector intrinsics, and BLAS routines on the SPEs. It is obvious that double buffering and a 128 Byte alignment of the data are essential for high performance. The same applies to the usage of BLAS 1 routines, especially for the application of stencil routines. Surprisingly, for the scalar product performance of scalar operations is superior compared to the results based on vector intrinsics. This behavior is attributed to the cost-intensive reduction operation. Maximal performance on 8 SPEs is 1.5 GFlop/s for the vector update, 2.5 GFlop/s for the scalar product, and 7.5 GFlop/s for the 3D Laplacian stencil. The first two routines are clearly bandwidth-bound when using more than two SPEs. In contrast, the stencil routine is not bandwidth-bound in the sense that no saturation effects are observed when increasing the number of SPEs. This is mainly due to the cost-intensive local vector re-organization via shuffle operations for the interactions.

5.2 Performance of Various Stencil Routines

On our applied structured grids, all matrix-vector-multiplications are implemented by stencil routines. Hence the full weighting restriction, the linear and cubic interpolation, the smoothers, and the CG-step use stencil routines. They basically differ in their data access pattern and the dimensions of source and target vector. The Gauß-Seidel smoother with red-black parallelization is accessing data with a stride of two with associated performance degradation. A performance comparison of these routines is given in Figure 4. In theory, all stencil routines are bandwidth-bound with upper bounds due to computational intensity resulting in 12.8 GFlop/s for the 3D Laplacian stencil, 5.0 GFlop/s for the linear interpolation, 12.4 GFlop/s for the cubic interpolation and 10.7 GFlop/s for the smoothers and the full weighting restriction (assuming a memory bandwidth of 25.6 GByte/s). Sustained stream-bandwidth for read operations on the QS22 for a single Cell and its local memory banks is only 21 GByte/s. For simultaneous read and write accesses bandwidth decreases to roughly 18 GByte/s. Thus, actual performance results can only reach about 70 % of the theoretical upper limits. Keeping this fact in mind, performance of the 3D Laplacian stencil, of the linear interpolation, of the full weighting restriction, and of the Jacobi smoother are close to optimal. Only the cubic interpolation, and especially the Gauß-Seidel smoother with red-black-enumeration are not that efficient. This has to be attributed to repeated local vector re-organization due to non-continuous access patterns and nearest neighbor interactions in unit stride direction.

5.3 Comparison of Gauß-Seidel and Jacobi Smoother

By comparing the performance of the Gauß-Seidel and the Jacobi smoother we find that the latter is faster in terms of runtime for a single smoothing step (see e.g. Table 1). In Figure 5 performance numbers for both smoothers are presented for a different number of SPEs and different problem sizes. However, improved smoothing properties of the Gauß-Seidel method [17] result in reduced runtime of the complete multigrid scheme, since fewer iterations have to be performed (e.g. two GS-steps instead of eight J-steps) for the same error reduction. An analysis of the runtime for the complete multigrid solver is illustrated in Table 2. For the Jacobi solver with two smoothing steps, significantly more V-cycles and coarse grid CG-iterations are necessary until the same error bound is reached. For eight smoothing steps the number of iterations is similar to the Gauß-Seidel case, but total runtime is slightly worse.

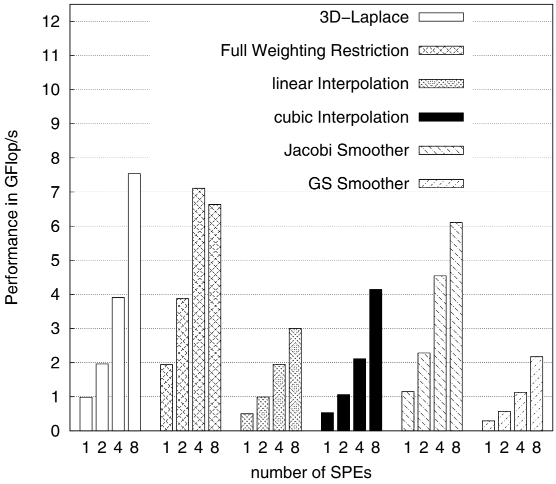


Fig. 4. Performance comparison of various stencil routines

Table 1. Detailed runtime analysis of the multigrid method on 8 SPEs

	Interpolation		Smoother (one iteration)		Residual	Error-	Restriction
	cubic	linear	Gauß-Seidel	Jacobi		correction	
Unknowns	[ms]	[ms]	[ms]	[ms]	[ms]	[ms]	[ms]
63^3	0.5	0.3	0.9	0.5	0.4	0.3	0.3
127^3	2.1	1.2	7.1	3.6	3.2	2.9	1.4
255^3	17.6	9.7	76.8	27.4	27.1	23.5	9.6

Table 2. Runtime of the complete 4-grid multigrid solver, number of required V-cycles, and coarse grid CG iterations for the Jacobi- and the Gauß-Seidel smoother on 8 SPEs, 255^3 grid points on the finest level, and 31^3 grid points on the coarsest grid, error bound 10^{-6}

	Gauß-Seidel				Jacobi			
Smoothing	V-Cycles	CG Iterations		Time [s]	V-Cycles	CG-Iterations		Time [s]
Steps		per cycle	total			per cycle	total	
2	8	≤ 71	186	4.6	37	≤ 74	815	15.5
4	6	≤ 68	176	4.9	18	≤ 73	376	8.6
6	5	≤ 65	160	5.3	12	≤ 71	253	6.8
8	5	≤ 63	148	6.4	9	≤ 70	219	6.2
10	5	≤ 60	137	7.5	8	≤ 69	205	6.3
12	4	≤ 57	126	7.2	8	≤ 69	199	6.7

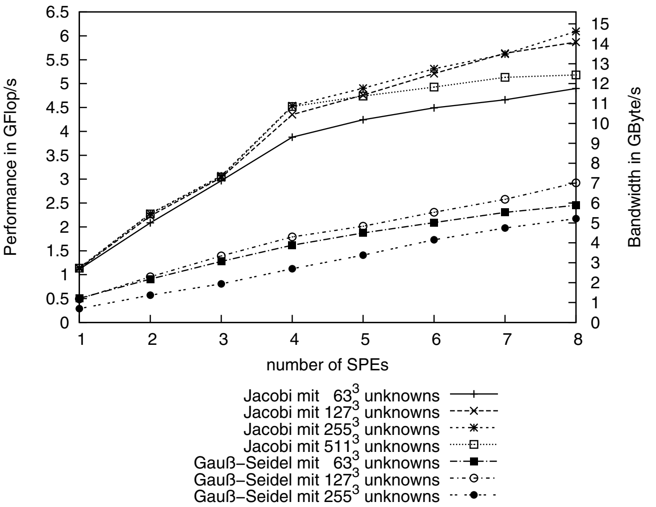


Fig. 5. Performance behavior of relaxed Jacobi and Gauß-Seidel smoothers

5.4 Results for the Multigrid Method

The Gauß-Seidel smoother with two steps for pre- and post-smoothing results in a runtime of 4.58 seconds for 255^3 unknowns and an error bound of 10^{-6} for the approximate solution of the complete multigrid solver. In Table 3 the runtime for the multigrid with Gauß-Seidel smoothing is compared to the runtime for the multigrid with Jacobi smoothing, and the plain CG method. In the multigrid method, the coarsest grid is 31^3. As expected, the CG method without preconditioning is by far too slow for huge problems. Only for small problem sizes the CG solver is faster than our multigrid implementation based on a FMG method followed by V-cycles. This is due to the fact, that the costs for thread management, SPE program launch, and coarse grid overhead are much higher for the multigrid solver. The LS of each SPE is shared by both the application data for the operations at issue and the program code. Using the GNU compiler and the optimization flag -O3, the size of the executable code of one of the four main steps of the multigrid method is at least 80 KByte. Combination of several routines would result in a code size exceeding a reasonable size. Thus, we decided to use a separate SPE-program for each subroutine of the multigrid method. However, this approach leads to a management overhead which reduces the efficiency of our implementation. In contrast, the complete code for the CG has a size of 100 KByte – fitting perfectly in the LS and reducing the management overhead.

For a problem size of more than one million unknowns our multigrid implementation is 16 times faster than the CG solver expressing the reduced work complexity which is increasing linearly in the problem size for the multigrid method and polynomially with exponent 4/3 for the CG method [2]. The dominating part of the multigrid method is the smoothing step – especially when

Table 3. Runtime for the 2-grid (63^3), 3-grid (127^3) and 4-grid (255^3) multigrid method with Gauß-Seidel (2 smoothing steps) and Jacobi (2 and 8 smoothing steps) smoothing, and the CG method on 8 SPEs.

	Gauß-Seidel, 2 Its.		Jacobi, 2 Its.		Jacobi, 8 Its.		CG	
Unknowns	V-Cycles	Time [s]	V-Cycles	Time [s]	V-Cycles	Time [s]	CG-Its.	Time [s]
63^3	7	0.75	33	2.92	8	0.88	170	0.05
127^3	7	1.53	35	6.28	8	1.99	349	4.03
255^3	9	4.58	38	15.91	9	6.16	716	64.45

using the Gauß-Seidel smoother (see Table 1). Hence, the smoother is the routine that should be optimized primarily. Calculation of the residual and error correction nearly take the same amount of time as the Jacobi smoother.

Further details on the applied multigrid methods, implementation aspects and performance results can be found in [15].

6 Related Work

Detailed discussions on data locality techniques for stencil operations and optimization techniques for the Cell BE architecture are presented in [4,3]. An evaluation of a single precision multigrid solver using the Jacobi smoother on the Cell BE can be found in [16]. The implementation there is based on the IBM Cell SDK without using BLAS routines for the SPEs. Performance results have the same quality like ours. In [8,9] an alternative approach for Cell programming is exemplified by means of the RapidMind development platform. Comparison of both performance results (although obtained in single precision only) indicate a non-negligible performance degradation by the additional software layer. A performance analysis of a GPU-based multigrid method is provided in [5]. The measurements are done on a NVIDIA GeForce GTX 280 in double precision. Performance is significantly better than for our Cell implementation – proving unrivaled computing capabilities and memory bandwidth of modern GPUs.

7 Conclusion

The landscape of multicore and accelerator technologies is multi-faceted. The future is about to bring further performance progression by increasing core counts. The Cell BE is representing the branch of chip-level heterogeneous multicore configurations and ring-based interconnects. Although the Cell project is believed not to be continued, its basic ideas will be seen in future devices like Intel's Larrabee (ring and specialized cores) and Rock Creek (software-managed caches). On the software side and for the algorithms a lot of research is still necessary in order to overcome the current multicore dilemma and to express a high degree of multi-level parallelism within the applications.

Numerical simulation of scientific or technological problems requires tremendous computing power that can be provided by the Cell BE. But much more important than pure GFlop/s-numbers is the available memory bandwidth. For bandwidth-bound applications like the considered multigrid method sustained bandwidth is the limiting factor for nearly all basic subroutines. Hence, an efficient utilization of the available bandwidth is performance-critical. Our approach for a parallel multigrid solver points out the capabilities of the Cell processor, but also shows the price in terms of complexity and extra coding effort. A lot of responsibility for hardware-specific handling is put on the programmer.

For best performance on the Cell architecture, a bunch of optimization techniques has to be applied. This includes blocking techniques for the SPEs and their Local Store memory, overlapping of computation and communication by manual double buffering, and vectorization of the computations on the SPEs. Moreover, data has to be aligned at 128 Byte boundaries in the main memory as well as in the Local Stores. By following these rules, coding complexity and code size increases, but all existing resources can be utilized beneficially. By following Cell's vendor-specific programming approach, portability to other platforms is not given.

As an alternative, OpenCL provides a generic programming environment. However, feasibility and performance competitiveness (based on immature compilers) has still to be proven. With the first pre-release version 0.1 of IBM's OpenCL SDK [10] available we have started to evaluate the suitability of OpenCL. For achieving optimal performance utilization of double buffering and SIMD operations is again essential. This reduces the portability of implementations but only the computing kernels have to be arranged for different platforms. Regarding the early status of the first release sustained performance for a vector update and scalar product is pretty impressive. The OpenCL implementation reaches approximately 90 to 100 % of the non-OpenCL performance. Currently, we are implementing and evaluating more complex routines by means of OpenCL.

Acknowledgements

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology (KIT) in the framework of the German Excellence Initiative and the industrial collaboration partner Hewlett-Packard. We also thank the Steinbuch Centre for Computing (SCC) at KIT for its support and for providing the IBM QS22 BladeCenter.

References

1. AMD. ATI Radeon HD 5870 GPU Feature Summary (February 2010), http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx
2. Axelsson, O., Barker, V.A.: Finite element solution of boundary value problems. Academic Press, London (1984)
3. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. SIAM Review 51(1), 129–159 (2009)

4. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12 (2008)
5. Göddeke, D., Strzodka, R., Turek, S.: Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations (part 2: Doubleprecision GPUs). Technical report, Technical University Dortmund (2008)
6. Khronos Group. OpenCL (February 2010), http://www.khronos.org/opencl/
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitive Approach. Elsevier Academic Press, Amsterdam (2006)
8. Heuveline, V., Lukarski, D., Weiss, J.-P.: RapidMind Stream Processing on the Playstation 3 for a Chorin-based Navier-Stokes solver. In: Proc. of 1st Int. Workshop on New Frontiers in High-performance and Hardware-aware Computing, Lake Como, pp. 31–38. Universitätsverlag Karlsruhe (2008)
9. Heuveline, V., Lukarski, D., Weiss, J.-P.: Performance of a Stream Processing Model on the Cell BE NUMA Architecture Applied to a 3d Conjugate Gradient Poisson Solver. International Journal of Computational Science 3(5), 473–490 (2009)
10. IBM. OpenCL Development Kit for Linux on Power (February 2010), http://www.alphaworks.ibm.com/tech/opencl
11. IBM. Programming the Cell Broadband Engine Architecture: Examples and Best Practices (August 2008), http://www.redbooks.ibm.com/abstracts/sg247575.html
12. Intel. Single-chip Cloud Computer (February 2010), http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC-Overview.pdf
13. NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: FERMI (February 2010), http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
14. NVIDIA. Tesla C1060 Computing Processor (February 2010), http://www.nvidia.com/object/product_tesla_c1060_us.html
15. Oboril, F.: Parallele 3D Mehrgitter-Methoden auf der STI Cell BE Architektur. Diploma thesis, Karlsruhe Institute of Technology, Germany (2009)
16. Ritter, D.: A Fast Multigrid Solver for Molecular Dynamics on the Cell Broadband Engine. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2008)
17. Trottenberg, U., Oosterlee, C.W., Schüller, A.: Multigrid. Elsevier Academic Press, Amsterdam (2001)

Applying Classic Feedback Control for Enhancing the Fault-Tolerance of Parallel Pipeline Workflows on Multi-core Systems

Tudor B. Ionescu, Eckart Laurien, and Walter Scheuermann

Institute of Nuclear Technology and Energy Systems, Stuttgart, Germany

Abstract. Nuclear disaster early warning systems are based on simulations of the atmospheric dispersion of the radioactive pollutants that may have been released into the atmosphere as a result of an accident at a nuclear power plant. Currently the calculation is performed by a series of 9 enchained FORTRAN and C/C++ sequential simulation codes. The new requirements to our example early warning system we focus on in this paper include a maximum response time of 120 seconds whereas currently computing a single simulation step exceeds this limit. For the purpose of improving performance we propose a pipeline parallelization of the simulation workflow on a multi-core system. This leads to a 4.5x speedup with respect to the sequential execution time on a dual quad-core machine. The scheduling problem which arises is that of maximizing the number of iterations of the dispersion calculation algorithm while not exceeding the maximum response time limit. In the context of our example application, a static scheduling strategy (e.g., a fixed rate of firing iterations) proves to be inappropriate because it is not able to tolerate faults that may occur during regular use (e.g., CPU failure, software errors, heavy load bursts). In this paper we show how a simple PI-controller is able to keep the realized response time of the workflow around a desired value in different failure and heavy load scenarios by automatically reducing the throughput of the system when necessary, thus improving the system's fault tolerance.

Keywords: pipeline scheduling, queuing theory, feedback control, dispersion simulation, fault tolerance.

1 Introduction

In many engineering and natural sciences, modelling and simulation play an important role in understanding the behaviour and limitations of certain technical facilities, natural phenomena, and other physical or abstract systems. One type of simulation software component, called *simulation code*, has been established as the standard way of encapsulating a simulator for a certain physical aspect of a real system. A simulation code is a command line executable which uses file-based communication with the outer world. The codes are often written in FORTRAN or C/C++ and are usually developed at research institutes by domain experts. The simulation of complex systems like, for example, a nuclear

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 79–91, 2010.

power plant, and complex physical phenomena like the atmospheric dispersion of radioactive pollutants, asks for a multi-physics approach. In software terms, this means that for computing one simulation step several simulation codes are involved. Usually, there exists an exact predefined calling order (workflow) of the codes.

Many of the simulation codes are based on old sequential implementations of complex algorithms. The parallelization of these codes appears to be a straightforward approach for improving their performance. But, as experience shows, parallelization turns out to be a very costly and time consuming operation. Furthermore, in order to speedup the entire workflow one must parallelize every code involved in the workflow; otherwise the unparallelized codes will represent performance bottlenecks. Fortunately, when the workflow contains several modular simulation codes another speedup technique can be used, namely the parallel process pipeline method. The main advantage of this technique is that it does not require any changes made to the simulation codes. In addition, it is complementary to the parallelization method.

In this paper we focus on a dispersion calculation simulation workflow which is used by a nuclear disaster early warning system. The purpose of the simulation is, firstly, to trace the airborne radioactive materials that may have been released into the atmosphere as a result of an accident at a nuclear power-plant and, secondly, to calculate the effects of the radiation upon humans and the environment. In case of an emergency, the calculation has to be performed in real time using measured weather and emission data. The new requirements to the system include a hard deadline of 120 seconds[1] for the simulator's response time. If this deadline is missed once, the entire simulation is compromised. On the other hand, in order to assure a high quality of the results, it is desirable for the algorithm to perform a great number of iterations without exceeding the response time limit. In other words, the system's throughput (i.e., the number of iterations) must be maximized while the response time must be kept within the imposed limit. Implementing the dispersion calculation workflow as a parallel process pipeline on a multi-core system leads to a 4.5x speedup with respect to the sequential computation time. For the dispersion calculation workflow, a static scheduling strategy (e.g., a fixed rate of firing iterations) is inappropriate because if a software or hardware fault occurs, the scheduling algorithm must react so as to reduce the system's throughput in order to meet the response time deadline at the cost of a lower quality of the results.

This is where the PI controller comes into play by providing a robust algorithm for dynamically scheduling the execution of the process pipeline stages. Given a desired response time for one calculation step (i.e., workflow run), the controller attempts to keep the realized response time to the desired value by acting upon the throughput. This way the response time can be automatically controlled regardless of the characteristics of the underlying hardware environment, of the

[1] Although this value seems to be very high for the response time of a real-time system, the dispersion calculation application can be regarded as one according to the definitions of a real-time system from [1].

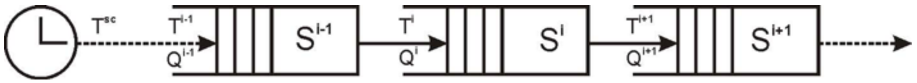


Fig. 1. A linear N-station queueing network

number of concurrently running calculations, and of the level of disturbances and other faults in the system. The only parameter that needs to be fed to the controller is a *reasonable* desired response time. By reasonable we mean a value that is to be determined experimentally depending on the theoretical peak load of the system and the envisioned hardware platform as well as on different failure scenarios and projected worst case execution times.

In short, our contribution is two-fold: (i) We model a dispersion calculation workflow using queuing network theory from a pure computational point of view and implement it in the discrete events domain of computation as a parallel process pipeline; (ii) We derive a discrete mathematical model for the response time in a parallel process pipeline and then reformulate the scheduling problem in terms of control theory. Finally, we propose a controller design and test its capabilities.

2 Pipeline Workflow Scheduling: The Case of Dispersion Calculation Workflows

In our approach, the dispersion calculation workflow is regarded as a process pipeline. The pipeline stages are represented by the different computation steps carried out by different simulation codes. Each stage receives its input from the previous stage and outputs data for the next stage. Such a pipeline can be modelled as a *queueing network* with discrete events [2].

The following subsection presents the queueing network metrics which will be used throughout the paper. We use the notations from [3].

2.1 Queueing Network Terminology

We consider a generic linear N-station queueing network as the one shown in figure 1. It is assumed that stations do not have to share resources and can thus work in parallel. There are three events related to each request: $e^i_{arrival}[k]$ – the arrival and en-queueing of request k at station i, $e^i_{process}[k]$ – starting to actually process request k, and $e^i_{complete}[k]$ – the completion of request k at station i. Considering that the occurrence of each event e has a time-step $\tau(e)$, then $\tau(e^i_{complete}[k]) = \tau(e^{i+1}_{arrival}[k])$ if the queue of station i is not empty at that time. For simplicity we will sometimes write $\tau^i_{arrival}[k]$ instead of $\tau(e^i_{arrival}[k])$.

Definition 1. *Let $A^i \geq 0$ be the number of observed arrivals and $C^i \geq 0$ the number of observed completions at station i for a timespan $T > 0$. Then the* ***arrival rate*** *at station i is given by $\lambda^i = A^i/T$ and the* ***throughput*** *of station*

i by $X^i = C^i/T$. The arrival rate of the entire queueing network is given by $\lambda = \lambda^1$. The network's throughput is given by $X = \min\{X^i\}$ for $i = 1..N$.

Definition 2. *Let $B^i \geq 0$ represent the length of time that station i was observed to be busy and $C^i \geq 0$ the number of jobs that have been completed in a timespan T. Then the* ***service time*** *of station i is given by $S^i = B^i/C^i$. The service time of the entire network is given by $S = \sum_1^N S^i$.*

The theoretical speedup with respect to the sequential service time (i.e., if all stations share a single set of resources and thus work sequentially) when all stations work in parallel is $S/\max\{S^i\}$.

Definition 3. *Let B^i represent the length of time that station i was observed to be busy in a time interval T. Then the* ***utilization*** *of the station's resource(s) is given by $U^i = B^i/T$. Considering that the stations possess identical resources, the utilization of the network's resources is given by $U = \sum_1^N U^i/N$.*

Definition 4. *Let i be a station in a queueing network and let S^i be its service time. Further, let $W^i \geq 0$ denote the average time spent by a request in the queue of station i (**waiting time**). Then the station's* ***response time (latency)*** *is given by $R^i = S^i + W^i$. The response time of the entire network is given by $R = \sum_1^N R^i$.*

Assuming S^i to be constant, upon $\tau(e^i_{arrival}[k])$ the expected waiting time of request k at station i is given by

$$W^i_k = Q^i(\tau^i_{arrival}[k])S^i - (\tau^i_{arrival}[k] - \tau^i_{process}[k-1]) \tag{1}$$

where Q^i represents the number of enqueued requests at station i (including the one currently being processed).

Considering that for a pipeline workflow $X = \min\{X^i\}$, the latency depends on $\lambda = \lambda^1$ which, in term, depends on the type of workload considered. In our case we are able to control the arrival rate and therefore the model is said to be a *closed* one and the workload of *batch* type.

We will now relate the queueing network metrics to the dispersion calculation workflow.

2.2 The Dispersion Calculation Workflow

The dispersion simulations we target in this paper rely on the Lagrangian particle model [4]. The goal of the simulation is, firstly, to trace airborne particles (radioactive aerosols) that have been released into the atmosphere as a consequence of an accident at a nuclear power plant and, secondly, to calculate the effects of radioactivity upon humans and the environment. In the Lagrangian particle model, the trace species are represented by particles spreading across an area surrounding the point of emission. The dispersion is determined by wind speed, wind direction, and atmospheric turbulence. The monitored area is divided into a regular grid and only those cells containing at least one particle are processed by the dispersion calculation algorithm.

The workflow is composed of 9 autonomous simulation codes (FORTRAN executables) which communicate with the outer world only through ASCII and binary files. The first group of codes (WIND) is concerned with generating a 3-dimensional wind field based on weather data. The second group (RAIN) processes weather data and prepares the information about the influence of precipitation upon the concentration of the transported radioactive substances. A third group of codes (EMIS) processes the data about the type and quantity of the released radioactive substances and determines which proportion of particles corresponds to which nuclide type. The data from the first three groups of codes are used by the dispersion module (DISP) to simulate the transport of about 10000 particles. Its output is then fed to the equivalent dose module (AIRD) which determines the amount of radiation in the air. Finally, the dose module (DOSE) computes the time-integrated effects of radiation upon the organs of humans belonging to different age groups.

It is important to note that in the following execution chain all codes or groups of codes only need the results from previous code groups for any given time-step: (WIND, RAIN, EMIS) $>$ DISP $>$ AIRD $>$ DOSE. This makes the pipeline parallelization possible. In this context, the notion of *simulation step or time-step* refers to a complete run of the workflow for a set of input weather and emission data. The *input interval Δt* refers to the timespan (or time interval) for which the input data for the current time-step are valid. Thus, a simulation step can use an input interval of any length (e.g., 10 minutes or 10 seconds) and can be computed in a time which depends on the underlying hardware resources and the semantics of the workflow (i.e., parallel vs. sequential).

In terms of queueing network theory, the sequential computation time needed by the workflow to process one simulation step represents the total service time $S = \sum_1^N S^i$ of a linear queueing network. If the workflow is parallelized using the pipeline method then one simulation step k will be computed in a time $S_k^{max} = \max\{S_k^i\}, i = 1..N$.

In some application contexts (e.g., an accident at a nuclear power plant) dispersion calculations have to be performed in real-time with currently measured weather and radioactive emission data. During real-time calculations, at the beginning of each time-step current weather and emission data are fetched from the National Weather Forecast Center and from the source of emission, respectively, in form of average values for a given real timespan $\Delta t_k = t_{now} - t_{past} = t_k - t_{k-1}$ whereby the index k identifies the simulation step. Δt_k is dictated by the workflow and can take values from S_k^{max} to 120 seconds. The upper limit of 120 seconds is imposed by law and ensures a certain standard of accuracy for the results. In general, the smaller Δt_k is the more accurate the results will be. But more importantly, the 120 seconds limit applies to the total latency of the workflow: $R_k < 120s$. This means that regardless of the value of Δt_k and of the characteristics of the computing environment, the results of one simulation step must become available with a maximum delay of two minutes with respect to the timespan for which the input data are valid. This is the *hard* real-time condition imposed to the system.

In order to comply with the real-time condition the simulation clock must be accurately set according to the service times observed in the queueing network. The simulation clock determines the arrival rate of requests which means that the tick period of the simulation clock is defined as $T^{sc} = 1/\lambda$. However, in a real computing environment the service times of the dispersion calculation modules cannot be accurately determined for several reasons. Firstly, depending on the input data S^{DISP}, S^{AIRD}, and S^{DOSE} can experience variations of up to 50% from the observed average values. Secondly, the service times depend on the server load determined, for example, by the number of concurrently running dispersion calculations. Furthermore, in case of a disturbance (e.g., a process reclaiming 100% of the system's processing capacity) or a hardware failure (e.g., a defect processor in a multi-core system) the service times of the simulation codes can be unpredictable.

The fact that we do not always have *a priori* knowledge about the service times in an arbitrary computing environment already suggests the need for some sort of feedback mechanism able to provide at least estimations for the stations' service times based on measured values from the past. However, we will not attempt to obtain estimations for the service times but rather for the latencies of the stations since this way we can use the waiting times as buffers that can be freely manipulate. For example, if the service times of stations increase for some reason, we can reduce the waiting times to compensate the increase of the service times in order to keep the latency as invariant as possible.

3 Feedback Control of Parallel Process Pipelines

The two main goals of a controller are: (1) *reference tracking*, i.e., eliminating the steady state error e between a measured process variable y and a desired setpoint (or reference variable) r so that eventually $e = r - y = 0$ and (2) *disturbance rejection*, i.e., minimizing the effects of perturbations upon the controlled process. In addition to these goals our aim is to maximize the server utilization U while obtaining an optimal trade-off between the length of the time-step T^{sc} and the total latency of the network R. For this purpose, the controller output will be used to set T^{sc} whereas R will be the measured process variable. The choice of an appropriate controller depends on the dynamics of the process to be controlled. The process dynamics are usually described by one or several differential equations which characterize the behaviour of the system.

In the following we will first derive a general-purpose model of a process pipeline and then propose a controller able to meet all the requirements mentioned before.

3.1 A Discrete Model for Latency in Parallel Process Pipelines

Consider an N-station queueing network as the one shown in figure 1 where the service times of all stations are assumed to be time-step invariant, i.e., $S^i_k = S^i = constant$. We define the *job arrival period* at station i as the amount of time that passes between two consecutive job arrivals at that station: $T^i = 1/\lambda^i = \max\{S^{i-1}, T^{i-1}\}$ for $i > 1$ and $T^i = T^{sc}$ for $i = 1$. Under these conditions, the following recursion holds for any station i of the queueing network:

$$\Delta R^i = R^i_k - R^i_{k-1} = S^i - T^i \tag{2}$$

where k designates the last enqueued request at station i. A natural choice for the discretization time-step is $\Delta t^i = \max\{S^i, T^i\}$ since in a real computing environment where S^i can vary, the actual value of R^i_k can only be accurately measured upon occurrence of $e^i_{complete}[k]$.

We now sketch an intuitive proof for this first order difference equation which will be backed up by experimental results in section 3.3. Considering that $R^i_k = W^i_k + S^i_k$, equation 2 simply states that when $T^i < S^i$ a new request, say k, arrives "too soon" at station i and must therefore wait in line $S^i - T^i$ more time units than the previous request before being processed. In this case $\Delta t^i = S^i$. Conversely, when $T^i > S^i$ the request arrives "too late" because the completion rate of the station is higher than the arrival rate of requests. In this case, the newly arrived request k will have to wait $W^i_k = \min\{T^i - S^i, W^i_{k-1}\}$ less time units than the previous request. $\Delta t^i = T^i$ if $W^i_{k-1} = 0$ and $\Delta t^i = S^i$ otherwise. Please also note that, by convention, if $T^i > S^i$ and $W^i_{k-1} = 0$ then $\Delta R^i = 0$ since in this model both the latency and the waiting time are positive values. A steady state with $W^i_k = W^i_{k-1}$ is reached when $T^i = S^i$.

In a real computing environment the accurate value of the latency R^i_k can only be obtained with a time delay equal to R^i_k itself. However, for being able to efficiently control the latency we need to obtain at least an estimation of R^i_k upon the arrival of request k at station i. For this purpose we can use relation 1 to compute the waiting time in $R^i_k = W^i_k + S^i_k$.

Extending the model to represent the total latency of the queueing network is straightforward and yields:

$$\Delta R = R_k - R_{k-1} = \sum_{i=1}^{N} \Delta R^i \frac{\Delta t}{\Delta t^i} \tag{3}$$

where $\Delta t = \max\{S^i, T^{sc}\}$ for $i = 1..N$. The delay with which we can measure R_k is given by $\tau_{complete}[k] - \tau_{arrival}[k] = R_k$. An estimation of the total latency, denoted $\tilde{R}_k$, can be obtain by summing up $\tilde{R}^i_k$ computed using relation 1 for all stations i and updating this value upon arrival of a request at any of the stations in the network.

As an illustration of how the total latency in a given queueing network can be computed, we will consider the assembly line example from [2]. Given a queueing network of 3 stations with $S^1 = 10$, $S^2 = 15$, $S^3 = 3$, and $T^{sc} = 7$,

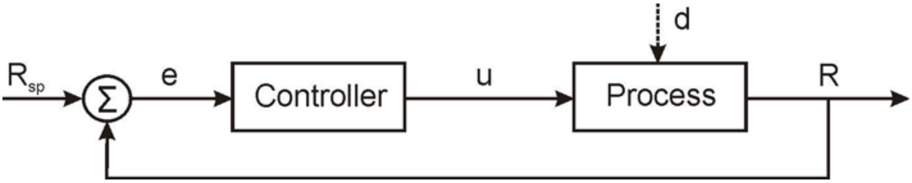


Fig. 2. The closed loop block diagram of a process pipeline and a controllers

compute R_k for $k = 5$. First, we determine $\Delta R^1 = S^1 - T^{sc} = 3$, $\Delta R^2 = S^2 - \max\{S^1, T^{sc}\} = 5$, and $\Delta R^3 = S^3 - \max\{S^2, T^2\} = S^3 - \max\{S^2, S^1, T^{sc}\} = 3 - 15 < 0$ which, by convention, results in $\Delta R^3 = 0$. Next, we determine $\Delta t = \max\{S^1, S^2, S^3, T^{sc}\} = 15$, $\Delta t^1 = \max\{S^1, T^{sc}\} = 10$, $\Delta t^2 = \max\{S^2, T^2\} = 15$, and $\Delta t^3 = \max\{S^3, T^3\} = 15$. Finally, by using equation 3 we obtain $\Delta R = 3\frac{15}{10} + 5\frac{15}{15} + 0\frac{15}{15} = 12.5$ and, taking into account that $R_1 = S^1 + S^2 + S^3 = 28$, $R_5 = R_1 + 4\Delta R = 78$.

Before proceeding to the controller design phase, two remarks regarding the limitations and the applicability of the presented model are necessary.

Firstly, we have considered the service times of all stations to be time-step invariant. Any variations from the observed averages for the service times are regarded as disturbances in the model. We found this simplification to be necessary because the service times of the simulation codes depend on the input data and the computing environment. Rather than claiming the impossibility of deriving a model for the service times in a queueing network, we conjecture that the presented simplified model suffices for achieving the goal of controlling the latency in a queueing network and prove it through experiments in section 3.3.

Secondly, the applicability of the model is limited by the fact that equations 2 and 3 can be only be used in a setup where the cost of the communication between the stations can be neglected. For example, on a multi-core system we can neglect this cost by employing a RAM-drive solution. If the communication cost cannot be neglected, the service time of any of the stations will have an additional communication component with own dynamics, i.e., $S = S^{proc} + S^{com}$, for which the presented model does not hold any more.

3.2 Controller Design

It is well known from classic control literature (see for example [5]) that for controlling a first order system – like the one described by equation 3 – it suffices to use a PI (proportional-integral) controller. Figure 2 depicts the block diagram of the controller together with the process pipeline feedback loop and its characterizing parameters. The discrete version (also known as the velocity form) of the PI controller is given by:

$$u_k = u_{k-1} + K_c(e_k - e_{k-1}) + \frac{K_c T_s}{T_i} e_k \tag{4}$$

with u_k the controller output (which in our case is used to set $T_k^{sc} = u_k$), R_{sp} the controller setpoint (i.e., the desired total response time), R the observed response time, K_c the controller gain, T_i the integral time, and T_s the error estimation period.

As previously discussed, the waiting time is given by relation 1 based on which we can compute an estimation of the total response time as $\tilde{R_k} = \sum_{i=1}^{N} (\tilde{W_k^i} + \tilde{S_k^i})$. Here, $\tilde{S_k^i}$ represents the latest value of S^i which has been observed at station i. $\tilde{W_k^i}$ is also computed on the basis of $\tilde{S_k^i}$. For each request k, the estimated value of the response time R_k is updated repeatedly over a time period of Δt and such a change is triggered by the arrival of a request at any of the stations in the network. Thus, T_s does not directly reflect Δt but the time interval between two request arrivals at any of the stations and $\tilde{R_k}$ is updated incrementally every T_s. Consequently, $e_k = \tilde{R_k} - R_{sp}$ and is also updated every T_s.

The parameters K_c and T_i must be adjusted so as to stabilize the closed loop system. We applied the Ziegler-Nichols frequency response method [5] to find proper values for the controller gain and the integral time. This method states that K_c and T_i should be chosen as follows: $K_c = 0.4K_u$ and $T_i = 0.8T_u$ where K_u represents the value of the gain for which the process starts to oscillate (while $T_i = \infty$) and T_u the period of oscillation.

3.3 Experimental Validation of the Approach

We conducted a series of experiments to examine the performance of the dispersion calculations workflow with and without feedback control. For this purpose, we implemented the workflow using the Ptolemy II Scientific Workflow System [6]. The PI controller and the simulation clock have been implemented using standard Ptolemy II actors whereas the simulation codes have been wrapped by customized Ptolemy II server actors. All simulations have been conducted within the discrete event (DE) domain provided by the Ptolemy II system.

The simulation code actor is threaded and has a first-come first-served queue of unlimited capacity. When a request arrives it is immediately processed if the actor's state is IDLE and enqueued otherwise. The actor outputs the service time of the last request upon finishing its processing, the queue length upon arrival of a new request, and the currently elapsed time upon arrival of a new request if the actor is not in IDLE state. All these values are used to compute $\tilde{R_k}$.

The PI controller receives the current value of the error and T_s as inputs and outputs T^{sc}. The simulation clock generates a new request every T^{sc} time units. This mechanism works as follows: A timer is reset every time a new request is generated and then compared to T^{sc} every tenth of a second. When the timer reaches the current value of T^{sc} a new request is issued. T^{sc} is updated every T_s time units where T_s ranges from 100 milliseconds to tenths of seconds, depending on CPU load and the level of perturbations in the system.

Test Setup. The tests were conducted under Windows Server 2008 installed on a dual quad-core Intel machine with 8GB of RAM. The amount of RAM was sufficient for installing a RAM-Drive large enough to entirely support the I/O exchange of the dispersion calculation workflow. Hence, the I/O overhead could be neglected in both the model and the experiments.

We ran a dispersion calculation with real weather and emission data for 1000 seconds in two disturbance scenarios using two scheduling strategies: feed-forward scheduling with a constant $T^{sc} = 3.071s$ and feedback scheduling using a PI controller with $R_{sp} = 30s$, $K_c = 0.75$, and $T_i = 100$. The values for T_i and K_c were determined using the Ziegler-Nichols method. The setpoint was chosen by considering the maximum total service time of the queueing network at 100% CPU utilization which yielded a value of about 25 seconds for S (while normally at low CPU utilization $S = 13.5s$). Hence, R_{sp} was set to 30 seconds in order to allow the error to take both positive and negative values even under heavy CPU load conditions. With no disturbances in the system, we obtained a 4.5x speedup with respect to the sequential workflow and an average 59% CPU utilization using the PI controller with $R_{sp} = 30s$ which yielded an average T^{sc} of $3.071s$. This value was then used for the tests with constant T^{sc} in order to obtain a similar CPU utilization.

In the first disturbance scenario, we simulated a server load burst using Prime95 [7], a CPU "torture" program that searches for very large prime numbers. Prime95 does not rely on cycle scavenging and, thus, constitutes an ideal application stress testing tool. We ran the program with the large FFT (Fast Fourrier Transformation) option and 8 torture threads corresponding to the 8 available CPU cores. The large FFT option heavily uses the CPU while only requiring some RAM and is considered to be the hardest CPU stress test facilitated by Prime95. In the second disturbance scenario, we simulated a permanent CPU core failure using a simple fault injection technique: one in 8 simulation code actor firings was deemed to fail after a random time period ranging from 1 to 4 seconds and had to be retried.

Test Results. The results of the tests are presented in figure 3. The first diagram (i.e., top-left) corresponding to a workflow run with a fixed T^{sc} of 3.071 seconds shows that, when Prime95 is started using 8 parallel threads (one on each CPU core) after an undisturbed period of about 850 seconds, the response time starts to increase with a steep linear slope; R reaches over 200 seconds after 1000 seconds of total simulation time. In fact this constitutes a proof for the correctness of our mathematical model for the total response time in a queueing network.

When a PI controller is used to control T^{sc} (see top-right diagram) we observe a sudden rise in R up to a value of 100 seconds, after which the response time settles back to around 30 seconds while T^{sc} is kept around 20 seconds. The controller action prevented the response time to exceed the 120 seconds limit imposed by law under the effect of a sudden CPU load burst. During the undisturbed phase the controller was able to follow the reference value $R_{sp} = 30s$ with a maximum deviation of $\pm 3.5s$ (excluding the initial overshoot) at an average $T^{sc} = 3.071s$ and an average CPU utilization of 59%.

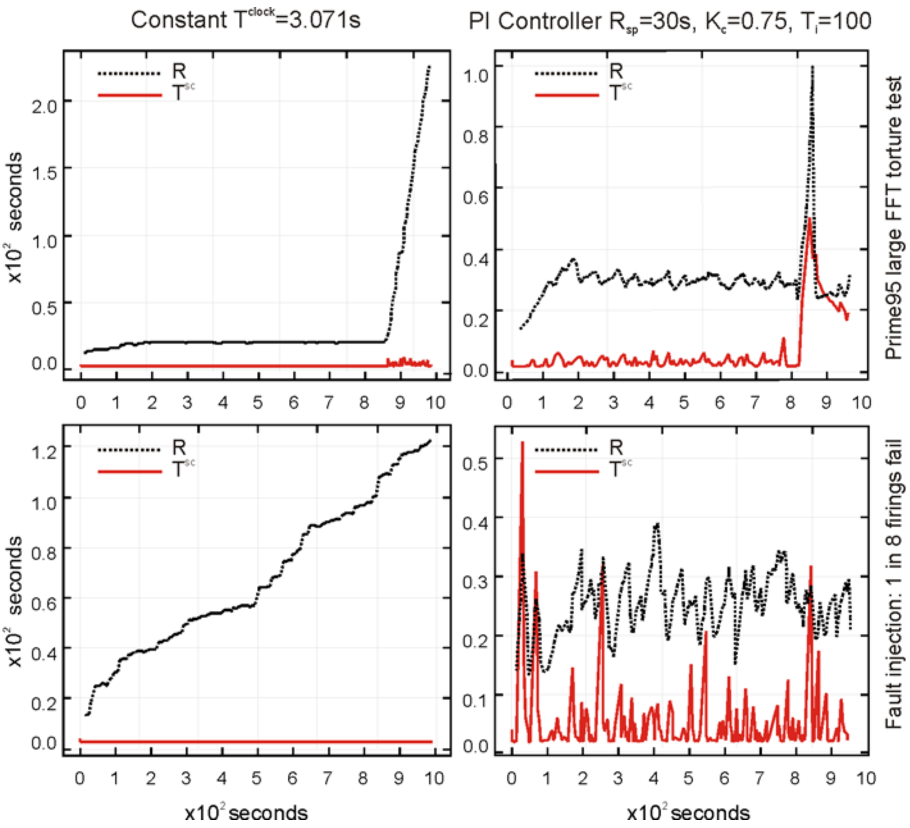


Fig. 3. Test results for fixed T^{sc} (left) and PI-controlled T^{sc} (right). Note that the diagrams were scaled to fit the boxes.

By contrast, when T^{sc} is fixed to a value of 3.071 seconds which equals the average value of the controller output during the undisturbed phase we obtain an unexpected steady state response time of 20 seconds. This shows that without feedback it is nearly impossible to obtain a desired response time. This statement is also backed up by additional failed attempts to obtain a response time of exactly 30 seconds by directly manipulating T^{sc}. For example, when $T^{sc} = 3.08s$ we obtain a steady state response time of 17.4 seconds whereas when $T^{sc} = 3.06s$ the steady state response time becomes 23.5 seconds. Also, the reason for the discrepancy between the projected response time for a given constant T^{sc} and the R_{sp} that leads to that particular T^{sc} when using a PI controller remains an open question.

The bottom diagrams show what happens when one in 8 jobs fails and needs to be restarted after a random time period between 1 and 4 seconds. Once again, when T^{sc} is fixed at 3.071 seconds (i.e., the bottom-left diagram) after 1000 seconds of simulation time the response time has a linearly increasing trend and exceeds the upper limit of 120 seconds. The PI controller (see the bottom-right diagram) is able to leverage the effects of this type of perturbation as well by preventing the response time to exceed 39 seconds which is far below the upper limit of 120 seconds.

4 Novelty of Our Approach and Related Work

Our work stands at the intersection between classic feedback control applied to computer science problems and pipeline workflow scheduling on multi-core platforms.

Classic control theory has received more attention in the computer science community since it was shown in [8] how the PI controller outperforms other algorithms for optimizing TCP/IP traffic in routers. This became possible after an exact mathematical model of the TCP/IP protocol was derived [9]. In [10] it has been shown that a PI controller was able to keep the load of a web server to a certain desired level in order to improve the quality of the service. A similar approach can be found in [11] where a queueing model based feedback control solution is introduced.

In [12] a feedback control real-time scheduling (FCS) framework for adaptive real-time systems is presented. In [13] a process pipeline for JPEG encoding is proposed whereby the pipeline workflow scheduling problem is solved using integer linear programming algorithm.

In addition to the quality of service, real-time and pipeline scheduling, queueing network modelling, and feedback control problems discussed in these papers, we also addressed the issue of fault-tolerance in early warning simulation systems. In short, the novelty of our approach consists of applying classic feedback control theory to the end of solving the scheduling problem in a discrete-event based pipeline workflow for dispersion calculations.

5 Conclusion and Future Work

We have presented a new method for scheduling discrete-event based pipeline workflows on multi-core systems. As an example application we have considered an early warning system based on legacy simulation codes for tracing the dispersion of radioactive pollutants. We successfully applied classic feedback control theory to the pipeline workflow scheduling problem. Our aim was to use a PI (proportional-integral) controller for reliably controlling the latency in the dispersion calculation workflow while also obtaining an optimal trade-off between CPU utilization, response time, and the accuracy of the results. We validated our approach through tests which showed how the PI controller achieves both the reference tracking and the disturbance rejection goals while also meeting the imposed real-time and fault-tolerance requirements, which is not the case when fixing the tick period of the simulation clock to a constant value. Overall, the method leads to a 4.5x speedup with respect to the sequential execution time with reduced implementation efforts and without modifying the implementations of the simulation codes.

As part of our future work, we intend to reduce the overshoot caused by a sudden load burst in the system (which is visible in the top-right diagram figure 3) using adaptive control and to scale out the dispersion simulation application by using a cluster of multi-core systems. For this purpose, the mathematical model presented in this work has to be extended so as to take into consideration the communication component of the service time.

References

1. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages, 4th edn. Addison Wesley, Reading (2009)
2. Misra, J.: Distributed discrete-event simulation. ACM Computing Surveys 18, 39–65 (1986)
3. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc., Englewood Cliffs (1984)
4. Legg, B.J., Raupach, M.R.: Markov-chain simulation of particle dispersion in inhomogeneous flows: The mean drift velocity induced by a gradient in eulerian velocity variance. Boundary-Layer Meteorology 24, 3–13 (1982)
5. Åström, K.J., Hägglund, T.: Advanced PID Control. ISA (2006)
6. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the ptolemy approach. Proceedings of the IEEE 91, 127–144 (2003)
7. Woltman, G.: Prime95 v25.9 (2009), http://www.mersenne.org/freesoft/
8. Hollot, C., Misra, V., Towsley, D., Gong, W.: On designing improved controllers for aqm routers supporting tcp flows. Technical report, University of Massachusetts, Amherst, MA, USA (2000)
9. Kelly, F.: Mathematical modelling of the internet. In: Proceedings of the 4th Int. Contr. on Industrial and Applied Mathematics (2000)
10. Abdelzaher, T., Lu, C.: Modeling and performance control of internet servers. In: Proceedings of the 39th IEEE Conference on Decision and Control, pp. 2234–2239 (2000)
11. Sha, L., Liu, X., Lu, Y., Abdelzaher, T.: Queueing model based network server performance control. In: Proceedings of the 23rd Real-Time Systems Symposium, pp. 81–90 (2002)
12. Lu, C., Stankovic, J., Son, S., Tao, G.: Feedback control real-time scheduling: Framework, modeling, and algorithms. Real-Time Systems 23, 85–126 (2002)
13. Benoit, A., Kosch, H., Rehn-Sonigo, V., Robert, Y.: Multi-criteria scheduling of pipeline workflows (and application to the jpeg encoder). The International Journal of High Performance Computing Applications 23, 171–187 (2009)

Lattice-Boltzmann Simulation of the Shallow-Water Equations with Fluid-Structure Interaction on Multi- and Manycore Processors

Markus Geveler, Dirk Ribbrock, Dominik Göddeke, and Stefan Turek

Institut für Angewandte Mathematik, TU Dortmund, Germany
markus.geveler@math.tu-dortmund.de

Abstract. We present an efficient method for the simulation of laminar fluid flows with free surfaces including their interaction with moving rigid bodies, based on the two-dimensional shallow water equations and the Lattice-Boltzmann method. Our implementation targets multiple fundamentally different architectures such as commodity multicore CPUs with SSE, GPUs, the Cell BE and clusters. We show that our code scales well on an MPI-based cluster; that an eightfold speedup can be achieved using modern GPUs in contrast to multithreaded CPU code and, finally, that it is possible to solve fluid-structure interaction scenarios with high resolution at interactive rates.

Keywords: High performance computing; Lattice-Boltzmann methods; shallow water equations; fluid-structure interaction; CUDA; Cell BE; multithreading.

1 Introduction and Motivation

In many practical situations, the behaviour of a fluid can be modelled by the *shallow water equations (SWE)*, e.g., for tidal flows, open water waves (such as tsunamis), dam break flows and open channel flows (such as rivers). In such cases, vertical acceleration of the fluid is negligible because the flow is dominated by horizontal movement, with its wavelength being much larger than the corresponding height. In the SWE, vertical velocity is replaced by a depth-averaged quantity, which leads to a significant simplification of the general flow equations (like the Navier-Stokes equations which are derived from general conservation and continuity laws). In the inhomogeneous SWE, source terms are employed to internalise external forces, e.g., wind shear stress and, more importantly, forces resulting from the interaction between the fluid and the *bed topography.* Using such source terms, the two-dimensional SWE can be used for the simulation of a fluid given by its free surface, which significantly reduces the computational cost and makes them a popular method for instance in (ocean-, environmental- and hydraulic) engineering.

The Lattice-Boltzmann method (LBM) is a modern numerical technique that starts with a fully discrete model rather than discretising a set of partial differential equations and solving them directly. One of the key features of the

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 92–104, 2010.

LBM is that an implementation in parallel is comparably easy, which makes it a promising method, especially in view of modern computational hardware, which evolves towards massive fine-grained parallelism (see below).

Besides efficiency, a key feature of a method for advanced simulations involving a fluid is the capability of letting it interact with its environment. This interaction includes the internalisation of the 'world geometry' in terms of the surface the fluid is streaming over and interaction with rigid bodies that move through and are moved by the fluid (fluid-structure interaction, FSI). An algorithm that provides both reasonable performance on modern commodity based computer systems on the one hand and FSI functionality on the other hand is very attractive in engineering and for example in computer graphics, ranging from feature film to games and interactive environments.

During the past few years, computer architecture has reached a turning point. Together, the memory, power and instruction-level parallelism (ILP) wall form a 'brick wall' [1], and performance is no longer increased by frequency scaling, but by parallelisation and specialisation. Commodity CPUs have up to six cores, the Cell processor is heterogeneous, and throughput-oriented fine-grained parallel designs like GPUs are transitioning towards becoming viable general purpose compute resources. On the software side, programming models for fine-grained parallelism are subject to active discussion and are rapidly evolving. Programmers have to adapt to this inevitable trend, because compiler support is on the far horizon if at all, in particular for computations with low arithmetic intensity (ratio of arithmetic operations per memory transfer). Established parallelisation strategies for both shared and distributed memory architectures have to be revisited, and different strategies are necessary for different architectures.

1.1 Related Work

Fan et al. [2] were the first to implement a Lattice-Boltzmann solver on a cluster of GPUs. Advanced Lattice-Boltzmann solvers on CPUs and GPUs have been implemented by Tölke and Krafczyk [3], Thürey [4] and Pohl [5]. Many publications are concerned with interactive and (visually) accurate simulations of fluid flow [6–9].

1.2 Paper Contribution and Paper Overview

In Section 2.1 and Section 2.2 we briefly review the shallow water equations, and their solution using the Lattice-Boltzmann method with support for internal boundaries. In Section 2.3 we present modifications to the LBM to incorporate more realistic simulation scenarios with nontrivial bed topologies, in particular the dynamic flooding and drying of areas. Furthermore, this section describes our approach to couple the simulation of fluids with moving solid objects that influence the behaviour of the fluid.

Section 3 is dedicated to parallelisation and vectorisation techniques for the FSI-LBM solver. We present efficient algorithms for all levels of parallelism encountered in modern computer architectures. In Section 4 we demonstrate the applicability and performance of our approach for several prototypical benchmark problems. Performance is evaluated on a cluster of conventional CPUs communicating via MPI, on multi-socket multi-core SMP systems, on a Cell blade, and on modern fully programmable GPUs. We are convinced that such algorithmic studies with respect to exploiting parallelism on various levels for a given application are necessary at this point, in particular in view of the challenges outlined in this section. We conclude with a summary and a discussion in Section 5.

2 Mathematical Background

2.1 Shallow Water Equations

Using the Einstein summation convention (subscripts i and j are spatial indices) the two-dimensional shallow water equations in tensor form read

$$\frac{\partial h}{\partial t} + \frac{\partial (hu_j)}{\partial x_j} = 0 \quad \text{and} \quad \frac{\partial hu_i}{\partial t} + \frac{\partial (hu_i u_j)}{\partial x_j} + g\frac{\partial}{\partial x_i}(\frac{h^2}{2}) = S_i^b, \tag{1}$$

where h is the fluid depth, $\mathbf{u} = (u_1, u_2)^T$ its velocity in x- and y-direction, and g denotes the gravitational acceleration. In addition, we apply a source term S_i^b which internalises forces acting on the fluid due to the slope of the bed and material-dependent friction:

$$S_i^b = S_i^{\text{slope}} + S_i^{\text{friction}}. \tag{2}$$

The slope term is defined by the sum of the partial derivatives of the bed topography, weighted by gravitational acceleration and fluid depth (b denotes the bed elevation), and we define the friction term using the Manning equation (as suggested by Zhou [10]).

$$S_i^{\text{slope}} = -gh\frac{\partial b}{\partial x_i}, \qquad S_i^{\text{friction}} = -gn_b^2 h^{-\frac{1}{3}} u_i \sqrt{u_j u_j}, \tag{3}$$

n_b denotes a material-specific roughness coefficient. Using the inhomogeneous SWE with source term (2) enables the simulation of a fluid (bounded by its surface) with a two-dimensional method due to the coupling of fluid motion with the bed topography.

2.2 Lattice-Boltzmann Method

In order to solve problem (1) with some initial conditions $h(\mathbf{x}, t = 0)$, $\mathbf{u}(\mathbf{x}, t = 0)$ and a constant bed topography, $b(\mathbf{x})$, we apply the Lattice-Boltzmann method (LBM) with a suitable equilibrium distribution to recover the SWE. In the LBM, the fluid behaviour is determined by particle populations residing at the sites of a regular grid (the *lattice*). The particles' movement (*streaming*) is restricted to fixed trajectories $\mathbf{e}_\alpha$ (*lattice velocities*) defined by a local neighbourhood on the lattice. We use the D2Q9 lattice, which defines the lattice velocities in the direction of the eight spatial neighbours as

$$\mathbf{e}_\alpha = \begin{cases} (0,0) & \alpha = 0 \\ e(\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}) & \alpha = 1,3,5,7 \\ \sqrt{2}e(\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}) & \alpha = 2,4,6,8, \end{cases} \tag{4}$$

with $e = \frac{\Delta x}{\Delta t}$ being the ratio of lattice spacing and timestep. Particle behaviour is defined by the Lattice-Boltzmann equation and a corresponding *collision* operator. Here, the Lattice-Bhatnagar-Gross-Krook (LBGK) collision operator [11] is used, which is a linearisation of the collision-integral around its equilibrium state with a single uniform relaxation time τ. Using this relaxation, the Lattice-Boltzmann equation can be written as

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + \frac{\Delta t}{6e^2} e_{\alpha i} S_i^b \ , \alpha = 0, \ldots, 8, \tag{5}$$

where f_α is the particle distribution corresponding to the lattice-velocity $\mathbf{e}_\alpha$ and f_α^{eq} a local equilibrium distribution, which defines the actual equations that are solved. In order to recover the SWE, a suitable f_α^{eq} has to be defined for every lattice-velocity. Zhou [10] has shown that the equilibria can be written as

$$f_\alpha^{\mathrm{eq}} = \begin{cases} h(1 - \frac{5gh}{6e^2} - \frac{2}{3e^2} u_i u_i) & \alpha = 0 \\ h(\frac{gh}{6e^2} + \frac{e_{\alpha i} u_i}{3e^2} + \frac{e_{\alpha j} u_i u_j}{2e^4} - \frac{u_i u_i}{6e^2}) & \alpha = 1,3,5,7 \\ h(\frac{gh}{24e^2} + \frac{e_{\alpha i} u_i}{12e^2} + \frac{e_{\alpha j} u_i u_j}{8e^4} - \frac{u_i u_i}{24e^2}) & \alpha = 2,4,6,8 \end{cases} \tag{6}$$

and that the SWE can be recovered by applying Chapman-Enskog expansion on the LBGK approximation ([5]). Finally, macroscopic mass (fluid depth) and velocity can be obtained by

$$h(\mathbf{x}, t) = \sum_\alpha f_\alpha(\mathbf{x}, t) \quad \text{and} \quad u_i(\mathbf{x}, t) = \frac{1}{h(\mathbf{x}, t)} \sum_\alpha e_{\alpha i} f_\alpha, \tag{7}$$

respectively. We use the popular bounce-back rule as boundary conditions, where particles are reflected using opposite outgoing directions and which therefore implements no-slip boundary conditions. In the following, this basic method is used as a starting point for our more sophisticated solver, capable of dealing with complex flow scenarios, including the interaction with moving solid obstacles.

2.3 Dry-States and Fluid Structure Interaction

The LBM for the shallow water equations presented above can interact with the bed surface and therefore is not restricted to simple scenarios (as it would be if an equilibrium distribution function corresponding to the two-dimensional Navier-Stokes equations had been used). However, it is restricted to subcritical[1]

[1] Usually, the term *critical flow* is associated with a Froude number being smaller than one. Throughout this paper, we use it for flows over a bed topography with possibly very small (or even zero-valued) fluid depth.

flows, i. e., the fluid depth is significantly greater than zero. The first extension to the method aims at allowing so-called *dry-states*, since the dynamic drying and wetting of the bed topography is a feature desired by many applications. In our approach, we define a fluid depth below a specified small threshold parameter as *dry* and set the macroscopic velocity at dry sites to zero, to avoid the division by zero in the extraction phase of the original algorithm (the evaluation of equation (7)). Secondly, local oscillations caused by critical non-zero fluid-depths are confined with an adaptive limiter approach. Due to space constraints, we refer to Geveler [12] for details.

In order to simulate rigid bodies moving within the fluid, the method described so far is extended by three major algorithmic steps: In the first step, the forces acting on the fluid due to a moving boundary have to be determined. We use the so-called BFL-rule [13], which interpolates the momentum values for the consistency with non-zero Dirichlet boundary conditions induced by a moving boundary. The interpolation is achieved by taking values in opposite direction of the solid movement similar to the bounce-back rule, see Figure 1. In the original BFL formulation, the interpolation needs four coefficients depending on the distance $q = \frac{|\mathbf{b}_\beta - \mathbf{x}|}{\Delta x}$ where $\mathbf{e}_\beta$ is the lattice-velocity approximating the direction of the solid movement, $\mathbf{b}_\beta$ the corresponding point on the solid boundary and $\mathbf{x}$ a location in the fluid (and on the lattice) in opposite direction. In our approach, we use a piecewise linear approximation of the boundary, and set $q = \frac{1}{2}$, which reduces three of the four coefficients to zero. We obtain a very simple formula[2] for the missing momentum, that still respects the moving boundary. Let $\mathbf{u}_B$ be the macroscopic velocity of the solid, our modified boundary condition then reads:

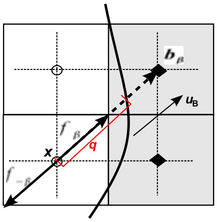


Fig. 1. Modified *bounce-back* scheme for moving boundaries

$$f_{-\beta}^{\text{temp}}(\mathbf{x}, t + \Delta t) = 6\Delta x \; w_{-\beta}(\mathbf{u}_B(\mathbf{b}_\beta) \cdot \mathbf{e}_{-\beta}). \tag{8}$$

The superscript temp indicates the distribution function after the collision step. The w_α are weights depending on the lattice, which can be set to $\frac{4}{9}$ for $\alpha = 0$ and $\frac{1}{9}$ for uneven α and $\frac{1}{36}$ in the even case for our D2Q9 lattice, see Caiazzo [14].

The second major step in performing FSI is the extrapolation of missing macroscopic quantities. Moving solids imply that the lattice is in general not

[2] It should be noted, that this simplification increases performance but reduces the spatial convergence order to one.

invariant over time: Lattice sites that belong to a solid region at time t may become fluid sites at time $t + \Delta t$. In this case, the missing quantities have to be reconstructed. We use an indirect so-called *equilibrium refill* method proposed for example by Caiazzo [14], which uses a three point-backward approximation after calculating the opposite direction of the solid's movement in order to use one-dimensional extrapolation only. Again, the value $q = \frac{1}{2}$ is used and we obtain

$$\begin{aligned}\tilde{h}(\mathbf{x}, t + \Delta t) = {} & 3h(\mathbf{x} + \mathbf{e}_{-\beta}\Delta t, t + \Delta t) - 3h(\mathbf{x} + 2(\mathbf{e}_{-\beta}\Delta t), t + \Delta t) \\ & + h(\mathbf{x} + 3(\mathbf{e}_{-\beta}\Delta t), t + \Delta t)\end{aligned} \tag{9}$$

for the extrapolated fluid depth and

$$\begin{aligned}\tilde{\mathbf{u}}(\mathbf{x}, t + \Delta t) = {} & 8/15\Delta x \mathbf{u}_B(\mathbf{b}_\beta, t + \Delta t) + 2/3\mathbf{u}(\mathbf{x} + \mathbf{e}_{-\beta}\Delta t, t + \Delta t) \\ & - 2/5\mathbf{u}(\mathbf{x} + 2(\mathbf{e}_{-\beta}\Delta t), t + \Delta t)\end{aligned} \tag{10}$$

for the macroscopic velocities, respectively.

Finally, the force acting on the solid due to fluid movement is determined by the Momentum-Exchange algorithm (MEA) [15], in order to be able to couple the method with a solid mechanics (CSM) solver. The MEA uses special distribution functions to compute the moments resulting from incoming particles and outgoing particles corresponding with a single lattice-velocity $-\beta$ at a solid (boundary) point $\mathbf{b}$:

$$f_{-\beta}^{\mathrm{MEA}}(\mathbf{b}, t) = e_{\beta i}(f_{\beta}^{\mathrm{temp}}(\mathbf{x}, t) + f_{-\beta}^{\mathrm{temp}}(\mathbf{x}, t + \Delta t)). \tag{11}$$

The forces can be aggregated into the total force acting on $\mathbf{b}$:

$$F(\mathbf{b}, t) = \sum_{\alpha} f_{\alpha}^{\mathrm{MEA}}(\mathbf{b}, t). \tag{12}$$

3 Implementation and Parallelisation

3.1 Modular FSI-LBM Solver

The combination of all functionality presented in Section 2 results in the solver given by Algorithm 1. Note that the algorithm is designed in a modular way in order to be able to activate/disable certain functionality, for instance to disable the FSI components in scenarios without moving objects.

3.2 Efficient Parallelisation and Vectorisation

It can be seen in Algorithm 1 that parallelism is trivially abundant in the modified LBM solver: All work performed for each lattice site is independent of all other sites (in the basic algorithm). However, this general observation does not lead in a straightforward manner to an efficient parallelisation, and in particular vectorisation. Our implementation supports coarse-grained parallelism for distributed memory systems, medium-grained parallelism on multicore shared memory machines, and fine-grained parallelism corresponding to

Algorithm 1. LBM solver for SWE with FSI

```
perform preprocessing → $h(\mathbf{x},0), \mathbf{u}(\mathbf{x},0)$
for all timesteps
  approximate extrapolation direction → $-\beta$
  determine lattice sites to be initialised
  for all lattice sites to be initialised
    initialise fluid sites (equations (9) und (10))
  for all fluid sites
    for $\alpha = 0$ to 8:
      compute equilibrium distribution functions (equation (6))
      perform LBGK collision
      compute momentum exchange (equations (11) and (12))
      for all fluid sites adjacent to moving boundary
        apply modified BFL-rule (equation (8))
      compute and apply source-terms (equations (2) and (3))
      perform LBM streaming
      for all boundary fluid sites not adjacent to moving solid
        apply standard bounce-back scheme
      extract physical quantities (equation (7))
```

the SIMD paradigm. The latter is important not only in the SSE units of conventional CPUs, but also on graphics processors. For instance, the SIMD width is 32 on current NVIDIA CUDA-capable GPUs. We apply the same techniques for coarse- and medium-grained parallelism on CPUs; and for fine-grained parallelism within CPU cores and GPU multiprocessors, respectively. Only the actual implementation of the algorithms varies for different architectures, see Section 3.4.

The SIMD paradigm implies that branches should be avoided in the innermost loops, because otherwise serialisation of the branches occurs. In the context of our FSI-LMB solver, sites can be fluid, solid, dry or moving boundary, and each type has to be treated differently. Furthermore, different computations are performed in the collision steps for the nine lattice velocities of the D2Q9 model. For an efficient vectorisation, we want to store all data contiguously in memory. A special packing algorithm is used to determine the largest connected areas in the given domain: For the basic solver without source terms and FSI, all obstacle sites can be eliminated, as they contain no fluid throughout the entire calculation. In a second step, all remaining sites are classified with respect to their neighbours in all nine directions in a similar way as it has been proposed by Krafczyk et al. [16]. For example, if the northern neighbours of two adjacent lattice-sites are also adjacent and have the same boundary conditions, the solver can process these sites in a vectorised manner without branches. However, for the advanced algorithm employing FSI, this *lattice-compression* technique is not suitable since the lattice is dynamically altered by the movement of the solids. In this case, the packing algorithm is only run once in the preprocessing phase, packing only *stationary* obstacles as in the original algorithm. Dynamic lattice transformation in the actual simulation is achieved by tagging lattice sites either as *fluid*, *solid* or *fluid-boundary*, etc. In all cases, the packed data is stored in one-dimensional arrays that contain areas of lattice-sites with related neighbours and the same boundary conditions. Despite vectorisation, the approach also ensures good spatial and temporal locality of the computations.

To be able to distribute the solver calculation across various cores and to calculate the solution in parallel, the domain (the packed data vectors) is partitioned into different nearly independent parts. We pad each local array with a few ghost entries, allowing it to synchronise with its predecessor and successor. As we use a one-dimensional data layout, each part has only two direct neighbour parts to interact with. After each time step every part sends its own results corresponding to subdomain boundaries to the ghost sites of its direct neighbours. As soon as every part has finished these independent, non-blocking transfers, the next time step calculation can begin. On shared memory architectures, the synchronisation phase does not involve message passing, but can be realised via locks on shared data, and thus the procedure is conceptually the same. Consequently, the communication between the different parts is very efficient, because it involves no serialisation.

3.3 Source Terms and FSI Implementation

The partial derivatives in the slope source term (3) are evaluated by means of the semi-implicit *centred* scheme proposed by Zhou [10], where the slope is computed at the midpoint between the lattice-site and one neighbouring lattice-site and therefore includes the interpolation of the fluid depth:

$$S_i^{\text{slope}} = S_i^{\text{slope}}(\mathbf{x} + \frac{1}{2}\mathbf{e}_\alpha \Delta t, t) \tag{13}$$

With this approach, we are able to achieve a horizontal steady-state even when nonplanar bed topographies are involved, see Section 4.1.

Besides the source terms, two additional solver modules are needed to provide FSI functionality. To increase efficiency, Algorithm 1 can be reformulated, resulting in a fused kernel that performs LBGK collision and LBM streaming and all steps between these two. The corresponding module also automatically corrects the streaming of particles that are influenced by a moving boundary, i. e., a boundary that is not treated as a solid obstacle by our packed lattice data structure. In addition, the BFL rule is applied and the MEA distribution functions are computed. The second FSI module performs the initialisation of fluid sites with all necessary extrapolations.

Keeping track of the lattice flags is achieved by boolean vectors directly corresponding to the compressed data vectors needed by the basic solver and calculations concerning these flags, e.g., determining fluid sites that need to be initialised, are performed on the fly.

3.4 Hardware-Oriented Implementation

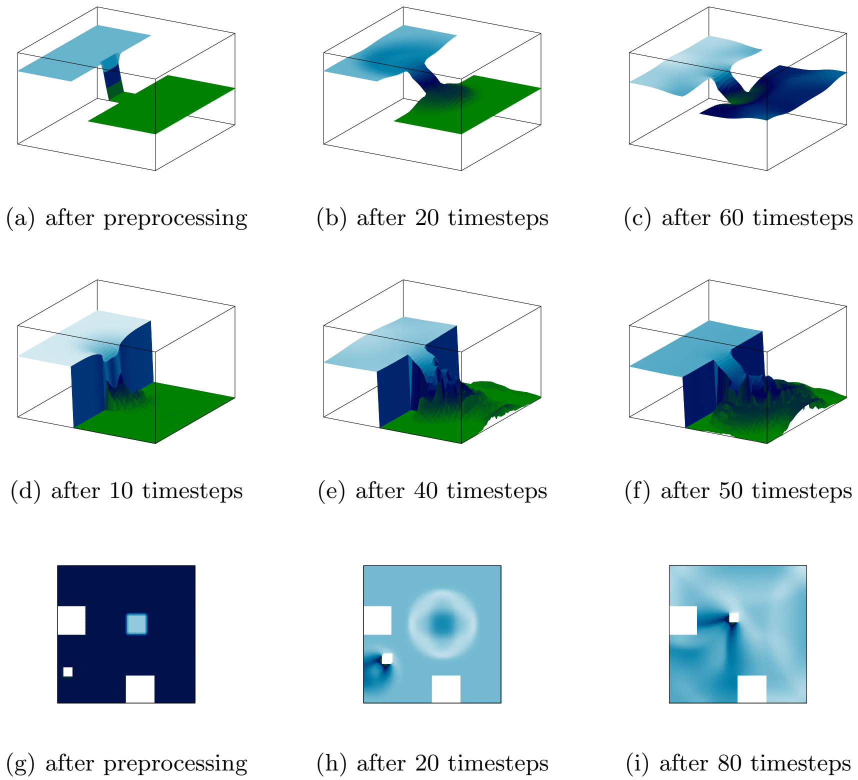
The solver is built on top of the HONEI libraries [17] to be able to use the different target hardware plattforms efficiently. HONEI provides a wide range of software backends to access different hardware via a unified interface. Its generic backend approach enables the programmer to develop code without having to care about specific hardware details, and applications built on top of the libraries are written only once and can directly benefit from hardware acceleration by simply designating them with a hardware tag. Furthermore, the backend-specific infrastructure eases the development of application-specific functionality, because hardware specific optimisation has not to be done from scratch: The CPU backend is built around SSE intrinsics. The multicore backend uses PThreads to provide an abstract thread type and tools to execute and synchronise these threads. The GPU backend is based on NVIDIA CUDA and provides simplified access to any CUDA-enabled GPU. In addition, all memory transfers between main memory and the GPU device memory are done automatically and executed only if necessary. The Cell backend enables support for the IBM Cell BE and grants a comfortable way to create custom SPE programs on top of the IBM SPE libraries. Finally, the MPI backend encapsulates the common message passing interface.

4 Results

4.1 Validation

Figure 2 demonstrates the applicability of our solver for various dam break scenarios including the flooding of dry areas and self-propelled objects moving through the resulting fluid surface. We abstain from giving detailed numerical test results here, and refer to the theses of the first two authors [12, 18]. There, it is shown that all solver configurations provide good accuracy in terms of mass conservation, smoothness of quantity-fields and stability, as well as a comparison in terms of accuracy with a solver using a finite element discretisation of the Navier-Stokes equations. The treatment of dry-states by the experimental limiter methods combined with the cutoff of particle populations at the fluid-solid interface may lead to a small loss of mass. Nonetheless, the FSI-LMB solver always computes a stable and visually accurate solution.

The first scenario we present (top row in the figure) is a partial dam break simulation, a standard benchmark problem for shallow water solvers. For this test case, the modules treating source terms, dry-states and FSI are deactivated in our modular solver. The middle row in the figure depicts the same partial dam break simulation, but with supercritical initial fluid depth (dry-states). The results show that such configurations can be stabilised by our solver with no significant loss of mass. The third scenario (bottom row) contains two stationary obstacles and one moving solid coupled with a full cuboidal dam break simulation. Furthermore, this configuration includes the nontrivial bed topography given by $f_\gamma(x, y) = \gamma(x^2 + y^2)$ and is therefore, even though no dry-states are present, a test case for the full functionality of our solver, because all solver modules including FSI and the treatment of source terms are activated. In all simulations with an uneven bed geometry present, the steady-state solution always cancels out the non-vanishing forces and converges to a horizontal plane.



(a) after preprocessing (b) after 20 timesteps (c) after 60 timesteps

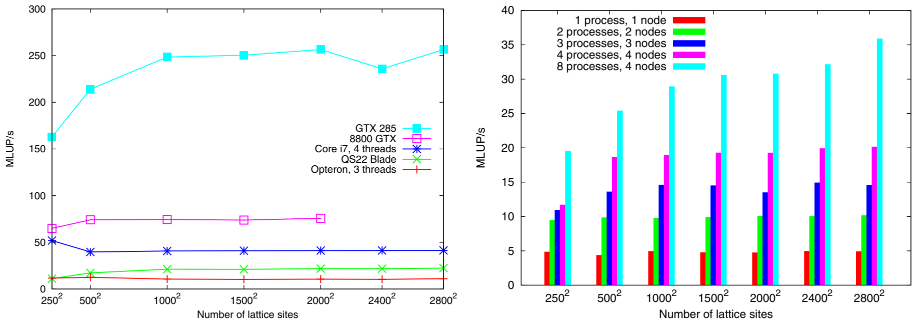
(d) after 10 timesteps (e) after 40 timesteps (f) after 50 timesteps

(g) after preprocessing (h) after 20 timesteps (i) after 80 timesteps

Fig. 2. Partial dam break simulations. Top: without source terms and FSI (the lower basin is filled with water initially); middle: with dry-states (the lower basin is empty initially), without FSI; bottom: full FSI simulation (bird's eye view) with cuboidal dam break.

4.2 Performance Benchmarks

We first evaluate the performance on different multi- and manycore architectures of the basic solver without its source term and FSI modules. Figure 3 (left) shows the mega lattice updates per second (MLUP/s) for increasing lattice size, simulating a partial dam break scenario like the one depicted in Figure 2 (top row). The CPU SSE/multicore backend is evaluated on a dual-socket dual-core AMD Opteron 2214 system and an Intel Core i7 quadcore workstation. The Cell backend is tested with an IBM QS22 blade with two Cell BE processors. The GPU-based solver is executed on a NVIDIA GeForce 8800 GTX and a GeForce GTX 285. The QS22 blade executes twice as fast as the Opteron system, but is outperformed by a factor of two by the Core i7 system. Even the older GTX 8800 outperforms all CPU systems but is restricted to small lattice sizes due to its comparatively small amount of device memory. Finally, the GTX 285 reaches eight times the performance of the fastest CPU system. These speedup factors are



(a) Performance on different architectures

(b) Strong scalability with MPI

Fig. 3. Partial dam break benchmark on different architectures

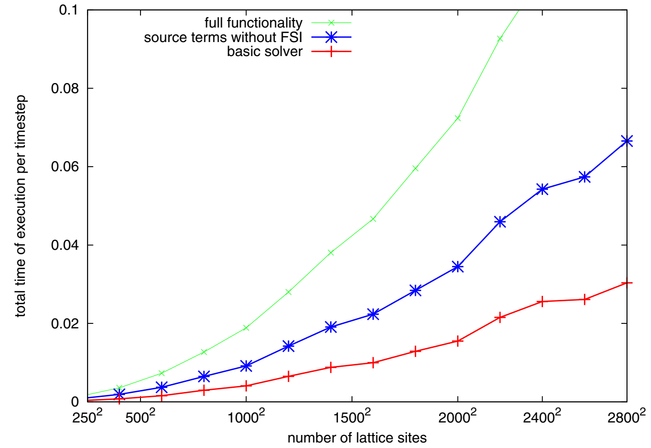


Fig. 4. Time per timestep (in seconds) for the three scenarios with corresponding solver configurations (basic solver, with source terms, full functionality with FSI) on the GeForce GTX 285

proportional to the bandwidth to off-chip memory of the various architectures, ranging from 6.4 GB/s per socket (Opteron), 25 GB/s (Cell) and 33 GB/s (i7) to 87 GB/s and 160 GB/s for the two GPUs. Detailed measurements reveal that only the kernel responsible for computing the equilibrium distribution functions is compute-bound, all other operations are limited in performance by the memory bandwidth [18]. Figure 3 (right) shows almost perfect strong scaling of the MPI backend on a small cluster of Opteron 2214 nodes.

Our last experiment compares the performance of increasingly complex solver configurations on the GeForce GTX 285 by simulating the three test cases described above. We emphasise that the speedup of the GPU over other architectures is in line with the measurements for the basic LBM solver, even with full FSI functionality. The timing measurements in Figure 4 demonstrate that all solver configurations can cope with high resolutions in reasonable time. For example, even the full algorithm can compute a single timestep on a lattice with approximately 1.7 million lattice-sites in less than 0.04 seconds, corresponding to 30 timesteps per second. The more advanced solvers do not benefit from the static lattice compaction (cf. Section 3.2) to the same extent as the basic solver, because the domain changes in the course of the simulation. Besides the additional computational effort, the loss in performance compared to the basic solver is therefore certainly due to the increase in conditional branches in the code.

5 Conclusions and Future Work

The combination of the shallow water equations and a suitable Lattice-Boltzmann approach with methods to stabilise dry-states as well as for fluid-structure interaction has been used for a novel approach to simulate 'real-world' free surface fluids. With the presented algorithm, the problem of lacking computational resources for the direct solution of flow equations can be overcome if quantitative accuracy is less important than, for example, visual appearance as it is the common case in computer graphics or entertainment.

In addition to this numerical modeling, we implemented our method on a wide range of parallel architectures, addressing coarse, medium and fine-grained parallelism. In future work, we will explore heterogeneous systems, e.g., the simultaneous use of the CPU cores and GPUs in cluster nodes, to maximise the computational efficiency.

Acknowledgements

We would like to thank Danny van Dyk, Sven Mallach and all contributors to HONEI. This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under the grant TU 102/22-2, and by BMBF (call: HPC Software für skalierbare Parallelrechner) in the SKALB project (01IH08003D / SKALB). Thanks to NVIDIA for generous hardware donations, and to IBM Germany for access to QS22 blades.

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
2. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: SC 2004: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, p. 47 (2004)

3. Tölke, J., Krafczyk, M.: TeraFLOP computing on a desktop PC with GPUs for 3D CFD. International Journal of Computational Fluid Dynamics 22(7), 443–456 (2008)
4. Thürey, N., Iglberger, K., Rüde, U.: Free Surface Flows with Moving and Deforming Objects for LBM. In: Proceedings of Vision, Modeling and Visualization 2006, pp. 193–200 (2006)
5. Pohl, T.: High Performance Simulation of Free Surface Flows Using the Lattice Boltzmann Method. PhD thesis, Universität Erlangen-Nürnberg (2008)
6. Molemaker, M.J., Cohen, J.M., Patel, S., Noh, J.: Low viscosity flow simulations for animations. In: Gross, M., James, D. (eds.) Eurographics / ACM SIGGRAPH Symposium on Computer Animation (2008)
7. van der Laan, W.J., Green, S., Sainz, M.: Screen space fluid rendering with curvature flow. In: I3D 2009: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, pp. 91–98. ACM, New York (2009)
8. Baboud, L., Décoret, X.: Realistic water volumes in real-time (2006)
9. Krüger, J.: A GPU Framework for Interactive Simulation and Rendering of Fluid Effects. PhD thesis, Technische Universität München (2006)
10. Zhou, J.G.: Lattice Boltzmann methods for shallow water flows. Springer, Heidelberg (2004)
11. Higuera, F.J., Jimenez, J.: Boltzmann approach to lattice gas simulations. EPL (Europhysics Letters) 9(7), 663–668 (1989)
12. Geveler, M.: Echtzeitfähige Interaktion von Festkörpern mit 2D Lattice–Boltzmann Flachwasserströmungen in 3D Virtual–Reality Anwendungen. Diploma thesis, Technische Universität Dortmund (2009)
13. Bouzidi, M., Firdaouss, M., Lallemand, P.: Momentum transfer of a Boltzmann-lattice fluid with boundaries. Physics of Fluids 13(11), 3452–3459 (2001)
14. Caiazzo, A.: Asymptotic Analysis of lattice Boltzmann method for Fluid-Structure interaction problems. PhD thesis, Technische Universität Kaiserslautern, Scuola Normale Superiore Pisa (2007)
15. Caiazzo, A., Junk, M.: Boundary forces in lattice Boltzmann: Analysis of momentum exchange algorithm. Computaters & Mathematics with Applications 55(7), 1415–1423 (2008)
16. Krafczyk, M., Lehmann, P., Philippova, O., Hänel, D., Lantermann, U.: Lattice Boltzmann Simulations of complex Multi-Phase Flows. Springer, Heidelberg (2000)
17. van Dyk, D., Geveler, M., Mallach, S., Ribbrock, D., Göddeke, D., Gutwenger, C.: HONEI: A collection of libraries for numerical computations targeting multiple processor architectures. Computer Physics Communications 180(12), 2534–2543 (2009)
18. Ribbrock, D.: Entwurf einer Softwarebibliothek zur Entwicklung portabler, hardwareorientierter HPC Anwendungen am Beispiel von Strömungssimulationen mit der Lattice Boltzmann Methode. Diploma thesis, Technische Universität Dortmund (2009)

FPGA vs. Multi-core CPUs vs. GPUs: Hands-On Experience with a Sorting Application

Cristian Grozea[1,⋆], Zorana Bankovic[2], and Pavel Laskov[3]

[1] Fraunhofer Institute FIRST,
Kekulestrasse 7, 12489 Berlin, Germany
cristian.grozea@first.fraunhofer.de

[2] ETSI Telecomunicación, Technical University of Madrid,
Av. Complutense 30, 28040 Madrid, Spain
zorana@die.upm.es

[3] Wilhelm Schickard Institute for Computer Science
University of Tuebingen, Sand 1, 72076 Tuebingen
pavel.laskov@uni-tuebingen.de

Abstract. Currently there are several interesting alternatives for low-cost high-performance computing. We report here our experiences with an N-gram extraction and sorting problem, originated in the design of a real-time network intrusion detection system. We have considered FPGAs, multi-core CPUs in symmetric multi-CPU machines and GPUs and have created implementations for each of these platforms. After carefully comparing the advantages and disadvantages of each we have decided to go forward with the implementation written for multi-core CPUs. Arguments for and against each platform are presented – corresponding to our hands-on experience – that we intend to be useful in helping with the selection of the hardware acceleration solutions for new projects.

Keywords: parallel sort, FPGA, GPU, CUDA, multi-core, OpenMP, VHDL.

1 Introduction

Low-cost high-performance computing is a recent development that brings computing power equivalent to former supercomputers to ordinary desktops used by programmers and researchers. Harnessing this power, however, is non-trivial, even with a growing availability of tools for facilitating the transition from traditional architectures. Successful use of possibilities offered by modern parallel architectures is still largely application-dependent and more often than not necessitates rethinking of programming paradigms and re-design of software.

In this contribution, we describe the experience of a practical transition to multi-core architecture in a specific application that requires high performance

[⋆] Corresponding author.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 105–117, 2010.

and low latency - real-time network intrusion detection. The goal of a network intrusion detection system (IDS) is to detect malicious activity, e.g. buffer overflow attacks or web application exploits, in incoming traffic. Both performance and latency are of crucial importance for this application if decisions must be made in real time whether or not to allow packets to be forwarded to their destination.

An anomaly-based network IDS ReMIND [3] developed in our laboratory is designed for detection of novel, previously unseen attacks. Unlike traditional. signature-based IDS which look for specific exploit patterns in packet content, our IDS detects packets with highly suspicious content. The crucial component of our detection algorithms is finding of matching subsequences in packet content, a problem that requires efficient algorithms for sorting such subsequences.

The goal of the project described in this contribution was to accelerate existing sequence comparison algorithms (see e.g. [26,18] for details) to work at a typical speed of an Ethernet link of 1 Gbit/s by using parallel architectures.

There exist at least four alternatives in this field: FPGA devices and various boards using them (ranging from prototyping boards to FPGA based accelerators, sometimes general purpose, sometimes specialized) [29]; multi-core CPUs, with 2, 3, 4 cores or more [10]; many-core GPUs, with 64,128, 240 or even more cores [25]; the Cell processor [13,32]. The former three solutions were considered and implemented in the course of the current study.

Let us now describe the specific setting of the problem for which an acceleration was sought. The ReMIND system receives packets from a network interface and, after some preprocessing, transforms them into byte strings containing application-layer payload. For a decision to be made, a set of N-grams (substrings of length N) must be extracted from each string (the values of N that work best are in typically in the range of 4 to 8), and compared to the set of N-grams in the prototype; the latter set has been previously extracted from payload of normal packets. The comparison essentially amounts to computing the intersection of two sets of N-grams. An efficient linear-time solution to this problem involves lexicographic sorting of all N-grams in the incoming string (linear-time, low constants). The sorting component takes a string and is supposed to return some representation of sorted N-grams in this string, for examples an index set containing the positions of the respective N-grams, in a sorted order, in the original string[1]. The incoming strings can be up to 1480 bytes long (maximal length of an Ethernet frame), and to achieve 1 Gbit/s speed, approximately 84,000 full-length packets must be handled per second. Processing of single packets can be assumed independent from each other since, in the simplest case, decisions are made independently on each packet.

We will now proceed with the description of the particular methods and implementations followed by the presentation and discussion of experimental results.

[1] Returning a sorted set of N-grams itself blows up the size of the data by a factor of up to N.

2 Methods

The following hardware platforms were available for our implementation:

- FPGA: Xilinx evaluation board ML507 equipped with a Virtex-5 family XC5VFX70T FPGA, maximum clock frequency 0.55 GHz, 70 thousand logic cells.
- CPUs: Dell Precision T7400 with two Xeon 5472 quad-core, clock speed 3GHz and 16GB of RAM.
- GPUs: Two Nvidia Quadro FX 5600, with 1.5 GB of RAM and 128 v1.0 CUDA shaders each - the clock of the shaders: 1.35 GHz.

The details of a hardware design and/or algorithms for each platform are presented below.

2.1 FPGA

The ML507 board used offers a PCIe 1x connection to the hosting PC, and the setup was to capture data from network on the PC, send data over the PCIe to the FPGA board, sort it there, and send the results back to the PC. All communication with the board had to be implemented as DMA transfers, for efficiency reasons and in order to overlap communication and processing both on the PC side and on the FPGA side. This requirement proved to be very difficult to fullfil as the support of Xilinx did not include the sources of a complete application + driver solution, just a performance demo with some of the components only in binary form [4].

Let L be the length of the list to sort. It is interesting to note that the FPGA could do all comparisons between the elements of an L-long sequence in $O(1)$ time complexity and $O(L^2)$ space complexity, by specifying a comparator for every pair of elements in the list.

When one targets efficiency on a serial or multi-core CPU implementation, a good order and good constant algorithm is chosen and this usually solves the problem. When working with FPGAs, the things are more complicated. A smart but more complex algorithm can solve the problem in less steps, but the maximum clock speed usable depends on the complexity of the design/algorithm implemented, so the net effect of using a smarter algorithm can be of slowing down the implementation. This is exactly what has happened to us.

The complex sort-merge sorting. This algorithm was specialized in sorting $L = 1024$ elements lists. In a first stage, the incoming data is grouped into groups of 8 items, sorted in $O(1)$ with the extensive comparators network described above. Every 4 such groups are merged with a 4-way merging network producing a 32-elements group. The 32 groups of 32 elements each are repeatedly merged with a tree of parallel 4-way merging nodes, which outputs on its root in sorted sequence the 1024 items. Despite the clever design and the tight and professional VHDL coding of this algorithm, it went past the resources of the FPGA chip we used. After a synthesis process (the rough equivalent for FPGA of compiling to object files) that took two days (not unheard of in the FPGA world), the numbers reported for resources usage were much beyond the available resources: 235% LUTs, 132% block RAMs, max frequency 60MHz. While the maximum frequency was above the 50MHz needed with this design, everything else was beyond physical limits. Only a bigger FPGA chip could have accommodated the design – and indeed it did fit (if only barely) on the biggest FPGA of the Virtex-5 family, the FX200T. But, as we didn't have this much more expensive one, we had to look for possible alternatives.

The bitonic sort. Batcher's bitonic sort algorithm is an early algorithm [5] that has implementations on most types of parallel hardware. Its time complexity is $O((log_2 L)^2)$, its space complexity is $O(L(log_2 L)^2)$, as it consists in $(log_2 L)^2$ stages of $L/2$ comparators. While the time complexity was maybe acceptable for our problem (although for L as low as 1480, $(log_2 L)^2$ is only about 10 times smaller than L), the space complexity was not acceptable. We have used the perfect shuffling technique of Stone [30] to collapse all these stages to a single one through which the data is looped $(log_2 L)^2$ times. But, the shuffling circuits where still using too much of the FPGA area (more than available), so we considered one more option.

The insertion sort. The parallel insertion sort algorithm is very simple: the sequence to sort is given element by element. At every step all stored elements equal to or bigger than the current input element shift replacing their right neighbor, leaving thus an empty place for the insertion of the current input element.

We implement the needed comparison and conditional shifting by creating a "smart cell" that can store one element and does the right thing when presented with an input. The whole parallel insertion sort is implemented as a chain of such smart cells, as long as the maximum length of the list to sort, and with appropriate connections.

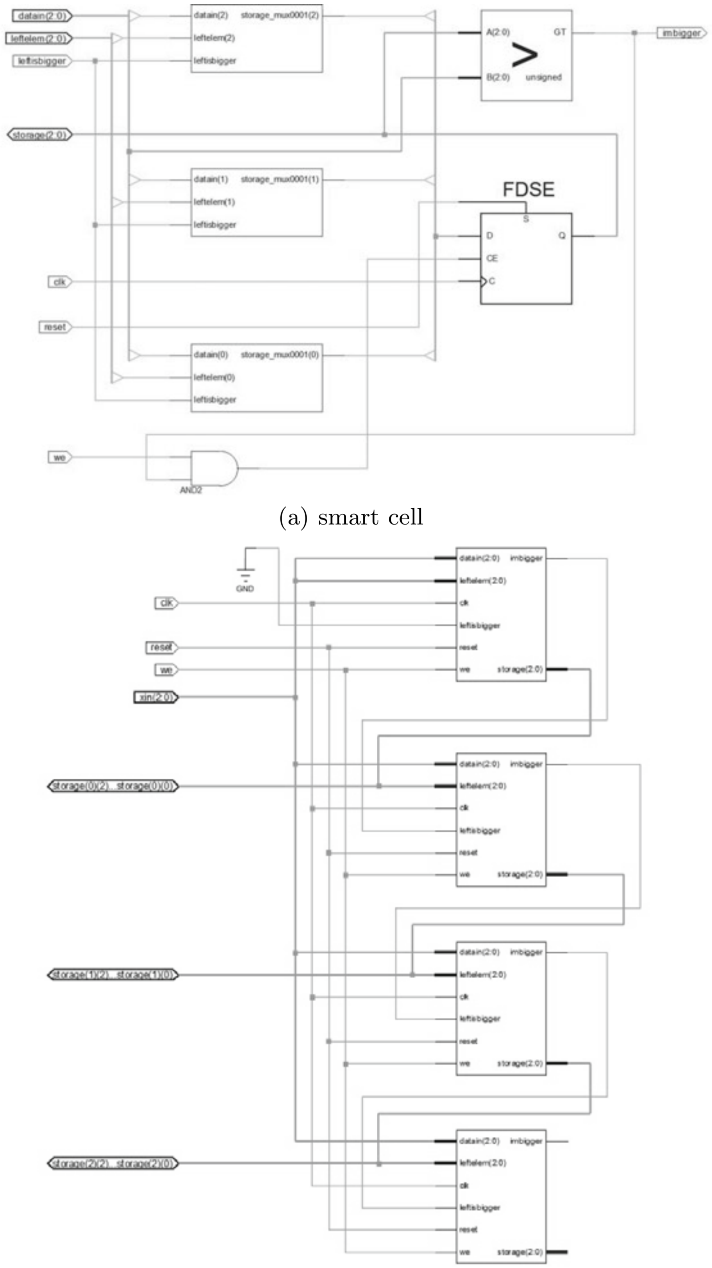
The circuit for a "smart cell" able to process elements of three bits in size is given in Figure 1(a). The circuit for a full chain of 4 smart cells that can sort lists of length 4 of 3-bit elements is shown in Figure 1(b).

Please note that the time complexity of one step is O(1), i.e. constant, but the amount of resources needed (comparators, multiplexers, gates) is directly proportional to the length of the list to sort. Overall, the time complexity is O(L), the space complexity is O(L) as well.

The VHDL code for this implementation is given in Appendix 1. It uses the construction "generate" to create multiple instances of "smart cells" and to connect appropriately their pins.

2.2 Multi-core CPUs

We have used OpenMP, which makes parallelizing serial programs in C/C++ and Fortran fairly easy [9]. A fragment from the implementation of an early benchmark is given in Appendix 2. As one can see in the code we provide, we don't try



(a) smart cell

(b) chain of smart cells

Fig. 1. (a) The circuits of a smart cell used in the parallel insertion sort (here for 3 bit elements); (b) Parallel insertion sort as a structure composed of many smart cells (here for processing 4 elements of 3 bits each)

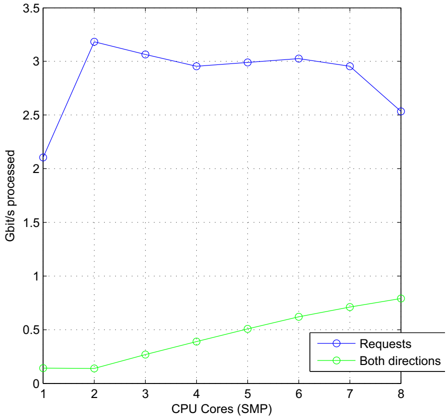


Fig. 2. The performance of the multicore solution: the highest volume of network traffic that can be processed as a function of the number of cores used

to split the individual sorting tasks, but to dispatch them to multiple threads (in groups of 16) that get dispatched on the cores and on the CPUs. While for forking threads and splitting the workload of a *for loop* to those threads OpenMP is very easy to use, for the anomaly detection system we needed a producer-consumer setup, where the network data acquisition would acquire data and send it in some load-balancing fashion to the worker threads. Producer-consumer setups are notably difficult to implement in OpenMP, because of limited locking and signaling mechanisms, mainly because it lacks condition variables. Therefore, while we kept using OpenMP, we have added message_queue from the open-source library *boost* to implement the queues needed for the producer-consumer paradigm.

2.3 GPUs

Our Nvidia GPUs are programmable with CUDA [24,20], which combines C/C++ on the host side with C-like kernels that run in parallel on the cores of the GPU. A C++ STL like framework on top of CUDA is the open-source library *Thrust* [1]. We have used this library to implement our test programs. As a result these are remarkably concise, as can be see in Appendix 3.

3 Results and Analysis

The performances obtained have been: FPGA – processing 1.875 Gbit/s, communication 1 Gbit/s; multi-core CPU – 2 Gbit/s overall; GPU – 8 Mbit/s overall, including processing and communication.

The FPGA communication implemented achieved 2 Gbit/s with Bus-Master DMA transfers (approx. 1 Gbit/s in each direction). The parallel insertion sort was the only one fitting into our FPGA chip, when restricted to sorting of 256 64bits elements (8-byte-grams) or sorting 512 48bits elements (6-byte-grams). It did so for a maximum clock speed of 240MHz. This was more than the 128MBytes/s needed to process a full-speed 1 Gbit line. Still, retrieving data from FPGA board requires more than 8 bits of output per input byte, when there are more than 256 elements to sort. This would then require more than 2 Gbit/s communication speed between the CPU and the FPGA board. The communication constraints and the difficulty to adapt sorting to FPGA led us to investigate the alternatives.

The multi-core CPU implementation achieved from the first tests 2 Gbit/s. The results of the complete network intrusion detection system prototype, where time is spent also on tasks other than the N-gram extraction and sorting are shown in Figure 2, where the numbers are estimated offline by running on previously captured network traffic. Added to the graph is a curve "both directions" which models the worst-case traffic, completely unbalanced such that the requests are much longer than the replies. Please note that in general (for example for the HTTP protocol), the requests are short and the replies longer. We have also tested the prototype on a quad-core QuickPath Interconnect-enabled Intel E5540@2.53GHZ machine; the QPI architecture did not lead to a supplementary acceleration, probably because our solution doesn't require much synchronization or data passing from one core to another.

The GPU solution had the latency so high that only about 1% of the desired speed has been obtained (1000 sorts/second of 1024 element long lists). Our profiling linked most of this latency to the memory transfers between the memory space of the CPU and that of the GPU. While the radix sorting in *Thrust* has been reportedly outperformed [19], speeding it up will not have a big impact on the total time. For this reason we decided that it makes little sense to investigate further the GPU alternative as a possible improvement over the multi-core CPU version. The GPUs are a better fit for sorting large vectors.

4 Related Work

Previous work on sorting on FPGA include generating automatically optimal sorting networks by using quickly reconfigurable FPGAs as evaluator for a genetic programming system [17], with a follow-up where the genetic search is also done on the FPGA [16]. Human designed sorting networks were published e.g. in [12] (where the authors write that "The results show that, for sorting, FPGA technology may not be the best processor choice") and [21]. A recent paper where the tradeoffs involved in implementing sorting methods on FPGA are carefully considered is [7].

The state-of-the-art GPU sorting algorithms in CUDA (radix sort, merge sort) are explained in [27], and are included in the open-source library CUDPP[28] starting with version 1.1.

Most GPU application papers compare to a CPU implementation, although most often with a single core one - reporting in this way the best speed-up values. In [8] the GPU, FPGA and multi-core CPU implementations for solving three problems (Gaussian Elimination, DES - Data Encryption Standard and Needleman-Wunsch) are compared. Unfortunately, although the GPU used was the same family with the one we used, and the same holds true for the CPU, the authors used a much older FPGA (Virtex II Pro, maximum frequency 100MHz), which could have biased the results. Another paper comparing all three platforms we tested (and supplementarily a massively parallel processor array, Ambric AM2000) is [31] where various methods for generating random numbers with uniform, Gaussian and exponential distributions have been implemented and benchmarked. Another interesting factor is introduced in the comparison, the power efficiency (performance divided by power consumption) and here FPGAs were the leaders with a big margin. A very extensive overview of the state of the art in heterogeneous computing (including GPU, Cell BEA, FPGA, and SIMD-enabled multi-core CPU) is given in [6]. The obvious conclusion one gets after surveying the existing literature is that there is no clear winner for all problems. Every platform has a specific advantage. What is most interesting, beyond the price of the devices – which is fairly balanced for the devices we used – is how much progress is one researcher expected to make when entering these low-cost HPC technologies fields on the particular problem of interest, and this depends most on how easy is to develop for these platforms.

5 Discussion and Conclusion

5.1 Comparing the Difficulty of Programming and Debugging

As far as the FPGA is concerned, it can be configured in the language VHDL in two styles: “behavioral” (corresponding to the procedural style of CPU programming) and “structural” (one abstraction layer lower). If in the beginning the ”behavioral” style might look appealing, after hitting the area limitation hard barrier one is supposed to grasp such definitions “FDSE is a single D-type flip-flop with data (D), clock enable (CE), and synchronous set (S) inputs and data output (Q)” [2] and work almost exclusively in a structural fashion, where the allocation of the very limited resources is more under the control of the designer and not of the compiler – so previous experience with digital electronics helps. In other words, “programming” FPGAs is not really for regular computer programmers, as the programming there is actually describing a structure, which only sometimes can be done by describing its desired behavior. If the general purpose logic cells are to be saved, then explicit modules on the FPGA such as DSP units and memory blocks have to be referenced and used and this shifts the competencies needed even more towards the field of digital electronics and further away from the one of computer science.

Some of the FPGA tools are overly slow. We mentioned that it took us two days to synthesize the most complex sorting algorithm we created for FPGA – by using Xilinx ISE 10. Switching to Synplify reduced the synthesis time to

50 minutes. Still a lot by software engineering practice (software compilation rarely takes that long). Even worse, while alternative tools can cover the chip-independent stages in the FPGA workflow, the chip-dependent stages like map, place and route can be done usually only with the vendor's software tools – and these stages are slower than the synthesis.

We think FPGA is still an interesting platform, for being energy efficient and to some extent scalable: when one needs more FPGA resources, one can simply use more FPGA chips connected to each others. The FPGA chips have hundreds of I/O pins that could facilitate the communication. It is unlikely that the speed issues with the FPGA workflow tools can be solved completely, as the problems they try to solve are NP-complete (resource allocation, place & route)[33]. This issue gets worse with the size of the FPGA. While our FPGA drifts towards entry-level ones, bigger ones could have been worse for this reason.

As far as the multi-core CPUs are concerned, programming them can be done with standard compilers – newest versions of the main C/C++ compilers like GNU gcc and Intel's one have all OpenMP support. Adding parallelism to loops is fairly easy in OpenMP. The possibility to implementing a correct and complete producer-consumer setup is unfortunately considered outside of the scope of the OpenMP framework. Debugging and tuning OpenMP multi-threaded code is not as easy as for the serial code, of course, but it's not overly difficult, especially with the aid of tracing tools like VampirTrace [22] used by us. The entry level barrier is not too high, but having parallel programming knowledge helps – especially understanding race conditions and the synchronization mechanisms.

One of the main problems with programming the GPUs is the instability of the frameworks and the fact that most are vendor-specific. OpenCL [23] may change this in time, but for now we have used CUDA, which is the natural choice for Nvidia graphic chips. CUDA itself has now reached its third major version in two years, following the advances of the hardware. While CUDA is suitable for developing code, debugging code is much more complicated. It used to be the case that on Microsoft Windows XP machines memory errors that on CPU programs are caught by supervisors led to lock-ups and the need to restart the whole machine. The situation has been better in Linux, where only the X Window had to be restarted sometimes, in order to force the reinitialization of the graphic card. There is an emulated device mode which turns to some extent debugging CUDA into debugging many CPU-side threads, but it is much slower than the non-emulated mode. Apart from the difficulty to debug code, another criticism to the CUDA framework was that it is (still) too low-level, making the implementation and the tuning of complex systems overly difficult. Systems like *Thrust*, PyCUDA [15] and BSGP [14] aim to fix this. The need to transfer data between the CPU's memory and GPU's memory is also a major disadvantage of the GPU when used as a computing coprocessor, as these transfers introduce undesirable latencies. In CUDA, for a limited set of devices, which share the memory with the CPU (and are thus not top performing ones), page-locked host memory can be mapped to the GPU memory space, reducing these latencies. On the other hand, the dedicated GPU memory is higher speed (at least bandwidth-wise), so this is just a trade-off with outcomes to be tested case by case. While knowing C/C++ is enough to start CUDA programming, getting good performance requires leveraging the hardware primitives/structures meant for graphics (e.g. texture memory) - the compilers do not do this for the user, so having experience with graphics programming does still help.

5.2 Conclusion

To conclude, FPGA is the most flexible but least accessible, GPU comes next, very powerful but less flexible, difficult to debug and requiring data transfers which increase the latency, then comes the CPU which might sometimes be too slow despite multiple cores and multiple CPUs, but is the easiest to approach. In our case the multi-core implementation offered us the best combination of compatibility, high bandwidth and low latency, therefore we have selected this solution for integration into the ReMIND prototype.

References

1. Thrust, http://code.google.com/thrust
2. Xilinx FDSE, http://www.xilinx.com/itp/xilinx7/books/data/docs/s3esc/s3esc0081_72.html
3. Project ReMIND (2007), http://www.remind-ids.org
4. Xilinx application note XAPP1052, v1.1 (2008), http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf
5. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the Spring Joint Computer Conference, April 30-May 2, pp. 307–314. ACM, New York (1968)
6. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J., Storaasli, O.O.: State-of-the-Art In Heterogeneous Computing. Journal of Scientific Programming (draft, accepted for publication)
7. Chamberlain, R.D., Ganesan, N.: Sorting on architecturally diverse computer systems. In: Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications, pp. 39–46. ACM, New York (2009)
8. Che, S., Li, J., Sheaffer, J.W., Skadron, K., Lach, J.: Accelerating compute-intensive applications with gpus and fpgas. In: Symposium on Application Specific Processors (2008)
9. Dagum, L., Menon, R.: Open MP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering 5(1), 46–55 (1998)
10. Dongarra, J., Gannon, D., Fox, G., Kennedy, K.: The impact of multicore on computational science software. CTWatch Quarterly (February 2007)
11. Grozea, C., Gehl, C., Popescu, M.: ENCOPLOT: Pairwise Sequence Matching in Linear Time Applied to Plagiarism Detection. In: 3rd Pan Workshop. Uncovering Plagiarism, Authorship And Social Software Misuse, p. 10

12. Harkins, J., El-Ghazawi, T., El-Araby, E., Huang, M.: Performance of sorting algorithms on the SRC 6 reconfigurable computer. In: Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, pp. 295–296 (2005)
13. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, pp. 258–262 (2005)
14. Hou, Q., Zhou, K., Guo, B.: BSGP: bulk-synchronous GPU programming. In: ACM SIGGRAPH 2008 papers, p. 19. ACM, New York (2008)
15. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A., Sarma, A.D., Nanongkai, D., Pandurangan, G., Tetali, P., et al.: PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. Arxiv preprint arXiv:0911.3456 (2009)
16. Korrenek, J., Sekanina, L.: Intrinsic evolution of sorting networks: A novel complete hardware implementation for FPGAs. LNCS, pp. 46–55. Springer, Heidelberg
17. Koza, J.R., Bennett III, F.H., Hutchings, J.L., Bade, S.L., Keane, M.A., Andre, D.: Evolving sorting networks using genetic programming and the rapidlyreconfigurable Xilinx 6216 field-programmable gate array. In: Conference Record of the Thirty-First Asilomar Conference on Signals, Systems & Computers, vol. 1 (1997)
18. Krueger, T., Gehl, C., Rieck, K., Laskov, P.: An Architecture for Inline Anomaly Detection. In: Proceedings of the 2008 European Conference on Computer Network Defense, pp. 11–18. IEEE Computer Society, Los Alamitos (2008)
19. Leischner, N., Osipov, V., Sanders, P.: GPU sample sort. Arxiv preprint arXiv:0909.5649 (2009)
20. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 39–55 (2008)
21. Martinez, J., Cumplido, R., Feregrino, C.: An FPGA-based parallel sorting architecture for the Burrows Wheeler transform. In: ReConFig 2005. International Conference on Reconfigurable Computing and FPGAs, p. 7 (2005)
22. Muller, M.S., Knupfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In: Proceedings of the Minisymposium on Scalability and Usability of HPC Programming Tools at PARCO (2007) (to appear)
23. Munshi, A.: The OpenCL specification version 1.0. Khronos OpenCL Working Group (2009)
24. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA (2008)
25. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings-IEEE 96(5), 879 (2008)
26. Rieck, K., Laskov, P.: Language models for detection of unknown attacks in network traffic. Journal in Computer Virology 2(4), 243–256 (2007)
27. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, pp. 1–10. IEEE Computer Society, Los Alamitos (2009)
28. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, p. 106. Eurographics Association (2007)
29. Smith, M.C., Vetter, J.S., Alam, S.R.: Scientific computing beyond CPUs: FPGA implementations of common scientific kernels. In: Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices, MAPLD 2005, Citeseer (2005)

30. Stone, H.S.: Parallel processing with the perfect shuffle. IEEE Transactions on Computers 100(20), 153–161 (1971)
31. Thomas, D.B., Howes, L., Luk, W.: A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 63–72. ACM, New York (2009)
32. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the cell processor for scientific computing. In: Proceedings of the 3rd Conference on Computing Frontiers, pp. 9–20. ACM, New York (2006)
33. Wu, Y.L., Chang, D.: On the NP-completeness of regular 2-D FPGA routing architectures and a novel solution. In: Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design, pp. 362–366. IEEE Computer Society Press, Los Alamitos (1994)

Appendix 1: Parallel Insertion Sort in VHDL for FPGA

```
library IEEE;use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;use IEEE.STD_LOGIC_UNSIGNED.ALL;
constant nrbitselem:integer:=3;
subtype elem is STD_LOGIC_VECTOR((nrbitselem-1) downto 0);
subtype elemp1 is STD_LOGIC_VECTOR(nrbitselem downto 0);
type vect is array(natural range<>) of elem;
type vectp1 is array(natural range<>) of elemp1;
entity insertsortsmartcell is
    Port ( datain : in elem;we: in std_logic;
        leftelem: in elem;leftisbigger: in std_logic;
        clk: in std_logic;reset:in std_logic;
        imbigger:buffer std_logic;storage:inout elem);
end insertsortsmartcell;
architecture Behavioral of insertsortsmartcell is begin
  imbigger<=’1’ when storage>datain else ’0’;
  process(clk,reset)
  begin if clk’event and clk=’1’ then
   if reset=’1’ then storage<=(others=>’1’);
   else if we=’1’ then if imbigger=’1’ then if leftisbigger=’0’ then
      storage<=datain; -- insertion right here
      else storage<=leftelem;
     end if;end if;end if;end if;end if;
end process;end Behavioral;
entity insertsort is
  generic(abw:integer:=2);
    Port ( xin:elem;storage : inout vect(2**abw-1 downto 0);
        clk:std_logic;reset:std_logic;we:std_logic;xout:out elem);
end insertsort;
architecture Behavioral of insertsort is
signal isbigger: std_logic_vector(2**abw-1 downto 0);
begin
  a:for i in 0 to 2**abw-1 generate
    b:if i=0 generate
    cell0:entity work.insertsortsmartcell port map(xin,we,
     xin,’0’,clk,reset,isbigger(i),storage(i));
    end generate;
    c:if i>0 generate
    cell:entity work.insertsortsmartcell port map(xin,we,
     storage(i-1),isbigger(i-1),clk,reset,isbigger(i),storage(i));
    end generate;end generate;
  xout<=storage(2**abw-1);
end Behavioral;
```

Appendix 2: OpenMP Benchmark for Sorting

This is a code fragment from a benchmark that proves that it is possible to extract the N-grams and sort those on the multi-core CPUs we used, at a speed higher than 1 Gbit/s. The sorting is virtual in the sense that no data is moved around, just indexes are reordered; full details including code not reproduced here are given in [11]. The OpenMP influence on the code is minimal: a header is included, the number of threads is specified, then through one or two pragmas the tasks are split between threads.

```
#include "omp.h"
#define fr(x,y)for(int x=0;x<y;x++)
omp_set_num_threads(4);//how many threads openmp will use
//fork the threads
#pragma omp parallel private(counters,startpos,ix,ox,v)
{fr(rep,125000/16){
#pragma omp for schedule(static,1)
  fr(rep2,16){
   ... //generate an array, then sort it with serial radix sort
}}}
```

Appendix 3: Benchmark of Sorting on GPU Using Thrust

```
//included: <thrust/device_vector.h>, <thrust/host_vector.h>, <thrust/functional.h>, <thrust/sort.h>
int main(void){const int N = 1024;int elements[N] = {1,3,2};
  thrust::host_vector<int> A(elements,elements+N);thrust::device_vector<int> B(N);
  int s=0;for(int rep=0;rep<1000;rep++){
    thrust::copy(A.begin(), A.end(), B.begin());
    thrust::sorting::radix_sort(B.begin(), B.end());
    thrust::copy(B.begin(), B.end(), A.begin());
    s=s+A[0]+1;}
  std::cout<<s<<" ";
  return 0;}
```

Considering GPGPU for HPC Centers: Is It Worth the Effort?

Hans Hacker[1], Carsten Trinitis[1], Josef Weidendorfer[1], and Matthias Brehm[2]

[1] Department of Informatics, Technische Universität München, Germany
{hacker,trinitic,weidendo}@cs.tum.edu
[2] Leibniz Rechenzentrum, Garching bei München, Germany
brehm@lrz.de

Abstract. In contrast to just a few years ago, the answer to the question "What system should we buy next to best assist our users" has become a lot more complicated for the operators of an HPC center today. In addition to multicore architectures, powerful accelerator systems have emerged, and the future looks heterogeneous. In this paper, we will concentrate on and apply the abovementioned question to a specific accelerator with its programming environment that has become increasingly popular: systems using graphics processors from NVidia, programmed with CUDA. Using three benchmarks encompassing main computational needs of scientific codes, we compare performance results with those obtained by systems with modern x86[1] multicore processors. Taking the experience from optimizing and running the codes into account, we discuss whether the presented performance numbers really apply to computing center users running codes in their everyday tasks.

1 Introduction

In recent years, the strategy for an HPC center to get best performance out of their users' codes has undergone significant changes. Ten to fifteen years ago, a great variety of processor types and systems existed, e.g. MIPS, SPARC, Alpha, or vector systems like Cray, NEC, Fujitsu, etc.. Today, the TOP500 list [1] (the list of the 500 fastest supercomputers in the world) is dominated by x86 Intel or AMD processor based systems with high speed interconnects like e.g. Infiniband [2], and only a small niche remains for other architectures, such as IBM's Power architecture. Every few years, the next generation of a parallel system or compute cluster was put into operation, consisting of faster general purpose processor architectures. For many users, recompilation was only needed if the new system brought a change in the processors' Instruction Set Architecture (ISA).

Today, the number of options to choose from for a future system has increased again. On one hand, this is driven by technical boundaries in processor design, leading to multicore architectures with a wide variety of design options even inside a chip. Currently available multicore processors are often not best suited

[1] By x86 we also refere to the 64-bit extension of the x86 ISA.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 118–130, 2010.

to characteristic requirements of scientific code, e.g. only raising computational power but neglecting data transfer between components often results in reduced scalability for parallel code. On the other hand, the computer gaming industry has put a lot of effort into developing specially tuned graphics hardware devices, which get more similar to standard processors with every new generation [3]. The capability of graphics processing units (GPU) has reached a stage where they seem general enough to be useful to areas other than graphics, establishing the term GPGPU (General Purpose Graphics Processing Units) [4,5,6]. Companies such as NVidia or ATI/AMD see the HPC community as potential customers. Actually, the pure computational power achievable with tuned codes fitting a GPU's hardware capabilities is astonishing, even more so as this hardware is quite cheap since it is build for the mass market. However, graphics hardware traditionally is programmed in a way significantly different from regular software programming. It uses interfaces tuned for graphics requirements such as OpenGL and MS Direct3D. Due to this fact, GPU manufacturers introduced new proprietary programming models and according SDKs (e.g. NVidia CUDA [7], ATI Stream [8]).

While the available accelerator systems are attractive, a closer look should be taken at the requirements of scientific/HPC codes to judge usefulness. Ranging from requirements for IEEE floating point compliance and high bandwidth for data transfer, to stability and reliability, there is still room for improvement. Hence, adaption of accelerator hardware as part of the computer equipment of an HPC computing center has multiple aspects. From a purely operational point of view, these are:

- Improvement regarding *performance/cost* ratio in contrast to standard hardware: The cost that has to be taken into account must include power consumption and cooling needs both for GPU accelerators and standard systems, respectively.
- Expected *exploitation* of accelerator systems: What is the best way to fully utilize resulting heterogeneous systems? How many users are expected to actually run code which can utilize the accelerator parts?
- *Stability/reliability* of hardware and software, e.g. device driver, for use in multiuser environments.

To be able to answer the second aspect above regarding user acceptance of accelerator systems, it should be kept in mind that, besides performance, their main concerns are:

- *Ease of use*: What amount of studying effort is required to make optimal use of hardware? Is there a need for rewriting parts of the code from scratch, or can existing C/Fortran code be incrementally tuned to use additional features? Problems for adaption for the user could comprise a new programming model, new languages and new tools to be used. Similar, if the hardware does not completely conform to IEEE floating point standards (exactly same result, exception support), checking the correctness of a port can require a significant amount of time.

– *Persistency* and long-term support: Is it worth for users to tune their code for a given accelerator system? If the system involves a new programming model/-language, this should be part of a standardization process.

In this paper, we concentrate on the reachable performance of graphics processors from NVidia using CUDA. In order to achieve this, we have tuned a set of codes defined within work package (WP) 8 in the PRACE project. The work presented here was carried out in the context of the EU PRACE project,which prepares the creation of a persistent pan-European HPC service, consisting of several tier-0 centers providing European researchers with access to capability computers and forming the top level of the European HPC ecosystem. PRACE is a project funded in part by the EU 7th Framework Programme [9]. If available, we also measured optimized library codes with similar functionality. The results were compared with numbers obtained from systems consisting of recent x86 processors. In addition to performance results, we take a look at the usefulness of employing NVidia hardware as part of the computer equipment in an HPC center, taking the aspects presented above into account. In the process of optimizing and running the code, we obtained sufficient experience to be able to give profound arguments for discussion.

The paper is structured as follows: In the next section, we give an overview of the broader scope of PRACE where this work was done, and present some related work. Then, the programming model of CUDA is shortly presented. Afterwards, the different codes from the benchmark are shown, followed by optimization and tuning tips relevant for the codes. The next section presents performance numbers in detail. Finally, we discuss the advantages and disadvantages of using such accelerator systems at all, before we give a conclusion and an outlook on future work.

2 Related Work

In this paper, we concentrate on CUDA. However, the porting of the benchmarks was carried out as part of the “technology watch” subproject of PRACE with a broader focus. There, all kind of different architectures relevant for computing centers are taken into account, as can be read in the respective project deliverables [10]. The PRACE benchmarks represent simple kernels quite easily to port to various architectures, allowing for evaluation in the tight time frame of the project, i.e. striving for medium term statements. The Rodinia benchmark suite [11] takes a different approach. It specifically emphasizes on heterogeneous computing, fusing multicore architectures with GPU components. Further, it tries to cover the “dwarfs” application categorization of Berkeley [12]. This approach could become relevant for follow-up subprojects of PRACE.

There are quite a few projects world-wide that work towards the next-generation supercomputer, and collaborate in the International Exascale Software Project (IESP) [13]. The web page provides presentations and reports about the current status of these projects. While we expect that accelerators such as

GPUs to play an important role in this context, concrete statements are still pending.

There are a lot of papers presenting ports of all kind of applications to GPUs (see gpgpu.org [4]). However, it is quite difficult to estimate how much of these research works will result in code to be used by users of an HPC center.

3 CUDA as an Example for GPGPU Programming

In recent years, outsourcing compute intensive vector operations to graphics cards has become a popular pastime activity, especially in numerical simulation. Many people achieved quite good performance and scalability on non trivial problems by the so called Compute Unified Device Architecture (CUDA), a parallel programming model developed by NVidia. CUDA is an extension to the C programming language and allows programmers to utilize NVidia graphics cards for scientific computations. Each CUDA program consists of a so called *host* section and the *device* sections. The *host* section is executed sequentially and calls one or more so called *kernels*. These kernels are executed as threads in parallel on a *device*. Threads are grouped into blocks with up to three dimensions, where threads within the same block can be synchronized. The maximum number of threads per block is 512.

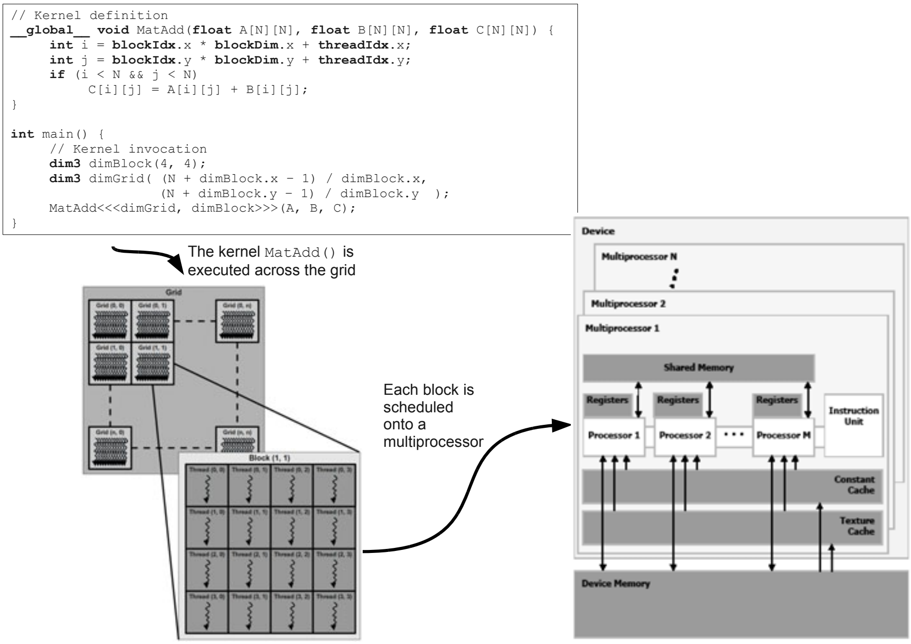
Thread blocks are grouped in one- or two dimensional *grids*. A grid may contain up to 65535 blocks per dimension. Thread blocks are distributed over the available SMs (Streaming Multiprocessors) upon execution. Each SM can serve up to 1024 coexisting threads. Current NVidia hardware contains up to 30 SMs; with each block being executed on exactly one SM. An SM consists of eight scalar cores with 2048 32-bit registers and 16 KB shared memory. Each of these cores is capable of performing integer and single precision floating point operations. Upon execution, blocks are divided into so called *warps*. CUDA currently groups 32 threads into *warps*. Within one *warp*, threads are always synchronous. Thread creation and scheduling are completely done in hardware.

Figure 1 shows a CUDA program for matrix addition ($C = A + B$). Each thread computes exactly one value of the result matrix. Blocks of 4x4 threads each are created and grouped in a two dimensional grid. Blocks are then executed by multiprocessors.

4 Methodology

4.1 The PRACE WP8 Benchmarks

In order to be able to assess and compare the performance among the vast amount of mainly heterogeneous PRACE prototypes, it was decided to select a set of small, easy to port, yet meaningful computational kernels. As a guideline for the selection, the principle of 'dwarfs' [14][12] was consulted. A dwarf is an



```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     int j = blockIdx.y * blockDim.y + threadIdx.y;
     if (i < N && j < N)
          C[i][j] = A[i][j] + B[i][j];
}

int main() {
     // Kernel invocation
     dim3 dimBlock(4, 4);
     dim3 dimGrid( (N + dimBlock.x - 1) / dimBlock.x,
                   (N + dimBlock.y - 1) / dimBlock.y  );
     MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Fig. 1. CUDA example with mapping to hardware

algorithmic method that captures a pattern of computation and communication. The underlying numerical methods may change over time, but the claim is that the underlying patterns persist through generations. Three important dwarfs are namely dense linear algebra, sparse linear algebra and spectral methods.

Due to its modularity, the codes from the EuroBen benchmark suite[2] were chosen. This benchmark suite provides synthetic programs for scientific and technical computing. It is organized in modules of increasing complexity. Module 1 is concerned with the performance of important basic computational kernels like the dot-product or the axpy operation. The second module considers basic numerical algorithms that use the operators from module 1. These include matrix-vector multiply, solution of full and sparse linear systems or FFTs, etc. In the 3rd module skeleton applications like ODE and PDE solvers are tested that in turn use the algorithms from module 2.

As a representation for dense linear algebra, mod2am was selected. This is a dense matrix-matrix multiplication in the form of $C = AB$. In the area of sparse linear algebra, the choice fell on mod2as, a sparse matrix-vector product in the form of $c = Ab$. The matrix is stored in the compressed row storage (CRS) format. Finally mod2f, a 1-D complex FFT represents the dwarf spectral methods.

[2] http://www.euroben.nl

4.2 Porting and Optimization Strategies

mod2am. The obvious way to implement this benchmark is to apply the library calls `cublasDgemm` / `cublasSgemm`. As with every accelerator-card, one needs to transfer the data to the cards' main-memory. CUDA offers various ways to copy data. For this application, two methods were chosen. The first method is to allocate the host data-structures with malloc and the same data structures on the GPU and then copy the data. However, the MMU is involved in every page-request. The second method allocates the host data-structures via a special CUDA call (`cudaMallocHost`) which marks the allocated pages as non-swappable and allows data transfer to take place without the interference of the MMU. Since the mod2am-benchmark uses a typical C-'row-major' data arrangement, the transpose option for the input matrices was used. Unfortunately, there is no transpose option for the output so one has to write a kernel that transposes the matrix. There is an example code for a matrix transposition in the SDK that was modified accordingly. The transposition is done out-of-place by blocking the matrix, transposing the block and storing it back to the main memory of the card. NVidia gives a small example of a blocked dense matrix-matrix multiply in the programming guide. For comparing to CUBLAS, this kernel was also implemented. As the original NVidia example kernel only allowed matrices with the multiple size of the blocks, a few changes (mainly if clauses) had to be applied to deal with matrices of arbitrary size. Due to the existence of CUBLAS, the porting of this benchmark was straightforward. Only the different data arrangement (column-major vs. row-major) required slightly more effort.

mod2as. NVidia offers no library for sparse matrix operations. However, on CUDA Zone [7] the publication 'Efficient Sparse Matrix-Vector Multiplication on CUDA' by Nathan Bell and Michael Garland [15] can be found. The proposed CSR-kernel was modified to fit the needs of the C reference implementation of mod2as. It was not necessary to change the data structures. The CSR-kernel exploits the use of the shared memory within the Streaming Multiprocessor. Because of the warp concept (a warp consists of 32 threads that are all synchronous) a very efficient reduction can be performed. Furthermore, this kernel puts the x-vector into the texture-cache of the GPU. Since the x-data is reused multiple times this improves efficiency.

mod2f. By the time when the porting of mod2f has been started, cuFFT (the NVidia FFT implementation) only supported single precision. The single precision port therefore was straightforward. For double precision, a port of the given Radix-4 FFT C-code was carried out.

The given code works out-of-place, which means that additionally to the input vectors (real and imaginary), two other vectors are used for intermediate values. It also uses a precalculated vector of sines and cosines. The sine/cosine vector is being calculated by two nested loops that leads potentially to a major load-imbalance. However, as the loops form a geometric sequence, they can be unrolled. Thus, each thread calculates exactly one element of the result.

The calculation consists of multiple Radix-4 rounds plus additional Radix-2 rounds if the input data is not a power of 4. The Radix-4 rounds always use four real and four imaginary values, resulting into eight values to be calculated. Since a SM has eight scalar units and a warp consists of 32 threads, this fits perfectly (4x 8 threads).

As a GPU does not have caches like a CPU, access to the global memory is very slow (400-600 cycles). To hide this latency, NVidia recommends starting between 64-192 threads (oversubscribing). Additionally the memory accesses of a half-warp (16 threads) are coalesced in bundles of 128/64 or 32 bytes. In order to minimize memory access, the data types double2/float2 (struct of two doubles/floats) should be used. Each round always requires both the real- and the imaginary value. The .x value is the real- and the .y value is the imaginary-part of the vector/number. To fully exploit the usage of the double2/float2 type one can use the texture cache. This loads both the .x/.y-values to the cache. Hence, the .y-value read is almost for free. The readability of the code suffers considerably by using the cache. However, by using float2/double2 and texture-cache, the accesses to global-memory could be halved (9-10% speedup).

There are also many (integer) intermediate values that are calculated at the beginning of each round. Each SM is able to perform eight (integer) additions or bit-operations each clock-cycle. However, an SM can only perform two 32-bit multiplications per cycle. Therefore, one can use the `__mul24/__umul24` intrinsic (8 instructions / cycle). Both inputs require a value less or equal to 24 bit. The result is 32 bit. The first implementation also calculated those values ahead and stored the results in the shared memory of the SM. The use of the CUDA profiler indicated that it is faster (5-6%) to have each thread calculate the value itself and store the results in registers.

The double precision port was written generically and can calculate single precision as well. However, it turned out that a few kernels had to be specialized. The first implementation used two intermediate values, which were stored in the shared memory of the SM. This led to bank-conflicts with double precision. For single precision, because of being only 32-bit wide, this works perfectly. Furthermore, for double precision, the intermediate values are substituted twice in order to have only one calculation (the intermediates are stored in registers). If using single-precision the usage of 'fused multiply-add' (just cuts the result of the multiply instead of rounding) results in slightly different values than without. One can use an intrinsic to avoid 'fused multiply-add' (compiler-default). In order to avoid the rounding errors a considerable coding overhead is needed. Two FFT-rounds require two distinct loop variables. Here the grid-feature of CUDA can be used. One can start the blocks of calculation in a grid of up to two-dimensions. Each dimension is used as a loop variable.

The optimal number of threads per block is 64. 32 threads are not sufficient to hide the latency of the global memory (400-600 cycles). With more than 64 threads, the overhead of the calculations outweigths the latency hiding.

5 Results

The CUDA results were obtained on the PRACE prototype 'uchu'. A node consists of a Intel Harpertown (Xeon E5462 at 2.8 GHz) with 16 GB of system memory and two NVidia Tesla C1060 cards each with 4 GB of GDDR3 memory. The cards are attached via PCIe x16 (gen2) with a theoretical maximum throughput of 8 GB/s. However the measured maximum is only 5.7 GB/s. The installed NVidia/CUDA driver version was 190.18 and the toolkit version 2.3.

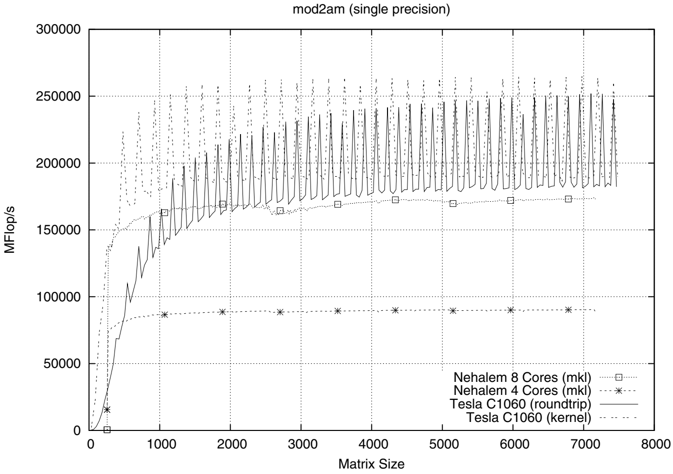


Fig. 2. mod2am in single precision

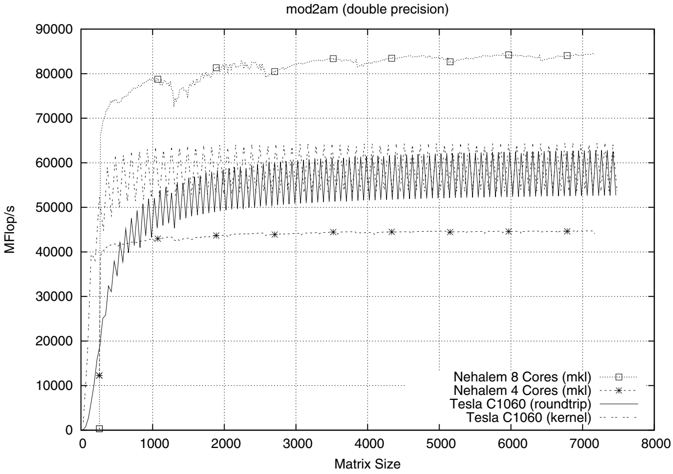


Fig. 3. mod2am in double precision

The comparison results were obtained on the SGI Altix ICE System (another PRACE prototype). Each node consists of two sockets, each with a Intel Nehalem EP (at 2.53 GHz). The node has 16 GB of system memory. The benchmarks were compiled using the Intel `icc` in version 11.1.064 and the Math Kernel Library (MKL) in version 10.1.

Figure 2 and Fig. 3 show the results of `mod2am`. Due to its highly parallel nature, this kernel is the sweet-spot for GPGPUs. The observed jitter in the CUDA results is due to underutilization of the hardware because of the input

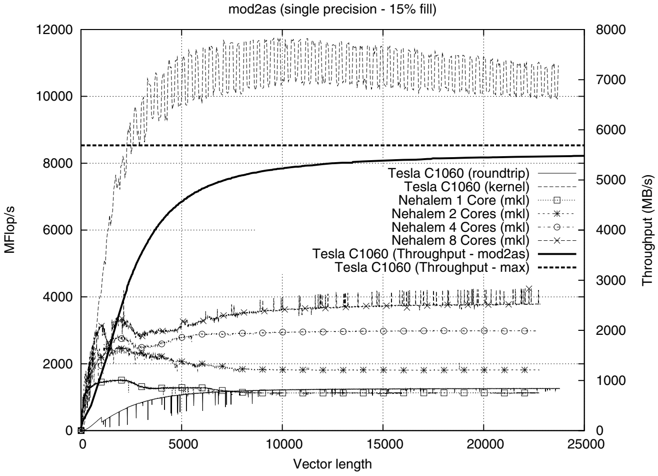


Fig. 4. mod2as in single precision

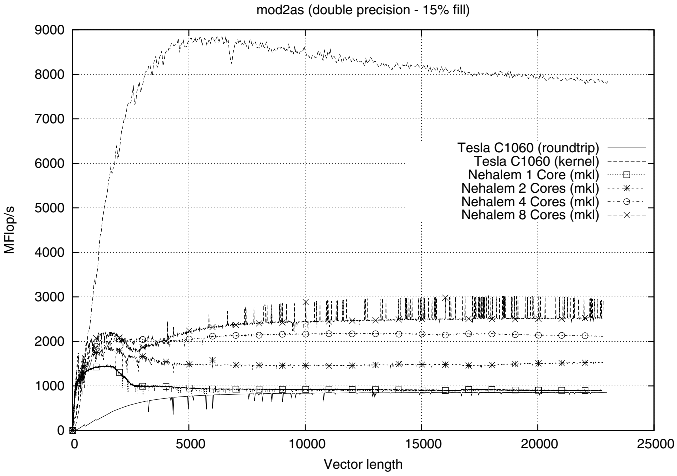


Fig. 5. mod2as in double precision

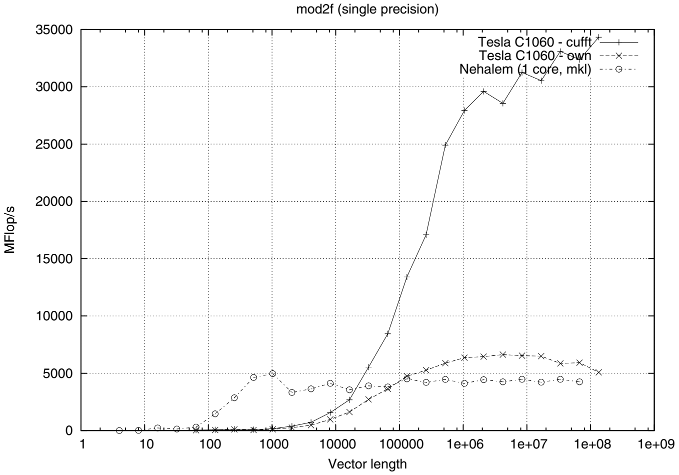


Fig. 6. mod2f in single precision

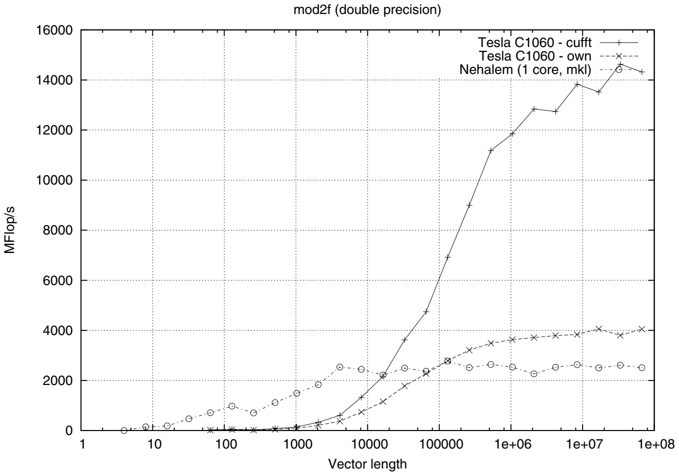


Fig. 7. mod2f in double precision

data size. The two different CUDA curves show achieved performance of the pure kernel and the performance with the inclusion of the data transfer (roundtrip).

Figure 4 and Fig. 5 show the results of `mod2as`. The main issue with this benchmark is that the data-transfer as well as the calculation are both of complexity O(n). The overall performance is limited by the throughput of the PCIe bus as shown in Fig. 4. The horizontal line (”throughput max”) indicates the maximum PCIe throughput on the ’uchu’ system. The tangential line shows the measured throughput of `mod2as`. The performance numbers for the pure kernel, with the data already on the card, indicate the potential of the Tesla card.

Therefore the suitability here clearly depends on whether the data can stay on the accelerator and be reused for multiple iterations or if the results have to be processed outside the accelerator after each calculation.

Figure 6 and Fig. 7 compare the results of `mod2f` with cuFFT, a hand coded version of a radix-4 FFT and MKL on Nehalem. The MKL provides only a serial version for the 1D-FFT. Recently NVidia released a new version of cuFFT (v.2.3) which supports double precision. The single precision performance was enhanced substantially. The cuFFT versions are an excellent choice for big inputs.

6 Discussion

Taking into account the results obtained by our benchmarks, this section will discuss the issues pointed out in the introduction.

From the HPC center's point of view, the following conditions apply:

- *Performance/Cost*: In terms of acquisition costs, graphics cards turned out to be an extremely cheap option for HPC, but in terms of operating costs, it must be taken into account that these nodes might be idle for a non-negligible part of their lifetime, still consuming additional power for cooling etc. . Therefore, future GPGPU accelerators should allow turning off idle parts of the system in order to guarantee efficient use.
- *Exploitation*: Despite the fact that only few users are willing and able to optimize their code the way described in this paper, there has been a strong demand for computing centers to offer compute nodes equipped with the respective hardware. However, as this applies only to a fraction of the users in a computing center, only a fraction of cluster nodes should be equipped with high end graphics cards for HPC usage.
- *Stability/reliability* of hardware: At the time of writing, existing graphics cards do not support error correcting codes (ECC), as this is not an issue in graphics programming - if one of several million pixels in an image has the wrong color, the user will not notice. DRAM failures are discussed in detail in [16]. Even if not all memory failures can be corrected, it is of major importance in an HPC Center to at least detect failing DRAMs. However, NVidia has announced that its new Fermi card will support ECC.

From the users' point of view, the situation looks as follows:

- *Ease of use*: Comprehensive study is certainly involved for users and/or programmers who intend to port their applications to CUDA. However, CUDA is relatively easy to learn for experienced programmers, as web pages provide plenty of examples. From a computing center's point of view, it would be a lot easier for users to upgrade to hardware that supports existing standards like e.g. OpenMP, PThreads, etc. or libraries like e.g. MPI. Programs should still run in standard C, C++, or FORTRAN environments without major modification, i.e. downward compatibility should be maintained to avoid unnecessary porting overhead. Regarding libraries, CUBLAS provides almost the complete set of BLAS routines for numerical operations. However, when it comes to more complex operations like e.g. sparse matrix computations, the user has to implement the underlying numerical algorithms. Also, current GPGPUs are not yet fully IEEE compliant, which can lead to errors when carrying out floating point operations. However, this might change with future GPGPU generations.

– *Persistency*: From our point of view, it is hard to foresee if a programming environment like CUDA, which is heavily derived from proprietary NVidia hardware (see section 3), will still prevail in five years. The upcoming standard for GPGPU programming, OpenCL [17], at least ensures that code is not bound to the success of only one vendor.

7 Conclusions and Future Work

In this paper, the practicability of using GPGPUs as accelerator cards in HPC centers has been investigated. We used several benchmarks from the PRACE WP8 benchmark suite and implemented these under CUDA on contemporary NVidia hardware. The benchmark results were compared with contemporary standard x86 based systems. In the discussion section, the issues raised in the introduction were answered according to the obtained results. We came to the conclusion that, at the time of writing, graphics cards are not yet a suitable alternative to ”standard” HPC architectures, as several issues from both the computing center's and the users' point of view indicate that the effort is still too much. However, the picture might change with future GPGPU architectures like e.g. NVidia Fermi, which will be subject to future investigations.

While one can argue whether results from three numerical kernels are sufficient to draw any conclusions regarding the benefit of GPU usage in computer centers, the same procedure was chosen within PRACE for comparing different architecture types. In future investigations additional benchmark kernels should be examined to cover different types of applications, e.g. based on the dwarfs classification [12].

Acknowledgments

This work was financially supported by the PRACE project funded in part by the EU's 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528. We specially thank our colleague Iris Christadler for valuable input from the HPC center's view.

References

1. Top500 Consortium: The Top 500 supercomputing sites, http://www.top500.org/
2. Infiniband Trade Association: Infiniband Interconnect Homepage, http://www.infinibandta.org/

3. Novakovic, N.: CPU and GPU now, the convergence goes on. The Inquirer (October 2009), http://www.theinquirer.net/inquirer/opinion/1560330/cpugpu-convergence-goes
4. GPGPU.org: A central resource for GPGPU news and information, http://gpgpu.org
5. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
6. Harris, M.: Mapping computational concepts to GPUs. In: ACM SIGGRAPH 2005 Courses. ACM Press, New York (2005)
7. CUDA Zone: The resource for CUDA developers, http://www.nvidia.com/object/cuda_home.html
8. Advanced Micro Devices, Inc.: ATI Stream Software Development Kit (SDK), http://developer.amd.com/gpu/ATIStreamSDK
9. PRACE: Partnership for Advanced Computing in Europe, http://www.prace-project.eu
10. PRACE: Public deliverables, http://www.prace-project.eu/documents/public-deliverables-1
11. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISW). IEEE, Los Alamitos (October 2009)
12. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley (December 2006)
13. IESP: International exascale software project homepage, http://www.exascale.org/
14. Colella, P.: Defining software requirements for scientific computing (2004)
15. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (December 2008)
16. Schroeder, B., Pinheiro, E., Weber, W.D.: DRAM errors in the wild: a large-scale field study. In: SIGMETRICS 2009: Proceedings of The Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, pp. 193–204. ACM, New York (2009)
17. Khronos Group: OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/

Real-Time Image Segmentation on a GPU

Alexey Abramov[1], Tomas Kulvicius[1,2],
Florentin Wörgötter[1], and Babette Dellen[3,4]

[1] Georg-August University, Bernstein Center for Computational Neuroscience,
Department for Computational Neuroscience, III Physikalisches Institut,
Göttingen, Germany
{abramov,tomas,worgott}@bccn-goettingen.de
[2] Department of Informatics Vytautas Magnus University, Kaunas, Lithuania
[3] Bernstein Center for Computational Neuroscience, Max-Planck-Institute for
Dynamics and Self-Organization, Göttingen, Germany
[4] Institut de Robòtica i Informàtica Industrial (CSIC-UPC), Barcelona, Spain
bdellen@iri.upc.edu

Abstract. Efficient segmentation of color images is important for many applications in computer vision. Non-parametric solutions are required in situations where little or no prior knowledge about the data is available. In this paper, we present a novel parallel image segmentation algorithm which segments images in real-time in a non-parametric way. The algorithm finds the equilibrium states of a Potts model in the superparamagnetic phase of the system. Our method maps perfectly onto the Graphics Processing Unit (GPU) architecture and has been implemented using the framework NVIDIA Compute Unified Device Architecture (CUDA). For images of 256 × 320 pixels we obtained a frame rate of 30 Hz that demonstrates the applicability of the algorithm to video-processing tasks in real-time[1].

1 Introduction

Image segmentation, i.e. the partitioning of an image into disjoint parts based on some image characteristics, such as color information, intensity or texture is one of the most fundamental tasks in computer vision and image processing and of large importance for many kinds of applications, e.g., object tracking, classification and recognition [1]. As a consequence, many different approaches for image segmentation have been proposed in the last twenty years, e.g. methods based on homogeneity criteria inside objects of interest [2], clustering [3,4,5], region-based growing [1], graph cuts [6,7] and mean shift segmentation [8]. We can distinguish between parametric (model-driven) [6,7] and nonparametric (data-driven) techniques [1,3,4,5,8]. If little is known about the data being segmented, nonparametric methods have to be applied. The methods of superparamagnetic clustering is a nonparametric method which solves the segmentation problem by

[1] By real-time we understand processing of a full frame at 25Hz or faster.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 131–142, 2010.

finding the equilibrium states of the energy function of a ferromagnetic Potts model (without data term) in the superparamagnetic phase [9,10,11]. By contrast, methods which find solutions by computing the minimum of an energy function usually require a data term – otherwise only trivial solutions are obtained. Hence, the equilibrium-state approach to the image segmentation problem has to be considered as fundamentally different from approaches which find the minimum energy configuration of energy functions in Markov random fields [12].

The Potts model [9], which is a generalization of the Ising model [13], describes a system of interacting granular ferromagnets or spins that can be in q different states, characterizing the pointing direction of the respective spin vectors. Depending on the temperature, i.e. disorder introduced to the system, the spin system can be in the paramagnetic, the superparamagnetic, or the ferromagnetic phase. In the ferromagnetic phase, all spins are aligned, while in the paramagnetic phase the system is in a state of complete disorder. In the superparamagnetic phase regions of aligned spins coexist. Blatt et al. (1996) applied the Potts model to the image segmentation problems in a way that in the superparamagnetic phase regions of aligned spins correspond to a natural partition of the image data [11]. Finding the image partition corresponds to the computation of the equilibrium states of the Potts model.

The equilibrium states of the Potts model have been approximated in the past using the Metropolis-Hastings algorithm with annealing [14] and methods based on cluster updating, which are known to accelerate the equilibration of the system by shortening the correlation times between distant spins. Prominent algorithms are Swendsen-Wang [3], Wolff [4], and energy-based cluster updating (ECU) [5]. All of these methods obey detailed balance, ensuring convergence of the system to the equilibrium state. However, convergence has only been shown to be polynomial for special cases of the Potts model.

Since the real-time aspect is getting more and more important in image processing and especially in image segmentation, parallel hardware architectures and programming models for multicore computing have been developed to achieve acceleration [15]. In this paper, we investigate opportunities for achieving efficient performance of superparamagnetic clustering using the Metropolis algorithm with annealing [14], and propose a real-time implementation on graphics processing units (GPU). For images of size 256×320 pixels the Metropolis procedure on GPU is 160 times faster than on CPU. Furthermore, a novel short-cut, consistent with the relaxation procedure of the Metropolis algorithm, is introduced for fast cooling.

The remainder of the paper is organized as follows: Section 2 describes the proposed segmentation algorithm. In Section 3, segmentation results for several test images are presented and the respective processing times on GPU and CPU are reported. In Section 4, the results are discussed and directions for future work are given.

2 The Segmentation Algorithm

The overall algorithm consists of several major steps as illustrated in Fig. 1. First, a parallel Metropolis procedure is developed and used to partition the image into disjoint regions (see 2.1). To reduce the total number of required Metropolis iterations, we developed a parallel algorithm that distinguishes between true object boundaries and boundaries caused by domain fragmentation (this is: uniform areas are split into meaningless sub-segments, see 2.2). The corresponding true segments are then relabeled (see 2.3), and the Metropolis algorithm is reapplied (see 2.4) for another short relaxation process after which steady state is achieved.

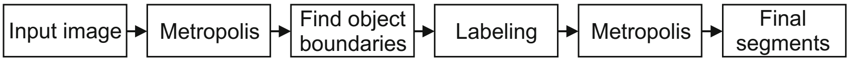


Fig. 1. Block diagram of the proposed segmentation method

2.1 Metropolis Algorithm

In the Potts model, a spin variable σ_k, which can take on q discrete values $v_1, v_2, \ldots, v_q$, called spin states, is assigned to each pixel of the image. The energy of the system is described by

$$E = -\sum_{<ij>} J_{ij}\delta_{ij} \quad , \tag{1}$$

with the Kronecker sign

$$\delta_{ij} = \begin{cases} 1 \text{ if } \sigma_i = \sigma_j, \\ 0 \text{ otherwise.} \end{cases} \tag{2}$$

where σ_i and σ_j are the respective spin variables of two neighboring pixels i and j. The function

$$J_{ij} = 1 - |\mathbf{g_i} - \mathbf{g_j}|/\overline{\varDelta} \tag{3}$$

is a coupling constant, determining the interaction strength, where $\mathbf{g_i}$ and $\mathbf{g_j}$ are the respective color vectors of the pixels, and

$$\overline{\varDelta} = \alpha \cdot \big(\sum_{<i,j>} |\mathbf{g_i} - \mathbf{g_j}|/\sum_{<i,j>} 1\big) \tag{4}$$

computes the averaged color vector difference of all neighbors $< i, j >$. The factor $\alpha \in [0, 10]$ is a system parameter.

The Metropolis algorithm allows generating spin configurations S which obey the Boltzmann probability distribution [16]

$$P(S) \sim \exp\left[-\beta E(S)\right] \quad , \tag{5}$$

where $\beta = 1/kT$, T is the temperature parameter, and k is the Boltzmann constant.

Initially, values are assigned randomly to all spin variables. According to the Metropolis algorithm, each spin-update procedure consists of the following steps [17]:

1. The system energy E_A of the current spin configuration S_A is computed according to Eq. 1.
2. A pixel i with spin variable σ_i in spin state v_l is selected and for each possible move to a new spin state $\sigma_i \neq v_l$ the energy E_B of the resulting new spin configuration S_B is computed according to Eq. 1. The number of possible moves is $(q-1)$.
3. Among all new possible configurations we find the configuration with the minimum energy

   $$E_{new} = \min(E_1, E_2, \ldots, E_{q-1}) \quad , \tag{6}$$

   and compute the respective change in energy

   $$\Delta E = E_{new} - E_A \quad . \tag{7}$$

4. If the total energy of the configuration is decreased by this move, i.e. $\Delta E < 0$, the move is always accepted.
5. If the energy increased, i.e. $\Delta E > 0$, the probability that the proposed move will be accepted is given by

   $$P_{A \to B} = \exp\left(-\frac{|\Delta E|}{kT_n}\right) \quad , \tag{8}$$

   and

   $$T_{n+1} = \gamma T_n \qquad \gamma < 1 \quad , \tag{9}$$

   where γ is the annealing coefficient. We draw a number ξ randomly from a uniform distribution in the range of $[0, 1]$. If $\xi < P_{A \to B}$, the move is accepted.

Each spin update involves only the nearest neighbors of the considered pixel. Hence, spin variables of pixels that are not neighbors of each other can be updated simultaneously [18]. Therefore the Metropolis algorithm fits very well to the GPU architecture.

The energy function may contain many local minima in which the system can get trapped. This problem can be resolved by slow annealing of the spin system. An annealing schedule allows to simulate a cooling process by decreasing the temperature after each iteration (see Eq. 9). While slow cooling leads to an undesired increase in computation time, fast cooling faces the problem of domain fragmentation. In the next section, we present an algorithm for resolving the domain-fragmentation problem.

2.2 Resolving Domain Fragmentation

Domain fragmentation describes the fact that large uniform areas are being split into sub-segments despite high attractive forces within them [10]. It happens in the case of a too fast annealing process when the temperature decreases rapidly and the system arrives too early at the "frozen" state. For illustration, the spin configuration with $q = 6$ after 20 Metropolis iterations is presented for an example image (Fig. 2(a-b)). Large interaction forces within the apple and the background lead to the creation of domains that try to cover each other. This effect has its origin in the finite interaction range and local dynamics of the Metropolis algorithm. The fragmented domains, however, carry all the required information to resolve this problem. For this we consider the result after an initial fast cooling phase consisting of 20 Metropolis iterations only and find that domain-fragment boundaries are unstable and clear-cut whereas true segment boundaries are stable and characterized by a noisy local neighborhood (Fig. 2(b)). This holds true for real images due to their finite image gradient at true boundaries and it allows us to distinguish true segment boundaries from those caused by domain fragmentation.

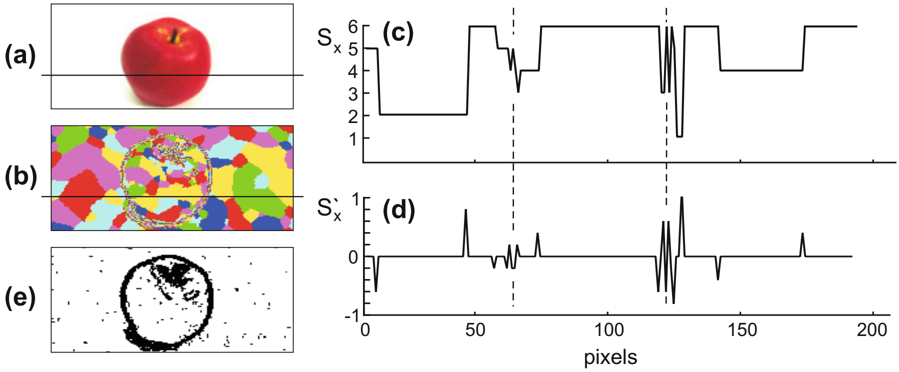


Fig. 2. Detection of real boundaries after using the Metropolis algorithm. (a) Input image. (b) Configuration of spin states after 20 Metropolis iterations. (c) Function of spin states for one image row as marked by a horizontal line in panels (a) and (b). (d) Changes of spin state for the same row where each peak represents a changing spin state. (e) Detected object boundaries.

The procedure works as follows. After a fixed small number of Metropolis iterations, we compute the spatial derivatives along the x and y direction of the spin-state configuration $S(x, y)$ according to

$$S'_x = \frac{\Delta S(x,y)}{\Delta x} \qquad \text{and} \qquad S'_y = \frac{\Delta S(x,y)}{\Delta y} \quad . \tag{10}$$

In Fig. 2(c,d) functions S_x and S'_x are depicted for one row of the original image. Each peak of S'_x represents a change in the spin state. Here we are interested only in the number of peaks rather than in the derivative values, because the Potts model does not penalize differences between certain spin states stronger than others (see Eq. 2). The frequency of peaks increases significantly at real boundaries (depicted by dashed lines). Thus considering couples of pixels in parallel we find boundaries

$$B(x_i, y_j) = \begin{cases} 1 \text{ if } S'(x_i, y_j) \neq 0 \text{ and } S'(x_{i-1}, y_j) \neq 0, \\ 1 \text{ if } S'(x_i, y_j) \neq 0 \text{ and } S'(x_i, y_{j-1}) \neq 0, \\ 0 \text{ otherwise.} \end{cases} \tag{11}$$

The result of this procedure is a binary image where objects are depicted by white and boundaries by black (see Fig. 2(e)). This step can also be implemented completely in parallel. Erroneous noisy speckles arising from this procedure are corrected by applying the Metropolis algorithm a second time for recovery (see 2.4). We used a fixed parameter $\alpha = 0.7$ for all images. For images which have not much texture a larger parameter $\alpha > 1$ can be used to obtain even better results.

Note, one cannot easily use a conventional edge detector (on the original image) for this. An edge detector would indeed find many segment boundaries, but it would also find others which are unrelated to the segments that come out from the Metropolis procedure. As we need to continue the relaxation process, we should do this using only "the correct" segments. Otherwise relaxation would have to undo all wrong segments to finally reach the minimum. Moreover the proposed procedure yields closed object boundaries while many edge detectors produce borders having gaps. The method of using the noisiness to distinguish real edges from domain edges is consistent within our algorithmic framework and, thus, allows continuation of the Metropolis procedure without problems.

2.3 Labeling of Connected Components

After resolving the domain fragmentation described in Sec. 2.2, all connected components, i.e. areas having a closed boundary, have to be labeled in order to get the spin states configuration back.

As our segmentation algorithm has to be sufficient for real-time applications, we decided to use a procedure suggested by He et al. (2009) which is, to our knowledge, among many algorithms proposed for the labeling of connected components in a binary image, the fastest labeling algorithm to date [19]. All steps of the employed labeling procedure are represented in Fig. 3.

The chosen algorithm completes labeling in two scans of an image: during the first scan it assigns provisional labels to object pixels (see Fig. 3(b)) and records label equivalences for labels, belonging to the same object. Label equivalences are being resolved during the first scan choosing one of the equivalent labels as a representative label. All representative labels are stored in the representative label table where provisional labels act as indexes. During the second scan, all equivalent labels are replaced by their representative label obtained from the representative label table (see Fig. 3(c)). The detailed description of the algorithm and its optimizations can be found in [19].

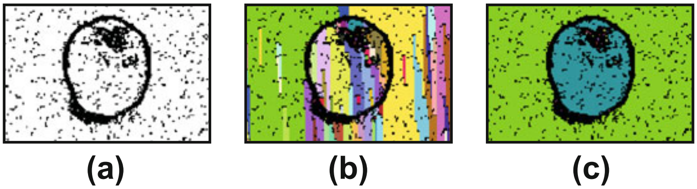


Fig. 3. Fast labeling of connected components. (a) Defined object boundaries. (b) Provisional labels after the first image scan. (c) Representative labels assigned after the second image scan.

Both image scans run on the CPU and are extremely fast for image sizes that are being used in our work. Especially the second scan can be accelerated on the GPU architecture, since representative labels can be assigned simultaneously to all pixels by independent parallel processing threads.

2.4 Employment of Metropolis for Final Relaxation

After the labeling of connected components we assign spin states to all pixels according to

$$\sigma(x_i, y_j) = L(x_i, y_j) \bmod q \quad , \tag{12}$$

where mod means that the segment label $L(x_i, y_j)$ of the pixel is divided by the number of possible spin states q and the new spin state σ is the remainder of the division. After this assignment we apply five more Metropolis iterations to obtain the final spin configuration after which final segments can be extracted.

2.5 Experimental Environment

As hardware platforms for testing of our segmentation algorithm we used

- NVIDIA card GeForce GTX 295 (using a single GPU) with 40 multiprocessors each having 8 cores, so 240 processor cores in total and 896 MB device memory.
- CPU 2.2GHz AMD Phenom Quad 9550 (using a single core) with 2 GB RAM.

3 Experimental Results

3.1 Segmentation Results

We applied the developed algorithm to a set of real images, i.e. *Cluttered scene*, *Lampshade* from the Middlebury dataset[2] and *Skier* from the Berkeley dataset[3] (see Fig. 4(a)). The results at the different stages of the algorithm are shown in Fig. 4(b-e).

[2] Available under *http://vision.middlebury.edu/stereo/*

[3] Available under *http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/*

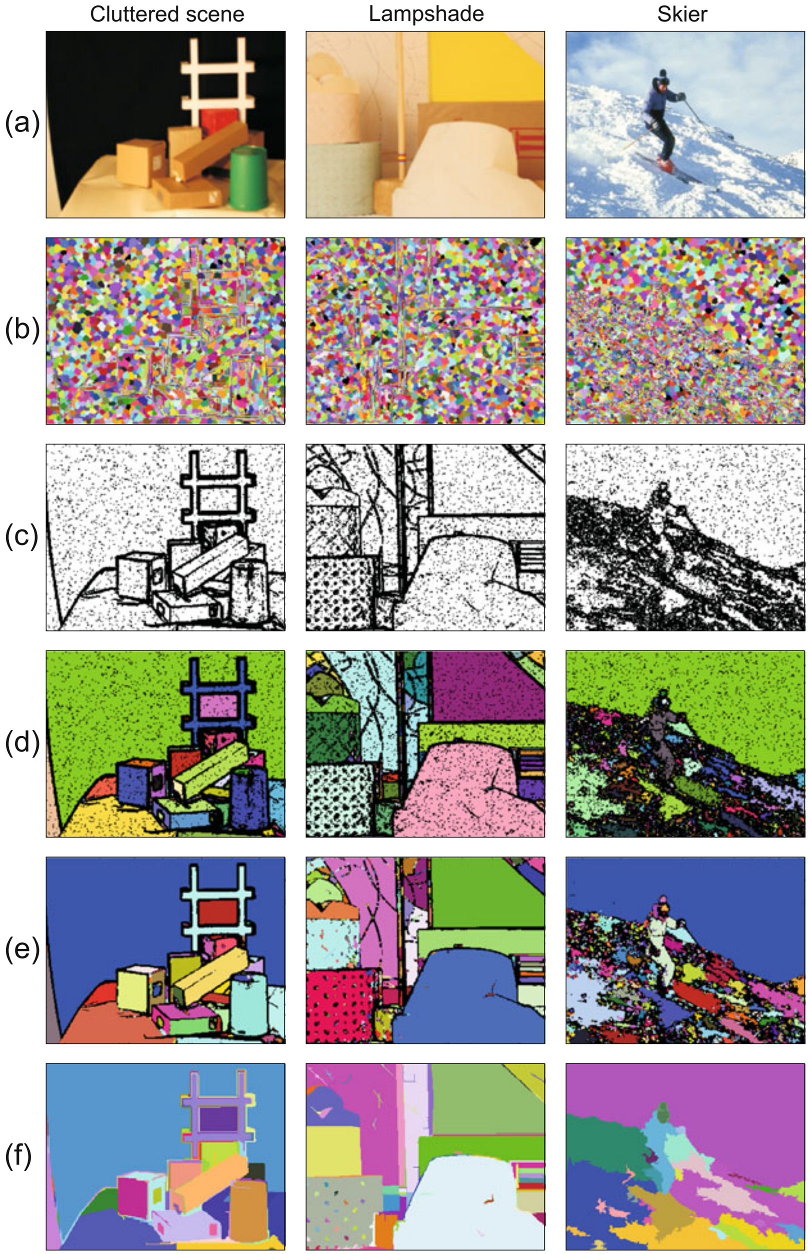


Fig. 4. Intermediate and final results of the segmentation algorithm for three example images. (a) Test images. (b) Results after 10 Metropolis iterations with $q = 256$. (c) Found objects boundaries. (d) Labeling of connected components. (e) Extracted segments after the final relaxation. (f) Results of graph-based image segmentation approach of Felzenszwalb and Huttenlocher [7].

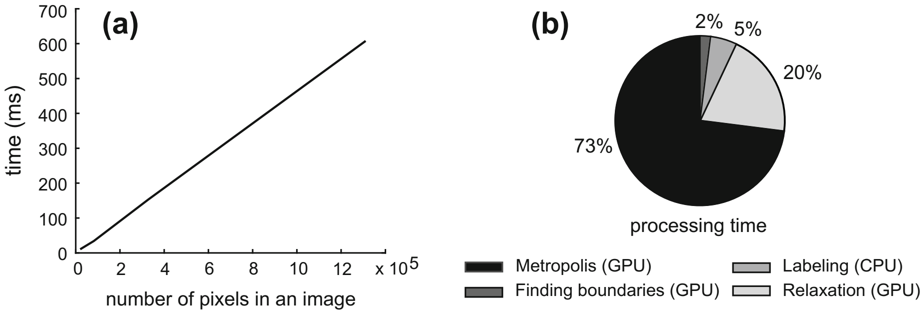


Fig. 5. Timing performances of the algorithm. (a) Execution time on GPU versus the number of pixels in an image. (b) Total computation time of all algorithmic steps in percentage.

In Fig. 4(b), the spin states after 10 Metropolis iterations are shown. Domain fragmentation is clearly visible, characterized by noisy boundaries, in all three images. The more textured an input image is, the more noisy entities are arising. Objects borders are found, resulting in a binary image (see Fig. 4(c)). Since the *Lampshade* and *Skier* images contain much more texture than *Cluttered scene*, more boundaries and consequently more boundary errors are visible at this stage (Fig. 4(c), middle and right panels). Experimentally it was determined that 10 Metropolis iterations are enough to obtain closed object boundaries which are acceptable for the labeling of connected components.

In Fig. 4(d) the results after the labeling of connected components are represented. Errors after the resolving domain fragmentation, resulting in noisy speckles, are removed by reapplying the Metropolis procedure for system relaxation. The respective final segments extracted after the final relaxation are shown in Fig. 4(e). Fig. 4(f) shows a comparison to a conventional segmentation algorithm.

3.2 Execution Time

In Fig. 5(a), the dependence of the segmentation runtime on the number of pixels in an image for the GPU architecture is shown. We can see that the dependence is almost linear, as the Metropolis algorithm, resolving the domain fragmentation and the labeling procedure have almost the ideal linearity property versus image size (i.e., for $N \times N$ images, its complexity is $O(N^2)$).

Among all algorithmic steps only the runtime of the labeling depends on the structure of the input image, but deviations are in the range of two milliseconds for images up to 256 × 320 pixels and of ten milliseconds for images up to 1024 × 1240 pixels. For very textured images like *Skier* the labeling takes longer, since shapes of objects are more difficult and more provisional labels are being assigned, so more time is needed to solve label equivalences (see 2.3). The most time-consuming step is the Metropolis procedure, taking together with the

Table 1. Total computation times obtained for GPU and CPU for different sizes of the test images

Image size (px)	GPU / CPU (ms)		
	"Cluttered scene"	"Lampshade"	"Skier"
128 × 160	9.55 / 1.4 × 10^3	10.5 / 1.4 × 10^3	11.0 / 1.5 × 10^3
256 × 320	33.8 / 5.8 × 10^3	34.3 / 5.9 × 10^3	33.7 / 5.9 × 10^3
512 × 640	150.5 / 24.3 × 10^3	153.1 / 24.4 × 10^3	154.6 / 24.3 × 10^3
1024 × 1280	601.3 / 100.8 × 10^3	612.2 / 102.2 × 10^3	609.8 / 102.5 × 10^3

Table 2. Comparison of computation times obtained for the proposed method on GPU and graph-based method by Felzenszwalb and Huttenlocher on CPU [7]

Image size (px)	GPU method / graph-based on CPU (ms)		
	"Cluttered scene"	"Lampshade"	"Skier"
128 × 160	9.55 / 10.0	10.5 / 10.0	11.0 / 10.0
256 × 320	33.8 / 75.0	34.3 / 75.0	33.7 / 75.0
512 × 640	150.5 / 510.0	153.1 / 500.0	154.6 / 470.0
1024 × 1280	601.3 / 3020.0	612.2 / 2950.0	609.8 / 2920.0

relaxation process more than 90 percent of the total execution time (see Fig. 5(b)). Processing times of our segmentation algorithm on GPU and CPU are compared in Table 1. The comparison of processing times for the proposed GPU method and the efficient graph-based method on CPU of Felzenszwalb and Huttenlocher is shown in Table 2.

4 Discussion

We introduced a novel parallel nonparametric image segmentation algorithm based on the method of superparamagnetic clustering. Using the highly parallel GPU architecture we obtained processing times which are sufficient for real-time applications. For images of size 256 × 320 pixels the algorithm can be used for real-time processing tasks and of size up to 512 × 620 for close to real-time applications. The algorithm has been adapted to fit the parallel architecture of GPUs, including a novel procedure to resolve the domain-fragmentation problem.

The proposed method has been applied to several real images. Obtained segmentation results for a single frame are comparable with conventional approaches. In Fig. 4(f) results of graph-based image segmentation proposed by Felzenszwalb and Huttenlocher (2004) are shown. We can see that our results (see Fig 4(e)) look very similar with the difference that our method yields more small segments for very textured images like *Skier*. This happens because our method takes into account only color information of interacting pixels. Therefore our algorithm has a better performance for large segments than for small ones, since the color segmentation works best for large uniform image regions. For textured areas, corresponding to small regions, the performance of our algorithm decreases, because the gray-value similarity of neighboring pixels is too low. Towards better results for very textured images, in the future texture segmentation can be incorporated into the algorithm. With respect to processing time our method outperforms the mentioned graph-based approach (see Table 2).

Also it is necessary to point out that the processing time is almost independent of image structure, number of segments, and image density, i.e. the relation between object pixels and boundary pixels during the labeling of connected components. The slowest part of the algorithm is Metropolis updating, since some annealing iterations have to be executed.

Before parallel hardware architectures became widespread, most image segmentation methods running on CPU either delivered precise segmentation results at low speed or real-time processing with relatively poor accuracy. Nowadays different types of parallel architectures are used for the real-time image segmentation: digital signal processors (DSP) with field programmable gate arrays (FPGA) and GPUs [15,20,21]. Using of DSPs and FPGAs makes it possible to achieve real-time processing [15] but requires far more development time than in the case of GPUs with CUDA. Furthermore, software developed for FPGAs is highly dependent on the used chip type and as a consequence has a limited portability while CUDA applications run on a wide range of GPUs without any problems.

After release of the framework CUDA by NVIDIA in 2007, some image segmentation algorithms were implemented on the GPU [20,21]. A method proposed by Kim et al. (2009) segments cervicographic images using the spatially coherent deterministic annealing, but not in real-time. The real-time algorithm of Vineet and Narayanan (2008) performs a binary segmentation of the image into objects of interest and background, which is a different problem. In our case, the whole image is segmented into similar regions according to a similarity criterion, here color.

Currently, we are investigating whether alternative approaches for computation of equilibrium states of the Potts model can be parallelized efficiently as well [3,4,5]. In the future, we will apply the proposed algorithm to the problem of image-sequence segmentation with the aim to track image segments in real time in a model-free way [22].

Acknowledgment

The work has received support from the German Ministry for Education and Research (BMBF) via the Bernstein Center for Computational Neuroscience (BCCN) Göttingen and the EU Project PACO-PLUS. B.D. also acknowledges support from Spanish Ministry for Science and Innovation via a Ramon y Cajal fellowship. We thank Karl Pauwels for valuable discussion.

References

1. Shapiro, L.G., Stockman, G.C.: Computer Vision. Prentice-Hall, Englewood Cliffs (2001)
2. Sahoo, P.K., Soltani, S., Wong, A.K.C., Chen, Y.: A survey of thresholding techniques. Computer Vision, Graphics and Image Processing 41(2), 233–260 (1988)
3. Swendsen, R.H., Wang, S.: Nonuniversal critical dynamics in Monte Carlo simulations. Physical Review Letters 76(18), 86–88 (1987)
4. Wolff, U.: Collective Monte Carlo updating for spin systems. Physical Review Letters 62(4), 361–364 (1989)
5. von Ferber, C., Wörgötter, F.: Cluster update algorithm and recognition. Physical Review E 62(2), 1461–1464 (2000)
6. Boykov, Y., Funka-Lea, G.: Graph cuts and efficient N-D image segmentation. International Journal of Computer Vision 70(2), 109–131 (2006)
7. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient graph-based image segmentation. International Journal of Computer Vision 59(2), 167–181 (2004)
8. Comaniciu, D., Meer, P.: Mean shift: a robust approach toward feature space analysis. Pattern Analysis and Machine Intelligence 24(5), 603–619 (2002)
9. Potts, R.B.: Some generalized order-disorder transformations. Proc. Cambridge Philos. Soc. 48, 106–109 (1952)
10. Eckes, C., Vorbrüggen, J.C.: Combining data-driven and model-based cues for segmentation of video sequences. In: Proc. of World Congress on Neural Networks, pp. 868–875 (1996)
11. Blatt, M., Wiseman, S., Domany, E.: Superparamagnetic clustering of data. Physical Review Letters 76(18), 3251–3254 (1996)
12. Boykov, Y., Kolmogorov, V.: An experimental comparison of min-cut/max-flow algorithms for energy minmization in vision. Pattern Analysis and Machine Intelligence 9, 1124–1137 (2004)
13. Ising, E.: Beitrag zur Theorie des Ferromagnetismus. Z. Phys. 31, 253–258 (1925)
14. Geman, D., Geman, S.: Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. Pattern Analysis and Machine Intelligence 6, 721–741 (1984)
15. Meribout, M., Nakanishi, M.: A new real time object segmentation and tracking algorithm and its parallel architecture. Journal of VLSI Signal Processing 39(3), 249–266 (2005)
16. Carnevali, P., Coletti, L., Patarnello, S.: Image processing by simulated annealing. IBM Journal of Research and Development 29(6), 569–579 (1985)
17. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. J. of Chem. Phys. 21(11), 1087–1091 (1953)
18. Barkema, G.T., MacFarland, T.: Parallel simulation of the ising model. Physical Review E 50(2), 1623–1628 (1994)
19. He, L., Chao, Y., Suzuki, K., Wu, K.: Fast connected-component labeling. Pattern Recognition 42, 1977–1987 (2009)
20. Kim, E., Wang, W., Li, H., Huang, X.: A parallel annealing method for automatic color cervigram image segmentation. In: Medical Image Computing and Computer Assisted Intervention, MICCAI-GRID 2009 HPC Workshop (2009)
21. Vineet, V., Narayanan, P.J.: CUDA cuts: fast graph cuts on the GPU. In: Proc. CVPRW 2008, pp. 1–8 (2008)
22. Dellen, B., Aksoy, E.E., Wörgötter, F.: Segment tracking via a spatiotemporal linking process including feedback stabilization in an n-d lattice model. Sensors 9(11), 9355–9379 (2009)

Parallel Volume Rendering Implementation on Graphics Cards Using CUDA

Jens Fangerau and Susanne Krömker

Interdisciplinary Center for Scientific Computing - IWR
Heidelberg University
Visualization and Numerical Geometry Group
Im Neuenheimer Feld 368, 69120 Heidelberg, Germany
Phone: +49 6221 54 8844, phone: +49 6221 54 8883
{jens.fangerau,kroemker}@iwr.uni-heidelberg.de,
www.iwr.uni-heidelberg.de

Abstract. The ever-increasing amounts of volume data require high-end parallel visualization methods to process this data interactively. To meet the demands, progamming on graphics cards offers an effective and fast approach to compute volume rendering methods due to the parallel architecture of today's graphics cards.

In this paper, we introduce a volume ray casting method working in parallel which provides an interactive visualization. Since data can be processed independently, we managed to speed up the computation on the GPU by a peak factor of more than 400 compared to our sequential CPU version. The parallelization is realized by using the application programming interface CUDA.

Keywords: volume rendering, ray casting, GPGPU, parallel computing, CUDA.

1 Introduction

Direct volume rendering, or volume rendering for short, is a visualization method in computer graphics to create colored and semitransparent segments from a 3D scalar dataset and to project them onto a 2D image. Applications are found in medicine, biology, geology and fluid dynamics for measured as well as simulated data. Although a lot of efforts and simplifications were made to speed up the algorithms, volume rendering has a huge computing time because of large cubic data sets which do not always allow for an interactive visualization.

Another approach to investigate a volume works by detecting isosurfaces using the *Marching Cubes* algorithm by Lorensen and Cline [1]. This *indirect volume rendering* method generates triangle meshes of 2D surfaces that separate 3D areas with lower values from those with higher than the isovalue. But considering blurred or cloud-like objects with smoothly varying values, direct rendering techniques are better suited. This follows from the possibility to get direct access to a volume without assigning it a geometrical structure with a triangulation that shows arbitrary surfaces.

R. Keller et al. (Eds.): Facing the Multicore-Challenge, LNCS 6310, pp. 143–153, 2010.

Volume rendering has become an important scientific tool and applications are for example in medical science where data are acquired by *computed tomography* (CT) or *magnetic resonance tomography* (MRT). These methods compute a stack of gray-value images which are assembled to a volume. Another application area can be found in biology, where volume data are generated by virtual optical slices via *confocal microscopy*.

By the volume rendering technique, single elements of a volume, the *voxels*, are assigned a certain color and opacity. The opacity is used for determining the density and the transparency of a voxel and consequently for the whole volume. Thus, certain areas can be made diaphanous or hid entirely and allow diagnosis without making a surgical intervention. Figure 1 shows two examples of a CT scan visualized with our ray casting algorithm. Here, volume clipping is used to reveal inner parts of the head.

Datasets resulting from simulations of aerodynamic behavior on different car shapings consist of scalar values on an irregular grid and therefore more information about connectivity and adjacencies of the data is needed. Generally, it is not necessary that 3D data exist on a regular grid. However, for a fast computation in volume rendering, we only regard datasets on regular grids.

Today 3D datasets have a dimension[1] of $512 \times 512 \times 512$ pixels or even bigger and despite the fact that the processing power is growing permanently, volume rendering makes great demands on every system visualizing these data. Even with existing speed-ups for miscellaneous render methods, a limit is reached at a certain size of the dataset such that an interactive operation is not possible anymore.

This limitation can be overcome by exploiting the parallel architecture of today's *graphics processing units* (*GPUs*). They are specialized on massive parallel data problems and belong to the field of *General-Purpose computation on Graphics Processing Units* (*GPGPU*). In [2], a massively parallel volume rendering algorithm is introduced. This algorithm, called *2-3 swap*, allows an arbitrary number of processors to be used for image compositing to overcome the bottleneck of interprocessor communication. There also exist algorithms on GPU clusters [3] [4] to process large data sets.

To employ this computing device we realize a volume rendering method with the *Compute Unified Device Architecture* (*CUDA*) by NVIDIA [5]. Since the *ray casting* method of volume rendering consists of independent calculation steps, it is well-suited to be parallelized on the graphics card. More precisely, NVIDIA's graphics cards have many *multiprocessors* (MP) where each multiprocessor consists of eight scalar processors. For example, the GeForce GTX 280 features 30 MPs and so in total $30 \times 8 = 240$ *streaming processors* (SP) that concurrently operate in parallel. Thus, data can be evaluated much faster than on the *central processing unit* (*CPU*) which can only operate sequentially (or at most with four cores in parallel like today's quad-core CPUs).

[1] The size does not have to be a power of 2 but this is an ideal case for computer graphics when working with textures.

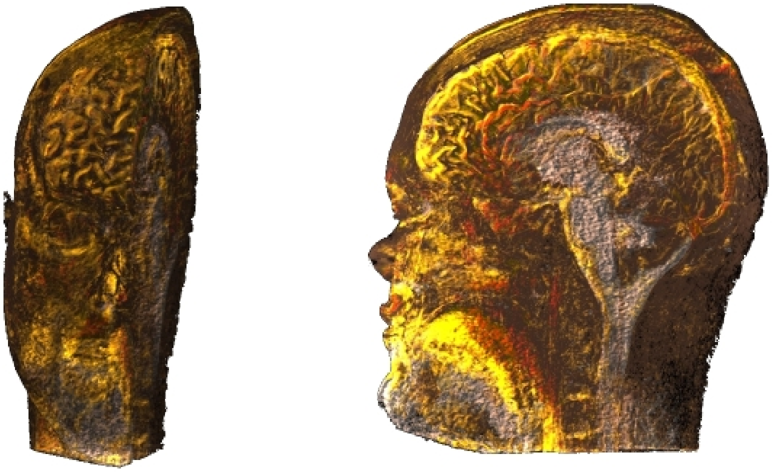


Fig. 1. CT volume data of dimension 336 × 384 × 132 pixels from a human head visualized in *GVvis* with our ray casting method. The size of the projected image is 2048 × 2048 pixels.

2 State-of-the-Art in Volume Rendering

The basis for all volume rendering methods is the physical model of transport of light through a colored, semitransparent and bounded data medium which is represented by the volume. This transport of light is approximated by the time-independent *equation of transfer* which describes the transport of light against absorption, emission and scattering,

$$n \cdot \nabla I = -\chi I + \eta, \tag{1}$$

where I is the *specific intensity* and n denotes the direction of the radiation field. χ describes the *extinction coefficient* or *total absorption coefficient* and η specifies the *emission coefficient*.

Equation (1) can be seen as the *volume rendering equation* which is equivalent to the rendering equation of Kajiya [6] if a series of simplifications of the interaction of light are made, like setting the boundary conditions, the assumption that light spreads out in vacuum and considering the elastic case, i.e., ignoring the frequency of light. Lacroute listed in his technical report [7] explicit restrictions to generate the volume rendering equation.

The behavior of light described by the equation of transfer can be approximated by several *optical models* like the density-emitter model of Sabella [8] and (1) can then be computed by integrating along one ray which leads to the *volume compositing equation* in its most common notation by using the *over* operator. In image processing and computer graphics this equation is also known as *alpha blending*,

$$\begin{aligned} C_{out} &= C_m \alpha_m + (1 - \alpha_m) C_{in} \\ &= C_m \,\mathrm{over}\, C_{in}, \end{aligned} \tag{2}$$

where α_m describes the *opacity* of the current voxel, C_{in} denotes the previous density or color, C_m defines the current color of the voxel and C_{out} denotes the color that has to be computed. More details about light transport and the derivation of the volume rendering equation are given in [9].

The volume rendering method *ray casting* follows immediately from (2). The analysis varies in computing from *back-to-front* or *front-to-back*. In the first case the ray has to be evaluated until it reaches the final projection image but in proceeding front-to-back, *early ray termination* can be used to abort the ray traversal at an early stage. In this case the volume compositing equation looks as follows:

$$\begin{aligned} C_{out} &= C_m (1 - \alpha_{in}) \alpha_m + C_{in}, \\ \alpha_{out} &= (1 - \alpha_{in}) \alpha_m + \alpha_{in}, \end{aligned} \tag{3}$$

where α_{in} is the previous opacity compared to the current opacity α_m and α_{out} states the finally computed one.

As mentioned before, volume rendering is a visualization method to create a projection of a 2D image from a 3D scalar dataset. The different algorithms for volume rendering can be categorized into three main classes: *Image-based* techniques generate a projection based on the image plane whereas by contrast *object-based* techniques create a projection based on the object. The third class deals with *frequency domain methods* which process volume data in the frequency domain and transform them into the image domain.

The most common representative of the image-based methods is *ray casting* in which for each pixel of the image plane, a ray is cast into the bounded volume.

Volume data can also be displayed with the object-based method *texture slicing* by slicing the volume into a stack of 2D textures and the generation of polygons for each slice. The *cell projection* method does also belong to the object-based methods and will be used to visualize volume data on an irregular grid. The idea of Shirley and Tuchman [10] consists of generating tetrahedrons from a field of voxels or *cells* and their projection onto the image plane.

A hybrid technique is the *shear warp* algorithm by Lacroute [7] in which the volume is first *sheared* and after applying the volume compositing equation the distorted intermediate image is *warped* to form the final image. The shear warp algorithm is considered to be one of the fastest CPU methods because it makes optimal use of the CPU's caching ability. Thus the algorithm makes efficient use of coherence data structures and traversal occurs in storage order. There is very little overhead due to addressing arithmetic or inefficient memory access patterns.

3 Parallelization in Visualizing Volume Data

For accelerating volume rendering techniques there already exist many approaches on the GPU. Krüger and Westermann [11] present their well-known stream model for volume ray casting by using the fragment shader of the graphics card. Their adept idea is the storage of front faces and back faces of the bounding box in 2D RGB and 2D RGBA textures respectively. This technique can be seen as the state-of-the-art volume rendering technique for interactive volume rendering.

There are accelerated ray casting systems for an interactive light field display [12] that provides the possibility to manipulate virtual volumetric objects floating in the display workspace. Kim provides in his dissertation [13] a stream model for ray casting implemented in CUDA and focuses on the decomposition of fine-grain task parallelism that achieves load balancing among the multiprocessors.

Mensmann et al. [14] offer a slab-based ray casting method with CUDA and assert that common acceleration methods like bricking do not benefit from CUDA features such as shared memory. They compare their results with the technique from [11] where both approaches are implemented in the volume rendering engine *voreen*[2]. Moreover, there also exist techniques on CPU, GPU and many-core architectures for the purpose of medical imaging [15] and a scheme of data compression is given that reduces data-transfer overhead.

Next to ray casting accelerations there also exists a parallel shear warp method on distributed-memory multiprocessor systems [16] but since shear warp is well-suited for caching on the CPU, the speed-up gained on this processing unit cannot be transfered to the GPU which uses the transistors mainly for data processing. Beyond, optimizations for the algorithm require a nonuniform access to memory, which is not available on the GPU.

4 Implementation

Due to the evident fact of volume ray casting that each ray is cast into the volume independently, we realize the compositing equation (2) and (3) in parallel on the GPU. For each ray, equidistant samples are taken along the ray whereupon samples between voxels in general are interpolated trilinearly. The samples are shaded depending on their computed gradients and all sample contributions along one ray are evaluated by using (2) which then result in the final color of the according pixel in the image plane. Figures 1 and 2 show images generated by our ray casting method. We fulfill the parallelization using the *CUDA-API*, which distributes the calculations of the final colors on the streaming processors of the GPU. In the context of GPU programming with CUDA, the CPU is also called *host* and the GPU is known as the *device* [5].

The host loads the application or function that has to be computed as a *kernel* onto the device. The kernel will then be executed in parallel by several *threads*. A thread denotes one part of an execution in processing of a program. The

[2] http://www.voreen.org/

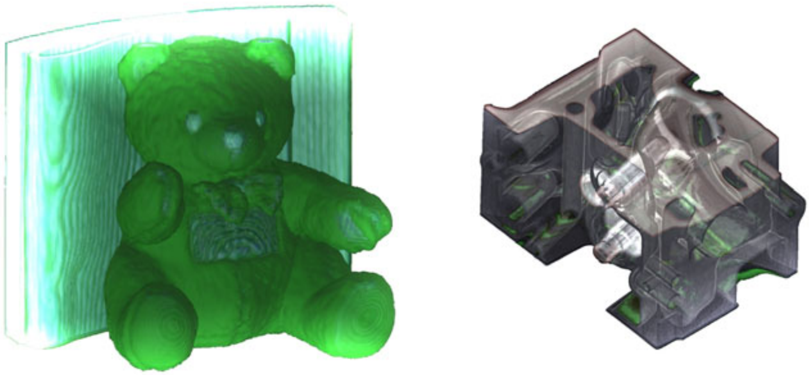


Fig. 2. Visualization of two volume examples with the program *GVvis* using our GPU ray casting method. In both cases the size of the projected image is 2048 × 2048 pixels. **Left:** CT volume data of dimension 128 × 128 × 62 pixels from a teddy bear. **Right:** CT volume data of dimension 256 × 256 × 110 from an engine block with transparent color properties.

threads are arranged hierarchically within a *block* and each has an ID number for addressing. Furthermore, a set of blocks is contained in a *grid* in which these blocks also have an ID-number. As a result we have a grid of thread blocks.

In the following we will discuss our algorithm in detail with main focus on the kernel. On the host, we define a built-in 3D vector in CUDA of dimension n, named `f3Sum`, where n is the amount of rays cast into the volume. This vector stores the results of the compositing equation for each pixel of the image plane in RGB values. It is initialized with 0 on each thread with associated `idx` and `idy`.

```
f3Sum[idx + imageWidth * idy] = (0,0,0);
```

On the device, each thread estimates the start and the direction vector `f3Start` and `f3Dir` of one ray, depending on `idx` and `idy` as well as the position and viewing direction of the camera. If the ray intersects the boundary box of the volume, then (3) will be evaluated in front-to-back order to use early ray termination. Now, for an arbitrarily chosen `stepSize` the ray will be traversed until it either exits the bounding box or the criteria of early ray termination is fulfilled. The call below is executed on each thread and the contribution of a voxel in position `f3Pos` along the ray will be added to the corresponding entry of `f3Sum`.

```
for step = 0 to MaxStep do
  if (!EarlyRayTerminationIsFulfilled) then
    f3Pos = f3Start + step * stepSize * f3Dir;
    f3Sum[idx + imageWidth * idy] += ContributionOnPosition(f3Pos);
```

By using early ray termination we get a percentaged speed-up between 5% and 30% depending on the volume and its position relative to the viewing camera. The final color of one ray traversal is stored in the associated entry in `f3Sum`. This vector will be sent back to the host when the whole processing on the device is done. When kernel execution is completed, a synchronization of all threads is done automatically so we can be certain that all results are stored in f3Sum. Afterwards the entries of f3Sum form a RGB texture on the CPU and will be visualized with the desired size of the image plane.

However, we do not always find the ideal case that all threads have equal computing times because each ray differs in length depending on volume orientation and viewing direction of the camera. In CUDA, a kernel is processed by distributing the blocks to multiprocessors with available execution capacity. The threads within a block execute concurrently on one multiprocessor and when they terminate, new blocks can be launched on the now idle multiprocessor. In our CUDA implementation, we find out that a block size of $(\texttt{dimBlock.x} = 8) \times (\texttt{dimBlock.y} = 8)$ and a grid size of

$$\left\lfloor \frac{\texttt{imageWidth} + \texttt{dimBlock.x} - 1}{\texttt{dimBlock.x}} \right\rfloor \times \left\lfloor \frac{\texttt{imageHeight} + \texttt{dimBlock.y} - 1}{\texttt{dimBlock.y}} \right\rfloor$$

result in the best computation times. Block sizes smaller than 8×8 yield lower framerates [FPS] since a multiprocessor has less work to do and data access on the global memory is expensive. For block sizes greater than 8×8 we observe not much of a difference in computation time. Note that this fluctuation depends on the amount and arrangement of input data to be parallelized.

5 Benchmarks and Results

The program we developed to visualize the volume data is called *GVvis* which among other methods, it contains an implemented version of the volume ray casting algorithm on the CPU as well as on the GPU. Both techniques are accelerated by using early ray termination. The program is built with g++ 4.3 on an unix system by using C++, OpenGL, CUDA 2.3 and Qt for the GUI.

We examine three different 8-bit volume data generated by CT which are illustrated in figure 1 and 2. The teddy bear and the engine block are chosen randomly from open source projects which are provided by the *Computer Graphics Group* of the University of Erlangen-Nuremberg. The human head is kindly provided by the *German Cancer Research Center* (DKFZ). The volumes differ in their volumetric size and in the amount of transfer functions affecting their visual appearance. We choose these data to compare the computing times of our CPU- and GPU-based ray casting version against the image size. There the image size denotes the resolution of the texture storing the final results. Consequently, the amount of rays cast into the volume is the product of currently used image width and height of the texture. Hence, a bigger image size yields a more resolutive visual output.

We consider volume sizes below 20 MB because the graphics card is limited to its own global memory. Our program needs to send the whole volume dataset to the device including transfer function elements. For allocating all variables (e.g. voxel field, gradient, transfer function) we need nearly 300 MB for the example of the human head. In addition, more than 200 MB of memory have

Table 1. Computing time on different computer systems

	System 1			**System 2**		
Image size [pixels]	**CPU time [sec]**	**GPU time [sec]**	$\frac{\textbf{CPU}}{\textbf{GPU}}$	**CPU time [sec]**	**GPU time [sec]**	$\frac{\textbf{CPU}}{\textbf{GPU}}$
Teddy bear, volume size: 128 × 128 × 62, **raw file size:** 0.99 **MB**						
32 × 32	0.038032	0.030635	1.24	0.027612	0.024472	1.13
64 × 64	0.122649	0.054660	2.24	0.122942	0.025764	4.77
128 × 128	0.463002	0.040498	11.43	0.402457	0.026366	15.26
256 × 256	1.771453	0.050477	35.09	1.501589	0.027952	53.72
512 × 512	6.991023	0.060850	114.89	5.989265	0.034157	175.35
1024 × 1024	27.872928	0.102042	273.15	24.052541	0.071986	334.13
2048 × 2048	111.324370	0.273764	**406.64**	95.429019	0.201665	**473.21**
Engine block, volume size: 256 × 256 × 110, **raw file size:** 6.9 **MB**						
32 × 32	0.048713	0.228838	0.21	0.032698	0.145385	0.22
64 × 64	0.146461	0.211831	0.69	0.114114	0.145455	0.78
128 × 128	0.456134	0.206123	2.21	0.433420	0.146993	2.95
256 × 256	1.700188	0.226943	7.49	1.710926	0.149610	11.44
512 × 512	6.778179	0.254925	26.59	6.795164	0.162959	41.70
1024 × 1024	25.650378	0.295635	86.76	27.249742	0.212555	128.20
2048 × 2048	102.400089	0.505312	**202.65**	108.903587	0.416474	**261.49**
Human head, volume size: 336 × 384 × 132, **raw file size:** 16.2 **MB**						
32 × 32	0.083478	0.482578	0.17	0.080035	0.373132	0.21
64 × 64	0.251737	0.467592	0.54	0.251024	0.366383	0.69
128 × 128	0.984302	0.485343	2.03	0.923825	0.387899	2.38
256 × 256	3.657701	0.526307	6.95	3.376500	0.410511	8.23
512 × 512	14.459413	0.644006	22.45	13.014291	0.476879	27.29
1024 × 1024	57.632532	0.820424	70.25	50.532764	0.592416	85.30
2048 × 2048	230.728418	1.2944500	**178.24**	198.635344	0.955145	**207.96**

to be allocated on the device to store the results in `f3Sum` with an image size of 4096 × 4096. The benchmarking is done on two different computer systems displaying image sizes from 32^2 to 2048^2 increasing in powers of 2. Hence, with increasing image size, we always get a scale factor of 4 for `f3Sum`. Thus, for an image size of e.g. 8192 × 8192 we have to allocate $(4 * 200\,\text{MB}) + 300\,\text{MB} = 1100\,\text{MB}$. Therefore, we are constrained to 3D sizes that can fit into the graphics memory.

In total, we have four processing units in two computer systems (considering a CPU and a GPU on each) which are compared in visualizing the data.

The two systems have the following specifications:

1. Intel Quad Core i5-750 with 4×2.666 GHz and 4 GB RAM, NVIDIA GeForce GTX 260 with 896 MB GDDR3 RAM and 27 MPs, OS: Ubuntu 9.10,

2. Intel Core 2 Duo E8500 with 2×3.16 GHz and 4 GB RAM, NVIDIA GeForce GTX 280 with 1024 MB GDDR3 RAM and 30 MPs, OS: Debian 5.0.3.

Table 1 gives an overview about the computing times of the ray casting method calculated on the CPU and GPU. By evaluating the table, we see the great speed-up with a peak factor of nearly 400 on system 1 and more than 470 on system 2 for the teddy bear and an image size of 2048 × 2048 pixels.

The weak start of the GPU is due to the fact that data of small image sizes can be cached on the CPU and therefore can be processed in a few clock cycles. In contrast, the GPU has to broadcast their computed data via the graphics bus to the main memory of the CPU where they are finally evaluated. In general, we are not interested in such small images. With increasing size, the visualization process is faster on the GPU. Considering the CPU computing times on both systems, their values vary only slightly due to the clock cycles of the CPUs. The small advantage in speed of the GTX 280 compared to the GTX 260 is because of its amount of multiprocessors and thus the amount of executing threads in parallel.

Figure 3 shows the computing time of the teddy bear for the different image sizes. We observe the linear progress of the CPU computing time in contrast to the almost constant time at the beginning but also slowly increasing time of the computation on the GPU. We can also see that with increasing image sizes the speed-up factor is getting even bigger and that the visualization process is only bounded on the memory resources available on the GPU.

We monitor rates between 5 and 15 FPS on the GPU for the image size of 2048 × 2048 for all three volumes. When comparing these results with the ray casting implementation in [11] using the fragment shader applied to our three volumes, our program reaches nearly equal frame rates. Since our code is not yet optimized in memory access and handling there is still room for improvements. Mensmann et al. [14] already compared a CUDA and fragment shader version of ray casting, both implemented in voreen. Their CUDA speed-up varies from −4% to 42% compared to the fragment shader version of [11]. The negative value

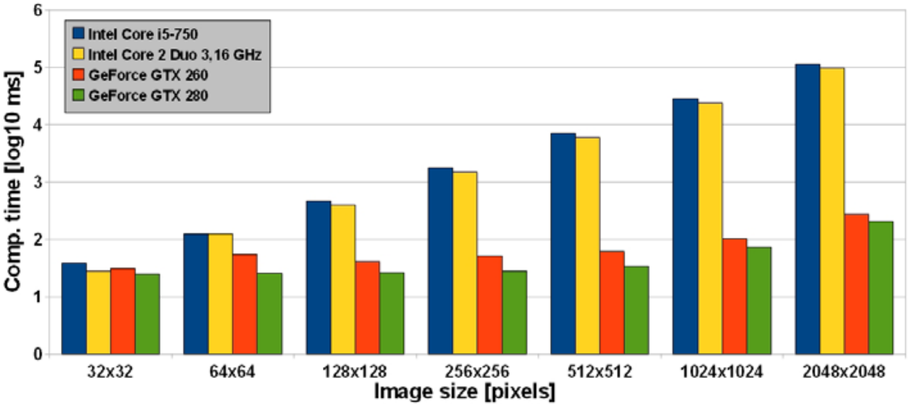


Fig. 3. Image size along computing time described by the logarithm in milliseconds for the teddy bear

appears in using phong shading because complex kernels need many registers and therefore less threads work concurrently.

6 Conclusion and Outlook

Above results show the effectiveness and rapidness of parallel computation of volume rendering methods and prove that volume ray casting is well-suited for computing on GPUs in parallel. It should be remembered that these huge time differences only occur when comparing our CPU and GPU ray casting versions and we are conscious of the fact that the CPU as well as the GPU version are not yet optimized. On the graphics card we are going to improve the perfomance by using 3D textures and deliberate shared memory handling for better memory efficiency on the host as well as on the device. Not considering the parallelism part, we can expand our ray casting algorithm in addition to early ray termination with *empty space skipping* to speed up the visualization process even more. This is also part of our current research.

Another important fact for GPU programming is the data exchange via the graphics bus. In our program we discover that this is a great bottleneck for the computing time and ultimately the visualization step. Although we use graphics cards with PCIe 2.0 x16, which corresponds to a bandwidth of 8 GB/s, we observe that the transfer time from CPU to GPU and backwards takes nearly 50% of the whole rendering process. Thus we also try to minimize the transfer rate even if this requires running kernels with low parallelism calculations.

Nevertheless, the physical limit of today's CPUs is reached sooner or later because of problems in miniaturization, manufacturing and radiation of heat. With APIs like CUDA we can implement available algorithms in parallel on the GPU, in case they can be parallelized (for example, when consisting of independent data) instead of proceeding them sequentially on the CPU. However, CUDA is limited to graphics cards from NVIDIA but with already existing open programming platforms like the *Open Computing Language* (*OpenCL*) created by the *Khronos Group*[3], parallel programming will be soon available on other hardware and hence platform independence is guaranteed. Note that not every process can be parallelized and just be released on the GPU, thus existing algorithms have to be examined whether they are suitable for parallelization. However, using the example of ray casting, we showed that the speed-up on the graphics card is enormous and the field of applications is multifunctional.

Acknowledgment

The authors like to thank Prof. Dr. Hans-Peter Meinzer and his research group, especially Mathias Seitel, at the *German Cancer Research Center* (DKFZ) for assisting and providing volume datasets.

[3] The Khronos group is a consortium which advocates the creation and administration of open standards in multimedia area.

References

1. Lorensen, W.E., Cline, H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. ACM SIGGRAPH Computer Graphics 21(4), 163–169 (1987)
2. Yu, H., Wang, C., Ma, K.-L.: Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing. In: Conference on High Performance Networking and Computing - Proceedings of the 2008 ACM/IEEE Conference on Super Computing, vol. 48, pp. 1–11 (November 2008)
3. Strengert, M., Magallón, M., Weiskopf, D., Guthe, S., Ertl, T.: Large Volume Visualization of Compressed Time-Dependent Datasets on GPU Clusters. In: Parallel Computing - Parallel Graphics and Visualization, vol. 31, pp. 205–219 (February 2005)
4. Strengert, M., Magallón, M., Weiskopf, D., Guthe, S., Ertl, T.: Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In: Eurographics Symposium on Parallel Graphics and Visualization (EGPGV 2004), pp. 41–48 (2004)
5. Compute Unified Device Architecture - Programming Guide 2.0, Nvidia (June 2008)
6. Kajiya, J.T.: The Rendering Equation. ACM SIGGRAPH Computer Graphics 20(4), 143–150 (1986)
7. Lacroute, P.G.: Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. Stanford University, Technical Report: CSL-TR-95-678 (September 1995)
8. Sabella, P.: A Rendering Algorithm for Visualizing 3D Scalar Fields. In: International Conference on Computer Graphics and Interactive Techniques, vol. 22(4), pp. 51–58. ACM, New York (1988)
9. Fangerau, J.: Volume Rendering auf Graphikkarten und parallele Implementierung unter CUDA. diploma thesis, Heidelberg University (2009)
10. Shirley, P., Tuchman, A.: A Polygonal Approximation to Direct Scalar Volume Rendering. ACM SIGGRAPH Computer Graphics 24(5), 63–70 (1990)
11. Krüger, J., Westermann, R.: Acceleration Techniques for GPU-based Volume Rendering. In: Proceedings of the 14th IEEE Visualization, p. 38. IEEE Computer Society, Los Alamitos (2003)
12. Agus, M., Gobbetti, E., Guitián, J.A.I., Marton, F., Pintore, G.: GPU Accelerated Direct Volume Rendering on an Interactive Light Field Display. In: Computer Graphics Forum, vol. 27(2), pp. 231–240 (April 2008)
13. Kim, J.: Volume Ray Casting with CUDA. dissertation, University of Maryland (2008)
14. Mensmann, J., Ropinski, T., Hinrichs, K.H.: Poster: Slab-Based Raycasting: Efficient Volume Rendering with CUDA. High Performance Graphics 2009 Posters (August 2009), http://viscg.uni-muenster.de/publications/2009/MRH09
15. Smelyanskiy, M., Holmes, D., Chhugani, J., Larson, A., Carmean, D.M., Hanson, D., Dubey, P., Augustine, K., Kim, D., Kyker, A., Lee, V.W., Nguyen, A.D., Seiler, L., Robb, R.: Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures. In: IEEE Educational Activities Department, vol. 15(6), pp. 1563–1570 (2009)
16. Sano, K., Kitajima, H., Kobayashi, H., Nakamura, T.: Parallel Processing of the Shear-Warp Factorization with the Binary-Swap Method on a Distributed-Memory Multiprocessor System. In: Proceedings of the IEEE Symposium on Parallel Rendering, p. 87 (July/August 1997)

Author Index