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

Facing the Multicore-Challenge II

Aspects of New Paradigms and Technologies in Parallel Computing

Springer

Volume Editors

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

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

Jan-Philipp Weiss
SRG New Frontiers in High Performance Computing
and Karlsruhe Institute of Technology (KIT)
Institute for Applied and Numerical Mathematics 4
Fritz-Erler-Straße 23
76133 Karlsruhe, Germany
E-mail: jan-philipp.weiss@kit.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-30396-8 e-ISBN 978-3-642-30397-5
DOI 10.1007/978-3-642-30397-5
Springer Heidelberg Dordrecht London New York

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

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

Preface

The Multicore Challenge is still causing agonizing pain on users of scientific computing, software developers, and vendors. While in theory the exponential increase of computing power is about to continue at least for the next couple of years, it is more and more difficult to harness the capabilities of parallel hardware in practical implementations. With the conference for young scientists "Facing the Multicore Challenge", which was held in Heidelberg in 2010, we initiated a platform for the mutual exchange between young researchers and experienced specialists in the domain of high-performance computing. The paper contributions, recent discussions, and the observations within the current computing landscape gave rise to the idea to have a second edition of the conference in 2011. The present proceedings are the outcome of this second conference for young scientists – "Facing the Multicore Challenge II" – held at the Karlsruhe Institute of Technology (KIT), September 28–30, 2011. The conference focused on the topics and the impact of multicore, manycore and coprocessor technologies in science and for large-scale applications in an interdisciplinary environment.

The 2011 conference – partially funded by KIT – placed emphasis on the support and the advancement of young researchers. It brought 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 in a pleasant atmosphere stimulating the exchange of ideas. 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. 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 toward personal supercomputing and very likely toward exascale computing. 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. In the contributed papers the status of the parallel evolution is investigated and theses for further development of hardware and software are discussed. Then, a load-balancing approach for hybrid programming models is considered and benefits of a task-based programming model are underlined. Research papers on parallel programming environments include a productivity and performance analysis and a case study of a programming model with high-level description of algorithms and automated vectorization. An application and performance study based on a simulator considers aspects of asymmetric manycore architectures. Furthermore, the mapping of a matrix estimation algorithm to an FPGA platform is investigated. Scheduling techniques for graphics processing units (GPUs) are presented in another paper. In the context of GPU computing, two research paper deal with the mapping and GPU acceleration of graph algorithms. The proceedings further describe the experience with high-level programming approaches for GPUs. Finally, the issues of parallel numerical methods in the manycore era are discussed in four research papers – highlighting aspects of a hybrid parallelization of a realistic simulation as well as of algebraic and geometric multigrid solvers and parallel preconditioners.

The conference organizers and editors would like to thank all the contributors for submitting exciting and novel work and providing multifaceted input to the discussions. Special thanks is devoted to the Technical Program Committee for their exhaustive work and effort in the reviewing process and their helpful feedback for the authors. Last but not least, we would like to acknowledge the financial support from Karlsruhe Institute of Technology in the context of the KIT Startup Budget 2011.

The conference *Facing the Multicore-Challenge II* was kindly funded and supported by the Karlsruhe Institute of Technology (KIT) in the context of the Startup Budget 2011. The Shared Research Group 16-1 of Jan-Philipp Weiss at KIT has received financial support from 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. The graphics on the cover were kindly produced by Dimitar Lukarski.

September 2011

Rainer Keller
David Kramer
Jan-Philipp Weiss

Organization

General Chairs

Jan-Philipp Weiss	Karlsruhe Institute of Technology, Germany
Rainer Keller	University of Stuttgart, Germany
David Kramer	Karlsruhe Institute of Technology, Germany

Program Committee

Michael Bader	University of Stuttgart, Germany
Rosa Badia	Barcelona Supercomputing Centre, Spain
Richard Barrett	Oak Ridge National Labs, Oak Ridge, USA
Mladen Berekovic	TU Braunschweig, Germany
Christian Bischof	RWTH Aachen, Germany
Arndt Bode	TU Munich, Germany
George Bosilca	University of Tennessee Knoxville, USA
Rainer Buchty	University of Tübingen, Germany
Mark Bull	EPCC, Edinburgh, UK
Hans-Joachim Bungartz	TU Munich, Germany
Franck Capello	LRI, Université Paris Sud, France
Jack Dongarra	University of Tennessee, USA
David Ediger	Georgia Tech, USA
Claudia Fohry	Kassel University, Germany
Dominik Göddeke	TU Dortmund, Germany
Georg Hager	University Erlangen-Nuremberg, Germany
Thomas Herault	Université Paris Sud, France
Hans Herrmann	ETH, Zürich, Switzerland
Vincent Heuveline	Karlsruhe Institute of Technology, Germany
Lee Howes	AMD, UK
Wolfgang Karl	Karlsruhe Institute of Technology, Germany
David Kramer	Karlsruhe Institute of Technology, Germany
Rainer Keller	Oak Ridge National Labs, Oak Ridge, USA
Paul H. Kelly	Imperial College, London, UK
Hiroaki Kobayashi	Tohoku University, Japan
Dieter an Mey	RWTH Aachen, Germany
Claus-Dieter Munz	Stuttgart University, Germany
Norihiro Nakajima	JAEA and CCSE, Tokyo, Japan
Fabia Oboril	Karlsruhe Institute of Technology, Germany
Victor Pankratius	Karlsruhe Institute of Technology, 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, Switzerland
Martin Schulz	Lawrence Livermore National Labs, USA
Christian Simmendinger	T-Systems SfR, Germany
Masha Sosonkina	Ames Lab, USA
Thomas Steinke	ZIB, Berlin, Germany
Mehdi Tahoori	Karlsruhe Institute of Technology, Germany
Walter Tichy	Karlsruhe Institute of Technology, Germany
Carsten Trinitis	TUM, Munich, Germany
Stefan Turek	Dortmund University, 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

Only the First Steps of the Parallel Evolution Have Been Taken Thus Far

James Reinders

Director, Evangelist, Intel Corporation,
Hillsboro, Oregon, USA
james.r.reinders@intel.com

1 Introduction

We are roughly fives years into the multicore era of computing, and it is safe to say that a parallel evolution is well underway. We have only taken the first steps along this evolution. In my talk, and this paper, I offer a brief history of the evolution and predictions of four major trends that will, or have, emerged and help characterize the future.

The four major trends that I have chosen to highlight and discuss are:

1. The future of *hardware* is *specialization and programmability*.
2. The future of *system design* is *imbalance*.
3. The future of *programming* is *parallelism*.
4. The future of *computing* is *data processing using parallelism*.

2 The Trends Driving Us to Parallelism

Parallel computers have been around for many years, but several recent trends have led to increased parallelism in even common everyday computers. These trends can be illustrated by looking at characteristics of Intel processors from 1971 to date. While all the data in these graphs are from Intel product lines, the same trends are apparent across the industry in the product lines of other vendors. From these trends, the assertion that hardware now generally requires explicit parallel programming to maximize performance would be difficult to refute.

In 1965, Gordon Moore observed that transistor densities on silicon devices were doubling about every two years. This observation has become known as Moore's Law. Consider Figure 1, which shows a plot of transistor counts for Intel microprocessors. Two data points at the extremes of this chart are approximately 0.001 million 10^{-3} transistors in 1971 and 1000 million 10^{3} transistors in 2011. That works out to 1.995x every two years, showing that 2x per year has been amazingly accurate over four decades.

This exponential growth has created opportunities for more and more complex designs for microprocessors. Until 2004, there was also a rise in the switching speed of transistors. This translated into an increase in the performance of microprocessors through a steady rise in the rate at which their circuits were clocked. An increase in clock rate, if the instruction set remains the same (as has mostly been the case for the Intel architecture), translates roughly into an increase in the rate at which instructions

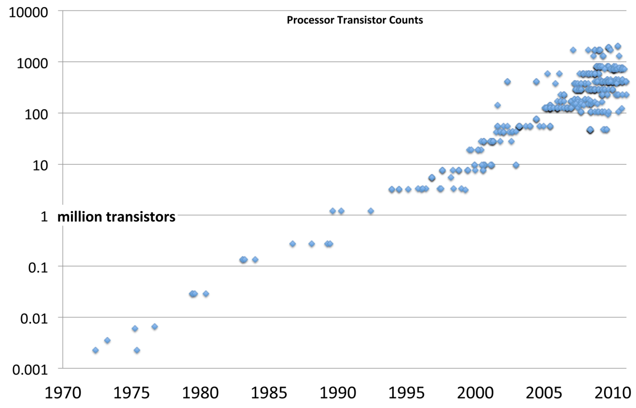


Fig. 1. Moore's Law, which observes that the number of transistors on a chip will double about every two years, continues to this day (log scale). In the multicore era, different versions of processors with different cache sizes and core counts has led to a greater diversity in processor sizes in terms of transistor counts.

are completed and therefore an increase in computational performance. This increase is shown in Figure 2.

Many of the increases in processor complexity have also been to increase performance, even on single core processors, so the actual increase in performance has been greater than this.

From 1973 to 2003, clock rates increased by three orders of magnitude (1000x), from about 1MHz in 1973 to 1GHz in 2003. However, as is clear from Figure 2, clock rates have now ceased to grow, and are now generally top out around 3GHz. In 2005, three factors converged to limit the growth in performance of single cores, and shift new processor designs to the use of multiple cores. These are known as the "three walls":

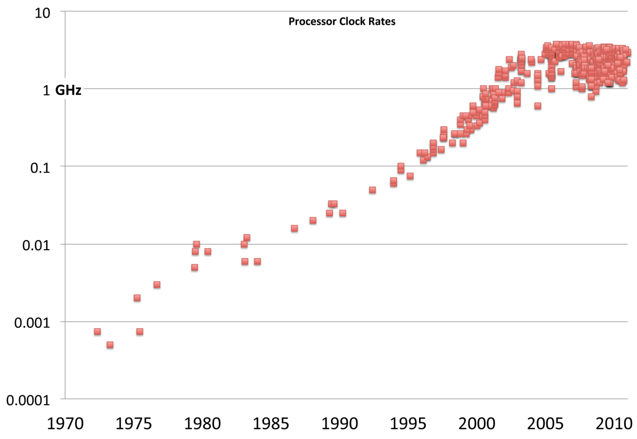


Fig. 2. Growth of processor clock rates over time (log scale). This graph shows a dramatic halt by 2005.

1. The Power Wall: Growth in power usage with clock rate that reached an unacceptable level.
2. The ILP Wall: Instruction level parallelism maximally exploited, with little remaining opportunity.
3. The Memory Wall: A growing gap between processor speeds and memory speeds leading to little gain from continued processor speed gains.

In order to achieve increasing performance over time for each new processor generation, we see these three as limiting factors for designs. We cannot depend on rising clock rates due to the power wall. We cannot depend on automatic mechanisms to find (more) parallelism in serial code, due to the ILP wall. In order to achieve higher performance, we *must* write explicitly parallel programs. When writing these parallel programs, the memory wall makes it very valuable to account for communication and memory access costs. It is often advantageous to use additional parallelism to hide the latency of memory access.

Instead of using the growing number of transistors predicted by Moore's Law for ways to maintain the *serial processor illusion*, architects of modern processor designs now provide multiple mechanisms for explicit parallelism. We must use them, and use them well, in order to achieve performance that will continue to scale over time.

The resulting trend in hardware is clear: more and more parallelism at a hardware level will become available and can benefit applications written to utilize it. However, unlike rising clock rates, non-parallelized application performance will not change without active changes in programming. The so-called "free lunch"[1] of automatically faster serial applications through faster microprocessors has ended. The "new free lunch" requires scalable parallel programming. The good news is that if we design a program for scalable parallelism, it can continue to scale as processors with more parallelism become available. Parallelism in hardware has been present since the earliest computers and reached a great deal of sophistication in mainframe and vector supercomputers by the late 1980's. For mainstream computation, miniaturization using integrated circuits started with designs that were largely devoid of hardware parallelism in the 1970s. Microprocessors emerged first using simple single threaded designs based on a very limited transistor budget. In 1971, the Intel 4004 4-bit microprocessor, designed to be used in an electronic calculator, was introduced. It used only 2,300 transistors in its design. The most recent Intel processors use over billions of transistors; a billion transistors would be enough for 434,782 Intel 4004 processors.

Hardware is naturally parallel since each transistor can switch independently. As transistor counts grew in accordance with Moore's Law, as shown in Figure 1, hardware parallelism, both implicit and explicit, gradually also appeared in microprocessors in many forms. Some variability in the number of transistors used for a processor can be seen in Figure 1, especially in recent years. Before multicore processors, different cache sizes were by far the driving factor in this variability. Today, cache size, number of cores, and optional core features (such as vector units) result in processors with a range of capabilities to be available. This is an additional factor that we need to take

[1] Herb Sutter. The free lunch is over: A fundamental turn towards concurrency in software. Dr. Dobbs Journal , March 2005.

into account when writing a program: even at a single point in time, it may need to run on processors with different numbers of cores, different SIMD instruction sets and vector widths, different cache sizes, and possibly different instruction latencies.

The amount of change, in software, needed for each kind of additional hardware mechanism using parallelism has grown because we naturally tackled the less intrusive (easier) first. Automatic mechanisms requiring the least software change, such as instruction-level parallelism (ILP), were generally introduced first. This worked well until several issues converged to force a shift to explicit rather than implicit mechanisms in the multicore era. As previously noted, the most significant of these issues was power.

Figure 3 shows a graph of total power consumption over time. After decades of steady increase in power consumption, the multicore era ushered in a halt to this growth. From this chart we can see that modern processors span a large range of power consumption, with lower consumption driven, in large part, by the growth of mobile and embedded computing.

The sudden rise in the number of hardware threads in the multicore era shows one aspect of the move toward explicit parallelism mechanisms. It is easy to argue that the use of multiple hardware threads requires more software changes than prior changes in hardware architecture. Another form of parallelism, vector parallelism, is growing in the multicore era as measured by the width of data operations as shown in Figure 5. While data width parallelism growth predates the halt in the growth of clock rates, the forces driving multicore parallelism growth are also adding motivation to increase data width. While some automatic parallelization (including vectorization) is possible, it has not been universally successful. Explicit parallel programming is generally needed to fully exploit both these forms of hardware parallelism capabilities.

Additional hardware parallelism will continue to be motivated by Moore's Law coupled with power constraints. This will lead to processor designs that are increasingly

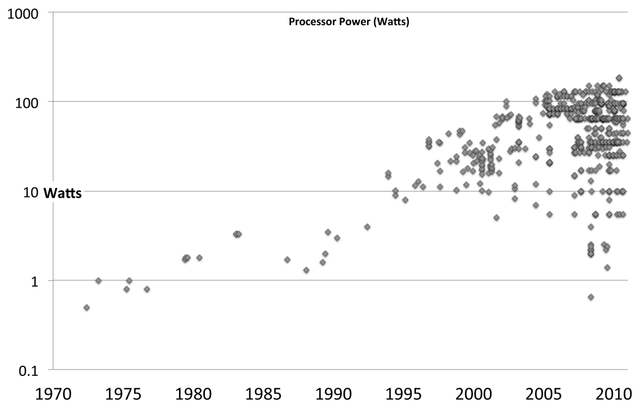


Fig. 3. Graph of processor total power consumption (log scale). The maximum power consumption of processors grew steadily for nearly two decades before the multicore era. This growth was brought to a halt by an interest in reduced power consumption, greater efficiencies, and mobile operation. We also simultaneously see more options at lower power points.

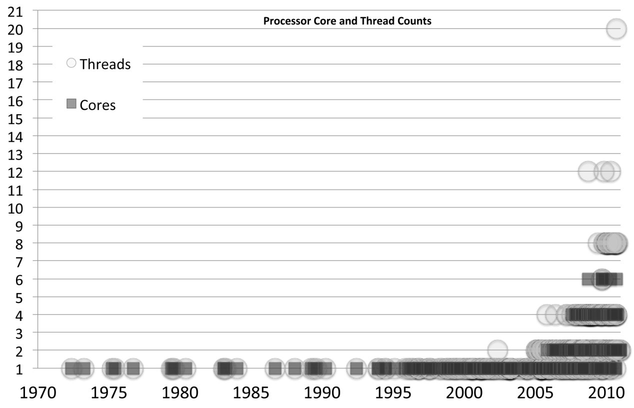


Fig. 4. Hardware threads emerged early as a trend in the multicore era

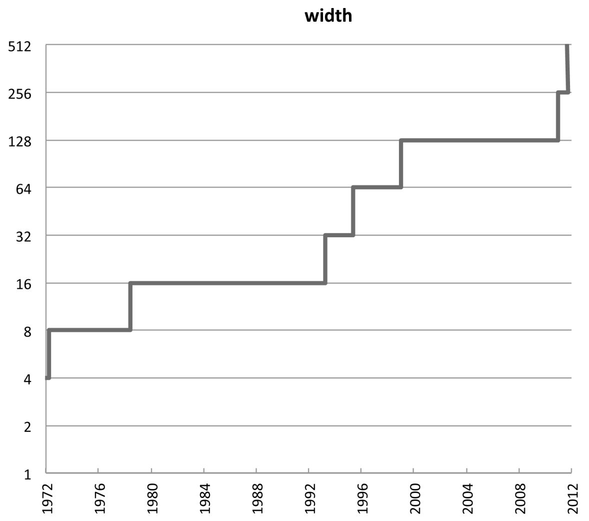


Fig. 5. Growth in data processing widths (log scale). At first the width of scalar elements grew, but now the number of elements in a register is growing with the addition of SIMD (vector) instructions that can specify the processing of multiple scalar elements at once.

complex and diverse. Proper abstraction of parallel programming methods are necessary to be able to deal with this diversity, and to deal with the fact that Moore's Law continues unabated for now, so the maximum number of cores (and the diversity of processors) will continue to increase.

3 Multicore Era Thus Far

None of the programming languages in widespread use were designed as parallel programming languages. All were created to serve a single-core world, and have many shortcomings and biases that interfere with parallel programming and offer little to express parallelism explicitly. It may surprise some to learn that Fortran has perhaps been the most aggressive at parallelism support with vectorizable array notations (Fortran90), FORALL (Fortran95), OpenMP (1997), DO CONCURRENT (Fortran 2008) and Coarray Fortran (Fortran 2008).

At the dawn of the multicore era, Intel introduced the Intel Threading Building Blocks (TBB) as a solution to extend C++ for parallelism. Well designed and implemented, it got a substantial boost from Intel's decision to open source it and from it being ported to numerous operating systems and processors. The TBB project has been widely used and is now the most popular abstraction for parallel programming.

There is considerable interest in extending TBB beyond what is possible in a C++ template library. Three key challenges that have emerged are: support for C programming, optimization support in compilers and support for explicit vectorization. To address these, Intel has created the Intel Cilk Plus project. Intel has implemented these very interesting extensions for C and C++ in the Intel compilers (Windows, Linux and Mac OS X versions). Intel has also published the full language specification, and the interface specifications to its runtime library and open sourced the runtime to help other compilers follow. There is work on a branch of gcc to explore adding these to gcc. Cilk Plus is a project (`http://cilkplus.org`) worth following as it develops.

Most recently, the new C++11 standard lays a solid foundation by addressing memory model issues that made parallelism in C++ suspect previously. This brings us to a point where it is time to debate how to formally extend our languages for parallelism. I summarized key thoughts in a blog "Parallelism as a First Class Citizen in C and C++, the time has come." (`http://software.intel.com/en-us/blogs/tag/citizenparallel/`) Many developers agree strongly with the idea that we are ready for this debate and standardization. It would represent a substantial step forward for programming in C and C++.

While you could argue that the multicore era is only getting started, there are signs that the rate of parallelism growth can grow very fast if we find uses for it. Multicore processors have gone from 2 to 4 to 8 highly programmable cores. Many-core processors promise to offer a much faster pace if we can use it. Intel has unveiled a new architecture: the Intel Many Integrated Core (MIC) Architecture. The early prototype (codenamed: Knights Ferry) using the MIC architecture has 32 cores. The product (codenamed: Knights Corner) will have more than 50 cores. Many-core processors give up serial performance in favor of a design with a higher degree of parallelism. Many-core processors are not designed to displace multicore processors, but rather offer a higher degree of parallelism specifically aimed at highly scalable workloads. With Moore's Law continuing to give us more and more transistors, today's octo-core multicore processors and 32 core many-core processors are just a start. The future is a lot of parallelism. Of course, cores is not the only dimension for parallelism. SSE has offered 128-bit wide data operations for a decade. Multicore processors from Intel offer 256-bit wide operations, called Advanced Vector Extensions (AVX), while the Knights Ferry many-core co-processor offers 512-bit wide operations. Again, these may just be starting points for the future.

Many-core processors from Intel highlight that more cores can be coupled with programmability while addressing the need for a greater ratio of performance per unit of power, higher system density, and high versatility.

Parallelism is driving the largest systems in the world to gain 80% more performance each year (see current graphs available on top500.org and updated each 6 month titled "Projected Performance Development"). This is twice the rate that you might expect from Moore's Law (which is about 40% gain per year). The difference between a 40% gain from more transistors and 80% gain in high performance computing comes from parallelism. Going parallel is happening like wildfire, making the need and opportunity of parallel programming ever greater.

4 Trends for the Future

There are four major trends that I would like to highlight to encourage thought and debate.

The future of *hardware* is *specialization and programmability*. Power considerations favor power efficient designs but at the expense of programmability. The most general-purpose designs offer high degrees of programmability. For instance, GPUs emerged as more power efficient ways to render graphics in visual displays. This they have succeeded in doing. Interest in using GPUs for non-graphical work has been vaulted into consideration as well. When reusing functionality, intended for graphics originally, the power efficiency may remain but the programmability is brought into question. Trying to fix programmability and stay power efficient are at odds with each other, especially if the original mission in graphics is also a requirement. The future is going to be more and more precise specialization, and the current interest in "highly parallel" workloads is going to be a very harsh critic of solutions. Solutions will be judged on programmability and power efficiency for "highly parallel" workloads alone. Being dual-purpose with graphics will not be enough if it means compromise. I believe this spells the end to GPU computing as we know it, because once we are looking for solutions for "highly parallel" workloads, we should expect the winning solution will have to maximize programmability and power efficiency. The market will decide what works best. My point, however, was not to predict the end of GPU computing. My point was to ask that we consider the success of the concept of a "GPU" for graphics procession as validating the concept of specialization, in this case for graphic programming. And then, consider "What does a specialized solution for "highly parallel" look like?" What other "specialized" solutions are in our future? I can imagine a future where processing chips have large number of transistors dedicated to functions that are highly efficient and programmable when used, and relatively low burden when not used. Of course, this has to be balanced with cost and design considerations. More transistors make this an important area to anticipate and study. Future hardware will see increased emphasis on specialization. "One size fits all" is under constant attack, and increases in transistor counts is definitely one such attack. Programmable but specialized hardware will become more common in the future.

The future of *system design* is *imbalance*. I decided to emphasize this only slightly in jest. It happened that someone recently complained about imbalance in computer systems to the point of suggesting they would rather have slower processors and balance than have processors that were faster than memory. The only reason anyone could agree with such a viewpoint would be the assumption that it must "cost something" to have processors faster than they would be in a system with "balance with memory." My thoughts on this are very different because I observe that the concept of "balance with memory" is a circular definition. "Balance" means that the capabilities of the memory system are sufficient to serve the demands of the processing on the CPU. If we believe once a program saturates the memory capabilities, that our job is done until we find more memory bandwidth then we are ignoring any additional CPU power that is waiting for us. New algorithms seem to be emerging when we take a different view. When an eye is put on what extra compute power can do, without adding burden to memory, interesting things happen. When more CPU power can be used for the same memory utilization, we find that one person's balance is another person's imbalance. One simple example: consider an application that is reading a data stream and processing it. Imagine it is memory bound. What if we reformulate the application to read a compressed data stream and output compressed data? The work of the application increases because of the decompression and compression work that is added. However, the amount of data processed may go up substantially if the processor has enough available compute power. Suddenly, we may have a use for the previously unused capability in this supposedly "unbalanced system." The only motivation for discovering such new methods comes in "unbalanced" systems. In the future, CPUs will keep driving "imbalance" and some uses of the "imbalance" will emerge thereby making "imbalance" appear to be less for some uses. We'll probably find that most applications will feel systems are "unbalanced" because they have not discovered use for all the compute power. Therefore, I propose it will be useful to a few, and an unused opportunity for many more. Hence, the future is "imbalanced" designs.

The future of *programming* is *parallelism*. Parallel hardware demands parallel programming. We should not expect that magically parallelizing compilers will ever appear. Nevertheless, not all programmers will not have to be parallel algorithm experts. Libraries and tools are emerging to support common patterns in solving problems in parallel. Most programmers can view parallel programming more as solutions they utilize than a deep area of study and focus. Parallel programming experts will be in demand, but not everyone will need to focus on parallelism.

The future of *computing* is *data processing using parallelism*. I like to say "the parallelism is in your data and not your code." That's a bit extreme, but it is the right place to look because data parallelism scales. Amdahl's Law is depressing without understanding Gustafson's Law. As long as problem sizes grow, we will see scaling. As a result, we are going to see more and more emphasis on learning, studying, using and supporting data parallel methods. An excellent place to start is making it more explicit and portable in our programming languages. Hence, my strong interest in adding array notations to C and C++ as we have in the Cilk Plus project.

5 The World Moves Fast

125 years ago, we had no airplanes, no radio, no TV, no computers, no cell phones, and no internet. In 1907, humans flew for the first time in a heavier than-air aircraft.

By 1962, humans were in space. By 1969, we were standing on the moon, and in 1976 we were seeing pictures from a spacecraft sitting on Mars. Imagine the person, like my grandfather, who saw all that – the dawn of radio, TV, automobiles, airplanes and spaceflight. What a change! Computers have advanced at a fantastic pace too, from vacuum tubes in the 1940s, to a million transistors in the late 1980s to billions today. What an amazing journey this was as well! The world is going faster now than ever. Today, starting with billions of transistors, the first decade of multicore era, the dawn of many-core processors... where will we steer our journey during the next several decades?

I hope you view this as a challenge. I do.

6 Summary

The future of computing is many things, and *parallel* is one of them. The ability to evolve to an all-parallel future combined with the need for parallel computing sets the stage for this evolution. I choose not to call it a revolution because it builds on so much of the past. Nevertheless, certain shackles in our serial-oriented past must become things of the past and disappear.

The future includes programmable uses for specialized hardware, adjusting to what we call imbalance today, parallel programming and widespread use of high degrees of data parallelism.

The world is moving fast, and parallelism is a key trend. We've really only seen the beginning of this evolution. It is an exciting challenge for us all.

A Dynamic Load Balancing Approach with SMPSuperscalar and MPI

Marta Garcia[1,2], Julita Corbalan[1,2], Rosa Maria Badia[1,3], and Jesus Labarta[1,2]

[1] Barcelona Supercomputing Center (BSC)
[2] Universitat Politecnica de Catalunya (UPC)
[3] Artificial Intelligence Research Institute (IIIA)
Spanish National Research Council (CSIC)

Abstract. In this paper we are going to compare the performance obtained with the hybrid programming models MPI+SMPSs and MPI+OpenMP, especially when executing with a Dynamic Load Balancing (DLB) library. But first we will describe the SMPSuperscalar programming model and how it hybridizes nicely with MPI. We are also explaining the load balancing algorithm for hybrid applications, LeWI, and how it can improve the performance of hybrid applications. We will also analyze why SMPSs is able to exploit the benefits of LeWI further than OpenMP. The performance results will show not only how the performance of hybrid applications can be improved with LeWI but also the benefit of using a hybrid programming model MPI+SMPSs for load balancing instead of MPI+OpenMP.

1 Introduction

In an HPC environment using a hybrid programming model is often a good approach to obtain a good performance when parallelizing an application. When we talk about hybrid programming models the first combination that comes to our mind is MPI+OpenMP.

The reason for MPI+OpenMP being the mostly used hybrid model is its success. Its success is not only because the performance obtained (which is the sum of the good performance obtained by the two programming models on their own). It is also because the flexibility it gives, being able to program clusters of shared memory nodes and the possibility to tackle the parallelization from different approaches, with more or less granularity, using shared memory or communication. Not only that but it has also shown how the use of OpenMP as the second level of parallelism can help load balance the MPI level [1].

In this paper we will talk about the load balancing library, DLB, and a balancing algorithm, LeWI, that can improve the performance of hybrid applications. DLB can load balance an application at runtime without modifying nor analyzing the application. In a previous work [2] we showed the potential of DLB and LeWI when executed with MPI+OpenMP applications. But we also found a limitation of OpenMP, that prevented the algorithm to obtain all the possible benefit. We are going to explain how using a hybrid programming model with SMPSuperscalar+MPI can overcome this limitation and obtain a better performance for most applications.

We will introduce the basics of SMPSuperscalar, a programming model aiming at shared memory systems. SMPSs is a programming model that has shown to obtain a performance comparable to OpenMP on its own [3]. Not only this but it can also hybridize with MPI offering a powerful hybrid model. We will explain how the MPI+SMPSs approach can help load balance applications with LeWI, and improve the performance obtained by the MPI+OpenMP version of the same application.

The paper is organized as follows: we will first explain the basics of the SMPSs programming model and the different techniques to combine it with MPI. In the next section we will introduce the DLB library and the LeWI algorithm and show how it can benefit from using MPI+SMPSs. In Section 4 we will present the performance evaluation comparing the performance of the MPI+OpenMP and MPI+SMPSs versions of an application with and without LeWI. Finally, we will conclude summing up the results presented in this paper and the future work.

2 Hybrid MPI+SMPSuperscalar Programming Model

2.1 SMPSuperscalar (SMPSs)

SMPSuperscalar is a task based programming model for shared memory systems. It was first released in 2007. A task is the basic parallel element. Each task will be executed by a thread and different tasks can run in parallel. The programmer should mark functions that can be executed as tasks (taskified) in the code with compiler directives and give all the parameters of the function (task) a directionality. The directionality of a parameter can be: *input*, *output* or *inout*. Each time a taskified function is called, a task is created and added to the task graph. The parallelism between tasks is controlled by tasks' dependencies. And dependencies will be computed at runtime based on the directionality of the parameters. The runtime environment will ensure that the dependencies are fulfilled when executing the tasks.

There are two kinds of code in an SMPSs application: the tasks and the serial code. The serial code is all the code that is outside a taskified function. SMPSs threads present a hierarchy. The master thread executes the serial code, creates the tasks and can execute tasks. The slave threads will only execute tasks.

In Figure 1 we can see an example of execution of an SMPSs application. In this example there is just one taskified function, *func_task*. The master thread starts executing some user code until it reaches the start directive for SMPSs (*css_start*), when the worker threads are created. At this point of the execution there are no tasks in the task graph yet so the worker threads are idle while the master thread continues executing the user code. When the master thread reaches a call to a taskified function it will create the task and add it to the task graph. The worker threads will get the tasks that are ready to run from the

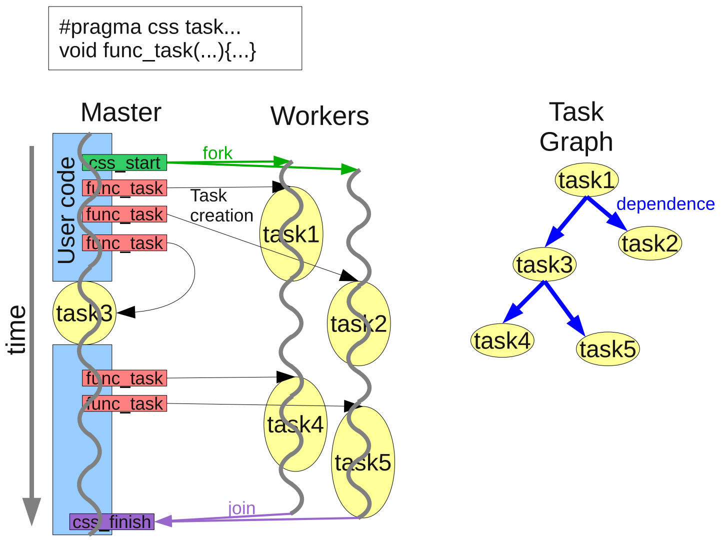


Fig. 1. SMPSs execution model

graph and execute them always respecting the dependencies. At some point of the execution the master thread can decide to execute a task (if it has reached a barrier or if there are too many of them in the graph). In the example the master thread executes task 3 after task 1 is finished. At the end of the execution the *css_finish* is called to join all the threads.

The number of threads can be changed at any time during the execution, the only limitation is that a thread will never leave a task before finishing it.

The SMPSs programming model does not need explicit synchronization between tasks, because the correctness of the execution is ensured by the dependencies. But it is important to notice that the serial code, that will be executed in a series by the master thread, can run in parallel with tasks executed by slave threads. To avoid race conditions between tasks and the serial code we need some synchronization mechanisms:

Barrier: All the tasks created before the barrier must be finished before the execution can proceed further in the serial code.

Wait on(*variable*): Creates a dependency with the given variable at this point of the code. Therefore, the execution can not proceed from this point until the dependencies for the given variable are fulfilled.

These are the basics of the programming model and the runtime. There are more features that we will not explain here, but you can find more details in the documentation [4].

2.2 Hybrid MPI+SMPSuperscalar

SMPSuperscalar hybridizes nicely with MPI. We can spawn MPI processes across nodes and exploit the node parallelism with SMPSs.

There are several ways of mixing MPI and SMPSs, the most obvious and easy one would be to have the MPI calls as serial SMPSs code. We must ensure that the data that we want to send or receive is ready with one of the synchronization mechanisms that SMPSs offers (*barrier, wait on*).

When using MPI+SMPSs we can also encapsulate MPI calls inside SMPSs tasks. This approach allows to overlap communication and computation. In this case the dependencies would be satisfied by the runtime if they were correctly set in the code as inputs and outputs of the task containing the MPI call. This approach presents some drawbacks that must be taken into account:

– **Possibility of several concurrent MPI calls:** we need to use thread-safe MPI
– **Reordering of MPI calls:** Can introduce a deadlock; we need to control the order of communication tasks.
– **Wasting a core while in a communication task.**

To avoid these limitations SMPSs includes a *Communication Thread*. The Communication Thread is a mechanism that allows to overlap computation and communication transparently for the programmer. The programmer must mark the tasks that include MPI communication calls with *device(comm_thread)* and the runtime will handle them. There is a special thread not counted among the general pool of threads that only executes communication tasks, and communication tasks can only be executed by the Communication Thread.

With this environment communication tasks are executed in order, therefore we ensure that there are no introduced deadlocks. And we do not need a thread-safe MPI because only one thread will be executing communication tasks.

3 Dynamic Load Balancing (DLB) Library

The Dynamic Load Balancing (DLB) is a shared library that helps load balance applications with two levels of parallelism. The current version provides support for:

– MPI+OpenMP
– MPI+SMPSs

The aim of DLB is to balance the MPI level using the malleability of the inner parallel level. One of its main properties is that the load balancing will be done at runtime without analyzing nor modifying the application previously. The algorithm that has showed better performance results is LeWI (Lend When Idle) [2]. And this is the algorithm that we are going to explain in the following section and use for the performance evaluation.

3.1 LeWI Algorithm

The philosophy of LeWI is based on the fact that when an MPI process is waiting in an MPI blocking call none of its threads is doing useful work. Therefore, we have one or several CPUs that are not being used. LeWI aims to use these CPUs to speedup other MPI processes running in the same node. The usual behavior of an MPI application is that if a process is blocked in an MPI call it is waiting for one or several other processes to finish. Speeding up processes that are more loaded helps to load balance the application and to speedup the whole application.

In Figure 2 we can see the behavior of the LeWI algorithm when balancing an unbalanced application. On the left hand side, Figure 2.a shows an unbalanced hybrid application with 2 MPI processes and 2 threads per process. In this example MPI process 1 is more loaded than MPI process 0 and this makes that MPI process 0 must wait in an *MPI_Send* for some time.

In the center, Figure 2.b shows the behavior of the same application when executed with the LeWI algorithm. When an MPI process reaches a blocking MPI call it will lend its CPUs to the other MPI processes running in the same node. With the lent CPUs the more loaded MPI processes will be able to finish their computation faster and the MPI process 0 will be less time waiting in the MPI call. The use of the computational resources will be better and the application will perform better.

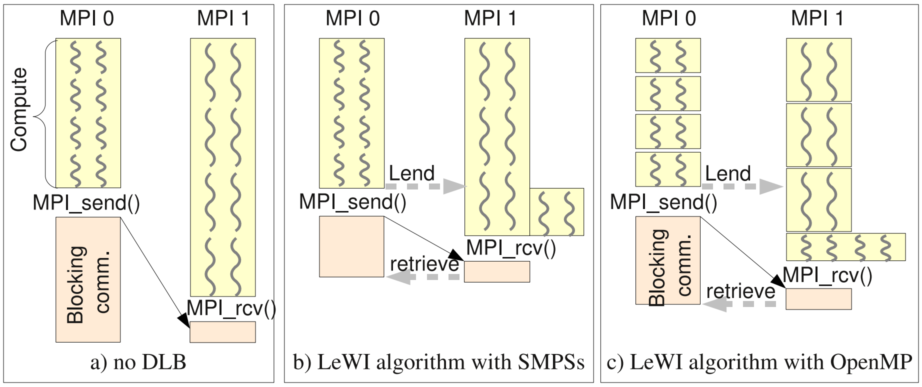


Fig. 2. LeWI behavior

When an MPI process that has lent its cores reaches the end of the blocking call it will retrieve the cores that it had before lending them. And it will be able to continue its computation with its threads.

3.2 SMPSs Potential for Load Balancing

The limitation that we detected in this algorithm is the fact that OpenMP can only change the number of threads outside a parallel region. This means that when an MPI process lends its cores the MPI process that wants to use them is not able to do so until reaching a new parallel region (i.e. the number of OpenMP threads can only be changed before spawning a parallel region). This is shown in Figure 2.c; when MPI 0 lends its cores to MPI 1, MPI process 1 is already executing its third parallel loop. Therefore, MPI 1 is not able to use the lent cores until the fourth loop starts.

This limitation makes the performance of the algorithm highly dependent on the number of parallel regions that the application presents between MPI blocking calls (i.e. if there is just one parallel loop between MPI blocking calls we cannot change the number of threads. Therefore, the application cannot be balanced and the performance cannot be improved).

This limitation was not inherent to the algorithm but introduced by the programming model used (in this case OpenMP). To overcome this limitation we chose a shared memory programming model that allowed us to change the number of threads at any time, such as SMPSuperscalar.

In Figure 2 we show the difference between using OpenMP or SMPSs when running with LeWI algorithm and how it can impact the performance. Figure 2.c shows the limitation in OpenMP that cannot start to use the threads until it reaches a new parallel region. In Figure 2.b we can see how SMPSs can start to use the new threads as soon as they are available. This example shows us how the performance can be improved by using SMPSs instead of OpenMP.

4 Performance Evaluation

4.1 Environment

The experiments have been executed on Marenostrum. Marenostrum has 10240 PowerPC processors. Its nodes are JS21 blades with two IBM PowerPC 970MP processors with two cores each and 8Gb of shared memory. This means that we have nodes of 4 cores with shared memory.

We have used the MPICH library as the underlying MPI runtime and the operating system is a Linux 2.6.5-7.244-pseries64. The OpenMP compiler used is IBM XL version 10.1 and SMPSs version used is 2.0.

4.2 Methodology

In this section we will use the speedup to compare the performance of each experiment. The speedup has been computed as the serial time divided by the parallel execution time.

The serial time used to compute the speedup is the execution time of the MPI only version of the application executed with a single MPI process. We are using the MPI version with one MPI process and not the serial version of the application because we want to focus on the performance obtained by the inner programming model, in our case OpenMP or SMPSs. By using this baseline we exclude the overhead introduced by the MPI runtime from the computation of the speedup.

In the following charts we will be comparing several configurations for each application. Their meaning is as follows:

OMP - ORIG: Execution of the original MPI+OpenMP application without any load balancing.

OMP - LeWI: Execution of the MPI+OpenMP application with DLB and LeWI algorithm.

SMPSs - ORIG: Execution of the original MPI+SMPSs application without any load balancing.

SMPSs - LeWI: Execution of the MPI+SMPSs application with DLB and LeWI algorithm.

All the executions have been run in Marenostrum (4 cores per node). We will present different configurations, running with 2 MPI processes per node or 4 MPI processes per node. Although it is not usual to run more than one MPI process in the same node sometimes it is done because the application is MPI only or for performance reasons. In our case it is necessary because we balance between MPI processes running in the same node. Furthermore, we showed in a previous work that running several MPI processes in the same node could improve the performance of applications when combined with the DLB library.

In the following sections we will present the results obtained with different applications.

4.3 PILS

PILS is a synthetic benchmark that we have developed to help us evaluate load balancing techniques. What we aim to reproduce with PILS is not a whole application but the parallel region between MPI calls of an application.

The core of the synthetic benchmark is a function that will do several floating point operations without data involved. In the case of the OpenMP version this function will be called in each iteration. In the case of SMPSs version this function will be taskified, meaning that each call to this function is a task. The cost of the core function in terms of time and computation is always the same. The imbalance between MPI processes will be introduced by the number of times that the function is executed which will be given by the loads of each MPI process.

In Figure 3 we can see a schematic representation of PILS. The amount of work is prescribed at the beginning of the execution for each MPI process. This work load is processed in the parallel regions. The number of parallel regions depends on the parallel grain parameter. A parallel region in the OpenMP version corresponds to a parallel loop. In the SMPSs version a parallel region is a loop that creates several tasks and finishes with an SMPSs barrier. At the end of the iteration there is an MPI barrier to synchronize all the processes.

The PILS benchmark has several configurable parameters that allow us to reproduce the behavior of different types of applications. The different parameters are the following:

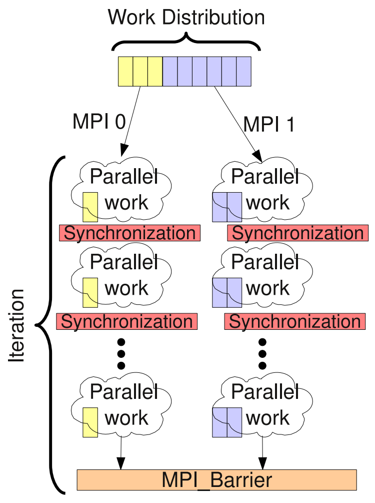


Fig. 3. PILS benchmark

Parallel grain	Parallel regions
1	1
0,5	2
0,1	10

Fig. 4. Parallelism grain and parallel regions

Programming Model: We can compile three different versions of the benchmark:
- MPI
- MPI+OpenMP
- MPI+SMPSs

Work Distribution: We can introduce the load balance of the application as the different loads for each MPI process. For all the executions the sum of the loads will be the same (giving always the same amount of work to the application).

Parallelism Grain: Represents the amount of computation that is parallel for the iteration. The value can go from 1 (everything is parallel) to 0 (nothing is parallel). You can see the relationship between parallelism grain and the number of parallel regions in Figure 4.

Iterations: The number of iterations that we will execute. Each iteration includes the computational part (parallel regions in OpenMP or SMPSs) and MPI communication.

In the following experiments we have executed PILS with the parameter *iterations* always equal to 1. We want to compare the performance of the different programming models with different works loads and parallelism grain.

In Figure 5 we can see the speedup obtained for PILS with the 4 different versions when running with 2 MPI processes in the same node. We executed with five configurations for the work load that go from a very unbalanced application (10-90) to a well balanced application (50-50) and for each of the imbalanced configurations we used 10 different values for the parallelism grain rangingfrom 1 to 0,1.

We can see that the executions without LeWI (SMPSs-ORIG and OMP-ORIG) have the same performance, this means that the baseline for both are the same. But we can see the difference when running with LeWI (SMPSs-LeWI and OMP-LeWI). While SMPSs obtains almost the ideal speedup of 4 for all the executions,

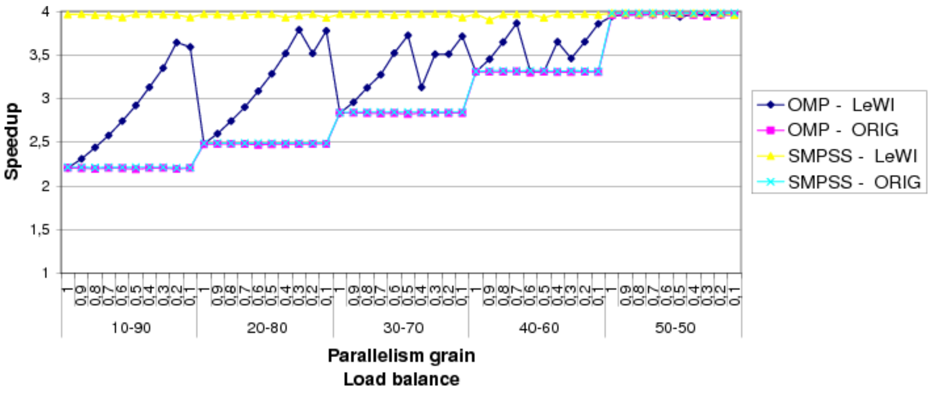


Fig. 5. PILS 2 MPIs per node in Marenostrum

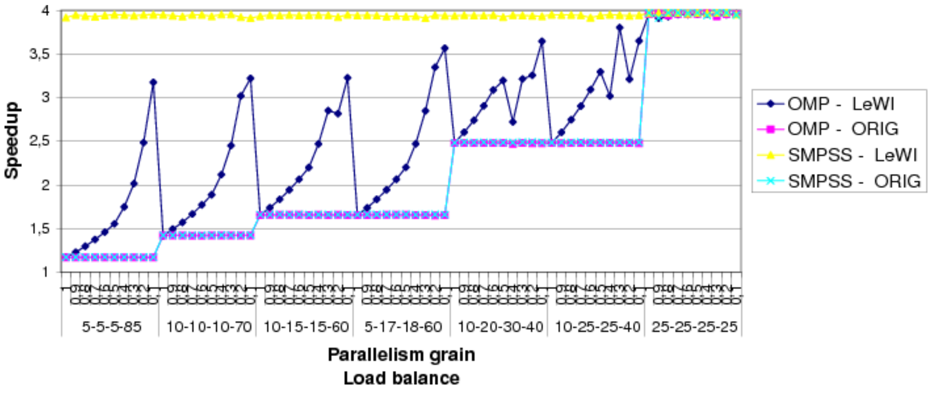


Fig. 6. PILS 4 MPIs per node in Marenostrum

the performance of the version with LeWI and OpenMP depends on the parallelism grain (or equivalent, on the number of parallel regions between MPI calls).

Figure 6 shows the performance of PILS when running with 4 MPI processes in the same node. In this case we can see 7 different configurations of work loads with different levels of imbalance between the MPI processes. Again, the original executions of the two models without LeWI have the same performance but when running with LeWI we can see how SMPSs performs much better. While SMPSs with LeWI can obtain an almost ideal speedup for all the configurations, OpenMP depends on the parallelism grain and almost never can reach the same performance as SMPSs.

4.4 LUB

LUB is a kernel performing an LU matrix factorization. The structure is a two dimensional matrix organized by blocks. The data is distributed by blocks of

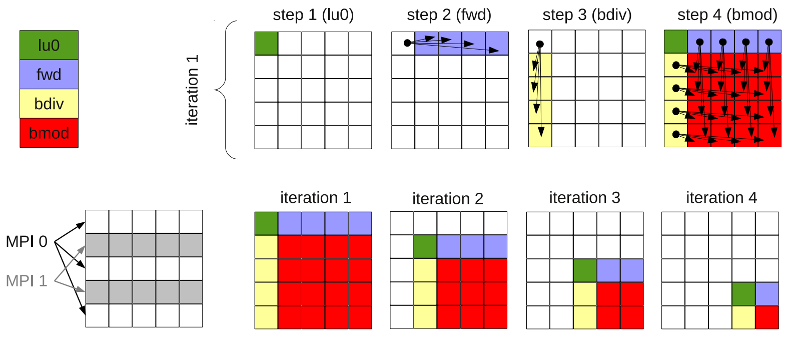


Fig. 7. LUB behavior

rows among the different MPI processes. In Figure 7 we can see a schematic representation of the LUB kernel. There are four functions that are applied to the data blocks, *lu0*, *fwd*, *bdiv* and *bmod*. The *fwd* blocks can be processed in parallel but depend on the lu0 computation. The *bdiv* blocks can be processed in parallel but depend on the *lu0* block. And the *bmod* blocks can be processed in parallel but each one depends on the *fwd* block in the same column and the *bdiv* block in its row.

We have two OpenMP parallelizations for this application. The first one (labeled as *OMP1* in the charts) is the "natural" one that parallelizes the outer possible level. But this parallelization is not the optimal for LeWI because it presents few loops between MPI calls. So it gives low malleability to change the number of threads. The second OpenMP parallelization (labeled as OMP2) parallelizes inner loops of the application giving more malleability to LeWI to change the number of threads. We can say that *OMP2* is a parallelization tuned to obtain the best of LeWI but that also introduces some overhead.

The SMPSs parallelization considers each algorithm applied to a block as a task which is the natural way of parallelizing this application with a task oriented programming model.

In the experiments we have worked with a matrix of 5000 x 5000 elements with block size of 200 x 200 elements.

In Figure 8 we can see the performance obtained when running LUB in a single node of Marenostrum (4 cores). We have executed with 2 or 4 MPI processes in the same node. We can see how the 3 original versions (*OMP1*, *OMP2* and *SMPSs*) have a similar performance, and in general all of them have a bad speedup when running with 4 MPI processes per node. The reason for this is that the imbalance of the application grows linearly with the number of MPI processes. But in all the cases the best performance close to the ideal one is obtained by the SMPSs version when running with LeWI. Not even the modified OpenMP version (*OMP2*) implemented to obtain the most out of LeWI is able to get the same performance as SMPSs with LeWI.

It is important to notice that with the OpenMP version *OMP1* LeWI is able to increase the speedup of the application from 2.8 to 3.5. This means executing the

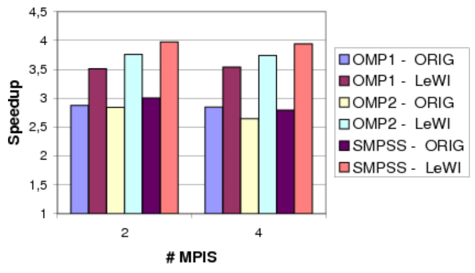


Fig. 8. LUB in 1 node in Marenostrum

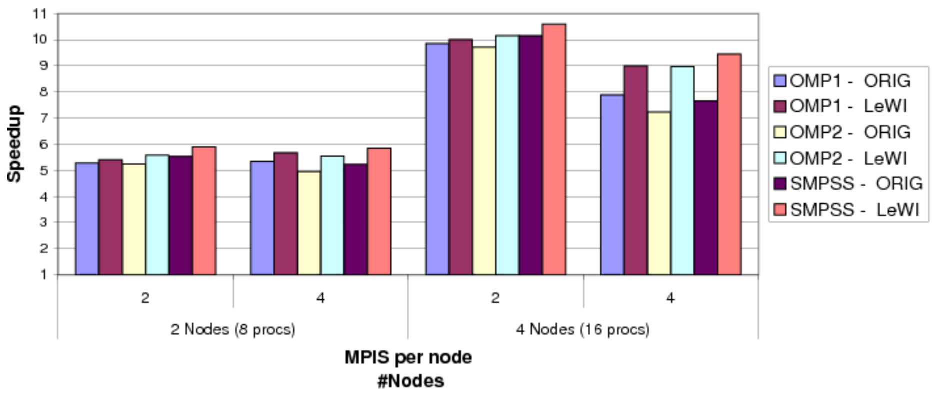


Fig. 9. LUB in 2 and 4 nodes in Marenostrum

application without modifying it nor analyzing it previously. The tuned version *OMP2* was implemented to give a chance to the OpenMP version compared to the SMPSs version.

The performance of LUB executed on 2 and 4 nodes of Marenostrum can be seen in Figure 9. We have also executed LUB with 2 and 4 MPI processes per node in each version. In this case we can see how the performance obtained is not so close to the ideal one. The two reasons for this performance drop are the following: first, the more MPI processes are running the more imbalance is presented by the application. Second, LeWI can only balance between MPI processes running in the same node. This means that when the imbalance is present between processes running in different nodes LeWI can not improve the performance. But still the version SMPSs with LeWI is the one that obtains the best speedup, even better than the OpenMP version modified to run with LeWI (*OMP2*).

4.5 BT-MZ (NAS Benchmarks)

The BT application is one of the benchmarks in the NAS Multizone suite [5]. The original version is a hybrid parallelization of MPI+OpenMP that we will see in the charts as *OMP1*. We have modified this benchmark to be parallelized with SMPSs (labeled *SMPSs*), and we have also modified the original MPI+OpenMP version because in some parts of the code it presented several loops inside a single parallel region. This prevents LeWI to change th number of threads often enough. We have labeled this tuned version for LeWI *OMP2*.

The NAS benchmarks can run different classes that correspond to the size of the problem that it is solving. In our experiments we have executed classes A, B and C (with the following relationship of size: $A < B < C$).

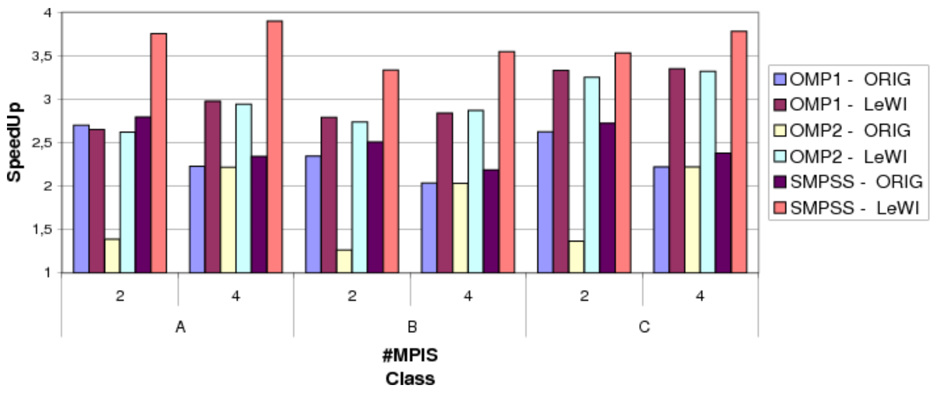


Fig. 10. BT-MZ in 1 node in Marenostrum

In Figure 10 we can see the performance obtained with BT-MZ when running in a single node in Marenostrum 3 different classes, A, B and C. In this case we see a big difference between the performance of the original OpenMP version (*OMP1*) and the tuned version for LeWI (*OMP2*). The reason of this difference is that the granularity of the loops is too small for *OMP2* and some data locality is lost. When running with 4 MPI processes in the same node we do not see this effect because the OpenMP level is only used to load balance, therefore, the performance is the same for both versions (*OMP1* and *OMP2*).

Looking at the executions with LeWI we see that in all the cases it improves the performance of the original application. But the best performance is always obtained by the SMPSs version with LeWI being close to the ideal performance in some cases.

In Figure 11 we can see the speedup obtained by the different executions of BT-MZ class C in 2 and 4 nodes. Again, the versions with LeWI improve the performance of the original executions. We can notice that the improvement in the performance when running in several nodes is less than when running in a single node. The reason is that LeWI can only balance MPI processes running in the same node.

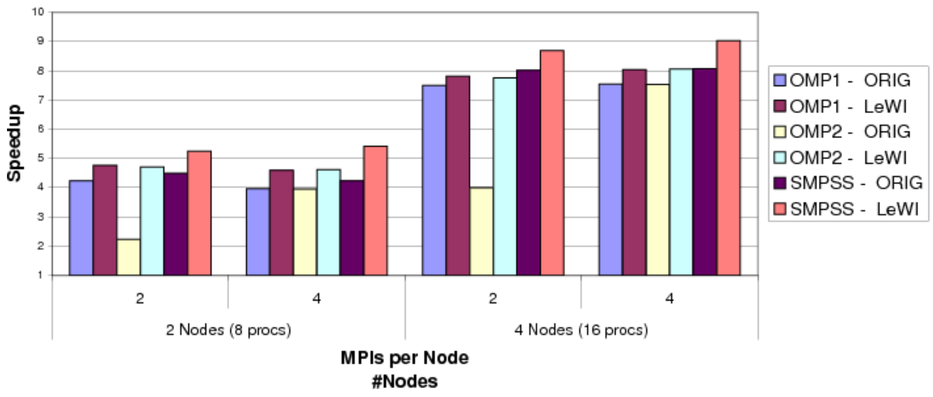


Fig. 11. BT-MZ class C in 2 and 4 Nodes in Marenostrum

5 Conclusions

In this paper we have explained the basics of SMPSuperscalar. SMPSuperscalar is a programming model for shared memory systems that hybridizes nicely with MPI. We have explained why MPI+SMPSs is a powerful hybrid programming model and we have underlined this observation in the performance evaluation section. We have also presented a load balancing algorithm for hybrid applications called LeWI. The LeWI algorithm improves the load balance and performance of hybrid applications. To achieve this it redistributes the computational resources assigned to the application between the different MPI processes of the application.

The current version of LeWI supports MPI+OpenMP and MPI+SMPSs applications. We have detected and explained a limitation of the algorithm when load balancing MPI+OpenMP applications related to the malleability of OpenMP. And we have seen that this limitation can be avoided when using SMPSs instead of OpenMP. In the performance evaluation section we have executed three different applications with different imbalance patterns. For all the applications we have an MPI+OpenMP version and an MPI+SMPSs version. In the performance results obtained we have seen that SMPSs can achieve the same performance as OpenMP when combined with MPI in all the applications we have tested.

We have also shown how the LeWI algorithm improves the performance of all the applications executed and does not penalize the performance of balanced applications. And we can use it without analyzing nor modifying the original application. But the most interesting remark from the performance evaluation is that for all the applications the best performance is obtained with the MPI+SMPSs version of the application running with LeWI. The malleability of SMPSs allows LeWI to improve the performance of unbalanced applications and obtains a higher speedup than the same application with MPI+OpenMP and LeWI.

Acknowledgments. We thankfully acknowledge the support of the European Commission through the TEXT project (FP7-261580), the support of the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980).

References

1. Henty, D.S.: Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (2000)
2. Garcia, M., Corbalan, J., Labarta, J.: LeWI: A runtime balancing algorithm for nested parallelism. In: International Conference on Parallel Processing, ICPP 2009 (2009)
3. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: 2008 IEEE International Conference on Cluster Computing (2008)
4. SMPSuperscalar official site and documentation, `http://www.bsc.es/smpsuperscalar`
5. Van der Wijngaart, R., Jin, H.: NAS parallel benchmarks, multi-zone versions. Technical report, NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA (2003)

Performance and Productivity of New Programming Languages[⋆]

Iris Christadler[1], Giovanni Erbacci[2], and Alan D. Simpson[3]

[1] Leibniz Supercomputing Centre, Garching, Germany
christadler@lrz.de
[2] CINECA Supercomputing Centre, Bologna, Italy
g.erbacci@cineca.it
[3] EPCC, The University of Edinburgh, United Kingdom
a.simpson@epcc.ed.ac.uk

Abstract. Will HPC programmers (have to) adapt to new programming languages and parallelization concepts? Many different languages are currently discussed as complements or successors to the traditional HPC programming paradigm (Fortran/C+MPI). These include both languages designed specifically for the HPC community (e.g. the partitioned global address space (PGAS) languages UPC, CAF, X10 or Chapel) and languages that allow the use of hardware accelerators (e.g. Cn for ClearSpeed accelerator boards, CellSs for IBM CELL and GPGPU languages like CUDA, OpenCL, CAPS hmpp and RapidMind).

During the project "Partnership for Advanced Computing in Europe – Preparatory Phase" (PRACE-PP), developers across Europe have ported three benchmarks to more than 12 different programming languages and assessed both performance and productivity. Their results will help scientific groups to choose the optimal combination of language and hardware to efficiently tackle their scientific problems. This paper describes the framework used for this assessment and the results gathered during the study together with guidelines for interpretation.

1 Introduction

A few developments have re-animated the discussion on HPC programming models. First, the physical limits of performance-gains through an increase in clock

[⋆] This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP/2007/2013) under grant agreement no. RI-211528 and RI-261557. We specially thank the following individuals for their time and effort to port the Euroben kernels and for reporting their experiences: Guillaume Colin de Verdière, CEA, France (CAPS hmpp); Maciej Cytowski , PSNC, Poland (CellSs); Jose Gracia, HLRS, Germany (Chapel); Hans Hacker, LRZ, Gernany (CUDA, MKL); Olli-Pekka Lehto, CSC, Finland (OpenCL, CUDA+MPI); Walter Lioen, SARA, Netherlands (X10); James Perry, EPCC, United Kingdom (VHDL, HCE); Sami Saarinen, CSC, Finland (UPC, CAF); Ramnath Sai Sagar, BSC, Spain (CellSs); Filippo Spiga, CINECA, Italy (OpenMP+MPI); Aad van der Steen, NCF, Netherlands (C, Fortran, MPI) and Volker Weinberg, LRZ, Germany (RapidMind).

speeds have been reached. Performance improvements now require the exploitation of the increased on-chip and off-chip parallelism. Second, hardware accelerators promise high performance, low energy consumption and/or smaller footprints but require device-specific languages. Third, the fastest HPC installations are now composed of hundreds of thousands of cores. Handling this level of scalability goes beyond the limits of most MPI codes.

High Performance Computing (HPC) is at a crossroads. New programming languages have not gained widespread acceptance; High Performance Fortran (HPF) being probably the best-known example. But HPC is in need for new programming and parallelization models to keep pace with architecture developments. Many different candidate languages are available, but it is very unclear which of these languages are mature enough, provide sufficient ease-of-use and high performance. To guarantee portability and sustainability, it is important to check the promises made for each language and ensure that the number of possible languages is narrowed down to a sensible set which could be widely supported by hardware, compiler and tool developers.

PRACE, the Partnership for Advanced Computing in Europe (PRACE) is an international non-profit association with its headquarters in Brussels. The PRACE Research Infrastructure provides a persistent world-class High Performance Computing service for scientists and researchers from academia and industry. The PRACE leadership systems form the apex of the performance pyramid and are well integrated into the European HPC ecosystem. On the road to enable software for the Petascale era, the PRACE-PP software work package has investigated the European HPC workload [1], chosen an application benchmark suite [2], optimized those codes [3] and improved their scalability [4]. In parallel, more than 12 new languages and paradigms have been chosen for a deep investigation of performance and productivity. Here, productivity was considered to be a combination of ease-of-use and achievable performance, a full definition can be found in [5]. The intention of the PRACE report was to give assistance to those scientific groups who are considering porting their codes to new languages or hardware accelerators; the objective was to check whether the promises of these languages could be trusted. The full results of this study are reported in [6]. The most important findings of the study are presented in this paper.

Three mathematical kernels, a dense matrix-matrix multiplication, a sparse matrix-vector multiplication and a one-dimensional Fast Fourier Transformation, have been chosen as synthetic benchmarks. The selection of those follows the "dwarves taxonomy" introduced in [7]. We describe the experimental setup in Section 2 and give an overview of the basic results in Section 3. Conclusions are given in Section 4 and a proposal how the work could be expanded to form a benchmark framework is given in Section 5. A benchmark suite which is able to handle both different languages and different hardware architectures is becoming more important. A first implementation for a GPU/CPU benchmark is given in [8] and has been discussed during a BoF session at SC09. The "PRACE First – Implementation Project" (PRACE-1IP) is currently working on an extended language benchmark suite.

2 Experimental Setup

This study was started with the selection of benchmark kernels. The selection was based on the taxonomy of the seven dwarves, which has been repeatedly used in PRACE for the classification of applications. To be able to cover as many languages as possible, three relatively easy and well-known benchmarks were chosen from the Euroben [9] benchmark suite. This synthetic benchmark suite had already been used for the assessment of the PRACE-PP hardware prototypes. The three kernels are:

1. MxM: Dense matrix-matrix multiplication.
2. SpMV: Sparse (CSR) matrix-vector multiplication.
3. FFT: 1-dimensional Fast Fourier Transformation.

Serial and parallel C and Fortran versions existed for all kernels. A version based on Intel's MKL (Math Kernel Library, [10]) was used as reference implementation and its performance on an 8-core Nehalem-EP board was used as baseline for performance comparisons. The developers involved could choose from a list of programming languages which included the PGAS (Partitioned Global Address Space) languages UPC (Unified Parallel C, [11]) and CAF (Coarray Fortran, [12]), the DARPA HPCS (High Productivity Computing Systems, [13]) languages (Chapel [14] and X10 [15]) and several low-level and high-level languages for hardware accelerators (Cn [16] for ClearSpeed accelerators, CellSs [17] for IBM CELL, CUDA [18], OpenCL [19], CAPS hmpp [20] and RapidMind [21] for Nvidia Tesla). Additionally, a mixed MPI+OpenMP port and a mixed MPI+CUDA port were done, to test the effectiveness of these hierarchical parallelization concepts.

The wide variety of languages was necessary to cover the most significant paradigms that introduce different ways of treating and abstracting parallelism. The languages were chosen mid of 2009, implementations were due approximately four months later, performance figures were then updated for a workshop in March 2010 [22]. Newer figures on the extended benchmark suite are currently prepared and should be available early 2012, first intermediate results have been reported in the PRACE-1IP deliverable [23]. Several facts show how fast evolving this field is; for all languages new compiler versions exists, several languages moved out of focus because of lacking hardware developments and one language (Rapidmind) is no longer available but its main concept and core development team has been included in Intel ArBB [24]. Table 1 gives an overview of the compiler versions used for the data presented in this study.

The most severe hurdle for the study was the fact that the languages were designed for quite different hardware architectures. There was neither a language which could be used on all different hardware, nor a hardware which could be used as a reference for all implementations. The selection of kernels to benchmark all languages was problematic as well, since some codes fit very well to some devices while others don't. We expectet that hardware accelerators could show off their potential wit MxM, a kernel with a high computational intensity,

Table 1. Compiler versions and mapping between hardware and languages

Language	Compiler version	1. MxM	2. SpMV	3. FFT
CAF	cce 7.1.2	cray xt	n.a.	cray xt
CAPS	CAPS hmpp v2.0.0	tesla	tesla	n.a.
CellSs	CellSs V2.2	cell	cell	cell
Chapel	Chapel V0.9	cray xt	cray xt	n.a.
Cn	Cn 3.11	clearspeed	clearspeed	clearspeed
CUDA	CUDA 2.2, V0.2.1221	tesla	tesla	tesla
CUDA+MPI	CUDA 2.2, gcc 4.1.2, OpenMPI 1.3.2	tesla	tesla	n.a.
MPI+OpenMP	Intel Version 10, OpenMPI 1.3.2	nehalem	nehalem	n.a.
OpenCL	CUDA 2.2, OpenCL 1.0 conf. release	tesla	n.a.	n.a.
RapidMind	RapidMind 4.0	tesla	tesla	n.a.
UPC	Berkeley UPC 2.8.0	itanium	itanium	itanium
X10	X10 v1.7.5	ibm pwr6	ibm pwr6	n.a.

but will perform dramatically worse on SpMV, a memory-bound kernel with a low computational intensity. The FFT kernel is somewhere between these two extremes and can potentially run well on accelerators but requires some optimization effort. Several PRACE-PP "Future Technology" prototypes were used for benchmarking. Final results of these prototypes can be found in [25]. An overview of the different hardware is given in Table 2; the mapping between languages and hardware can be found in Table 1.

We chose double-precision arithmetic and compared the whole time of the main mathematical routine, which included the time for data movements to and from accelerator devices. Not all languages supported parallel execution; e.g. most accelerator languages do not support multiple devices. The reference input data sets were chosen to reflect relevant matrix sizes or vector lengths for one board. They were also adapted to include both advantageous and disadvantageous data sets for the different hardware (e.g. for MxM matrix sizes which are a multiple of 512 will perform best on Tesla GPUs).

To measure productivity, the programmers were asked to fill in "developer diaries", which have been specially developed for this task but are inspired by the work carried out in the DARPA-HPCS project, see [26] especially [27] and [28].

Table 2. Hardware overview

Name	System	Processor	Peak perf. [GFlop/s]	Peak perf. for comp. (Fig. 5)
cell	QS22-blade cluster	PowerXCell8i(1PPC+8SPE)	102.4 per acc.	102.4 (1PXC8i)
clearspeed	CATS units	CSX700 (96PE)	96.0 per acc.	96.0 (1CSX700)
cray xt	Cray XT5	AMD Barcelona (4 cores)	9.2 per core	73.6 (8 cores)
itanium	SGI Altix	Itanium Montecito (2 cores)	6.4 per core	102.4 (16 cores)
ibm pwr6	IBM Power6 cluster	Power6 (2 cores)	18.8 per core	601.6 (32 cores)
nehalem	Intel processor	Nehalem EP (4 cores)	10.1 per core	80.8 (8 cores)
tesla	Nvidia Tesla S1070	C1060 GPU (240SP)	78.0 per acc.	78.0 (1C1060)

The diaries contained the development time needed and the performance obtained with a few data-sets for each development or optimization step. The information given in the diaries allows much insight into the problems encountered with the language, it shows the first time when an error-free version of the code was running, and the optimization potential and effort which lead from the first version to the finally submitted version. All submitted versions were kept in a subversion directory along with the performance results. After submission, programmers were asked to fill in a survey where they could briefly report their personal experience.

3 Results

Ideally, to obtain statistically significant results, a set of developers with equal level of knowledge for all languages and a thorough understanding of the three mathematical kernels would be needed. Furthermore, each developer would need to port the kernels to several languages to compute mean times of development or the mean lines of code. This is a task which requires enormous time and resources and might nevertheless fail since the level of maturity varies extremely between the languages (ranging from implementations on which some time had to be spent in finding compiler-bugs to well established products). The results of such a study would be partially invalidated by every new compiler release. In a field which evolves as fast as hardware-accelerator languages do today, we believe that our approach, which relies on only one programmer per language, is the only feasible approach that could lead to a separation of the promising languages from the immature ones. Since all kernels are easy to understand, all programmers have prior HPC knowledge and everyone had no or very little experience with the language, we believe that the study is able to give an insight into the current state of these languages and reveal remarkable tendencies between them.

Figure 1, 2 and 3 show the raw results of this study. All programmers were asked to start with the MxM kernel and, if time remains, subsequently port the SpMV and FFT kernel. Since all languages have different formal requirements, LoC (Lines of Code) was reported by the programmers themselves. The 3000 lines which were necessary for the CUDA FFT implementation show clearly how many lines of code are necessary if a kernel cannot simply use the CUDA libraries: a version based on a cuFFT library call contained only 75 LoC, but was restricted to single-precision arithmetic.

Figure 2 reports the development time in days and relies on the information given by the programmer in the developer diaries. Besides the total time given from the diaries it reports the time necessary for a first working version if this could be deduced from the diaries. Unfortunately, no estimate on the development time for the Cn ports exists because those were done by ClearSpeed-Petapath staff outside of PRACE.

Figure 3 gives an overview of the achieved performance. Maximum performance means the maximum performance which could be obtained for one of the approximately 20 different reference input data sets for each kernel. To allow

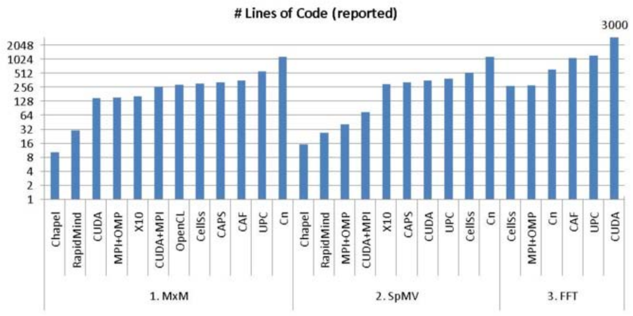


Fig. 1. Lines of Code as reported by the programmers

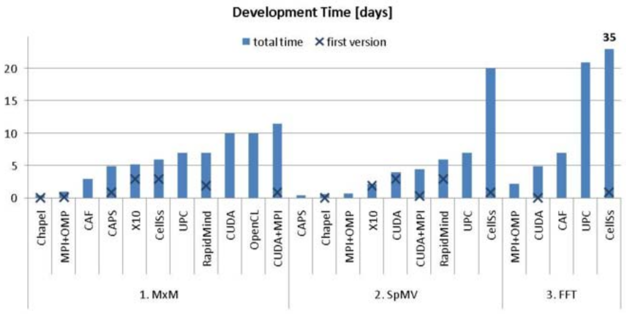


Fig. 2. Development time in days as reported by the programmers in their diaries

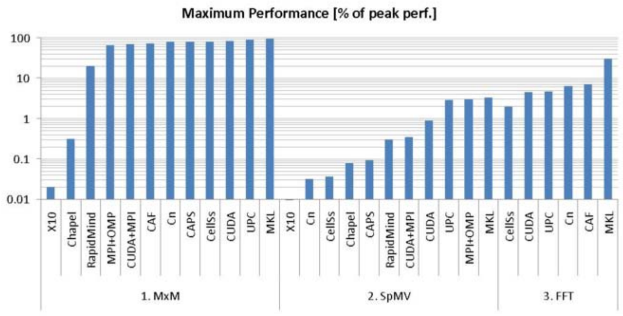


Fig. 3. Maximum Performance per kernel and language in percentage of peak performance. Maximum performance is achieved by one of the reference input data sets and usually based on single-core or single-accelerator measurements.

a comparison between the different languages on different hardware, the percentage of peak performance was used. This was usually based on single-core or single-accelerator runs (see second column from the right in Table 2). Many implementations made use of highly optimized mathematical libraries; especially in the case of MxM the ability to use those seems to be the most decisive performance factor.

Figure 4 shows a combination of the two productivity metrics development time versus lines of code. It confirms that the relation between both metrics is quite similar between most languages and the pattern stays roughly the same throughout all kernels. Chapel is the only language which positively sticks out; Chapel is very clear and concise and therefore allows fast programming with only a few lines of code, which minimises programming errors.

Finally, Figure 5 gives a comprehensive view on the productivity achieved with each language per kernel. This time, performance measurements are based on raw performance measured. We tried to keep the peak performance of the units as comparable as possible; an overview is given in the rightmost column of Table 2. It ranges from 73.6 GFlop/s for 8 AMD Barcelona cores up to 102 GFlop/s for a PowerXCell8i. For X10, data was available only per node, which has more than 600 GFlop/s peak performance. However, the performance of X10 was so low that the overall scale is not affected. Using a vertical line between the blue bars for one kernel will separate the languages which are not yet able to deliver enough performance to be used in HPC from those that are fast enough. Inserting an additional horizontal line will separate the languages which took too much development time from those with an acceptable development effort. This idea is illustrated by the dotted separation lines for performance in light blue and for productivity in dark blue. Please note that their placement is arbitrarily, it usually depends on the actual project, e.g. on the available man power or computing ressources.

Figure 6, 7 and 8 give a separate overview of the PGAS, GPGPU and accelerator languages to allow fast performance comparisons. Performance measurements for the PGAS languages have been performed on different architectures (Cray XT, Intel Itanium and IBM POWER6); only CAF and Chapel results are directly comparable. The percentage of peak performance was therefore used as the basis for the comparison. The overview of GPGPU languages was easier: all performance measurements have been done on the same Nvidia Tesla machine and are therefore directly comparable. The overview of accelerator languages is based on the fastest implementations available for each accelerator and directly compares results for one C1060, one PowerXCell8i, one ClearSpeed CSX700 with the performance achieved on one Nehalem-EP board (2 sockets with 4 cores each).

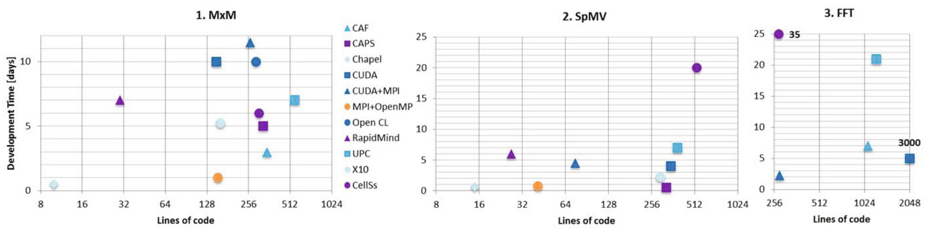


Fig. 4. Development time versus lines of code for all languages and kernels

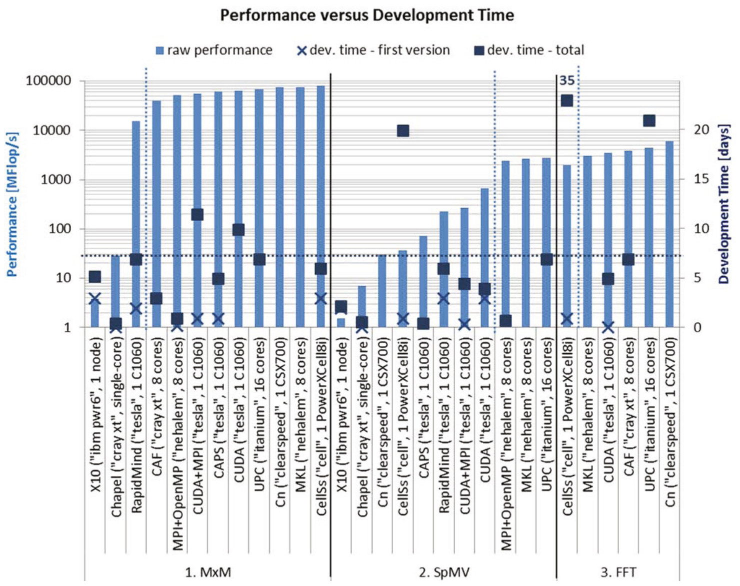


Fig. 5. Performance versus development time for all languages. Performance is measured in MFlop/s on the unit specified below. The units are chosen to have approximately the same peak performance; see rightmost column in Table 2.

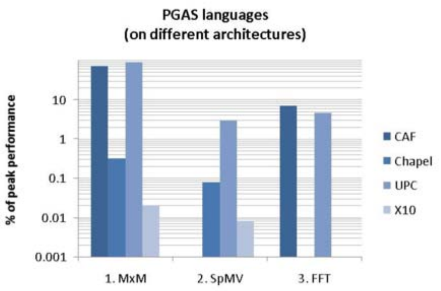


Fig. 6. Maximum performance achieved with the PGAS language ports for all kernels. Performance measurements have been performed on different hardware and are therefore given in percentage of peak performance.

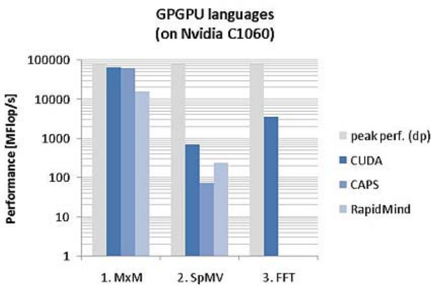


Fig. 7. Maximum performance achieved with different GPGPU languages for all kernels. Performance measurements have been performed on 1 Nvidia C1060 and given in MFlop/s.

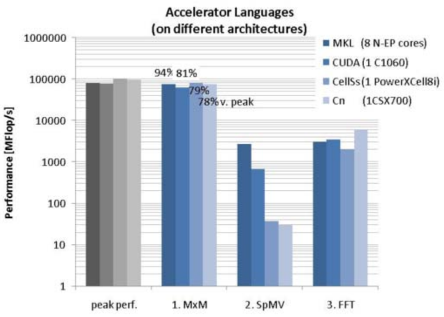


Fig. 8. Performance comparison of hardware accelerators for all kernels together with the reference performance obtained on 8 Nehalem-EP cores

4 Conclusions

The presented results are all based on one single port of the kernel to the language; one programmer was responsible for one language. Hence, the figures will mainly show certain trends and tendencies in the comparison of the languages. Since new hardware and new compilers are released nearly every quarter we like to consider the results as a snapshot showing the state of the languages in autumn 2009. Currently, we are in the process of establishing a constant evaluation of new languages and compiler releases which is further described in the following section.

Concerning the PGAS languages (Fig. 6), a remarkable difference exists between the more mature languages (UPC, CAF) and the latest HPCS languages (Chapel, X10), which were still too immature and did not even deliver one percent of peak performance. The difference in performance has been further intensified by the fact that Chapel and X10 were amongst the few languages which could not make use of library calls. The compiler releases used for Chapel and X10 were mostly proof-of-concepts. Looking at the reported development time and lines of code, Chapel clearly outperformed all other languages. Within PRACE-1IP, we are monitoring progress made with new Chapel compiler releases. UPC and CAF have been selected for studies on more complex benchmark codes.

Looking at the GPGPU languages (Fig. 7), CUDA is clearly the fastest language. The performance of CAPS hmpp was one of the positive surprises during this study while the RapidMind ports could not meet expectations. Further studies on the portability of RapidMind code and code performance [29] showed that a Cell optimized version of MxM is able to compete with the Cell SDK version, but performance is hardly portable across architectures (even if the programming model seems abstract enough to allow writing architecture-independent code and a highly optimized compiler which could add architecture-specific auto-optimizations). The OpenCL performance results were obtained on a beta-release of the compiler and are therefore not published; this version was not able to compete with the performance of the other languages. Meanwhile, a full set of the three OpenCL benchmarks is available and OpenCL will be further used to assess GPGPU performance of real applications.

Comparing MKL runs on 8 Nehalem-EP cores running at 2.53 GHz (peak performance: 80 GFlop/s) with 1 C1060 GPU (78 GFlop/s), 1 PowerXCell8i (102 GFlop/s) and 1 CSX700 (96 GFlop/s) seems reasonable and shows that even for the matrix-matrix multiplication benchmark, which is targeted to hardware accelerators, the performance increase is not overwhelming. If the code relies on double-precision arithmetic a significant performance improvement when using hardware accelerators will only appear in very special circumstances. We are currently evaluating the improvement gains by the new Nvidia "Fermi" hardware [30].

5 Future Work

The first pass in our performance and productivity assessment has unveiled a few shortcomings in the original setup that have been improved for the PRACE-1IP evaluation:

- The questionnaire should specify in more detail how to measure the lines of code. Tools, which could reliable measure this metric for all languages, should be employed.
- The developer diaries need to be adapted to assist programmers in deciding which time needs to be accounted under which category (development, testing, bug-fixing, etc.).
- The benchmarks should be chosen to represent important application domains and make it hard to simply use library calls.

The last point might be controversial since many production codes make use of vendor optimized (math) libraries. A separate assessment of the suitability and performance of vendor libraries and ISV codes is needed and will be very beneficial for certain communities, but the majority of highly-scalable codes are still home-brewed. These codes will only benefit from a comparative languages study if the chosen benchmarks are closely related to their most time-consuming kernel routines. Additionally, it is important that the performance and productivity measurements are based on real ported codes without any library calls to ensure that: firstly, the performance improvements do not depend on the highly architecture-optimized library version but are achievable with hand-ported code and a regular compiler; secondly, the benchmark code could be used as a guideline to port a full application and a rough estimate on the difficulty of the port.

In the follow-on project we have limited the number of languages under investigation, dismissed Cell and Clearspeed hardware and adopted Cilk [31], ArBB [24] and StarSs, a successor of CellSs, as new languages. We proposed a 3-level hierarchy for testing new languages:

1. The three mathematical kernels could be used for a first assessment of new languages.

2. Those languages that pass a certain performance threshold (e.g. 75% of peak for MxM) should form the middle level of the hierarchy. The languages of the middle level should then be ported to more complex mathematical kernels which can not be simply based on math library calls, to better reflect their true potential.
3. Only languages that are able to pass certain performance and productivity levels should then form the upper level. Porting of promising applications to the languages of the upper level should prove if the determined productivity improvements persist for full application codes.

Once this hierarchy has been set up it is relatively easy to add new languages or evaluate new compiler versions.

To ensure that these results are most beneficial to the scientific community we need to find a way to make them as transparent as possible. Making the hierarchy accessible online together with performance and productivity results will ensure that everyone who is considering to move to new languages or architectures could get instant advice. Bringing this data together with metrics like GFlop/s per Dollar or GFlop/s per Watt will allow further judgments. The PRACE work package on "Future Technologies" has already started to gather these values. How to complement the limited resources available within PRACE with external man power might be the most challenging question. Assessing performance of submitted codes is not the problem, but a network of trust is necessary to assess development time, the most important metric to determine productivity.

References

1. Simpson, A., et al.: PRACE-PP Deliverable D6.1 Identification and Cate- gorisation of Applications and Initial Benchmarks Suite, `http://www.prace-project.eu/documents/public-deliverables-1/Identification_and_Categorisatio_of_Applications_and_Initial_Benchmark_Suite_nal.pdf`
2. Michielse, P., et al.: PRACE-PP Deliverable D6.3.1 Report on available performance analysis and benchmark tools, Representative Benchmark, `http://www.prace-project.eu/documents/public-deliverables-1/public-deliverables/D6.3.1.pdf`
3. von Alfthan, S., et. al.: PRACE-PP Deliverable D6.5 Report on Porting and Optimisation of Applications, `http://www.prace-project.eu/documents/public-deliverables/d6-5.pdf`
4. Jowkar, M., et. al.: PRACE-PP Deliverable D6.4 Report on Approaches to Petascaling, `http://www.prace-project.eu/documents/public-deliverables/d6-4.pdf`
5. Kennedy, K., et al.: Defining and Measuring the Productivity of Programming Languages. International Journal of High Performance Computing Applications 18 (2004)
6. Cavazzoni, C., et al.: PRACE-PP Deliverable D6.6 Report on petascale software-libraries and programming models, `http://www.prace-project.eu/documents/public-deliverables/d6-6.pdf`

7. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report No. UCB/EECS-2006-183 (2006), http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf
8. Che, S., et al.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization IISWC 2009, pp. 44–54 (2009)
9. The Euroben benchmark home page, http://www.euroben.nl/
10. Intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl/
11. Carlson, W., et al.: UPC: Distributed Shared Memory Programming. Book of WileyInter-Science (2005)
12. Numrich, R.W., Reid, J.: Co-Array Fortran for Parallel Programming. ACM SIGPLAN Fortran Forum 17 (1998)
13. Dongarra, J., et al.: DARPA's HPCS Program: History, Models, Tools, Languages. Advances in Computers (2008)
14. Chamberlain, B.L., et al.: Parallel Programmability and the Chapel Language. International Journal of High Performance Computing Applications 21 (2007)
15. X10 language, http://x10-lang.org
16. ClearSpeed Cn language, http://www.clearspeed.com
17. Perez, J.P.: CellSs: making it easier to program the cell broadband engine processor. IBM Journal of Research and Development 51 (2007)
18. Nvidia CUDA, http://www.nvidia.com/object/cuda_home.htm
19. OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/
20. CAPS hmpp workbench, http://www.caps-entreprise.com/hmpp.html
21. RapidMind, https://www.rapidmind.com (forwarded to Intel ArBB)
22. Christadler, I., et al. (eds): PRACE Workshop New Languages and Future Technology Prototypes, Garching (2010), http://www.prace-project.eu/documents/prace_workshop_on_new_languages_and_future_technology_prototypes.pdf
23. Strumpen, V., et. al.: PRACE-1IP Deliverable D9.2.1 First report on multi-Peta to Exascale software, http://www.prace-project.eu/documents/public-deliverables-1/
24. Intel ArBB, http://software.intel.com/en-us/articles/intel-array-building-blocks/
25. Sai Saigar, R., et al.: PRACE-PP Deliverable D8.3.2 Final technical report and architecture proposal, http://www.prace-project.eu/documents/public-deliverables/d8-3-2-extended.pdf
26. Dongarra, J., et al. (eds): CT Watch Quarterly. High Productivity Computing Systems and the Path Towards Usable Petascale Computing Part A 2(4A) (2006), http://www.ctwatch.org/quarterly/pdf/ctwatchquarterly-8.pdf
27. Squires, S., et al.: Software Productivity Research in High-Performance Com putingi. CT Watch Quarterly 2(4A), 52–61 (2006)
28. Hochstein, L., et al.: Experiments to Understand HPC Time to Development. CT Watch Quarterly 2(4A), 24–32 (2006)
29. Christadler, I., Weinberg, V.: RapidMind: Portability across Architectures and Its Limitations. In: Keller, R., et al. (eds.) Facing the Multicore-Challenge. LNCS, vol. 6310, pp. 4–15. Springer, Heidelberg (2010)
30. Tesla M-Class GPU Computing Modules ("Fermi"), http://www.nvidia.com/docs/IO/105880/DSTesla-M2090LR.pdf
31. The Cilk Project, http://supertech.csail.mit.edu/cilk/

Towards High-Performance Implementations of a Custom HPC Kernel Using Intel® Array Building Blocks

Alexander Heinecke[1], Michael Klemm[2], Hans Pabst[2], and Dirk Pflüger[1]

[1] Technische Universität München, Boltzmannstr. 3, D-85748 Garching, Germany
[2] Intel GmbH, Dornacher Str. 1, D-85622 Feldkirchen, Germany

Abstract. Today's highly parallel machines drive a new demand for parallel programming. Fixed power envelopes, increasing problem sizes, and new algorithms pose challenging targets for developers. HPC applications must leverage SIMD units, multi-core architectures, and heterogeneous computing platforms for optimal performance. This leads to low-level, non-portable code that is difficult to write and maintain. With Intel® Array Building Blocks (Intel ArBB), programmers focus on the high-level algorithms and rely on an automatic parallelization and vectorization with strong safety guarantees. Intel ArBB hides vendor-specific hardware knowledge by runtime just-in-time (JIT) compilation. This case study on data mining with adaptive sparse grids unveils how deterministic parallelism, safety, and runtime optimization make Intel ArBB practically applicable. Hand-tuned code is about 40% faster than ArBB, but needs about 8x more code. ArBB clearly outperforms standard semi-automatically parallelized C/C++ code by approximately 6x.

Keywords: parallel languages, vector computing, high performance computing, Intel® Array Building Blocks, Intel ArBB, OpenCL.

1 Introduction

Moore's Law is still alive and well: the amount of transistors on a chip is still growing exponentially [2]. Because of well-known fundamental limitations in energy consumption [20], hardware vendors can no longer increase performance of single-core CPUs at the usual pace. Instead, the available transistor budget is used to build multi-core and many-core CPUs. This trend is expected to continue in future [2], making (efficient) parallelism a key requirement for software [21]. To complicate matters, software not only needs to cope with thread parallelism, but it is also required to exploit SIMD parallelism, e. g., Intel® Streaming SIMD Extensions (SSE) or Intel® Advanced Vector Extensions (AVX).

Classic programming approaches for recent CPUs use threading APIs (e. g., POSIX [5], OpenMP [16]) and augment it with vectorization hints for the compiler or low-level intrinsic functions to control SIMD parallelism. The drawbacks of this approach are manifold. First, new hardware generally comes with new features. The evolution of SSE to AVX is just one example. Code with low-level vectorization is not portable and may not exploit new hardware features. Second, it exposes machine specifics such as memory organization, SIMD length and instructions, and data alignment. Third, it becomes increasingly difficult for programmers to focus on the problem domain. Scientists cannot naturally express their algorithm in such low-level programming models, but need to reformulate the algorithm due to a lack of expressiveness of the programming language used.

Intel® Array Building Blocks [15] provides a generic parallel programming model for vector-parallel programming. It frees programmers from the dependencies to the current compute platform. Intel ArBB relies on a combination of standard C++ interfaces and a powerful runtime system with just-in-time (JIT) compilation. The JIT compiler generates an optimized vectorized and multi-threaded code from a single-source high-level program description. With this approach, ArBB may re-target an application to a new hardware platform without the need to recompile. In addition, its programming model provides a portable and deterministic parallel programming model with sequential semantics.

In this paper, we use SG^{++} as a case study. SG^{++} is a high-performance data mining application that solves high dimensional problems through sparse-grid discretization techniques. Possible application scenarios are data mining tasks such as classification and regression. SG^{++} learns the structure of the data from input sampling data and creates a smooth function to classify unknown data sets. We compare several highly tuned implementations in OpenMP and SSE/AVX with a high-level implementation in Intel ArBB.

2 Related Work

There are numerous (parallel) programming models for HPC applications. Due to space limitations, we focus on wide-spread languages. Fortran 2008 offers support for high-level data-parallel programming [10]. C++11 is expected to implement a threading-model for the first time [11]. OpenMP [16] offers data-parallel and task-parallel programming on top of C/C++ and Fortran. In contrast to ArBB, these traditional languages are compile-time optimized languages that do not allow for re-targeting the application to future hardware. In addition, the base languages make it hard for a compiler to select matching SIMD instructions and to vectorize the code effectively.

The high-level language given by ArBB aims to balance between productivity and forward-scaling high performance such that future hardware improvements can be exploited without re-compilation. A summary of ArBB can be found in [15] which describes the programming model, the JIT compiler, and the code optimization techniques in detail. The ArBB programming model integrates the common parallel pattern of elemental functions using the map-operator into the more general context of vector processing and is therefore not limited to traditional "kernel-based programming" (cf. OpenCL [7]).

While providing a framework for parallel programming is not a new idea, ArBB combines several interesting ideas into a single approach. Skeleton libraries such as Muesli [4], SkeTo [14], and OSL [12] (see [13] for a short survey) are framework-based approaches to parallel programming. None of them utilizes a VM infrastructure to retarget code to the compute environment. As fixed-skeleton libraries they cannot support unforeseen algorithmic patterns. In addition, all of them only target parallelization and omit vectorization. ArBB and OpenCL are more generic, as they do not restrict the algorithmic patterns and also target vector units of modern CPUs. ArBB and OpenCL as presented do not support MPI, which the mentioned skeleton libraries do.

In [18], the authors explore the “Ninja performance gap” of several important workloads by using directives (pragma hints) and compiler extensions up to algorithmic changes. These techniques mainly aim to exploit SIMD-level parallelism as found in today’s CPUs and utilize multiple cores through OpenMP. In this case study, we focus on a single workload but employ multiple hand-tuned baselines. We do not aim to augment existing code with parallelization/vectorization hints; our target are data-parallel programming languages with runtime code generation and assess their performance impact. We exclusively rely on elemental functions to not exclude OpenCL from our evaluation.

For a meaningful comparison between HPC languages, a proper metric is needed. The common trend shows that platforms become more complex and heterogeneous with every generation. Hence, there is an ongoing research how to score productivity in high performance computing. Many different aspects are surveyed in [6,19]. Unfortunately, we cannot apply these in our case study due to the fact that they are partly based on developer surveys, programmer diaries, and other qualitative measurements. Hence, we focus on an ease-of-use comparison and evaluate the readability of the different approaches. Performance also plays a crucial role in our assessment, as we are targeting the HPC domain. For the sake of simplicity, we restrict ourselves to Lines of Codes (LOC). While we know about fundamental limitations in using LOCs as a productivity measurement metric, we believe that LOCs are a good basis to measure how concisely a scientific algorithm can be represented in source code in terms of expressiveness.

3 Intel ArBB

ArBB is a data-parallel language inspired by functional paradigms [1]. It relies on a virtual machine (VM) [9] with a C API for direct usage or for use by a language binding. ArBB currently offers a language binding (header-only C++ API on top of the VM API) to embeds a data-parallel language into C++ (no own textual representation, no external executable). This library-based approach enables any ISO-compliant industry-standard C++ compiler. ArBB is embedded into C++ such it naturally extends the language features of C++. Class templates, functions, or user-defined types are supported orthogonally to what ArBB provides. Moreover, the host language can be used to drive program generation, i. e., meta-programming and specialization can be part of an application.

In C++, an ArBB parallel region is formed by a function pointer that is handed to `arbb::call` which in turn returns a function object (closure) of the same signature as the function pointer. This allows to capture an intermediate representation of the to-be-invoked function for compilation and optimization by the JIT compiler. An ArBB program may use any level of indirection (modularization) offered by the host language; code is optimized and inlined as if there were no call boundaries. The ArBB operators associated to ArBB collections embrace parallelism such that they itself can be used in a serial fashion. Thus, programmers can reason about their code as if it was sequential and let the ArBB VM create parallelism out of the operations on ArBB collections. The program is mapped to the hardware by propagating and fusing excessive copy operations as well as barrier synchronization. ArBB can be applied per region or grow into an application similar to other frameworks.

Transparently retargeting code requires the code to be expressed using ArBB, i. e., calling into statically compiled code might break this ability. Since ArBB aims to overcome the burden of adapting to different hardware, runtime compiler technology is be employed to avoid optimization by (manually) slightly varying an algorithm's formulation. Programmers are encouraged to focus on the problem domain and algorithmic quality. Sealing the parallelism inside of operators as well as high-level optimizations and canonization applied to an unified internal representation allows the JIT compiler to aim for performance portability.

4 Sparse Grid Data Mining

We choose data mining as an application due to its growing importance: in many scientific fields, there has been a shift to data-driven applications in recent years. More and more data is available and is being collected, and is thus harvested and explored. Generalizing, inferring from gathered data, is a frequent task, which arises in many diverse areas such as medicine, finance, traffic control, or astrophysics. If the outcome of a certain measurement or experiment is expensive to obtain but it is related to other properties, it is of interest to reconstruct or learn this dependency to be able to predict it in the future.

In other words, starting from m known observations, $S = \{(\boldsymbol{x}_i, y_i) \in \mathbb{R}^d \times K\}_{i=1,\ldots,m}$, the aim is to learn the functional relation $f(\boldsymbol{x}_i) \approx y_i$ as accurate as possible. Reconstructing f then allows to estimate $f(\boldsymbol{x})$ for new properties $\boldsymbol{x}$. For the task of binary classification (think of a bank discriminating potential customers into creditworthy and non-creditworthy based on previous customers) we choose as target labels $K = \{+1, -1\}$, for regression (learning a general function), we allow $K = \mathbb{R}$.

Our target is to find a function f, which satisfies these requirements, and which is a linear combination of N basis functions φ_j with coefficients α_j, $f = \sum_{j=1}^{N} \alpha_j \varphi_j(\boldsymbol{x})$. The basis functions are associated with grid points on some grid. Here, we are considering piecewise d-linear functions. The main advantage of this approach is that almost arbitrarily large data sets can be dealt with (our example contains more than 250,000 data points), as the resulting algorithms scale almost linearly with the number of training data. Unfortunately, regular grid structures suffer the "curse of dimensionality": a regular grid structure with equidistant meshes and k grid points in one dimension features k^d grid points in d dimensions. This exponential growth typically prevents more than 4 dimensions for a reasonable discretization.

We therefore employ *adaptive sparse grids* [3,17], which mitigate the curse of dimensionality to a large extent. Sparse grids employ a hierarchical basis with basis functions defined on several levels of discretization. For piecewise d-linear functions, they are $\varphi_{l,i}(x) := \varphi(2^l x - i)$, based on a reference hat function $\varphi(x) := \max(1 - |x|, 0)$. The d-dimensional basis functions are then created as products of one dimensional functions, $\varphi_{\boldsymbol{l},\boldsymbol{i}}(\boldsymbol{x}) := \prod_{k=1}^{d} \varphi_{l_k,i_k}(x_k)$, with $\boldsymbol{l}$ and $\boldsymbol{i}$ as multi-indices indicating level and index in each dimension.

This allows one to represent a function on several scales. To find out which scales contribute most to the overall solution, as plenty of grid points can be omitted in the hierarchical representation as they have only little contribution—at least for sufficiently smooth functions. The cost is reduced from $\mathcal{O}((2^n)^d)$ to $\mathcal{O}(2^n n^{d-1})$ while a similar accuracy as for full grids is maintained. For functions that do not meet the smoothness requirements (as for classification), or where the data points are clustered together (our regression example), we employ adaptive refinement, spending only grid points where necessary. This leads, as for the DR5 dataset later on, to highly non-balanced grids, which poses further challenges for algorithms and parallelization.

Back to the data-driven problem, we would like f to be as close to the known data points as possible, preferably minimizing the mean squared error. Additionally, we require close data points to very likely have similar function values. We thus want to generalize and to not learn potential noise in the data. In summary, we minimize a trade-off between both (the hierarchical basis allowing for a simple generalization functional), which leads to a system of linear equations,

$$\arg\min_f \frac{1}{m} \sum_{i=1}^{m} (f(\boldsymbol{x}) - y_i)^2 + \lambda \sum_{j=1}^{N} \alpha_j^2 \qquad \Rightarrow \qquad \left(\frac{1}{m} B B^T + \lambda I \right) \boldsymbol{\alpha} = \frac{1}{m} B \boldsymbol{y} \,,$$

with matrix B, $B_{i,j} = \varphi_i(\boldsymbol{x}_j)$, and identity matrix I.

The system matrix is rather densely populated for typical datasets; thus, we do not want to fully assemble it. On the other hand, the applications of the matrix B and its transposed version boil down to function evaluations (and somehow transposed ones), as $(B^T \alpha)_i = f(\boldsymbol{x}_i)$. Function evaluations can be implemented in a multi-recursive way in both level and dimensionality, leading to a computationally efficient algorithm.

Unfortunately, this algorithm is inherently recursive and requires random memory access, both of which impose severe restrictions on parallel systems. Especially accelerator cards penalize such algorithms severely. Alternatively, one could evaluate all basis functions for all data points and sum up the results. The resulting algorithm is computationally much less efficient, but it can be arbitrarily parallelized and vectorized. A recent study [8] has shown that excellent speedups can be achieved on hybrid systems, even though a typical application did require 15 times more evaluations than before. Streaming access of the data and the avoidance of recursive structures and branches easily evens this out.

Algorithm 1. Iterative version of the operation $\boldsymbol{v} = B^T\boldsymbol{\alpha}$: every instance $\boldsymbol{x}_m$ is evaluated at every grid point g; see [8]. We denote level and index of grid point g in dimension k by g_{l_k} and g_{i_k}, and the corresponding coefficient by α_g.

```
for all $\boldsymbol{x}_m \in S$ with $v_m \leftarrow 0$ do
  for all $g \in$ grid with $s \leftarrow \alpha_g$ do
    for $k = 1$ to $d$ do
      $s \leftarrow s \cdot \varphi_{l_k,i_k}(\boldsymbol{x}_{m_k}) = s \cdot \max(1 - |2^{g_{l_k}} \cdot \boldsymbol{x}_{m_k} - g_{i_k}|; 0)$
    end for
    $v_m \leftarrow v_m + s$
  end for
end for
```

5 Implementation

As pointed out in the previous section, the most crucial parts are fast (iterative) evaluations of the function and its transposed counterpart. They can be realized using flat array structures containing the sparse grids' points ($\boldsymbol{l}$ and $\boldsymbol{i}$ uniquely define them each, thus we have two matrices L and I with the jth row containing the jth grid point), the training data points (a matrix S of the data points $\boldsymbol{x}_j$, row-wise), the coefficient vector $\boldsymbol{\alpha}$, and the vector $\boldsymbol{y}$ containing the evaluation results y_j, see Fig. 1. Using such a data layout leads to an algorithmic structure similar to a band-matrix multiplication with differing element-wise functions.

The kernel is implemented by three nested loops, see Alg. 1. The needed operation inside the inner most loop is non-linear due to the tensor construction of the sparse grid's space (see [8]). A suitable vectorization and parallelization for this algorithm would be the evaluation of several dataset instance on one grid point. This leads to a vectorization and parallelization of the outer most loop. Unfortunately the Intel® Composer XE 2011 is only able to vectorize inner most loops. Since the loop which iterates over the dimensions contains a non-linear operation, the loops cannot be exchanged in order the to enable auto-vectorization by the compiler. Additional temporal data structures are needed for such an exchange, which blow up the code and introduce unnecessary operations and reduce the programmers productivity.

Results for hand-tailored versions exploiting this approach have been shown in [8] for several programming models and architectures, ranging from multi-core CPUs to hybrid systems using several CPU sockets and GPUs. Although excellent absolute performance and speed-ups have been achieved, they were only possible by the burden of creating a hardware-aware implementations. For each platform, a complete rewrite of the kernels was necessary, and all future hardware changes will require code adjustments and changes as well. The hand-tailored implementations used here feature standard and well-known performance optimization techniques like loop-unrolling and register-blocking.

Fig. 1 illustrates the Intel ArBB data containers to store L, I, S, $\boldsymbol{\alpha}$, and $\boldsymbol{y}$. Since multi-dimensional structures are required, the splitting is performed as

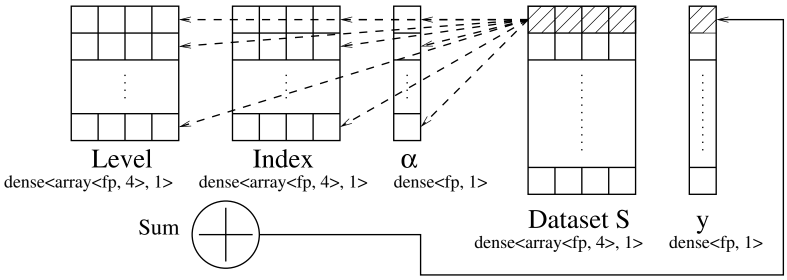


Fig. 1. ArBB data containers to manage adaptive sparse grids and data sets

follows: a grid point's level, index data, and single training data instances are coupled into ArBB d-dimensional vector types, whereas $\boldsymbol{\alpha}$- and $\boldsymbol{y}$-values are only one-dimensional, independent from the grid's dimensionality. All data elements are aggregated using the dense container type in ArBB.

After this step, five one-dimensional dense containers store the required data (Fig. 1). Besides this, Fig. 1 also gives an idea of how the language interface is designed: using templates, the member types of a container can be defined. This way, mixed precision calculations are easily realized since they can be performed by an additional template type. Here, *fp* is such a generalization. Moreover, Fig. 1 sketches how a data set is evaluated on the sparse grid: each instance has to be evaluated on every grid point as illustrated by the dashed arrows

Algorithm 2 provides the implementation details of a dataset's evaluation for an arbitrary number of dimensions. Evaluating an instance for all grid points is implemented using the `map` operator in ArBB. The code example shows, that the `map` operator is able to determine which element should be mapped on what argument by comparing the input parameters with the signature of the called function. Due to limitations in the current C++ standard, this kernel has to be defined in an inlined local structure. Inside this function an additional significant advantage of ArBB can be observed: element-wise functions and operators.

Unlike approaches as taken in [8], there is no loop iterating over the number of dimensions. Such a loop is replaced by calling functions and operators directly on the vector types that are used to store one grid point or one instance. Please note that the `_for(,,)` in the listing defines a sequential execution of the loop body. If a parallel loop is intended, the map operator should be used.

ArBB focuses on the mathematical problem description and hides the practical representation from the programmer. There is no blocking or reordering of the operations, leading to an easily readable source code. As stated in section 2, the JIT compiler in ArBB and the virtual machine are responsible for a highly efficient execution on a given hardware platform. This significantly simplifies a programmer's life and directly leads to better understanding and maintainability of the written code. If an ArBB code is compared to the OpenCL or intrinsics versions, much information about the scientific problem gets lost in the latter ones. As described in [8], in case of hand-tuned vectorization using intrinsics, *several* dataset instances are evaluated for *one* grid point simultaneously.

This requires duplicate loads or even shift operations inside the innermost kernels. Using OpenCL, the kernel implements the evaluation of *one* instance on *all* grid points. Moreover, since OpenCL includes an offloading model, concepts like device contexts, command queues, buffers, and explicit kernels have to be used, which require programmers to cope with additional boilerplate code.

In both OpenCL and intrinsics, the actual mathematical idea is not clearly represented which may cause difficulties in porting and maintaining the code. To be more precise, since several optimization techniques like register blocking and prefetching are employed, the code's readability decreases further. As it can be seen from this case study, ArBB requires no built-in assumptions about the underlying hardware, whereas intrinsics require a binary compatible platform and OpenCL relies on an execution model based on a complex memory hierarchy and small execution units (see [7] for details).

To discuss ease-of-use capabilities ArBB is compared in terms of lines of code (LOC) with these hardware-aware alternatives. For **ArBB, 10 LOC** are needed for the whole kernel (as shown in Alg. 2). These 10 lines are independent from the floating point precision that is used, since this can be handled by an additional template type. In case of Intrinsics, the kernel including an OpenMP parallelization requires 80 LOC for every floating point precision and vectorization method (SSE or AVX), so we end up with **320 LOC for intrinsics**. Although, the vectorized code can be transcribed semi-automatically between SSE and AVX, recompilation and individual testing is needed to ensure correctness. Due to its exposed offloading model, OpenCL is the most complex one in terms of LOC: for each floating point standard approximately 250 LOC are required which results in **500 LOC using OpenCL** for both single and double precision support. Different OpenCL targets may also cause a code rewrite since they offer hardware cache and same address spaces for host and target.

6 Results

The previous sections have introduced the scientific application, and its implementation with ArBB, hand-tuned vectorization, and OpenCL. These different programming languages have been compared in terms of their productivity. We now focus on the performance of all three approaches.

As workloads we use two different datasets with distinct properties and challenges. For classification, we use a synthetic 5-dimensional dataset. The 2^{18} data points have been drawn randomly from the domain, and have been assigned to the target values ± 1 based on a $3 \times 3 \times 3 \times 3 \times 3$ checkerboard pattern. For regression, we use a 5-dimensional real-world dataset (DR5) from astrophysics with measurements of more than $430,000$ galaxies to predict the galaxies' redshift as a measure for their distance. Without the data mining approach this data is very expensive to obtain.

We employ spatial adaptivity with six refinement steps for both workloads. This leads to completely different types of grids. Whereas the checkerboard dataset requires a regular distribution of grid points along the classification, the

Algorithm 2. Implementation of the innermost kernel using the Intel ArBB map-operator and element-wise functions. ArBB language extensions are underlined.

```
template<typename fp>
void arbb_mult(const dense<array<fp, DIM>>& Dataset,
               const dense<array<fp, DIM>>& Level,
               const dense<array<fp, DIM>>& Index,
               const dense<fp>& alpha, dense<fp>& y)
{
  struct local {
    static void evalGridPoint(const array<fp, DIM>& DataPoint,
      const dense<array<fp, DIM>>& Level,
      const dense<array<fp, DIM>>& Index,
      const dense<fp>& alpha, fp& y_point) {
        y_point = 0.0;
        _for (usize j = 0, j < Level.length(), j++) {
          y_point += alpha[j] * mul_reduce(
            max(1 - abs(Level[j] * DataPoint - Index[j]), 0));
        } _end_for;
      }
  };
  map(local::evalGridPoint)(storage_size,Dataset,Level,Index,alpha,y);
}
```

astrophysical DR5 dataset is highly clustered. The checkerboard dataset leads to a prediction accuracy of more than 92%, the redshift dataset to an MSE of $\approx 5.3 \cdot 10^{-4}$. As shown in [8], computation with single precision is sufficient. We use a dual-package Intel® Xeon™ X5650 (2.66 GHz, 24 GB memory) for the evaluation of the generated SSE code. In addition, we test on a recent Intel® Core™ i7-2600 (3.40 GHz, 8 GB memory) to evaluate AVX vectorization.

Table 1 compares the out-of-the-box performance for Alg. 2 to Intel® OpenCL and the hand-tuned intrinsic versions. The runtime includes the serial grid refinement with parallel calculation of B and B^T. Besides the timing, we also give results in terms of GFLOPS. SG^{++} is sensitive to rounding errors, which may lead to slightly different structures of the learned adaptive grid (see [8] for an in-depth discussion). While they achieve the same classification accuracy, runtime may vary significantly. Hence, GFLOPS give a better indication of the actual performance of the algorithm.

For both platforms, the optimal intrinsic version delivers about 40% more performance than ArBB in terms of execution time and between 1.6-1.9x if measuring in GFLOPS. In comparison to Intel OpenCL there is a mid-30% performance penalty for ArBB using the Intel Xeon workstation. On the Core i7 system, ArBB outperforms OpenCL by 11% due to a better support for AVX at the current time. Please note that both Intel ArBB and the Intel OpenCL SDK are currently in a beta phase thus further improvements can be expected for the release versions. A significant amount of performance for ArBB Beta 6 is lost due to a missing thread pinning when executing threaded code. Furthermore, we did single threaded experiments, here ArBB was able to yield up to 90% of the intrinsics' implementation, which confirms issues with the threading runtime in ArBB. For both test platforms the double-precision OpenCL performance drops clearly below the baseline using the static compiler. First analysis shows that the

Table 1. Performance for the 5-dim. checkerboard dataset and the DR5 dataset on a two-socket Intel® Xeon™ workstation with Intel® X5650 processors and on a one-socket Intel® Core™ i7-2600 desktop. For ArBB, the multi-threaded out-of-the-box performance is shown, single-threaded execution achieved up to 90% of the intrinsics' performance. The OpenCL numbers are based on the Intel® OpenCL SDK.

Tool	*5d ch.board SP*		*5d ch.board DP*		*DR5 SP*		*DR5 DP*	
	time [s]	GFLOPS	time [s]	GFLOPS	time [s]	GFLOPS	time [s]	GFLOPS
Intel® Xeon™ X5650 desktop								
ArBB	8300	83	18600	40	4200	91	5600	42
SSE	5400	140	10600	70	2300	138	2500	80
OpenCL	5700	117	81000	9	3500	109	23700	9
ICC 12	25200	28	26000	28	11100	28	7300	28
Intel® Core™ i7-2600 desktop								
ArBB	11000	63	24500	30	5500	70	7500	34
SSE	11600	65	21600	34	5000	76	5500	38
AVX	6400	118	13000	56	2600	145	3100	73
OpenCL	13800	49	196000	4	7800	50	68000	4
ICC 12	62000	12	60500	12	31500	12	18600	12

vectorizer seems to be the problem in this case hence the wrong data streams are vectorized.

In our experiments, ArBB clearly outperforms the static compiler (Intel C++ Composer XE 2011) by a factor of 6x for single precision on the Core i7 system. This huge speed-up is due to the fact that the JIT compiler in ArBB is able to vectorize the kernels with their non-linear inner most loop. Here a connection to Alg. 2 is useful: due to the `map` operator the ArBB compiler can analyze the data flow and choose a sufficient vectorization afterwards. A static compiler is not able to extract this knowledge from three nested loops.

For SSE, OpenCL (single precision only) exhibits a performance advantage over ArBB, because of additional manual optimizations like loop-unrolling, local store prefetching, and introduction of temporary variables. For OpenCL, this was necessary to achieve optimal performance on a GPU (see [8]). Without typical GPU optimizations, the straightforward CPU version of OpenCL achieves a significantly lower performance (about 15%), which results in an out-of-the-box performance that is slightly below ArBB.

Both tables additionally show that ArBB also automatically exploits the capabilities of new hardware without recompilation. Switching from one processor generation to a newer one yields the same speed-up as for low-level code which has to be rewritten to support new features.

Although ArBB is in beta phase (and work in progress), it already shows its enormous potential as a high-level programming language. The current release features an automatic parallel vectorized performance that is only around fifty percent slower than hand-tuned codes. Moreover, as stated in the previous section, the ArBB code is more readable than even the standard C++ code which gives an additional productivity advantage of ArBB.

7 Conclusions

We have demonstrated the capabilities of the novel programming model of Intel ArBB for a real-world application from the field of data mining, with adaptive sparse grids as the underlying data structure.

We have compared other implementations to an ArBB implementation. The Intel OpenCL implementation delivers performance close to hand-tuned SSE code. Programmers need to write some boilerplate code, and also need to handle buffer transfers. Orchestrating the memory hierarchy explicitly is mostly superfluous for architectures which are employing cache memory transparently. This, and other low-level language elements may harm performance-portability. The resulting complexity often obscures the underlying problem, but the code can be executed with excellent performance on today's GPUs.

Using platform-dependent implementations with vector intrinsics obviously provided the best performance. Intrinsics give control to hand-tune code in order to exploit the hardware. In terms of lines of code, this approach is to some extent even better than OpenCL, but it requires to reimplement the algorithm for new hardware architectures. The porting effort from SSE to AVX is little, but requires the availability of the new hardware (or simulators) and imposes additional correctness testing.

ArBB provides a high-level, hardware-oblivious programming approach, which allows to exploit data-parallelism as well as (nested) parallelism using general vector-parallelism and elemental functions. Its main advantage is its expressiveness which typically leads to a fraction of the code compared to loop-nested code (or compared to our OpenCL and Intrinsics code). ArBB guides programmers to write parallel algorithms which can be automatically parallelized. Furthermore, it is easy to generalize for different precisions or a different dimensionality of a problem because C++ language constructs such as templates are orthogonal to ArBB. ArBB is targeting the Intel® Many Integrated Core (MIC) architecture, or Intel® architecture in general (incl. non-Intel processors). Moreover, ArBB will target GPU architectures in the future. Being able to write maintainable code in a hardware-oblivious way with only slightly lower performance promises a significant gain for the parallelization of scientific problems.

References

1. Blelloch, G.E.: Vector Models for Data-Parallel Computing (Artificial Intelligence), 1st edn. The MIT Press (1990)
2. Borkar, S., Chien, A.A.: The Future of Microprocessors. Communications of the ACM 54(5), 67–77 (2011)
3. Bungartz, H.-J., Griebel, M.: Sparse Grids. Acta Numerica 13, 147–269 (2004)
4. Ciechanowicz, P., Kuchen, H.: Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In: Proc. of the 2010 IEEE 12th Intl. Conf. on High Performance Comp. and Comm., Orlando, FL, pp. 108–113 (2010)
5. Drepper, U., Molnar, I.: The Native POSIX Thread Library for Linux. Technical report, Redhat (2003)

6. Faulk, S., Porter, A., et al.: Measuring HPC productivity. Intl. J. of High Performance Computing Applications, 459–473 (2004)
7. Khronos OpenCL Working Group. The OpenCL Specification, Version 1.1, Document Revision 36 (2010)
8. Heinecke, A., Pflüger, D.: Multi- and Many-Core Data Mining with Adaptive Sparse Grids. In: Proc. of the 2011 ACM Intl. Conf. on Computing Frontiers (2011) (accepted for publication)
9. Intel Corp. Intel®Array Building Blocks Virtual Machine Specification, Version 1.0 Beta, Document Number 324820-002US (2011)
10. ISO/IEC. Information Technology – Programming Languages – Fortran – Part 1: Base Language, ISO/IEC 1539-1 (2010)
11. ISO/ISC. Standard for Programming Language C++, ISO/ISC DTR 19769, final draft (2011)
12. Javed, N., Loulergue, F.: OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 436–451. Springer, Heidelberg (2009)
13. Matsuzaki, K., Emoto, K.: Lessons from Implementing the biCGStab Method with SkeTo Library. In: Proc. of the 4th Intl. Workshop on High-level Parallel Programming and Applications, Baltimore, MD, pp. 15–24 (2010)
14. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In: Proc. of the 1st Intl. Conf. on Scalable Information Systems, Hong Kong (2006)
15. Newburn, C.J., McCool, M., et al.: Intel®Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language. In: Proc. of the Intl. Symp. on Code Generation and Optimization, Chamonix, France, pp. 224–235 (2011) (to appear)
16. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.0 (2008), `http://www.openmp.org/`
17. Pflüger, D.: Spatially Adaptive Sparse Grids for High-Dimensional Problems. Dissertation, Institut für Informatik, TU München, München (2010)
18. Satish, N., Kim, C., et al.: Can Traditional Programming Bridge the Ninja Performance Gap for Throughput Applications? In: Proc. of the 17th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, London, UK (2012) (submitted)
19. Sterling, T.L.: Productivity Metrics and Models for High Performance Computing. Intl. J. of High Performance Computing Applications 18(4), 433–440 (2004)
20. Sutter, H.: The Free Lunch Is Over—A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (2005)
21. Sutter, H., Larus, J.: Software and the Concurrency Revolution. ACM Queue 3, 54–62 (2005)

AHDAM: An Asymmetric Homogeneous with Dynamic Allocator Manycore Chip

Charly Bechara[1], Nicolas Ventroux[1], and Daniel Etiemble[2]

[1] CEA, LIST, Embedded Computing Laboratory, Gif-sur-Yvette, F-91191, France
charly.bechara@cea.fr
[2] Université Paris Sud, Laboratoire de Recherche en Informatique, Orsay, F-91405, France

Abstract. The future high-end embedded systems applications are characterized by their computation-intensive workloads, their high-level of parallelism, their large data-set requirements, and their dynamism. Those applications require highly-efficient manycore architectures. In response to this problem, we designed an asymmetric homogeneous with dynamic allocator manycore architecture, called AHDAM chip. AHDAM chip exploits the parallelism on all its granularity levels. It implements multithreading techniques to increase the processors' utilization. We designed an easy programming model and reused an automatic compilation and application parallelization tool. To study its performance, we used the radio spectrum sensing application from the telecommunication domain. On a simulation framework, we evaluated sequential and parallel versions of the application on 2 platforms: single processor, and AHDAM chip with a variable number of processors. The results show that the application on the AHDAM chip has an execution time 574 times faster than on the single-processor system, while meeting the real-time deadline and occupying 51.92 mm^2 at 40 nm technology.

Keywords: Manycore, asymmetric, multithreaded processors, dynamic applications, embedded systems.

1 Introduction

During the last decades, the computing systems were designed according to the CMOS technology push resulting from Moore's Law, as well as the application pull from ever more demanding applications [1]. The emergence of new embedded applications for mobile, telecom, automotive, digital television, communication, medical and multimedia domains has fuelled the demand for architectures with higher performances (order of TOPS), more chip area and power efficiency. These complex applications are usually characterized by their computation-intensive workloads, their high-level of parallelism, their large data-set requirements and their dynamism. The latter implies that the total application execution time can highly vary with respect to the input data, irregular control flow, and auto-adaptivity. Typical examples of dynamic algorithms are 3D rendering, high definition (HD) H.264 video decoder, and connected component labeling [2].

The parallelism can be exploited on multiple granularities, such as instruction level (ILP), loop level (LLP), and thread level (TLP). Those massively parallel high-end embedded applications, which can have more than 1000 parallel threads, require highly efficient microprocessor architectures. With the limits of ILP [3] and the low transistor/energy efficiency of superscalar processors for embedded systems applications, the chip manufacturers are increasing the overall processing power by integrating additional CPUs or "cores" to the microprocessor package. Such microprocessor chip architectures for the embedded systems world are called *MPSoC*. An MPSoC with large number of cores is called a *manycore* architecture. These architectures need to exploit all types of parallelism in a given application.

Unfortunately, the existing MPSoC/manycore architectures offer only partial solutions to the power, chip area, performance, reliability and dynamism problems associated with the embedded systems. For instance, an optimal static partitioning on an MPSoC cannot exist since all the tasks processing times depend on the input data that cannot be known off-line. [4] and [5] show that the solution consists in dynamically allocating tasks according to the availability of computing resources. Global scheduling maintains the system load-balanced and supports workload variations that cannot be known off-line. Only an asymmetrical approach can implement a global scheduling and efficiently manage dynamic applications. An asymmetric MPSoC architecture consists of one (sometimes several) centralized or hierarchized control core, and several homogeneous or heterogeneous cores for computing tasks. The control core handles the tasks scheduling. In addition, it performs load balancing through task migrations between the computing cores when they are homogeneous. The asymmetric architectures have usually an optimized architecture for control. This distinction between control and computing cores renders the asymmetric architecture more transistor/energy efficient than the symmetric architectures. However, one main drawback of asymmetric architectures is their scalability. The centralized core is not able to handle more than a specific threshold number of computing cores due to reactivity reasons.

In this paper, we present the AHDAM chip, an asymmetric manycore architecture that tackles the challenges of future massively-parallel dynamic embedded applications. Its architecture permits to process applications with large data sets by efficiently hiding the processors' stall time using multithreaded processors. Besides, the AHDAM chip has an easy programming model as it will be shown in this paper. The main contributions of this paper are:

- System design (architecture + programming model) of an efficient asymmetric manycore chip architecture for the embedded systems.
- Architecture evaluation using a significant application from radio telecommunication domain (radio sensing).

This paper is organized as follows: Section 2 introduces the AHDAM chip architecture, its system environment, its programming model, its different components functionality, their interoperability, and the execution model. The evaluation of the architecture using an embedded application from the radio telecommunication domain, and its performance speedup compared to a monothreaded core is done in section 3. And finally, section 4 concludes the paper by discussing the present results along with future works.

2 AHDAM Chip

AHDAM chip stands for Asymmetric Homogeneous with Dynamic Allocator Manycore chip. It is used as an on-chip accelerator component for high-end massively parallel dynamic embedded applications. Depending on the computation requirements, it can be used as a shared accelerator for multiple host CPUs, or a private accelerator for each host CPU. The host CPU is running an operating system or a bare-metal application. When a host CPU encounters a massively-parallel application, it sends an execution demand to AHDAM chip and wait for its acknowledgment. Then, the host CPU offloads the massively-parallel application to AHDAM chip. The application is already decomposed into concurrent tasks. AHDAM chip exploits the parallelism at the thread level (TLP) and loop level (LLP).

In this section, we illustrate AHDAM chip's programming model that supports the control-flow and streaming execution models (section 2.1). Then, we describe the overall architecture as well as the functionalities and interoperabilites between the hardware components in section 2.2. A typical execution model will be presented.

2.1 Programming Model

The programming model for AHDAM chip architecture is specifically adapted to dynamic applications and global scheduling methods. It is based on a streaming programming model. The chip's asymmetry is tackled on 2 levels: a fine-grain level and a coarse-grain level. The proposed programming model is based on the explicit separation of the control and the computing parts. As depicted in Figure 1, each sequential application is parallelized semi-automatically using the *PAR4ALL* tool from HPC Project [6]. It is manually cut into different tasks through pragmas from which explicit execution dependencies are extracted (TLP). Then, the generated parallel application follows a second path in *PAR4ALL*, where OpenMP pragmas are inserted at the beginning of possibly parallelized 'for-loop' blocks (fine-grain). In fact, OpenMP [7] is a method of loop parallelization (LLP) whereby the master thread forks a specified number of slave threads, and a task is divided among them. Then, the child threads run in parallel, with the runtime environment allocating threads to different cores. The *PAR4ALL* tool supports AHDAM chip Hardware Abstraction Layer (HAL) for proper tasks generation. *PAR4ALL* generates as output the computing tasks and the control task that are extracted from the application, so as each task is a standalone program. The greater the number of independent and parallel tasks that are extracted, the more the application can be accelerated at runtime, and the application pipeline balanced.

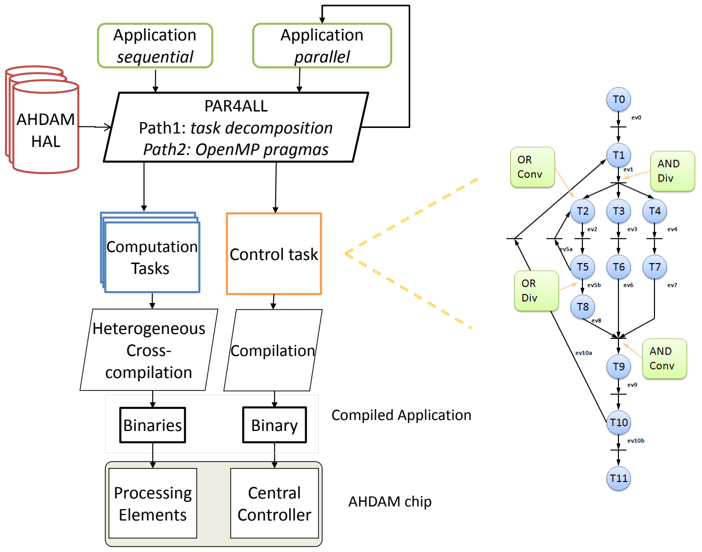


Fig. 1. AHDAM programming model and an example of a typical CDFG control graph

The control task is a Control Data Flow Graph (CDFG) extracted from the application (Petri Net representation), which represents all control dependencies between the computing tasks (coarse-grain). The control task handles the computing task scheduling and activations. A specific compilation tool is used for the binary generation from the CDFG generated by the *PAR4ALL* tool.

For the computing tasks, a specific HAL is provided to manage all memory accesses and local synchronizations, as well as dynamic memory allocation and management capabilities. A special on-chip unit called MCMU (Memory Configuration and Management Unit) is responsible for handling these functionalities (more details in section 2.2). With these functions, it is possible to carry out local control synchronizations or to let the control manager taking all control decisions. Concurrent tasks can share data through local synchronizations handled by the MCMU (streaming execution model). Each task is defined by a task identifier, which is used to communicate between the control and the computing parts. A task suspends/resumes its execution based on data availability from other tasks. It follows the producer/consumer execution model (streaming). When a data is produced by Task A, then Task B resumes its execution. When data is consumed by Task B, then it suspends its execution. Each task has the possibility to dynamically allocate or deallocate buffers (or double buffers) in the shared memory space through specific HAL functions. An allocated buffer is released when a task asks for it and is the last consumer. A buffer cannot be released at the end of the execution of the owner task. A dynamic right management of buffers enables a dataflow execution between the tasks: it is handled by the MCMU.

Once each application and thread has been divided into independent tasks, the code is cross-compiled for each task. For heterogeneous computing resources, the generated code depends on the type of the execution core.

2.2 Architecture Description

The AHDAM chip architecture separates control from computing tasks. This separation rends the architecture more transistor/energy efficient, since the tasks are executed on dedicated resources. AHDAM chip is composed of 3 main units: Memory units, control unit, and computation units, as shown in Figure 2. AHDAM chip has M Tiles, where a Tile represents a computation unit. In the following sections, we will describe in more details the functionality of each unit.

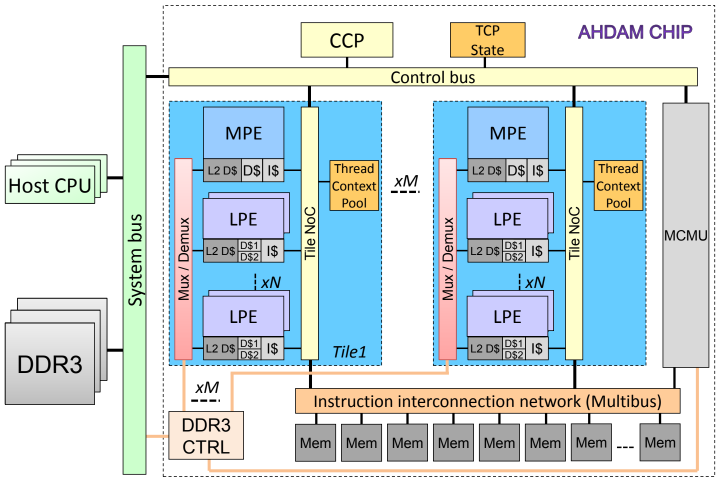


Fig. 2. AHDAM chip architecture

Memory Units. The AHDAM chip memory hierarchy is composed of a separated L2 instruction memory and data cache memory.

The instruction memory is a shared on-chip multi-banked SRAM memory that stores the codes of the tasks. The code size for all the tasks is known statically for a given application, thus the instruction memory size can be well dimensioned. In addition, the instruction memory is a shared memory, which is suitable for inter-tile task migration and load-balancing. Besides, it is implemented as a multi-banked memory instead of a single-banked multiple Read/Write ports memory, which is proven to be more efficient according to CACTI 6.5 tool [8]. In addition to a better area occupation, the multi-bank memory generates less contention per bank when multiple Tiles are accessing simultaneously the instruction memory. This happens when the instruction codes for the tasks are stored in different memory banks.

On the other hand, since we cannot know in advance the application data set size, we implement a L2 data cache memory instead of an on-chip SRAM memory. Cache memories have a bigger area and are less area/energy efficient than SRAM memories. But caches facilitate the programmability since the memory accesses to the external DDR3 memory are transparent to the programmer and independent from the data set size. This eliminates the need for explicit data prefetching using DMA engines, which hardens the tasks decomposition and synchronization as it happens with the IBM Cell processor [9] for instance. All the L2 data cache memories are connected to an on-chip DDR controller, which transfers the data memory access requests to the off-chip DDR3 memories.

A special unit called MCMU (Memory Configuration and Management Unit) handles the memory configuration for the tasks. It divides the memory into pages. In addition, MCMU is responsible of managing the tasks' creation and deletion of dynamic data at runtime, and synchronizing their access with other tasks. There is one allocated memory space per data. A data identifier is used by tasks to address them. Each task has a write exclusive access to a data buffer. Since all the tasks have an exclusive access to data buffers, the data coherency problems are eliminated without the need for specific coherency mechanisms. A data access request is a blocking demand, and another task can read the data when the owner task releases its right. Multiple readers are possible even if the memory latency will increase with the number of simultaneous accesses.

The *Instruction interconnection network* connects the M Tiles to the multi-banked instruction memory. It is a **multibus**. According to the author [10], the multibus occupies less die area that other types of NoCs for small to medium interconnections, and has less energy consumption and memory access latency. The shared on-chip instruction memory is the last level of instruction memory. As we will see later in section 2.3, the execution model assumes that the instructions are already prefetched in the instruction memory.

Control Unit. In the AHDAM chip, the CCP (Central Controller Processor) controls the task prefetching and execution. The application CDFG is stored in dedicated internal memories. The CCP is a programmable solution that consists of an optimized processor for control, which is a small RISC 5-stage, in-order, and scalar pipeline core. Thus, the RTOS functionalities are implemented in software. In addition, the CCP has special interfaces from receiving/sending interruption demands to the computation units.

Computation Units. The AHDAM chip supports M Tiles. The CCP views a Tile as 1 computation unit. But actually, a Tile has one MPE (Master Processing Element) and N LPEs (Loop Processing Element). In addition, it has a special scratchpad memory called *Thread Context Pool* that stores the thread contexts to be processed by the LPEs. The *Thread Context Pool* represents the tasks runqueue per Tile, thus AHDAM chip has M runqueues. The occupation status of all the Tiles' *Thread Context Pool* are updated in a special shared memory unit called the *TCP state*. The *TCP state* is shared by all the Tiles.

The MPE is the Master PE that receives the execution of a coarse-grain task or master thread from the CCP. It is implemented as a monothreaded processor with sufficient resources (ALUs, FPUs, etc...) for executing the tasks' serial regions. On the other hand, the LPE or Loop PE, is specialized in executing child threads that represent loop regions. The LPEs are implemented as blocked multithreaded VLIW processors with 2 hardware thread contexts (TC). In fact, the blocked multithreaded processor increases the LPE's utilization by masking the long access to the off-chip DDR3 memory that stalls the processors.

Each MPE and LPE has a private L1 I$, L1 D$, and L2 D$. For the multi-threaded LPE, the L1 I$ is shared by both TCs, while the L1 D$ is segmented per TC. In this way, we privilege the execution of 2 child threads from the same parent thread, while limiting their interferences on the data memory level. The *Tile NoC*, which is a set of multiple busses interconnecting all the units to each other and to the external world (control and memory busses), is responsible of forwarding the cores' request accesses to the corresponding external unit. However, for the memory data accesses, the requests are grouped by a special MUX/DEMUX unit that forwards the data request to the DDR controller, then to the off-chip DDR3 memory. The *Tile NoC* provides one serial connection of the Tile to the external world, which eases the implementation of the Control and Instruction busses.

2.3 Execution Model

In this section, we describe a typical execution model sequence in the AHDAM chip. At the beginning, the AHDAM chip receives an application execution demand from an external host CPU through the *System bus*. The CCP handles the communication. It fetches the application task dependency graph (CDFG), stores it in its internal memory, and checks the next tasks ready to run to be pre-configured by the MCMU. When the MCMU receives a task pre-configuration demand from the CCP, it configures the shared instruction memory space and allocates the necessary free space, then it fetches the tasks instruction codes from the off-chip DDR3 memory using an internal DMA engine, and finally it creates internally the translation tables. At this stage, the CCP is ready to schedule and dispatch the next tasks to run on available computation units through the *Control bus*.

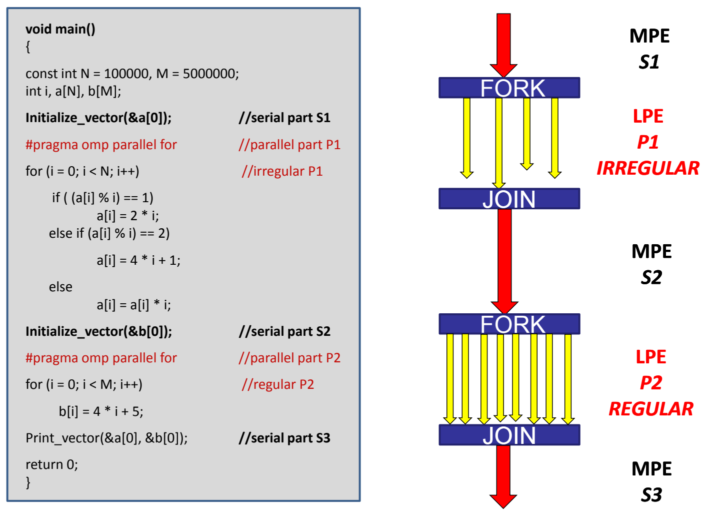
Each task has serial regions and parallel regions. The parallel regions are the parallelized loop codes using a fork-join programming model such as OpenMP pragmas. For instance, let us consider the code example shown in Figure 3. It consists of 3 serial regions (S1,S2,S3) and 2 parallel regions (P1,P2). The thread execution is processed in 4 steps: 1) executing the serial region 2) forking the child threads 3) executing the child threads in parallel 4) joining the child threads.

Fork: The MPE executes the serial region of the task (S1). When it encounters a loop region using OpenMP pragmas (P1), the MPE executes a scheduling algorithm that uses a heuristic to fork the exact number of child threads in the appropriate Tiles' *Thread Context Pool*. The scheduling algorithm is part of a modified OpenMP runtime. The heuristic determines the maximum number of parallel child threads required to execute the loop as fast as possible based on: 1) the data set size 2) the number of cycles to execute one loop iteration 3) the Tiles' *Thread Context Pool* occupation using the shared *TCP State* memory 4) the cost of forking threads in the local and other Tiles. If possible, the algorithm favors the local *Thread Context Pool* since the fork and join process are done faster by avoiding the access to multiple busses. However, in some cases, the local *Thread Context Pool* is full or not sufficient while the ones in other Tiles are empty. Therefore, the local MPE has the possibility of forking the child threads in others *Thread Context Pool* by verifying their availability using the shared *TCP state* memory. This can be the case for the parallel region P2 in Figure 3.

Execute: Then, each LPE (Loop PE) TC executes one child task instance from the local *Thread Context Pool* until completion. Forked parallel child threads are executed in a *farming model* by the LPEs. As soon as a LPE is idle, it spins on the local *Thread Context Pool* semaphore trying to fetch another child thread context. This type of execution model reduces the thread scheduling time and improves the LPEs occupation rate. In addition, it optimizes the execution of irregular for-loops. In fact, some for-loops have different execution paths that render their execution highly variable as shown in parallel region P1 in Figure 3. This scheduling architecture resembles the SMTC (Symmetric Multi-Thread Context) scheduling model, which has been shown to be the best scheduling architecture for multiple multithreaded processors [11].

In AHDAM, we implement a fork-join model with synchronous scheduling: the master thread forks the child threads, then waits to join until all the child threads have finished their execution. Therefore, during the execution of the parallel child threads, the MPE is in a dormant mode and is not preemptable by the CCP. There are 2 advantages from using this execution model: 1) the LPEs have a full bandwidth to the memory and are not disturbed by the MPE execution 2) easier 'join' process.

Join: When a child thread finishes execution, it sends a message to the corresponding MPE. The MPE waits until all the child threads have finished execution to join the process and continue execution of the serial region (S2).



```
void main()
{
const int N = 100000, M = 5000000;
int i, a[N], b[M];
Initialize_vector(&a[0]);           //serial part S1
#pragma omp parallel for            //parallel part P1
for (i = 0; i < N; i++)             //irregular P1
     if ( (a[i] % i) == 1)
             a[i] = 2 * i;
     else if (a[i] % i) == 2)
             a[i] = 4 * i + 1;
     else
             a[i] = a[i] * i;
Initialize_vector(&b[0]);           //serial part S2
#pragma omp parallel for            //parallel part P2
for (i = 0; i < M; i++)             //regular P2
       b[i] = 4 * i + 5;
Print_vector(&a[0], &b[0]);         //serial part S3
return 0;
}
```

Fig. 3. A task code example of serial and parallel regions using OpenMP pragmas

3 Evaluation

In this part, we provide a case study scenario in order to evaluate the AHDAM chip performance. The AHDAM chip is simulated in a modified SESAM framework [12], which is a SystemC framework for modeling and exploration of asymmetric MPSoC architectures. SESAM supports the AHDAM programming model already discussed in section 2.1. It also has a wide range instruction-set simulators (ISS) including monothreaded MIPS1 and MIPS32, and cycle-accurate multithreaded MIPS1 [13]. The latter implements the blocked multithreading protocol and has 2 thread contexts. In this framework, MPEs are implemented as monothreaded MIPS32 24K with FPU, while LPEs are implemented as 2-threaded 3-way VLIW processor (1 ALU, 1 FPU, 1 ld/st). In fact, the performance results of the monothreaded 3-way VLIW are extracted from Trimaran simulator [14] due to its high simulation speed compared to the RTL model, then the results are injected in SESAM simulator. The CCP is a 32-bit monothreaded 5-stage RISC processor similar to MIPS1 R3000.

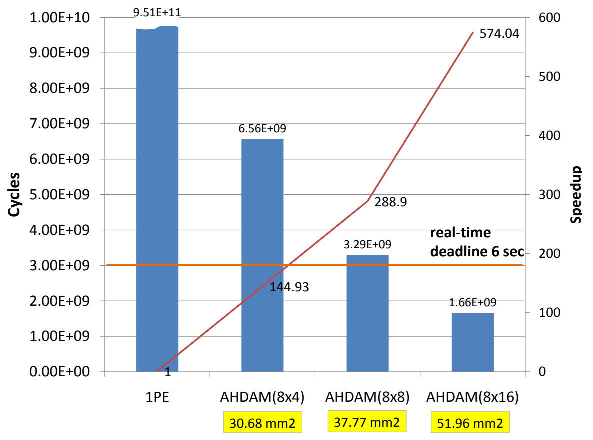
We choose an application that meets the future high-end embedded applications requirements as discussed in section 1. It is a radio spectrum sensing application from the telecommunication domain. This component, which can be found in cognitive radios, is developed by Thales Communications France (TCF). The radio spectrum sensing application sweeps the overall radio spectrum to detect unused spectrums. If an unused spectrum is found, it establishes a communication. We conducted a profiling on a modified version of the application, which provides a high degree of scalability for platform testing (number of cores, etc...) and is not fully optimized for sensing processing. This application has been developed within the SCALOPES project. The application is characterized by its high computation requirements and its adaptive reconfiguration (**dynamism**). The application supports different execution modes. For our evaluation, we choose the following options: high-sensitivity (frequency sample 102.4 MHz), 6 buffers, 100 ms buffer size every 1 sec. This gives us a computation requirement of *75.8 GOPS*, a data set of *432 MB*, and a real-time deadline of *6 seconds*. In addition, we examined the hot spots in the code where most of the application time is spent, and we noticed that 99.8 % of the loop regions can be parallelized by OpenMP.

Initially, the radio sensing application is built to run sequentially on a monothreaded processor. The task level parallelism is explicitly expressed by inserting specific pragmas. Then, *PAR4ALL* cuts the application in a set of tasks according to these pragmas, generates communication primitives to implement a double buffer streaming processing, and the corresponding CDFG control graph. For this paper, 19 tasks are generated. Once independent tasks are generated, *PAR4ALL* identifies netloops and inserts OpenMP pragmas. The loop parallelism is detected during runtime depending on the resources occupation. Also some loops are irregular, which means a variable execution time between the child threads.

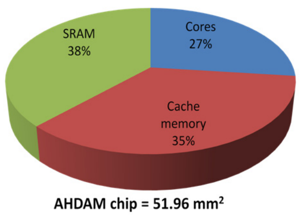
We run the radio-sensing application on 2 processing systems running at 500 MHz:

- Sequential version: 1 MIPS32 24K processor with a FPU, and a sufficient on-chip memory for data and instructions (432 MB). The memory access time to the on-chip memory is 10 cycles, as well as the L2$ memory. The processor has 4KBI and 8KBD L1$, and a 32KB L2$.
- Parallel version: AHDAM chip that is configured with 8 Tiles and 4/8/16 LPEs per Tile. The MPE has 4KBI and 8KBD L1$, while the LPE has 1KBI and 2KBD (1KB per TC) L1$, and a 32KBD L2$. The on-chip instruction memory is equal to 1.5 MB, which is the total size of the tasks' instructions and stack memories. The memory access to the on-chip instruction memory takes 10 cycles, as well as the L2 D$. For the off-chip DDR3 memory, the access time is 50 cycles.

In Figure 4, we compare the execution time of these processing systems. The results show that the AHDAM chip with 4 LPE/tile has a speed-up of 145 compared to a single-processor system. And since the application has lot of LLP, the acceleration goes up to 574 for 16 LPE/tile. In fact, Trimaran compiler optimizations contribute with the LLP exploitation to this high speed-up factor. It is clear that only AHDAM(8x16) with 136 processors is able to meet the real-time deadline constraints of 6 seconds for the radio-sensing application. In addition, we estimated the chip size (processors + memories) at 40 nm technology of both processor systems giving the radio-sensing application conditions. The cache memories and SRAM memories are estimated using CACTI 6.5 tool. As for the processors, we synthesized the multithreaded VLIW LPE and the CCP, while the MPE core area is given at MIPS website. AHDAM(8x4) with 40 processors has an estimated die area of 30.67 mm^2, while AHDAM(8x16) with



(a)



(b)

Fig. 4. a) Execution time of radio-sensing application on 1 PE v/s AHDAM chip with 8 Tiles and 4/8/16 LPEs on 6 buffers. The real-time deadline is 6 seconds b) Surface repartition of AHDAM with 8 Tiles and 16 LPE/tile at 40 nm technology

136 processors is 51.92 mm^2, which is only 69.2% bigger. The surface repartition of AHDAM(8x16) is shown in Figure 4(b). We can notice that the computing cores take 27% of the overall die area, which is quite a good number compared to recent MPSoC architectures. In fact, the key design parameter taken in AHDAM design is to reduce the size of the on-chip memory and integrate instead more efficient processors for computation.

4 Conclusion

This paper has presented an asymmetric homogeneous with dynamic allocator manycore architecture, called AHDAM. The AHDAM chip is designed to tackle the challenges and requirements of future high-end massively parallel dynamic embedded applications. The AHDAM chip units are chosen to increase the transistor/energy efficiency.

We presented the AHDAM chip programming model and its automatic compilation tool that takes as input a sequential application, then decomposes it into multiple parallel independent tasks (TLP), and finally inserts OpenMP pragmas at the loop level (LLP). To study its performance, we used the radio spectrum sensing application from the telecommunication domain. We simulated the execution of sequential and parallel versions of the application on 2 platforms: a single processor with sufficient on-chip memory for the application data set, and the AHDAM chip with a variable number of processors. The results show that the application on the AHDAM chip with 136 processors has an execution time 574 times faster than on the single-processor system, while meeting the real-time deadline and occupying 51.92 mm^2. The speed-up is almost scalable with respect to the number of LPEs since the application has lot of LLP.

For future enhancements, we would like also to compare AHDAM with other MPSoC architectures excluding the GPU architecture. In fact, with AHDAM chip, we are targeting applications that are highly dynamic with lot of TLPs. This execution model is not adequate with the GPU execution model, thus not comparable with AHDAM chip. In addition, we aim to explore AHDAM runtime environment and the heuristics for forking the child threads between the Tiles.

Acknowledgements. Part of the research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 100029. The authors would like to thank Florian Broekaert from Thales Communications France for providing the radio-sensing application. Also, we would like to thank Beatrice Creusillet, Ronan Keryel and Pierre Villalon from HPC Project for providing the *PAR4ALL* compilation tool.

References

1. Duranton, M., Yehia, S., De Sutter, B., De Bosschere, K., Cohen, A., Falsafi, B., Gaydadjiev, G., Katevenis, M., Maebe, J., Munk, H., Navarro, N., Ramirez, A., Temam, O., Valero, M.: The HiPEAC Vision. HiPEAC Network of Excellence (2010)

2. Lacassagne, L., Zavidovique, B.: Light speed labeling: efficient connected component labeling on RISC architectures. Journal of Real-Time Image Processing, 1–19 (2009), doi:10.1007/s11554-009-0134-0
3. Wall, D.W.: Limits of instruction-level parallelism. In: Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Santa Clara, USA (April 1991)
4. Bertogna, M., Cirinei, M., Lipari, G.: Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. IEEE Transactions on Parallel and Distributed Systems 20(4), 553–566 (2008)
5. Ventroux, N., David, R.: The SCMP architecture: A Heterogeneous Multiprocessor System-on-Chip for Embedded Applications. Eurasip (2009)
6. HPC Project: PAR4ALL tool, `http://hpc-project.com/pages/par4all.htm`
7. OpenMP, `http://www.openmp.org`
8. Muralimanohar, N., Balasubramonian, R.: CACTI 6.0: A Tool to Model Large Caches
9. Riley, M.W., Warnock, J.D., Wendel, D.F.: Cell Broadband Engine processor: Design and implementation. IBM Journal of Research and Development 51(5), 545–557 (2007)
10. Guerre, A., Ventroux, N., David, R., Merigot, A.: Hierarchical Network-on-Chip for Embedded Many-Core Architectures. In: 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip (NOCS), pp. 189–196 (May 2010)
11. Bechara, C., Ventroux, N., Etiemble, D.: Comparison of different thread scheduling strategies for Asymmetric Chip MultiThreading architectures in embedded systems. In: 14th Euromicro Conference on Digital System Design (DSD 2011), Oulu, Finland (September 2011)
12. Ventroux, N., Sassolas, T., David, R., Blanc, G., Guerre, A., Bechara, C.: SESAM extension for fast MPSoC architectural exploration and dynamic streaming applications. In: 2010 18th IEEE/IFIP on VLSI System on Chip Conference (VLSI-SoC), pp. 341–346 (September 2010)
13. Bechara, C., Ventroux, N., Etiemble, D.: Towards a Parameterizable cycle-accurate ISS in ArchC. In: IEEE International Conference on Computer Systems and Applications (AICCSA), Hammamet, Tunisia (May 2010)
14. Middha, B., Gangwar, A., Kumar, A., Balakrishnan, M., Ienne, P.: A Trimaran based framework for exploring the design space of VLIW ASIPs with coarse grain functional units, pp. 2–7 (2002)

FPGA Implementation of the Robust Essential Matrix Estimation with RANSAC and the 8-Point and the 5-Point Method

Michał Fularz*, Marek Kraft, Adam Schmidt, and Andrzej Kasiński

Poznań University of Technology, Institute of Control and Information Engineering,
Piotrowo 3A, 60-965 Poznań, Poland
Marek.Kraft@put.poznan.pl

Abstract. This paper presents a FPGA-based multiprocessor system for the essential matrix estimation from a set of point correspondences containing outliers. The estimation is performed using two methods: the 8-point and the 5-point algorithm, and complemented with robust estimation. The description of the architecture and the hardware-specific design considerations are given. Performance and resource use depending on the chosen method and the number of processing cores are also given.[1]

Keywords: FPGA, robust estimation, essential matrix, multicore.

1 Introduction

The problem of essential matrix estimation has been studied in the literature for decades, but is still a field of active research. This is because of wide spectrum of the possible practical applications depending on the knowledge of the essential matrix. Such applications include finding the relative orientation and position between cameras (used in visual odometry), finding the 3D coordinates of matched image points (allowing for 3D reconstruction), camera calibration etc. The input data for the essential matrix estimation algorithms are the image points matched across two overlapping views of the same scene. Such points must be however registered with a calibrated camera, i.e. inner camera parameters must be known in order to achieve the normalization. Field programmable gate array (FPGA) technology has in recent years evolved and became the tool that can successfully be used to tackle the challenges created by advanced image processing tasks [3][7][4]. The possibility to freely form parallel, pipelined architectures that are tailored to the requirements of the application at hand allows

[1] This project is supported by the Polish Ministry of Science and Higher Education, project number N N514 213238.

* Michał Fularz is the scholarship holder within the project "Scholarship support for Ph.D. students specializing in majors strategic for Wielkopolska's development", Sub-measure 8.2.2 Human Capital Operational Programme, co-financed by European Union under the European Social Fund.

for the creation of fast, power efficient embedded systems. In this paper, we present a FPGA based system for robust estimation of the essential matrix. In order to achieve it, we use the 8-point and the 5-point algorithm along with the RANSAC (random sample consensus) robust estimation framework running on a multicore system based on the Microblaze processors [13] with the dedicated coprocessor.

2 Robust Estimation of the Essential Matrix

This section describes the essential processing stages and the basic concepts underlying the implemented essential matrix estimation method.

2.1 The Fundamental and Essential Matrices

In computer vision, the fundamental matrix $\mathbf{F}$ is a matrix that relates corresponding points (described by homogeneous image coordinates) in two images of the same scene (a stereo pair). It is a 3×3 rank 2 matrix. Let us now denote the homogeneous coordinates of the projection of a real-world point to image one by $\mathbf{x}$, and the homogeneous coordinates of the projection of the same point on a different image of the same scene by $\mathbf{x}'$. The homogeneous coordinates of each corresponding point pair $\mathbf{x} \leftrightarrow \mathbf{x}'$ are related with the fundamental matrix $\mathbf{F}$. The relation is described by the so called *epipolar constraint* (1) [2], [6].

$$\mathbf{x'^T F x} = 0 \tag{1}$$

Any rank two matrix is a potential fundamental matrix, and it must satisfy the *cubic constraint* (2).

$$det(\mathbf{F}) = 0 \tag{2}$$

The essential matrix $\mathbf{E}$ is a specialization of the fundamental matrix. It is also a 3×3 rank two matrix that relates the locations of the projections of a real-word point on two images of the same scene. The difference is that in the case of the essential matrix $\mathbf{E}$ the cameras are assumed to be calibrated beforehand, so that we deal with the calibrated coordinate system. The relation between the essential matrix and the fundamental matrix is given in (3) [2], [6].

$$\mathbf{E} = \mathbf{K'^T F K} \tag{3}$$

The matrices denoted by $\mathbf{K}$ and $\mathbf{K}'$ are the camera matrices for both scene views, returned as the result of camera calibration. For the monocular case, in which both of the images of the scene are registered using the same camera, $\mathbf{K} = \mathbf{K}'$. The essential matrix has two equal, non-zero singular values. The algebraic constraint resulting from this fact is given in (4).

$$2\mathbf{E}\mathbf{E}^T\mathbf{E} - trace(\mathbf{E}\mathbf{E}^T)\mathbf{E} = 0 \tag{4}$$

The fundamental matrix allows for the 3D reconstruction with the accuracy up to a projective transformation, while the essential matrix allows for the reconstruction accurate up to a scale. The matrices can be computed using a set of corresponding points in two views of the same scene $\mathbf{x} \leftrightarrow \mathbf{x}'$. In our work, we are especially interested in the essential matrix, as it can be decomposed into the relative rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$.

The methods of computation of the essential matrix are a field of active research and numerous methods have been developed over the years. The common starting point for most of those method is given below.

As mentioned before, the essential matrix fulfills the epipolar constraint (1) if the corresponding points are normalized, i.e. transformed by multiplying them by the respective intrinsic camera matrices. The components of the equation (1) can be written down as:

$$\mathbf{E} = \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \mathbf{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}.$$

From this, for a single pair of corresponding points we get:

$$\begin{aligned} x'xe_{11} + x'ye_{12} + x'e_{13} + y'xe_{21} + y'ye_{22} + \\ +y'e_{23} + xe_{31} + ye_{32} + e_{33} = 0 \end{aligned} \tag{5}$$

By putting the elements of the matrix $\mathbf{E}$ in a column vector (row after row order) we can rewrite the equation for a single point pair (5) in the form given in (6).

$$\begin{bmatrix} x'x \; x'y \; x' \; y'x \; y'y \; y' \; x \; y \; 1 \end{bmatrix} \mathbf{e} = 0 \tag{6}$$

By stacking such equations for n point pairs, we get a system of equations given in (7).

$$\mathbf{Ae} = \begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{bmatrix} \mathbf{e} = 0 \tag{7}$$

Common to all the methods is the fact, that they use n observed point pairs in the epipolar constraint. In other words, we use a system of n equations of the form given in (7) to impose the linear constraints on the essential matrix. The names of the methods are derived from the minimum number of point pairs required to compute the essential matrix.

2.2 The 8-Point Algorithm

The 8-point algorithm is a classical method for the estimation of the essential (but also the fundamental) matrix. The essential matrix has 9 elements, but it is defined up to an unknown scale, so that 8 point pairs are sufficient to find a solution by using (7). If $rank(\mathbf{A}) = 8$, the solution is unique (up to a scale) and can be found by linear methods. In practice, this is rarely the case. The presence of image noise and quantization effects affect the accuracy of keypoint localization, so that the method of choice for finding the solution is the least squares method. The most common approach is to perform the singular value decomposition (SVD) of the matrix $\mathbf{A}$ ($SVD(\mathbf{A}) = \mathbf{UDV}^T$). The least squares solution is then the right singular vector corresponding to the smallest singular value.

2.3 The 5-Point Algorithm

Whenever the intrinsic camera parameters are known (i.e. the camera has been calibrated) the required number of point pairs can be reduced to 5 by deriving additional equations from the cubic (2) and the essential 4 constraints. Moreover, according to [10] enforcing the intrinsic calibration constraints improve the accuracy of the ego-motion and structure reconstruction. Additionally, the 5-point algorithm is also significantly more robust against the coplanarity of the analyzed points.

In this algorithm the matrix A (7) defines 5 linear constraints. The singular value decomposition is used to calculate the 4-dimensional null-space of the equation system. The essential matrix is a linear combination of four singular vectors corresponding to the zero singular values:

$$e = xe_1 + ye_2 + ze_3 + we_4 \tag{8}$$

where e_i are the vectors spanning the null-space, and x, y, z and w are some scalars. As the essential matrix can be estimated only up to scale the w is set to 1. Substituting e into the cubic (2) and the essential (4) constraints gives 10 third-degree polynomial equations consisting of the 20 monomials. The monomials ordered in the GrLex order form a monomial vector X and the equation system can be rewritten as:

$$MX = 0 \tag{9}$$

where M is a 10×20 matrix. According to [11] it is possible to solve this system of polynomial equations by defining a Gröbner basis from the constraints and using it to construct a $10x10$ action matrix. The Gröbner basis is obtained by the Gauss-Jordan elimination of (9):

$$\begin{bmatrix} I & B \end{bmatrix} X = 0 \tag{10}$$

Afterwards, the action matrix is created by extracting appropriate columns of the eliminated $\begin{bmatrix} I & B \end{bmatrix}$ matrix. The solutions of the equation system are encoded in the left eigenvectors of the action matrix corresponding to the real eigenvalues.

The detailed description of the 5-point algorithm using the Gröbner basis is beyond the scope of this paper and can be found in [11]. However, it is important to notice that the solution is obtained using standard linear algebra algorithms and is numerically more stable than the variant presented in [10] which requires finding the roots of the 10th degree polynomial. Other methods of solving the 5-point relative pose problem are described e.g. in [9], [1] and [8].

2.4 Robust Estimation with RANSAC

In real-life applications, the pairs of points $\mathbf{x} \leftrightarrow \mathbf{x}'$ used as the input data for computing the essential matrix are detected and matched automatically, using dedicated algorithms. Figure 1 gives an example output of automatic image feature detection and matching algorithm.

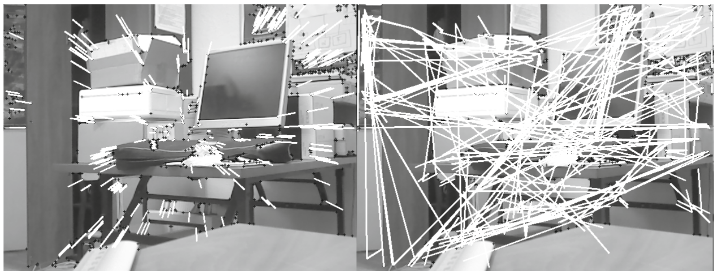


Fig. 1. Illustration of automatic feature detection and matching performance

The black crosses in the image indicate the features detected in the frame. The lines represent the displacement of the features from the previous (reference) frame to the current frame. The figure on the left contains only inliers – the tracks of the features converge towards the center of the image, as the frames were registered with the camera moving forward. The right image contains the outliers registered during the same matching procedure. It is evident, that the incorrectly matched point pairs (outlier) percentage is considerable.

As multiple point pairs are necessary to compute the essential matrix, the probability of selecting an incorrectly matched point pair as a part of the input data may be relatively high, even for a seemingly low outlier percentage. To avoid the corruption of results by invalid measurement data, robust estimation algorithms are used. In computer vision, the most commonly used method of robust estimation is the RANSAC algorithm and its variations.

RANSAC (ang. *Random SAmple Consensus*) is an iterative algorithm, that enables correct estimation of parameters of a mathematical model (the essential matrix in our case) from a set of measurement data (matched points) containing some percentage of outliers (false matches) ([5]). The algorithm is nondeterministic – the correct result is given with some probability. To perform robust estimation with RANSAC , we proceed as follows:

- randomly choose a sample s from a set of measurement data S; compute the model parameters $\hat{M}$ using this sample
- check the cardinality of a set of data consistent with the model $\hat{M}$ with an error less than Δ (cardinality of the set S_i, which is a subset of S)
- if the cardinality of S_i is higher than some threshold T, return $\hat{M}$ as the result (optionally, compute the model parameters using the data from S_i)
- if the cardinality of S_i is lower than T, repeat all the steps given above
- after N tries, choose the S_i with the highest cardinality and return the corresponding model $\hat{M}$ as the result (optionally, compute the model parameters using the data from the chosen S_i)

The least number of samples N that must be tested to get a model that is not corrupted with the outliers is given in equation (11). The term s stands for sample size, w is the probability that a randomly chosen element of S is an inlier (a correct match in our case), $\epsilon = 1 - w$ is the probability, that the a randomly chosen element of S is an outlier. The desired probability of getting the correct result is p.

$$N = \frac{\log(1-p)}{\log(1-(1-\epsilon)^s)} \tag{11}$$

There are numerous criteria for testing the consistency of the matched point pairs with the essential or fundamental matrix. The consistency measure, along with the user-selected threshold, is used as the Δ term in the RANSAC framework. For our implementation, we chose to use the the Sampson error (a first order approximation of the geometric error) (12).

$$\Delta_i = \frac{(\mathbf{x_i'}^T \mathbf{E} \mathbf{x_i})^2}{(\mathbf{E}\mathbf{x_i})_1^2 + (\mathbf{E}\mathbf{x_i})_2^2 + (\mathbf{E}^T\mathbf{x_i'})_1^2 + (\mathbf{E}^T\mathbf{x_i'})_2^2} \tag{12}$$

The $(\mathbf{E}\mathbf{x_i})_j$ stands for the j-th element of the vector. The Sampson error is widely used in similar applications, as it is a good compromise between accuracy and computation speed [6]. In the specific case of the essential matrix estimation, the execution time of the algorithm depends strongly on the quality of the matches. As the percentage of outliers ϵ grows, the number of samples that must be tested to get a correct result (with the desired probability) grows rapidly in a nonlinear manner. The minimum size of the sample is another factor, that has a very strong influence on the minimum number of hypotheses that need to be tested. Table 1 compares the minimum number of samples N that need to be tested at a fixed probability $p = 0.99$ for a sample size s of five and eight (as used the five- and the eight-point algorithm).

While the eight-point algorithm has the benefit of simplicity, the five-point algorithm is faster in real-life applications. The process of model generation takes more time in the case of the five-point algorithm, but the number of required RANSAC iterations is significantly smaller, especially when the percentage of outliers grows [9][10]. The hypothesis testing step is the same for both algorithms.

Table 1. Minimum number of samples N that need to be tested at a fixed probability $p = 0.99$ for a given sample size s

sample	outlier percentage ϵ				
size	10%	20%	30%	40%	50%
$s = 5$	6	12	26	57	146
$s = 8$	9	26	78	272	1177

3 Description of the Implemented System

The basic structure of the system is split into the part that is responsible for the dataflow and the blocks used to perform the computations. The control core (ConCore) consists basically of a single MicroBlaze soft-core microprocessor equipped with a fast local memory composed of the dedicated FPGA Block-RAM (BRAM) resources (256kB). Depending on the configuration, up to four processing cores (ProcCore) can be connected to the ConCore. The ProcCore cores are also MicroBlaze microprocessors, but in this case they are configured to perform computations, thus they are equipped with an IEEE-754 compatible single precision floating point unit (FPU). The ProcCores perform the task of hypothesis generation, both in the case of the 8-point and the 5-point algorithm implementations. As the hypothesis generation processes are independent from one another, the use of a few independent ProcCores results in almost proportional increase of processing speed due to the parallelization. Depending on the algorithm executed, the RAM configuration for ProcCores differs. The 8-point algorithm implementation requires significantly less program memory, so that it can be executed from fast, internal memory composed of BRAM blocks (64kB each). The 5-point algorithm is more memory consuming and requires the use of the external DDR3 RAM along with a dedicated multi-port memory controller (MPMC) created using FPGA resources. To speed up the operations using external memory, the ProcCores used within the 5-point algorithm implementation were equipped with 16kB data cache and 16kB instruction cache. The ConCore randomly selects a minimum sample (5 or 8 matched point pairs) from the dataset and sends it to the currently available ProcCore. Upon the completion of the hypothesis generation process, the ProcCore sends the results (the computed candidate essential matrix) to the ConCore. Upon receiving the candidate essential matrix from any of the ProcCores, the ConCore sends it to the dedicated coprocessor, along with the dataset that is to be tested against the candidate hypothesis. The testing of the hypotheses (candidate essential matrices) against the measurement set is considered a time consuming operation, but the computations required to perform this operation follow a simple data-flow scheme. The complexity results from the necessity of repeating the test for each interest point pair and the potentially large number of hypotheses that need to be tested. The formula given by equation (12) is therefore a good candidate for hardware implementation. Individual operations in the formula are easily decomposed to form a pipelined structure, fully utilizing the spatial and temporal parallelism enabled by the use of flexible, dedicated hardware. To provide fast communication between the elements of the system, the FSL (*Fast Simplex Link*) interface was used ([12]). FSL is a dedicated, fast, unidirectional point-to-point interface with FIFO buffering. Moreover, the Microblaze microprocessor can service the communication via FSL with dedicated instructions, which results in transfer speed increase. FSL communication is faster than communication using the available system buses (PLB or AXI). The block diagram of the system is given in Figure 2. The complete system contains additional support peripherals – a serial port for communication and a timer for profiling.

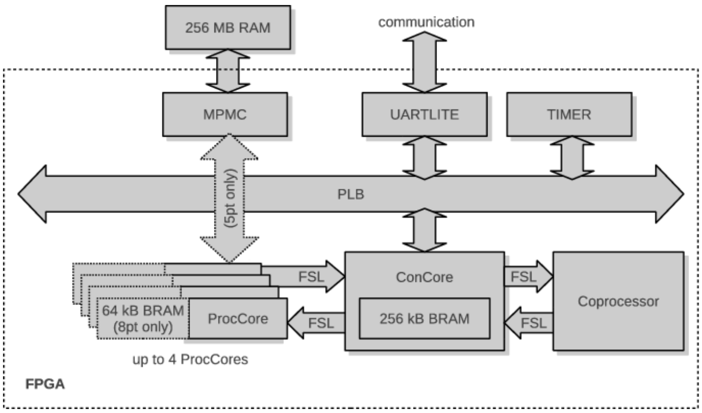


Fig. 2. Block diagram of the multiprocessor system

The microprocessor-based cores (ConCore, ProcCores) use single precision 32 bit format for floating point number representation. The Sampson error coprocessor uses 23-bit floating point number representation. The decision to switch to a reduced representation was made to reduce the use of hardware resources, e.g. the floating point multiplication of numbers in 23-bit format requires only a single dedicated DSP48E block, whereas multiplying two single precision (32-bit) floating point numbers requires three such blocks. For a detailed comparison of both floating point formats see Figure 3.

As mentioned before, the candidate essential matrix to be tested (nine floating point values) is sent to the coprocessor via FSL and stored in its internal

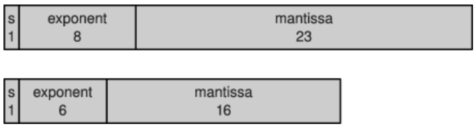


Fig. 3. Comparison of IEEE-754 single precision format (top) and reduced precision format used by the coprocessor (bottom); numbers stand for the number of bits; ’s’ stands for sign

registers for further processing. Afterwards, the consecutive point pair coordinates are transmitted to the coprocessor for testing against the currently stored candidate. The results (the error values for each point pair) are transmitted back to the microprocessor using FSL. As the Microblaze is a 32-bit microprocessor, and the FSL channels are 32-bit wide, the input data is converted into 32-bit floating point format before storing. The output data is converted back to the 32-bit single precision floating point format compatible with the FPU used in Microblaze-based ConCore and ProcCores. Both the input and the output data are transmitted in word by word order. The coprocessor consists of two main blocks – the controller block and the computational block. The controller is used to arrange and convert the incoming and outgoing data. The computational block is an exact implementation of the equation 12 using 23-bit floating point arithmetic blocks. The block diagram of the coprocessor is given in Figure 4. For the block diagram of the computational part of the coprocessor see Figure 5.

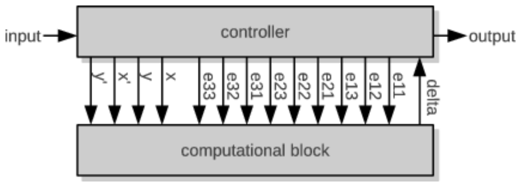


Fig. 4. Block diagram of the coprocessor

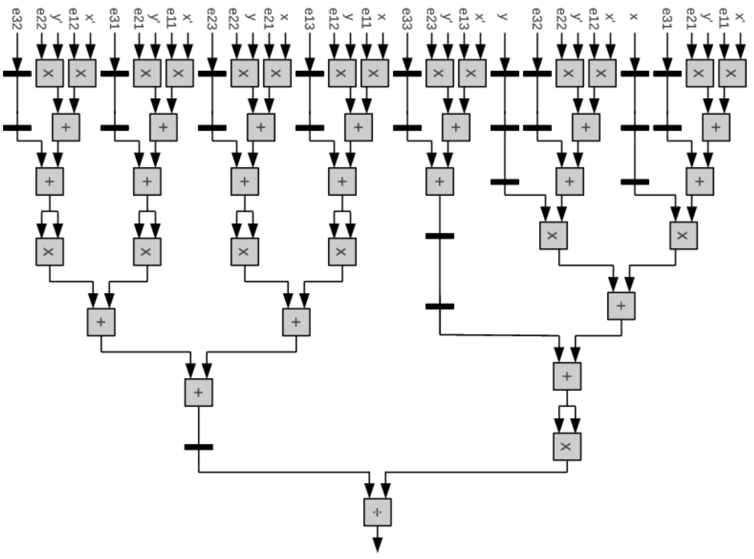


Fig. 5. Block diagram of the computational block; thick lines across datapaths denote delay balancing registers

To implement the hardware, we have used a XC6VLX240T Virtex-6 FPGA from Xilinx clocked at 100 MHz. This is a rather sizable device, but the results presented in the next section show, that the system could easily fit in a smaller, less expensive FPGA.

4 Results and Discussion

The tests were performed on a dataset consisting of 363 pairs of corresponding points. The points were detected and matched automatically by using the Harris corner detector and SSD (sum of squared differences) point neighborhood similarity measure on two images of the same scene. The camera used to register the image was calibrated beforehand and the point of interest image coordinates were normalized before passing to the system. As the RANSAC algorithm used in the estimation process of essential matrix is nondeterministic, a fixed scenario was used for comparison of processing speed against alternative platforms. The test consisted of 1000 algorithm iterations (model generation and testing against measurement data set), each iteration operated on the minimum required set of point pairs randomly selected from the dataset. The average execution time over 1000 runs for each configuration with the 8-point and the 5-point algorithm are given in Table 2

Table 2. Average procesing time (in miliseconds) for a single algorithm iteration – TPI stands for time per iteration, TPI/C stands for time per iteration for a single core

number of	**8-point**		**5-point**	
ProcCores	**TPI**	**TPI/C**	**TPI**	**TPI/C**
1	2.07	2.07	15.21	15.21
2	1.19	2.39	9.61	19.23
4	0.74	2.95	8.03	32.11

As mentioned before, the candidate essential matrices can be independently generated on multiple cores. Therefore, the achieved speedup should be almost proportional to the number of processors. The operations that cannot be performed in parallel are linked mainly with dataflow control and hypothesis testing and take only a small fraction of processing time compared to the hypothesis generation step. Hence, according to the Amdahl's law, they do not cause a significant slowdown of operation. This is the case for the implementation of the 8-point algorithm, as each ProcCore uses an independent, dedicated memory block. In the case of the implementation of the 5-point algorithm all ProcCores share the external memory through MPMC, which creates a bottleneck in the system and decreases performance. The reported power consumption for a system with 4 ProcCores is around 6 W. A software implementation of the algorithm on a machine with Intel Atom N270 1.6 GHz microprocessor with 1 GB RAM requires 0.59 ms for single iteration of the 8-point algorithm, and 1.53 ms for the 5-point algorithm. A single iteration of the 5-point algorithm takes more time, but consulting Table 1 reveals, that for a higher expected outlier percentage the 5-point algorithm may prove to be more time-efficient. Additionally, the 5-point algorithm is reported to return results that are more accurate, and is more robust to degenerate configurations of matched points [11]. The Table 3 gives the amount of resources (BlockRAM block (BRAM), flip-flops(FF), FPGA lookup tables (LUT) and the DSP blocks (DSP48E)) used to implement the 8-point and 5-point variants with a different number of ProcCores.

Table 3. The relation between resource usage and the number of ProcCores. The values in percent are given with respect to all corresponding resources available in the device.

Resource	**1 × ProcCore**	**2 × ProcCore**	**4 × ProcCore**
	8-point algorithm implementation		
BRAM	44 (11%)	53 (13%)	71 (17%)
FF	9372 (3%)	11322 (4%)	15222 (5%)
LUT	14617 (10%)	17601 (12%)	23569 (16%)
DSP48E	31 (4%)	36 (5%)	46 (6%)
	5-point algorithm implementation		
BRAM	96 (23%)	108 (26%)	132 (32%)
FF	16185 (5%)	18480 (6%)	23070 (8%)
LUT	18773 (12%)	22245 (15%)	29189 (19%)
DSP48E	31 (4%)	36 (5%)	46 (6%)

Clearly, the implementation of the 5-point algorithm requires more FPGA resources for implementation. This is due to the fact, that it uses a dedicated controller to make use of the external memory, as the code and data for ProcCores cannot fit in the internal FPGA memory blocks. The resource use shows, that the designs could easily be ported to a smaller, less expensive, lower power FPGA, and that it shows a good scalability potential. Adding additional ProcCores is possible, but only to a certain extent – the Microblaze microprocessor can service only up to 16 FSL channels, so only 3 more ProcCores can be added to a single ConCore.

5 Conclusions

The article presents the implementation of the essential matrix estimation algorithms in a single FPGA, using normalized feature correspondences detected in the image sequence as the input data. The use of flexible programmable hardware and specialized point-to-point interfaces allows to freely tailor the architecture of the system to the task at hand. This allows to achieve good performance in a low power, small footprint device. The results provided by the described device can be used further, e.g. for ego-motion estimation or 3D reconstruction. The availability of low power, small footprint solution performing the described task is desirable in applications such as unmanned aerial vehicles, small mobile robots, driver assistance systems etc. The results have been already tested for correctness, but more functional tests are needed to evaluate the usability of the system in real life scenarios. The authors also plan to integrate other worked out image processing blocks (e.g. feature detection and matching) within the same FPGA. The preprocessing and feature detection algorithms benefit significantly from hardware implementation, so the overall system performance as compared to analogous implementation on a PC will surely improve.

References

1. Batra, D., Nabbe, B., Hebert, M.: An alternative formulation for five point relative pose problem. In: Proceedings of the IEEE Workshop on Motion and Video Computing, pp. 21–26. IEEE Computer Society Press, Washington, DC, USA (2007)
2. Cyganek, B., Siebert, J.: An Introduction to 3D Computer Vision Techniques and Algorithms. Wiley (2009)
3. Draper, B.A., Beveridge, J.R., Bohm, A.P.W., Ross, C., Chawathe, M.: Accelerated image processing on FPGAs. IEEE Transactions on Image Processing 12(12), 1543–1551 (2003)
4. Farrugia, N., Mamalet, F., Roux, S., Yang, F., Paindavoine, M.: Fast and robust face detection on a parallel optimized architecture implemented on FPGA. IEEE Transactions on Circuits and Systems for Video Technology 19(4), 597–602 (2009)
5. Fischler, M.A., Bolles, R.C.: Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM 24(6), 381–395 (1981)
6. Hartley, R.I., Zisserman, A.: Multiple View Geometry in Computer Vision, 2nd edn. Cambridge University Press (2004) ISBN: 0521540518
7. Jin, S., Cho, J., Pham, X.D., Lee, K.M., Park, S.K., Kim, M., Jeon, J.W.: FPGA design and implementation of a real-time stereo vision system. IEEE Transactions on Circuits and Systems for Video Technology 20(1), 15–26 (2010)
8. Kukelova, Z., Bujnak, M., Pajdla, T.: Polynomial eigenvalue solutions to the 5-pt and 6-pt relative pose problems. Proceedings of the British Machine Vision Conference (2008)
9. Li, H., Hartley, R.: Five-point motion estimation made easy. In: Proceedings of the 18th International Conference on Pattern Recognition, ICPR 2006, vol. 1, pp. 630–633. IEEE Computer Society Press, Washington, DC, USA (2006)
10. Nistér, D.: An efficient solution to the five-point relative pose problem. IEEE Transactions Pattern Analysis Machine Intelligence 26, 756–777 (2004)
11. Stewénius, H., Engels, C., Nistér, D.: Recent developments on direct relative orientation. ISPRS Journal of Photogrammetry and Remote Sensing 60, 284–294 (2006)
12. Xilinx: Fast Simplex Link (FSL) Bus (v2.11c) Data Sheet (April 2010)
13. Xilinx Inc.: UG081 MicroBlaze Processor Reference Guide – Embedded Development Kit EDK 12.4, v11.4 edn. (November 2010)

Using Free Scheduling for Programming Graphic Cards

Wlodzimierz Bielecki and Marek Palkowski

Faculty of Computer Science, West Pomeranian University of Technology,
70210, Zolnierska 49, Szczecin, Poland
{bielecki,mpalkowski}@wi.zut.edu.pl
http://kio.wi.zut.edu.pl/

Abstract. An approach is presented permitting us to build free scheduling for statement instances of affine loops. Under the free schedule, loop statement instances are executed as soon as their operands are available. To describe and implement the approach, the dependence analysis by Pugh and Wonnacott was chosen where dependences are found in the form of tuple relations. The proposed algorithm has been implemented and verified by means of the Omega project software. Results of experiments with the NAS benchmark suite are discussed. Speed-up and efficiency of parallel code produced by means of the approach are studied. Problems to be resolved in order to enhance the power of the presented technique are outlined.

Keywords: fine-grained parallelism, free scheduling, parameterized affine loops, NVIDIA cards.

1 Introduction

Microprocessors with multiple execution cores on a single chip are typical computation platforms now. The lack of automated tools permitting for exposing parallelism for such systems decreases the productivity of programmers and increases the time and cost of producing a parallel program.

Because most computations are contained in program loops, automatic extraction of parallelism from loops is extremely important for multi-core systems, allowing us to produce parallel code from existing sequential applications and to create multiple threads that can be easily scheduled to achieve high program performance.

Given a loop, a schedule of loop statement instances is a function that assigns a time of execution to each loop statement instance preserving all dependences in this loop. There have been developed numerous approaches to form loop statement instance scheduling, for example [7,9,12,13,18].

Under the free schedule, loop statement instances are executed as soon as their operands are available that permits us to extract all fine-grained parallelism available in the loop, but well-known techniques based on linear or affine schedules do not guarantee finding free scheduling for non-uniform loops.

In this paper, we present a novel technique that permits for building free scheduling for both uniform and non-uniform loops. It is based on forming the exact transitive closure of a dependence relation describing all dependences in a loop. Experimental results with the NAS benchmarks suite demonstrate that the approach can be successfully applied to produce parallel programs for NVIDIA graphic cards.

2 Background

A discussed algorithm deals with *static-control loop nests*, where lower and upper bounds as well as conditionals and array subscripts are affine functions of symbolic parameters and surrounding loop indices. A *statement instance* $s(I)$ is a particular execution of a statement s of the loop for some loop iteration I.

Two statement instances $s_1(I)$ and $s_2(I)$ are dependent if both access the same memory location and if at least one access is a write. Provided that $s_1(I)$ is executed before $s_2(I)$, $s_1(I)$ and $s_2(I)$ are called the *source* and *destination* of the dependence, respectively. The sequential execution ordering of statement instances, denoted as $s_1(I) \prec s_2(J)$, is induced by the lexicographic ordering of iteration vectors and the textual ordering of statements when the instances share the same iteration vector.

Definition [11]. The free schedule is the function that assigns statement instances (for execution) as soon as their operands are available, that is, it is mapping $\sigma{:}LD \rightarrow \mathbb{Z}$ such that

$$\sigma(p) = \begin{cases} 0 \; if \; there \; is \; no \; p' \in LD \; s.t. \; p' \prec p \\ 1 + max(\sigma(p')), p' \in LD, p' \prec p \end{cases}$$

i.e., it is the fastest schedule, where *p*, *p'* are loop statement instances, *LD* is the loop domain.

The approach to find free scheduling, presented in this paper, requires an exact representation of dependences. To describe the approach and carry out experiments, we have chosen the dependence analysis proposed by Pugh and Wonnacott [25] where dependences are represented with dependence relations. This analysis is implemented in Petit [14] which returns a set of dependence relations describing all dependences in a loop. A dependence relation is a tuple relation of the form $\{[$*input list*$] \rightarrow [$*output list*$]$: *constraints*$\}$, where *input list* and *output list* are the lists of variables used to describe input and output tuples and *constraints* is a Presburger formula describing the constraints imposed upon *input list* and *output list* [20].

The general form of a dependence relation is as follows [25]:

$$\{[s_1, ..., s_k] \rightarrow [t_1, ..., t_{k'}] | \bigvee_{i=1}^{n} \exists \alpha_{i1}, ..., \alpha_{im_i} s.t. F_i\},$$

where F_i, i=1, 2, ..., n, are conjunctions of affine equalities and inequalities on the input variables $s_1, ..., s_k$, the output variables $t_1, ..., t_{k'}$, the existentially quantified variables $\alpha_{i1}, ..., \alpha_{im_i}$ and symbolic constants, k, k', m, n are integers.

An *ultimate dependence source* (resp. *destination*) is a source (resp. destination) that is not the destination (resp. source) of another dependence. A set, *UDS*, comprising all ultimate dependence sources can be found as domain(R)-range(R), where R represents all dependences in a loop.

Given dependence relations R_1, R_2,..., R_m, our approach requires first preprocessing these relations according to the procedure presented in [2]. Preprocessing makes the sizes of input and output tuples of dependence relations to be the same as well as inserts identifiers of loop statements in the last position of input and output tuples (this permits for applying the union, composition, and difference operations to relations describing dependences).

Positive transitive closure for a given relation R, R^+, is defined as follows $R^+ = \{[e] \rightarrow [e'] : \ e \rightarrow e' \in R \wedge \exists(e'' : \ e \rightarrow e'' \in R^+ \wedge e'' \rightarrow e' \in R)\}$.

Transitive closure, R^*, is defined as follows: $R^* = R^+ \cup I$, where I the identity relation. Details concerning these operations can be found in [16].

3 Finding Free Scheduling for Parameterized Loops

The idea of the algorithm presented in this section is as follows. Given preprocessed relations R_1, R_2, ..., R_m, we firstly calculate $R = \bigcup_{i=1}^{m} R_i$. Next we create the relation $R^{'}$ by inserting variables k and and k+1 into the first position of the input and output tuples of relation R; variable k is to present the time of a partition (a set of statement instances to be executed at time k). Next, we calculate the transitive closure of relation $R^{'}$, $R^{'*}$, and form the following relation

$FS = \{[X] \rightarrow [k,\ Y] : X \in UDS(R) \wedge (k,\ Y) \in \text{Range}((R^{'})^{*}\backslash\{[0,\ X]\}) \wedge \neg(\exists\ k^{'}{>}k \text{ s.t. } (k^{'},\ Y) \in \text{Range}(R^{'})^{+}\backslash\{[0,\ X]\})\}$,

where $(R^{'})^{*}\backslash\{[0,\ X]\}$ means that the domain of relation $R^{'*}$ is restricted to the set including ultimate dependences sources only (elements of this set belong to the first time partition); the constraint $\neg(\exists\ k^{'}{>}k \text{ s.t. } (k^{'},\ Y) \in \text{Range}(R^{'})^{+}\backslash\{[0,\ X]\})$ guarantees that partition k includes only those statement instances whose operands are available, i.e., each statement instance will belong to one time partition only.

It is worth to note that the first element of the tuple representing the set Range(*FS*) points out the time of a partition while the last element of that exposes what is the statement whose instance(iteration) is defined by the tuple elements 2 to n-1, where n is the number of the tuple elements of a preprocessed relation. Taking the above consideration into account and provided that the constraints of relation *FS* are affine, the set Range(*FS*) is used to generate parallel code applying any well-known technique to scan its elements in the lexicographic order, for example the techniques presented in papers[1,26].

The outermost sequential loop of such code scans values of variable k (representing the time of partitions) while inner parallel loops scan independent instances of partition k. Techniques aimed at calculating the transitive closure of dependence relations are presented in papers [4,5,16] and are out of the scope of this paper.

Finally, we expose independent statement instances, that is, those that do not belong to any dependence and generate code enumerating them. According to the free schedule, they are to be executed at time k=0.

Below we present the algorithm that realizes the presented above idea in a formal way.

Algorithm: Finding free scheduling for statement instances of a parameterized loop
Input: Preprocessed relation R describing all dependences in a loop.
Output: Code representing free scheduling.
Method:

1. Transform relation R of the form $\{[X] \rightarrow [Y] : \mathit{constraints}\}$, where X and Y are vectors representing the input and output tuple variables, respectively, to relation R' of the form $\{[k, X] \rightarrow [k{+}1, Y] : \mathit{constraints} \wedge k \geqslant 0\}$.
2. Calculate $(R')^{+}$ using any of known techniques, for example those presented in papers [4,5,16].
3. Form the following relation FS:
   $FS = \{[X] \rightarrow [k, Y] : [X] \in UDS(R) \wedge [k, Y] \in \mathrm{Range}((R')^{*}\backslash\{[0, X]\}) \wedge \neg(\exists\ k'{>}k \text{ s.t. } [k', Y] \in \mathrm{Range}(R')^{+}\backslash\{[0, X]\})\}$.
4. Generate code scanning elements of the set Range(FS). For this purpose, apply any well-known algorithm, for example that published in [14]. The outermost sequential loops of this code scan time partitions while the inner parallel loops scan instances to be executed in a particular partition.
5. Find set, IND, containing independent statement instances:
   $IND = IS$ - (domain(R) $\cup$ range(R)),
   where IS represents the union of preprocessed iterations sets of all loop statements. Generate code scanning elements of set IND. For this purpose, apply any well-known algorithm, for example that published in [14]. This code is to be executed at time k=0.

Let us illustrate the presented algorithm by means of the following imperfectly nested parameterized loop.

Example 1

```
for(i=1; i<=n; i++){
  a[i][0] = 1;   //s1
  for(j=1; j<=n; j++){
     a[i][j] = a[i-1][j] + a[i][j-1];    //s2
  }
}
```

There are the three dependence relations returned by Petit
R1 =$\{[i,-1,1] \rightarrow [i,1,2] : 1 \leqslant i \leqslant n\}$;
R2 =$\{[i,j,2] \rightarrow [i+1,j,2] : 1 \leqslant i < n \ \&\&\ 1 \leqslant j \leqslant n\}$;
R3 =$\{[i,j,2] \rightarrow [i,j+1,2] : 1 \leqslant i \leqslant n \ \&\&\ 1 \leqslant j < n\}$.

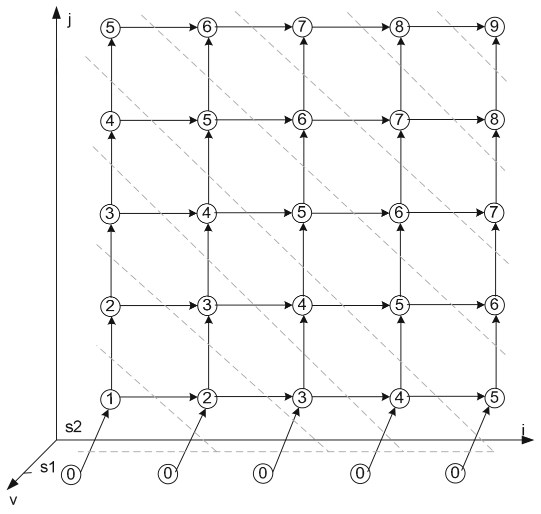


Fig. 1. The free schedule for Example 1 when n=5. The solid lines represent dependences, the dotted lines represent synchronization barriers between time partitions.

Figure 1 presents the free schedule for the loop of Example 1 when n=5.

Applying the presented algorithm, we get the following results being produced by means of the Omega calculator.

1. $R' = \{[k,i,-1,1] \rightarrow [k+1,i,1,2] : 1 \leqslant i \leqslant n \ \&\& \ 0 \leqslant k\} \cup \{[k,i,j,2] \rightarrow [k+1,i+1,j,2] : 1 \leqslant i < n \ \&\& \ 1 \leqslant j \leqslant n \ \&\& \ 0 \leqslant k\} \cup \{[k,i,j,2] \rightarrow [k+1,i,j+1,2] : 1 \leqslant i \leqslant n \ \&\& \ 1 \leqslant j < n \ \&\& \ 0 \leqslant k\}$.
2. $R'^{+} = \{[k,i,j,2] \rightarrow [k',i',i-k+j-i'+k',2] : 1 \leqslant i \leqslant i' \leqslant n \ \&\& \ 0 \leqslant k < k' \ \&\& \ 1 \leqslant j \ \&\& \ k+i' \leqslant i+k' \ \&\& \ i+j+k' \leqslant n+k+i'\} \cup \{[k,i,-1,1] \rightarrow [k',i',i-k+k'-i',2] : 1 \leqslant i \leqslant i' \leqslant n \ \&\& \ k+i' < i+k' \ \&\& \ 0 \leqslant k \ \&\& \ i+k' \leqslant n+k+i'\}$.
3. $FS = \{[1,-1,1] \rightarrow [k,i',k-i'+1,2] : 1 \leqslant i' \leqslant k, n \ \&\& \ k < n+i'\} \cup \{[i,-1,1] \rightarrow [0,i,-1,1] : 1 \leqslant i \leqslant n\}$.
4. $Range(FS) = \{[k,i,k-i+1,2]: 1 \leqslant i \leqslant k, n \ \&\& \ k < n+i\} \cup \{[0,i,-1,1]: 1 \leqslant i \leqslant n\}$.
   The loop scannig elements of the set Range(FS) for $k \geq 0$ and being produced by the codegen function of the Omega library is as follows.

```
for(t2 = 1; t2 <= n; t2++) {    // parallel loop
  a[t2][0] = 1;   // s1(0,t2,-1,1);
}
for(t1 = 1; t1 <= 2*n-1; t1++) {
  for(t2 = max(-n+t1+1,1); t2 <= min(n,t1); t2++) {  //parallel loop
     a[t2][t1-t2+1] = a[t2-1][t1-t2+1] + a[t2][t1-t2];
                                                  // s1(t1,t2,t1-t2+1,2);
}}
```

5. $IND = \varnothing$. There is no independent statements in the loop.

The pseudocode above was manually transformed to the parallel code for NVIDIA cards presented below. The main function of this code runs kernels of parallel loops. The value of variable *n_blocks* represents the number of threads that execute a single block of independent loop statement instances, i.e., the number of engaged CUDA cores. The value of variable *idx* defines the identifier of a block; the values of variables *lb* and *ub* indicate the lower and upper bounds of the parallel loop, respectively; variable *packet* is to represent the number of iterations in a block.

```
//Kernel definitions
__global__ void loop1_gpu(float (*a)[n])
{
   int idx = blockIdx.x, t2;
   int packet = (int)ceil(n / blockDim.x);
   int lb = idx*packet+1;
   int ub = ((idx+1)*packet < n) ? (idx+1)*packet : n;

   for(int t2 = lb; t2 <= ub; t2++)
     a[t2][0] = 1;

}

__global__ void loop2_gpu(float (*a)[n], t1)
{
   int idx = blockIdx.x, t2;
   int packet = (int)ceil((max(-n+t1+1,1) - min(n,t1)) / blockDim.x);
   int lb = idx*packet+max(-n+t1+1,1);
   int ub = ((idx+1)*packet < min(n,t1)) ? (idx+1)*packet : min(n,t1);

   for(int t2 = lb; t2 <= ub; t2++)
        a[t2][t1-t2+1] = a[t2-1][t1-t2+1] + a[t2][t1-t2];
}

int main(int argc, char * argv[]){
...
  int threads_per_block = 1;
  int n_blocks = atoi(argv[1]);  // number of CUDA cores

  // Kernel invocation
  loop1_gpu <<< n_blocks, threads_per_block>>> ((float(*)[n])d_A);
  for(t1 = 1; t1 <= 2*n-1; t1++)  {
    loop2_gpu <<< n_blocks, threads_per_block>>> ((float(*)[n])d_A, t1);
...
}
```

4 Experimental Results

The presented algorithm was implemented by us in a tool by means of the Omega library. It generates C-like pseudo-code scanning loop statement instances according to free scheduling. Using this tool, we have experimented with loops of the NAS 3.2 benchmark suite [23].

The NAS Parallel Benchmarks (NPB) have been developed at the NASA Ames Research Centre to study performance of parallel supercomputers. Benchmarks are derived from computational fluid dynamics and include [23]:

- EP - An *embarrassingly parallel* kernel, which evaluates an integral by means of pseudorandom trials.
- MG - Simplified multigird kernel, which solves a 3D Poisson PDE.
- CG - A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix.
- FT - A 3-D partial differential equation solution using FFTs.
- IS - A large integer sort.
- LU - A regular-sparse, block (5x5) lower and upper triangular system solution.
- SP - Solution of multiple, independent systems of non diagonally dominant, scalar, pentadiagonal equations.
- BT - Solution of multiple, independent systems of non diagonally dominant, block tridiagonal equations with a (5x5) block size.
- UA - Unstructured Adaptive, a new kernel solving scientific problems featuring irregular, dynamic memory accesses.
- DC, DT - Data Cube operator and Data Transfer benchmarks.

From 431 loops of the NAS benchmark suite, Petit is able to carry out a dependence analysis for 257 loops only, and it discovers dependences in 133 loops only. For 133 loops qualified for experiments, the tool is able to calculate the transitive closure of dependence relations for 96 loops. Scheduling and generating code are possible for 65 ones (for the rest 31 loops the algorithm fails to produce relation *FS* due to the time out limitation - maximum 10 seconds to produce set *FS*). For 19 from those 65 loops, the algorithm does not expose any parallelism (there exists a single statement instance for each time partition). Therefore parallelism is extracted for 46 loops only.

To assess the efficiency of code produced by the proposed algorithm, the following criteria were taken into account for choosing NAS loops: (i) a loop must be computatively heavy (there are many NAS benchmarks with constant upper bounds of loop indices, hence their parallelization is not justified), (ii) code produced by the algorithm must be parallel (there are NAS loops for which there exists a single statement instance for each time partition), (iii) structures of chosen loops must be different (there are many NAS loops of a similar structure). Applying these criteria, we selected the following NAS loops: *FT_auxfnct_2* (Fast Fourier Transform Benchmark), *UA_diffuse_4* and *UA_transfer_4* (both from Unstructured Adaptive benchmark).

Codes, produced for these loops by the presented approach, were manually converted by us to parallel programs to be executed on an NVIDIA graphic card 8800 GTS, 96 CUDA Cores, 1.6 GHz, GDDR3 512 MB (in the same way as it is done for Example 1). For this purpose, we have used the NVIDIA CUDA library [24].

Table 1. Results for NAS benchmarks

Loop	Upper bounds	data transf. time, s.	time (without data transfer time)			
			1 GPUs	2 GPUs	8 GPUs	96 GPUs
FT_auxfnct_2	N1, N2, N3 = 100	a: 0.00034 s: 0.00626 f: 0.00750 t: 0.01411	0.4632	0.2330	0.0572	0.0088
	N1, N2, N3 = 200	a: 0.00044 s: 0.04988 f: 0.04611 t: 0.09643	3.9193	1.9588	0.4952	0.0872
UA_diffuse_4	N1, N2 = 64; N3, N4 = 10	a: 0.00032 s: 0.00319 f: 0.00167 t: 0.00518	0.1959	0.0979	0.0275	0.0011
	N1,N2 = 128; N3,N4 = 10	a: 0.00035 s: 0.00972 f: 0.00561 t: 0.01567	0.8055	0.3970	0.1064	0.0159
UA_transfer_4	N1, N2 = 1000	a: 0.00032 s: 0.00632 f: 0.00264 t: 0.00928	0.4053	0.2458	0.0717	0.0202
	N1, N2 = 2000	a: 0.00037 s: 0.02510 f: 0.00981 t: 0.03528	1.6222	0.9294	0.2456	0.0476

Table 1 presents results of time measuring (in seconds) for executing the chosen loops on the graphic card. The execution time of a loop consists of the time of data transfer to/from a graphic card and the time of calculations. Experiments were carried out for two different values of the upper bounds of loop indices (see column 2). The time of data transfer (see column 3) comprises the times of [24]: allocation (a), sending data to the graphic card (s), and fetching data memory of the graphic card (f). Column 3 presents also the sum of those times as the time of data transfer (t). It is worth to note that the time of data transfer does not depend on the number of GPU cores [24]. Columns 4-7 show the time of calculations (not including the time of data transfer) for 1, 2, 8, and 96 GPU cores.

Table 2 presents the execution time (the sum of the time of data transfer and the time of calculations), speed-up, and efficiency for different numbers of GPU cores. The results in Table 2 demonstrate that parallel loops formed on the basis of parallel code produced by the algorithm: i) permit for utilizing many GPU cores (up to 96 under our experiments); ii) speed-up increases with increasing the number of GPU cores (up to 96 under our experiments). For loop *UA_diffuse_4*, increasing values of *N1* and *N2* leads to decreasing speed-up for 96 cores. This can be explained by increasing the number of synchronization evens and not

Table 2. Time, speed-up, and efficiency

Loop	Upper bounds	1 GPU	2 GPUs			8 GPUs			96 GPUs		
		time,s	time,s	S	E	time,s	S	E	time,s	S	E
FT_auxfnct_2	N1,N2,N3= 100	0.477	0.247	1.93	0.97	0.118	6.69	0.84	0.023	20.84	0.22
	N1,N2,N3= 200	4.016	2.055	1.95	0.98	0.592	6.79	0.85	0.184	21.86	0.23
UA_diffuse_4	N1,N2=64; N3,N4=10	0.201	0.103	1.95	0.98	0.033	6.15	0.77	0.006	31.87	0.33
	N1,N2=128; N3,N4=10	0.821	0.413	1.99	0.99	0.122	6.72	0.84	0.032	25.99	0.27
UA_transfer_4	N1,N2= 1000	0.415	0.255	1.63	0.81	0.081	5.12	0.64	0.030	14.04	0.15
	N1,N2= 2000	1.657	0.965	1.72	0.86	0.281	5.90	0.74	0.083	20.01	0.21

enough increasing the work running by each core. However, with increasing the number of cores, speed-up increases for the same values of the loop upper bounds.

To evaluate the time complexity of the algorithm presented in this paper, we measured the time that takes the tool, implementing the presented algorithm, from the beginning of a dependence analysis to the end of pseudo code generation on the machine with the following features: Intel Core 2 Duo 2.34 Ghz, 2 GB RAM, Ubuntu Linux.

Table 3. Time complexity of the algorithm

loop	dep. analysis, s	R'^+ and Range(FS), s	Final code, s	Total time, s
FT_auxfnct_2	0.01	0.07	0.01	0.09
UA_diffuse_4	0.01	0.05	0.01	0.07
UA_transfer_4	0.01	0.03	0.01	0.05

In our implementation, Petit was used as the dependence analyser. Calculating R'^+ and the set Range(FS) was carried out by a function written by us, and final code generation was realized on the basis of the Omega library *codegen* function. Table 3 presents the results for the three examined NAS loops. The first column contains the name of a loop while the remaining columns expose time measurement results (in seconds): of dependence analysis, calculating R'^+ and Range(FS), final code generation, and the total time.

Table 4 presents the comparison of time measurements for an Intel Core 2 Duo 2.34 Ghz computer (1 and 2 CPUs) and an NVIDIA 8800 GTS graphic card (1 and 96 CUDA cores). Loop *FT_auxfnct_2* is executed much faster on 96 CUDA cores than on two CPU cores. For the next two loops, free-scheduling introduce high volume of synchronization on two CPU cores that results in negative speed-up, while for the graphic card the speed-up is about 20 for 96 cores.

Based on the presented results, we can conclude that the presented algorithm can be successfully applied for producing parallel programs for many NAS benchmarks to be executed on graphic cards.

Table 4. CPU and GPU times comparison

Loop	Upper bounds	1 CPU	2 CPUs	1 GPU	96 GPUs
FT_auxfnct_2	N1,N2,N3= 100	0.174	0.091	0.477	0.023
	N1,N2,N3= 200	2.081	1.543	4.016	0.184
UA_diffuse_4	N1,N2=64; N3,N4=10	0.010	0.027	0.201	0.006
	N1,N2=128; N3,N4=10	0.038	0.046	0.821	0.032
UA_transfer_4	N1,N2= 1000	0.028	0.050	0.415	0.030
	N1,N2= 2000	0.071	0.101	1.657	0.083

5 Related Work

The approaches published in [8,17,22,21] build an explicit graph of a subset of the iteration space, with each node representing the instance of a statement. Free scheduling can be found by searching the graph or using the transitive closure of the graph, but dependences are restricted to uniform ones and the problem regarding boundary cases exists.

For the case of polyhedral approximations of dependences (including direction vectors), Darte and Vivien's algorithm is optimal but it does not guarantee forming free scheduling for parameterized loops [10].

For affine dependences, the most powerful algorithm for building scheduling is Feautrier's one based on multi-dimensional affine schedules [13]. But as mentioned by Feautrier, it is not optimal for all codes with affine dependences. However, the dimension of the schedules built by Feautrier's algorithm is minimal for each statement of the loop nest [27].

The approaches published in [7,15,19] present different ways of building affine partition mappings, but none of them guarantees producing free scheduling for the general case of loops with affine dependences.

Paper [3] presents a technique permitting for building free scheduling but only for non-parameterized loops.

The approach presented in [6] permits for extracting free scheduling within each synchronization-free slice but it does not produce free scheduling for all loop statement instances.

6 Conclusion and Outlook

In this paper, we presented the algorithm that permits us to build free scheduling for statement instances of parameterized arbitrarily nested loops. The necessary condition to apply it is the possibility of the calculation of the exact transitive closure of a relation describing all dependences in a loop.

There are tasks to be resolved in the future to strengthen the power of the presented algorithm and justify its application for parallelizing real-life codes: 1) developing algorithms and corresponding implementations permitting for automatic NVIDIA code generation from pseudo code produced by the presented algorithm; 2) when a free schedule is represented by non-linear forms, techniques should be developed to generate code enumerating statement instances under such a schedule.

References

1. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 13 IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, pp. 7–16 (September 2004)
2. Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel Computing 37, 479–497 (2011)
3. Beletskyy, V., Siedlecki, K.: Finding Free Schedules for Non-uniform Loops. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 297–302. Springer, Heidelberg (2003)
4. Bielecki, W., Klimek, T., Trifunovic, K.: Calculating exact transitive closure for a normalized affine integer tuple relation. Electronic Notes in Discrete Mathematics 33, 7–14 (2009)
5. Wlodzimierz, B., Tomasz, K., Marek, P., Beletska, A.: An Iterative Algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part I. LNCS, vol. 6508, pp. 104–113. Springer, Heidelberg (2010)
6. Bielecki, W., Palkowski, M.: Extracting Both Affine and Non-linear Synchronization-Free Slices in Program Loops. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 196–205. Springer, Heidelberg (2010)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Conference on Programming Language Design and Implementation, pp. 101–113. ACM (2008)
8. Chen, D.K.: Compiler optimizations for parallel loops with fine-grained synchronization. Ph.D. thesis, Champaign, IL, USA (1994), uMI Order No. GAX95-12325
9. Darte, A., Robert, Y.: Constructive methods for scheduling uniform loop nests. IEEE Trans. Parallel Distrib. Syst. 5, 814–822 (1994)
10. Darte, A., Vivien, F.: Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT 1996, pp. 281–291. IEEE Computer Society, Washington, DC, USA (1996)
11. Darte, A., Khachiyan, L., Robert, Y.: Linear scheduling is nearly optimal. Parallel Processing Letters 1(2), 73–81 (1991)
12. Feautrier, P.: Some efficient solutions to the affine scheduling problem: I. one-dimensional time. Int. J. Parallel Program. 21(5), 313–348 (1992)
13. Feautrier, P.: Some efficient solutions to the affine scheduling problem: II. multidimensional time. Int. J. Parallel Program. 21(5), 389–420 (1992)
14. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Tech. rep., College Park, MD, USA (1995)
15. Kelly, W., Pugh, W.: A framework for unifying reordering transformations. Tech. rep., Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-92-126.1, College Park, MD, USA (1993)

16. Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. Int. J. Parallel Programming 24(6), 579–598 (1996)
17. Krothapalli, V., Sadayappan, P.: Removal of redundant dependences in doacross loops with constant dependences. IEEE Transactions on Parallel and Distributed Systems 2, 281–289 (1991)
18. Le Gouëslier d'Argence, P.: Affine scheduling on bounded convex polyhedric domains is asymptotically optimal. Theor. Comput. Sci. 196, 395–415 (1998)
19. Lim, A.W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, pp. 228–237. ACM Press (1999)
20. Surhone, L.M., Tennoe, M.T., Henssonow, S.F.: Presburger Arithmetic. VDM Verlag Dr. Mueller AG & Co. Kg (2010); ISBN: 6133083557
21. Midkiff, S.P., Padua, D.A.: Compiler algorithms for synchronization. IEEE Transactions on Computers 36, 1485–1495 (1987)
22. Midkiff, S.P., Padua, D.A.: A comparison of four synchronization optimization techniques. In: ICPP (2), pp. 9–16 (1991)
23. NAS: Parallel Benchmarks Suite, Version 3.3 (February 2008), `http://www.nas.nasa.gov`
24. NVIDIA: NVIDIA CUDA C Programming Guide 4.0 (2011), `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`
25. Pugh, W., Wonnacott, D.: An Exact Method for Analysis of Value-Based Array Data Dependences. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1993. LNCS, vol. 768, pp. 546–566. Springer, Heidelberg (1994)
26. Verdoolaege, S.: Integer set library - manual. Tech. rep. (2011), `http://www.kotnet.org/~skimo//isl/manual.pdf`
27. Vivien, F.: On the Optimality of Feautrier's Scheduling Algorithm. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 299–308. Springer, Heidelberg (2002)

GPU Accelerated Computation of the Longest Common Subsequence

Katsuya Kawanami and Noriyuki Fujimoto

1-1 Gakuen-cho, Naka-ku, Sakai-shi, Osaka 599-8531 Japan
{mu301005,fujimoto}@mi.s.osakafu-u.ac.jp

Abstract. The longest common subsequence (LCS for short) for given two strings has various applications, e.g. comparison of DNAs. In this paper, we propose a GPU algorithm to accelerate Hirschberg's CPU LCS algorithm improved using Crochemore et al's bit-parallel CPU algorithm. Crochemore's algorithm includes bitwise logical operators which can be computed in embarrassingly parallel. However, it also includes an operator with less parallelism, i.e. an arithmetic sum. In this paper, we focus on how to implement these operators efficiently in parallel. Our experiments with 2.93GHz Intel Core i3 530 CPU, GeForce 8800 GTX, GTX 285, and GTX 480 GPUs show that the proposed algorithm runs maximum 12.77 times faster than the bit-parallel CPU algorithm and maximum 76.5 times faster than Hirschberg's LCS CPU algorithm. Furthermore, the proposed algorithm runs 10.9 to 18.1 times faster than Kloetzli's existing GPU algorithm.

1 Introduction

There are various metrics of the similarity between two strings, for example, the edit distance. The longest common subsequence [3] (LCS for short) is one of them. LCS can be applied to various problems, e.g., comparison of DNAs, inexact string matching, spell checking, and others.

An algorithm to compute the LCS of given two strings was proposed by Hirschberg [4]. The algorithm recursively computes LCS while computing the length of LCS (LLCS for short) between various substrings of the given two strings. When lengths of given two strings are m and n, Hirschberg's algorithm requires $O(mn)$ time and $O(m+n)$ space. A method to compute LLCS faster with bit-parallelism is well-known. The method requires $O(\lceil m/w \rceil n)$ time and $O(m+n)$ space [1] where w is the word size of a computer. Using this method, Hirschberg's LCS algorithm can be accelerated. However, much faster algorithms are desirable for strings of length more than one million characters, which are common in the field of the comparison of DNAs. So, we consider to accelerate the bit-parallel algorithm with a GPU (Graphics Processing Unit). The bit-parallel algorithm includes bitwise logical operations and arithmetic sums. Bitwise logical operations are suitable for GPU because they can be executed in embarrassingly parallel. However, arithmetic sums have less parallelism. So, we device to compute them efficiently in parallel.

Based on the method proposed in the paper, we implement a bit-parallel LCS algorithm on CUDA [5,8,7,2]. The experiment with GeForce GTX 480 and 2.93GHz Intel Core i3 530 CPU shows our algorithm is 2.1 to 12.77 times faster than the bit-parallel CPU algorithm, and 37.3 to 76.5 times faster than the non-bit-parallel CPU algorithm. Another experiment shows our algorithm is 37.3 to 76.5 times faster than Kloetzli et al's GPU algorithm over the same GPU (GeForce 8800 GTX).

The remainder of the paper is organized as follows. In Section 2, we briefly review the definition of LCS and existing algorithms for a CPU. In Section 3, we present the proposed algorithm. In Section 4, we conduct several experiments to compare our algorithm with the CPU algorithms and the existing GPU algorithm. In Section 5, we give some concluding remarks and show some future works. Due to the limited space, we illustrate neither the architecture nor the programming of GPUs in this paper. Readers unfamiliar with them are recommended the literatures [5,8,7,2].

2 LCS

2.1 The Definition of LCS

Let C and A be strings $c_1c_2\cdots c_p$ and $a_1a_2\cdots a_m$ respectively. In the following, we assume without loss of generality that characters in the same string are different from each other. If there exists a mapping from the indices of C to the indices of A subject to the following conditions C1 and C2, C is called a subsequence of A.

C1: $F(i) = k$ if and only if $c_i = a_k$
C2: If $i < j$, then $F(i) < F(j)$.

However, we define the null string as a subsequence of any string. We define a string which is a subsequence of both string A and string B as a common subsequence between A and B. The LCS between A and B is the longest one of all the common subsequences between A and B. For example, the LCS between "abcdefghij" and "cfilorux" is "cfi". LCSs between "abcde" and "baexd" are "ad", "ae", "bd", and "be".

2.2 How to Compute the Length of the LCS

The LLCS can be computed using the dynamic programming. This algorithm stores the LLCS between A and B in L$[m][n]$ if we fill the table L with $(m+1)\times(n+1)$ cells based on the following rules R1 to R3 where m is the length of A and n is that of B. To fill the table L, this algorithm requires $O(mn)$ time and $O(mn)$ space.

R1: If $i = 0$ or $j = 0$, then L$[i][j] = 0$.
R2: If A$[i-1]$ = B$[j-1]$, then L$[i][j]$ = L$[i-1][j-1] + 1$.
R3: Otherwise, L$[i][j]$ = max(L$[i][j-1]$, L$[i-1][j]$).

Listing 1.1. Hirschberg's LLCS algorithm

```
Input:string A of length m, B of length n
Output:LLCS L[j] of A and B[0...j-1] for all j(0<=j<=n)
llcs(A,m,B,n,L){
  for(j=0 to n)  K[1][j] = 0
  for(i=1 to m) {
    for(j=0 to n)  K[0][j] = K[1][j]
    for(j=1 to n) {
      if(A[i-1] == B[j-1])  K[1][j] = K[0][j-1] + 1
      else  K[1][j] = max(K[1][j-1], K[0][j])
    }}
  for(j=0 to n)  L[j] = K[1][j]
}
```

Listing 1.2. Hirschberg's LCS algorithm

```
Input:string A of length m, B of length n
Output:LCS C of A and B
lcs(A,m,B,n,C) {
  if(n==0)  C = ""(null string)
  else  if(m==1) {
    for(j=1 to n) if(A[0]==B[j-1])  { C = A[0]  return }
    C = ""
  }
  else {
    i = m/2
    llcs(A[0...i-1],i,B,n,L1)
    llcs(A[m-1...i],m-i,B[n-1...0],n,L2)
    M = max{j : 0<=j<=n , L1[j]+L2[n-j]}
    k = min{j : 0<=j<=n , L1[j]+L2[n-j] == M}
    lcs(A[0...i-1],i,B[0...k-1],k,C1)
    lcs(A[i...m-1],m-i,B[k...n-1],n-k,C2)
    C = strcat(C1,C2)
  }}
```

The rules R2 and R3 imply that the ith row ($1 \leq i \leq m$) of L can be computed only with the ith and $(i-1)$th rows. This property leads us to an algorithm which requires less memory, shown in Listing 1.1 [4]. K is a temporary array of size $2 \times (n+1)$ cells. L is an array for storing output of size $1 \times (n+1)$ cells. Hirschberg's LLCS algorithm shown in Listing 1.1 stores the LLCS between a string A and a string B$[0...j-1]$ (the substring of B from the 1st character to the jth character of B) in L$[j]$. This implementation reduces the required space to $O(m+n)$ with the same time complexity $O(mn)$.

2.3 Hirschberg's LCS Algorithm

Listing 1.2 shows the LCS algorithm proposed by Hirschberg [4] where S$[u...l]$ ($u \geq l$) represents the reversal of the substring S$[l...u]$ of a string S. This algorithm recurrently computes an LCS while computing the LLCS. The algorithm requires $O(mn)$ time and $O(m+n)$ space.

Listing 1.3. Crochemore et al's bit-parallel LLCS algorithm

```
 1  Input:string A of length m, B of length n
 2  Output:LLCS L[i] of A[0...i-1] and B for all i(0<=i<=m)
 3  llcs_bp(A,m,B,n,L){
 4    for(c=0 to 255) {
 5      for(i=0 to m-1)
 6        if(c==A[m-i-1]) PM[c][i] = 1
 7        else PM[c][i] = 0
 8    }
 9    for(i=0 to m-1) V[i] = 1
10    for(j=1 to n) V = (V + (V & PM[B[j]])) | (V & ~(PM[B[j-1]]))
11    L[0] = 0
12    for(i=1 to m) L[i] = L[i-1]+(1-V[i-1])
13  }
```

2.4 Computing the LLCS with Bit-Parallelism

There is an efficient LLCS algorithm with bit-parallelism. The algorithm shown in Listing 1.3 is Crochemore et al's bit-parallel LLCS algorithm [1] where V is a variable storing a bit-vector of length m. The notation & represents bitwise AND, | represents bitwise OR, ~represents bitwise complement, and + represents arithmetic sum. Note that, + regards V[0] as the least significant bit. First of all, Crochemore et al's algorithm makes a pattern match vector (PMV for short). PMV P of string S about c is the bit-vector of length m which satisfies following conditions.

C1: If S[i] = c, then P[i] = 1.
C2: Otherwise, P[i] = 0.

In the 4th to 8th lines of Listing 1.3, PMV of string A about each character c is made. Because we assume one byte character, $0 \leq c \leq 255$. Reversals of PMV about c are stored in PM[c] where a variable PM is a two-dimensional bit array of size $256 \times m$.

This algorithm represents the table of dynamic programming as a sequence of bit-vectors such that each bit-vector corresponds to a column of the table. The ith bit of each bit-vector represents the difference between ith cell and $(i-1)$th cell of the corresponding column. Repeating bitwise operations, the algorithm does the process which is equal to computing the table L from left to right.

The last column of the table L output by this algorithm is a bit-vector. However, we can easily convert it into an ordinary array of integer in $O(m)$ time. The converting process is in the 11th to 12th lines in Listing 1.3.

Crochemore et al's algorithm requires $O(\lceil m/w \rceil n)$ time and $O(m+n)$ space. where w is the word size of a computer.

3 The Proposed Algorithm

3.1 The CPU Algorithm to Be Implemented on a GPU

The dominant part of the LCS algorithm shown in Listing 1.2 is llcs(), which computes the LLCS. In the paper, we aim to accelerate the LCS algorithm shown in Listing 1.2 improved with the bit-parallel LLCS algorithm shown in Listing1.3 (in other words, we replace invocation of llcs() in Listing 1.2 with llcs_bp() in Listing 1.3). The algorithm requires $O(\lceil m/w \rceil n + m + n)$ time and $O(m + n)$ space. For this purpose, we propose a method to accelerate the LCS algorithm with a GPU. Even if we use 64 bit mode on a GPU, the length of every integer register is still 32 bits. So, the word size w is 32.

In the 16th line of Listing 1.2, llcs() computes the LLCS between the reversal of string A and the reversal of string B. However, if we reverse A and B in every invocation of llcs(), the overhead of reversing becomes significant. So, we make a new function llcs'() which traverses strings from tail to head. llcs'() is the same as llcs() except the order of string traverse. We also make the bit-parallel algorithm llcs_bp'() corresponding to llcs'().

The output of Hirschberg's LLCS algorithm shown in Listing 1.1 is the mth row of the table L. However, Crochemore et al's algorithm shown in Listing 1.3 represents a column of the table as a bit-vector and computes the table from 0th column to nth column. Hence, Crochemore et al's algorithm outputs the nth column. So, we change the original row-wise LLCS algorithm shown in Listing 1.1 into column-wise algorithm. In addition to this, we have to change the original LCS algorithm shown in Listing 1.2 into another form corresponding to column-wise LLCS algorithm. Because our algorithm embeds 32 characters into one variable of unsigned integer, length of string A must be a multiple of 32. Therefore, when length of string A is not a multiple of 32, we have to pad string A and make its length a multiple of 32. For this padding, we can use characters not included in both of string A and string B (e.g. control characters).

3.2 Outline of the Proposed Algorithm

A GPU computes only llcs() in the 11th to 12th lines of Listing1.2. Other parts of Listing 1.2 are computed by a CPU. The reason is that GPUs support recursive calls only within some levels although the algorithm shown in Listing 1.2 has recursive calls of lcs() in the 15th to 16th lines.

The LLCS algorithm shown in Listing 1.3 includes bitwise logical operators (&, |, ~) and arithmetic sums (+) on bit-vectors of length m. Bitwise logical operators are easily parallelized. However, an arithmetic sum has carries. Because carries propagate from the least significant bit to the most significant bit in the worst case, an arithmetic sum has less parallelism. So, we have to devise in order to extract higher parallelism from the computation of an arithmetic sum. We think to process the bit-vectors of length m in parallel by dividing them into sub-bit-vectors. On a GPU, each bit-vector is represented as an array of unsigned integer. A variable of unsigned integer on a GPU is of size 32 bits. In CUDA architecture, 32 threads in the same warp are synchronized at instruction level (SIMD execution). So, we set the number of threads in one block at 32 so that threads in one block can be synchronized with no cost. Since one thread processes one unsigned integer (32 bits), one block processes 1024 bits.

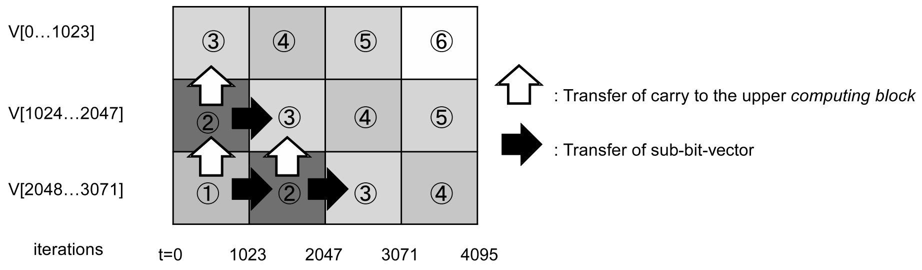


Fig. 1. Block partition in case of $m = 3072$ and $n = 4096$

During one invocation of the kernel function, our algorithm performs only 1024 iterations of n iterations. We call a group of those 1024 iterations one *step* in the remainder of the paper. For example, the jth step represents 1024 iterations from $1024 \times j$ to $1024 \times (j+1) - 1$. We set the number of bits processed in one block and the number of iterations in one invocation at the same value so that each thread can transfer carries by reading from or writing to only one variable.

Fig. 1 is an example of block-step partition in case of $m = 3072$ and $n = 4096$. Each rectangle in this figure represents one step of one block. We call it a *computing block* in the remainder of the paper. Each computing block can not be executed until its left and lower computing blocks have been executed. So, only the lowest leftmost computing block can be executed in the 1st invocation of the kernel function. The computing block is the 0th step of the block covering sub-bit-vector V$[2048 \cdots 3072]$. In the 2nd invocation of the kernel function, both the 1st step of the block covering V$[2048 \cdots 3072]$ and the 0th step of the block covering V$[1024 \cdots 2047]$ can be executed. In each invocation of the kernel function, we execute all computing blocks we can execute at that time. Then, computing blocks with the same number in Fig. 1 can be executed in parallel at the same time (the numbers represent execution order). Based on the above ideas, we invoke the kernel function $(\lceil m/1024 \rceil + \lceil n/1024 \rceil - 1)$ times to get the LLCS.

Black arrows in Fig. 1 indicate that the block covering sub-bit-vector V$[i \cdots i+1023]$ gives the value of V$[i \cdots i + 1023]$ to $(j + 1)$th step of itself at the end of jth step. On the other hand, white arrows indicate that the block covering sub-bit-vector V$[i \cdots i + 1023]$ gives carries in each iteration to jth step of the block covering V$[i-1024 \cdots i-1]$. Transfer of values from a computing block to another computing block, shown as black or white arrows in Fig. 1, is done out of the for-loop invoking the kernel function. During the for-loop, carries are stored in an array on the shared memory. After the loop, carries on shared memory are copied into the global memory. Reading is similar to this. Before the loop, carries on global memory are loaded into the shared memory. During the loop, we use carries on shared memory, not on the global memory.

Listing 1.4. A pseudo code of llcs_kernel() and llcs_gpu()

```
 1  __global__ void llcs_kernel(
 2    int m, n, char *dstr2, unsigned int *g_V, *g_PM, *car, int num) {
 3    index = global thread ID; count = step number of this block;
 4    cursor = 1024 * count; V = g_V[index];
 5    Load carries from car on global memory;
 6    for (j=0 to 1023) {
 7      if(cursor+j >= n) return;
 8      PM = g_PM[dstr2[cursor+j]][index];
 9      V = (V & (~PM)) | (V + (V & PM));
10    }
11    g_V[index] = V;
12    Save carries to car on global memory;
13  }
14
15  void llcs_gpu(
16    char *A, *B, int m, n, char *dstr1, *dstr2, int *out,
17    unsigned int *g_V, *car, *g_PM) {
18    dstr1 = padded copy of A;  dstr2 = B;
19    num_x = (m+1023)/1024;  num_y = (n+1023)/1024;
20    for (i=0 to ((m+31)/32)-1) g_V[i] = 0xFFFFFFFF;
21    Make PM vectors;
22    for(i=1 to num_x+num_y-1)
23      llcs_kernel() in Parallel on GPU(gridDim=num_x, blockDim=32);
24    for(i=0 to m)
25      out[i] = the amount of zeros from 0th bit to ith bit in g_V;
26  }
```

3.3 The Kernel Function

This section describes the kernel function llcs_kernel() which performs one step (1024 iterations) and the host function llcs_gpu() which calls llcs_kernel(). Listing 1.4 is a pseudo code of llcs_kernel() and llcs_gpu(). llcs_gpu() is a GPU implementation of llcs_bp() shown in Listing 1.3. In addition to them, we make llcs_kernel'() and llcs_gpu'() which is a GPU implementation of llcs_bp'(). However they are quite similar to llcs_kernel() and llcs_gpu(). So, we skip to explain them.

First, we explain the kernel function llcs_kernel(). Argument m and argument n respectively represent the length of string A and string B. dstr2 represents the copy of string B on the global memory. g_V is an array to store the bit-vector V. g_PM is a two-dimensional array to store PMV of string A about each character c ($0 \leq c \leq 255$). g_PM[c] is the PMV of string A about c. Argument car is an array to store carries. When we use the array car, we regard car as two-dimensional array and do double buffering. Argument num represents the number of invocation of llcs_kernel(). num is used to compute which step the block should process in llcs_kernel(). The for-loop in the 6th to 10th lines represents the process of one step (1024 iterations). Transfer of values from a computing block to another computing block, which is shown as black or white arrows in Fig. 1, is done in the 4th to 5th lines and the 11th to 12th lines.

Next, We explain the function llcs_gpu(). llcs_gpu() invokes the kernel function llcs_kernel() ($num\_x + num\_y - 1$) times in the for-loop of the 22nd to 23rd lines. Pre-processing is in the 18th to 20th lines. The string A is padded in the 18th

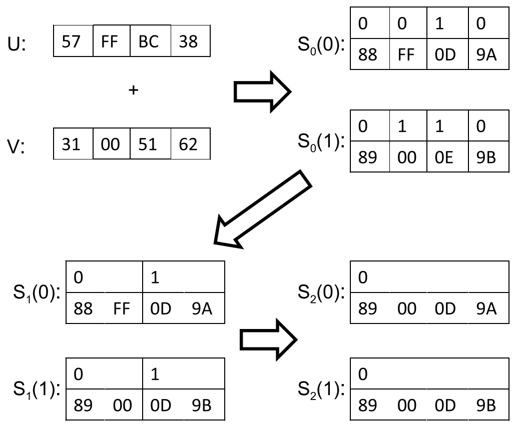


Fig. 2. Parallelization of n-bit full adder (in the case of $n = 32$)

line. The number *num_x* of blocks and the number *num_y* of steps are computed in the 19th line. In the 20th line, all bits of bit-vector V are initialized to one. The 24th to 26th lines are post-processing where bit-vector V is converted into an ordinary array and written in the output-array out.

3.4 Parallelization of an Arithmetic Sum

As we state in Section 3.2, + has less parallelism because it has carries. To parallelize +, we applied Sklansky's method to parallelize the full adder named conditional-sum addition [9].

Sklansky's method uses the fact that every carry is either 0 or 1. To compute the addition of n-bit-numbers, each half adder computes a sum and a carry to the upper bit in the both cases in advance. Then, carries are propagated in parallel. In our algorithm, we use 32 bit width half adders rather than one bit width half adders.

Using an example in Fig. 2, we explain the method to parallelize the n-bit full adder. Fig. 2 shows a 32 bit addition performed by four full adders of 8 bit width. Note that Fig. 2 is illustrative and our actual implementation uses 1024 bit full adders realized by 32 full adders of 32 bit width. In Fig. 2, we compute the sum and carry of U, V, and *l_car* (carry from the lower sub-bit-vector). To compute U+V+l_car, first, we compute one-bytewise sums and carries of each byte. $S_0(0)$ represents one-bytewise sums and carries when a carry from the lower byte is 0. $S_0(1)$ represents them when the carry from the lower byte is 1. Next, we consider computing two-bytewise sums and carries from $S_0(0)$ and $S_0(1)$. To get them, we copy the 3rd byte of $S_0(0)$ into the 3rd byte of $S_0(1)$. The results (two-bytewise sums and carries) are $S_1(0)$ and $S_1(1)$. Similarly, we get four-bytewise sums and carries from $S_1(0)$ and $S_1(1)$. The results are $S_2(0)$ and $S_2(1)$. $S_2(0)$ is U+V (the sums and the carry when a carry from the lower sub-bit-vector is 0). $S_2(1)$ is U+V+1 (the sums and the carry when a carry from the lower sub-bit-vector is 1).

The most important advantage of the method is to be able to execute the computation of $\mathrm{S}_{t+1}(0)$ and $\mathrm{S}_{t+1}(1)$ from $\mathrm{S}_t(0)$ and $\mathrm{S}_t(1)$ 2^t byte by 2^t byte in parallel. When the number of elements in U and V is l, we repeat this process $(log_2 l)$ times to get U+V+l_car.

The implementation is based on the above ideas. Input are two arrays of unsigned integer, which store 1024-bit-vector, and a carry from the lower bit-vector. Output are two arrays of unsigned integer and a carry to the upper bit-vector. The number of elements in one array is 32. In addition to them, we make an array of bool storing carries to the upper element on the shared memory. First, we compute one-bytewise sums for two arrays of unsigned integer. When a sum is smaller than two operands, we set a carry to the upper byte 1. Next, we get two-bytewise sums and carries from neighboring two one-bytewise sums and carries. This process can be done with one comparison and two substitutions. In the same way, we get four-bytewise ones, eight-bytewise ones, and finally 32-bytewise ones . "A carry to the upper byte" of the most significant byte is "a carry to the upper bit-vector." So, we store the carry in the global memory.

3.5 Other Notes

cudaMemcpy() of each array is done before the host function llcs_gpu() and recursive calls in Listing 1.2. Also, cudaMalloc() of each array is also done out of the recursive calls. If they are done in the recursive calls, overhead is too heavy.

When we pad the string on the device, we have to copy the original string from the host. However, transfer across host and device is much slower than device to device transfer or host to host transfer. So, we would like to reduce the number of transfer across host and device as few as possible. To do so, only once we perform host to device transfer to make a copy of the original string on the global memory. In llcs_gpu() or llcs_gpu'(), when we use the string, we copy it into working memories on the device. It is the copy that we pad.

If the lengths of given strings are shorter than some constant value, the cost to make PMV and copy data to device becomes larger than the cost to compute the LLCS with dynamic programming on a CPU. In such a case, execution speed becomes slower when we use a GPU. So, we check the lengths of strings before invoking llcs_gpu(). When the length of A is shorter than 4096, we compute the LLCS with dynamic programming on a CPU.

4 Experiments

In this section, we compare the execution time of the proposed algorithm with the execution times of Kloetzli et al's GPU algorithm, Hirschberg's CPU algorithm, and Crochemore's CPU algorithm. We execute our program on 2.93GHz Intel Core i3 530 CPU and Windows 7 Professional 64bit. Our development environment are CUDA 3.1 and Visual Studio 2008 Professional. We use default compile options in release mode and no SSE instruction is used.

Table 1. The speedup ratio of our algorithm to Hirschberg's CPU algorithm

m (10^6 chars)	n (10^6 chars)	CPU(Hi)	CPU(bp)	GPU(bp) (8800)	GPU(bp) (285)	GPU(bp) (480)
1.16	1.05	1.00	13.21	35.97	51.77	60.87
1.50	1.49	1.00	13.14	41.84	62.25	70.61
1.80	1.32	1.00	13.52	42.41	67.13	74.90
1.80	0.27	1.00	15.05	26.56	32.75	37.34
1.80	0.41	1.00	14.59	30.40	42.15	47.11
1.80	1.46	1.00	13.45	42.84	68.46	76.04
1.80	1.51	1.00	13.40	43.10	69.07	76.50

CPU:2.93GHz Intel Core i3 530, Hi:Hirschberg, bp:bit-parallel,
8800:GeForce 8800 GTX, 285:GeForce GTX 285, 480:GeForce GTX 480

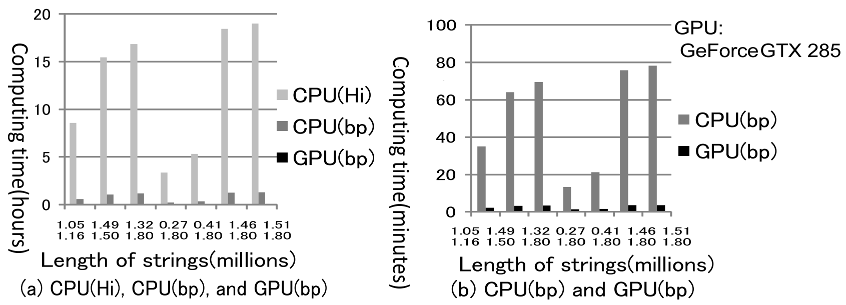


Fig. 3. A comparison of CPU(Hirschberg), CPU(Bit-Parallel), and GPU(Bit-Parallel)

4.1 A Comparison with the Existing CPU Algorithms

Fig. 3 shows the result of comparison among the non-bit-parallel CPU algorithm (CPU(Hi)), bit-parallel CPU algorithm (CPU(bp)), and bit-parallel GPU algorithm (GPU(bp)) over GeForce GTX 285. Fig. 3 (a) includes CPU(Hi), CPU(bp), and GPU(bp). Fig. 3 (b) includes only CPU(bp) and GPU(bp). The x-axes of these two graphs show length of string A and string B measured in millions. The y-axes show execution times. In Fig. 3 (a), times are measured in hours. In Fig. 3 (b), times are measured in minutes. Because we assume targets to be DNA sequences, we use strings of length 0.27 million to 1.8 million in the experiment. In addition, Table 1 shows the speedup ratio of CPU(bp) and GPU(bp) to CPU(Hi). In this table, we execute GPU(bp) on three types of GPUs, i.e., GeForce 8800 GTX, GeForce GTX 285, and GeForce GTX 480.

Fig. 3 and Table 1 show that CPU(bp) runs 14 to 15 times faster than CPU(Hi) in all of seven cases. GPU(bp) runs 2.1 times faster than CPU(bp) in the shortest case of 1.8 million and 0.27 million. In the longest case of 1.8 million and 1.51 million, GPU(bp) runs 5.1 times faster than CPU(bp).

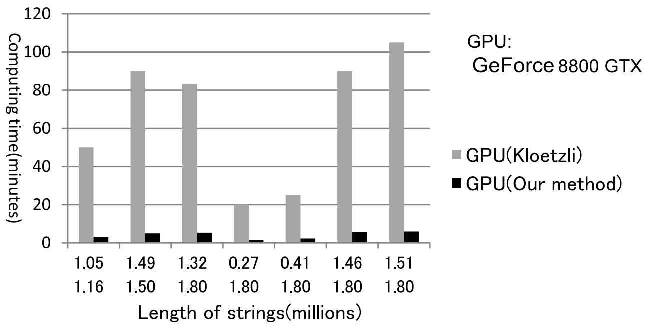


Fig. 4. A comparison with Kloetzli et al's GPU algorithm

Comparing with CPU(Hi), the proposed algorithm (GPU(bp)) runs 26.5 to 43.1 times faster over GeForce 8800 GTX, 32.7 to 69.0 times faster over GeForce GTX 285, and 37.3 to 76.5 times faster over GeForce GTX 480.

4.2 A Comparison with the Existing GPU Algorithm

We compared our algorithm with Kloetzli et al's GPU algorithm. Kloetzli et al used an AMD Athlon 64 CPU and a GeForce 8800 GTX GPU. To Compare over the same GPU, we used GeForce 8800 GTX GPU too. Because our CPU (2.93GHz Intel Core i3 530) is faster than Kloetzli et al's CPU, our environment is not completely equal to Kloetzli et al's. However, our algorithm scarcely depends on CPU. So, we can expect the result does not become quite different.

Fig. 4 shows the result. The x-axis shows the length of strings A and B. The y-axis shows execution times measured in minutes. GPU(Kloetzli) represents the execution time of Kloetzli et al's GPU algorithm. GPU(proposed) represents the execution time of our GPU algorithm.

In the shortest case (1.8 million and 0.27 million), the speedup ratio is 12.0. In the longest case (1.8 million and 1.51 million), the speedup ratio is 17.6. The speedup ratio ranges from 10.9 (1.8 million and 0.41 million) to 18.1 (1.8 million and 1.51 million).

4.3 A Comparison for Longer Strings

We show the result of comparison between the bit-parallel CPU algorithm and the proposed algorithm in Table 2. We set the lengths of given two strings are identical. The lengths are 1MB, 2MB, 4MB, 8MB, and 16MB. The leftmost column shows the speedup ratio of the proposed algorithm to the bit-parallel CPU algorithm. In Table 2, the longer the lengths of strings become, the larger the speedup ratio becomes. For example, the speedup ratio increases from 4.46 to 9.95 with increase of the lengths of strings from 1MB to 8MB. In the longest case of 16MB, the speedup ratio is 12.77.

Table 2. A comparison with the bit-parallel CPU algorithm

String Length(MB)	CPU(bp) (seconds)	GPU(bp) (seconds)	speedup ratio
1	468.55	105.04	4.46
2	1855.39	327.82	5.66
4	7424.83	940.72	7.89
8	29772.05	2992.01	9.95
16	131574.88	10306.66	12.77

CPU:2.93GHz Intel Core i3 530, GPU:GeForce GTX 285

5 Conclusions

In the paper, we have presented a method to implement the bit-parallel LCS algorithm on a GPU and have conducted several experiments on our program based on the method. As a result, the proposed algorithm runs 2.48 to 12.77 times faster than the bit-parallel algorithm on a CPU and 10.9 to 18.1 times faster than Kloetzli et al's algorithm on a GPU. Future works include optimization to the newest Fermi architecture and GPU implementation of other bit-parallel algorithms with a similar method.

References

1. Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J., Reid, J.F.: A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem. Information Processing Letters 80(6), 279–285 (2001)
2. Garland, M., Kirk, D.B.: Understanding Throughput-Oriented Architectures. Communications of the ACM 53(11), 58–66 (2010)
3. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
4. Hirschberg, D.S.: A Linear Space Algorithm for Computing Maximal Common Subsequences. Communications of the ACM 18(6), 341–343 (1975)
5. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann (2010)
6. Kloetzli, J., Strege, B., Decker, J., Olano, M.: Parallel Longest Common Subsequence Using Graphics Hardware. In: Proc. of the 8th Eurographics Symposium on Parallel Graphics and Visualization, EGPGV (2008)
7. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28(2), 39–55 (2008)
8. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional (2010)
9. Sklansky, J.: Conditional-Sum Addition Logic. IRE Trans. on Electronic Computers EC-9, 226–231 (1960)
10. Vai, M.: VLSI Design. CRC Press (2000)

Experiences with High-Level Programming Directives for Porting Applications to GPUs[*]

Oscar Hernandez, Wei Ding, Barbara Chapman,
Christos Kartsaklis, Ramanan Sankaran, and Richard Graham

Computer Science and Mathematics Division,
National Center for Computational Sciences,
Oak Ridge National Laboratory,
Dept. of Computer Science, University of Houston
{oscar,kartsaklisc,sankaranr,rlgraham}@ornl.gov,
{wding3,chapman}@cs.uh.edu

Abstract. HPC systems now exploit GPUs within their compute nodes to accelerate program performance. As a result, high-end application development has become extremely complex at the node level. In addition to restructuring the node code to exploit the cores and specialized devices, the programmer may need to choose a programming model such as OpenMP or CPU threads in conjunction with an accelerator programming model to share and manage the different node resources. This comes at a time when programmer productivity and the ability to produce portable code has been recognized as a major concern. In order to offset the high development cost of creating CUDA or OpenCL kernels, directives have been proposed for programming accelerator devices, but their implications are not well known. In this paper, we evaluate the state of the art accelerator directives to program several applications kernels, explore transformations to achieve good performance, and examine the expressivity and performance penalty of using high-level directives versus CUDA. We also compare our results to OpenMP implementations to understand the benefits of running the kernels in the accelerator versus CPU cores.

1 Introduction

Computational scientists performing research in a broad range of domains are demanding ever more powerful computing systems as they strive to solve some of the most pressing problems of today and prepare for those of tomorrow. To meet their needs, systems that provide significantly higher levels of peak performance

[*] This work was funded by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This research used resources of the Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This work was also funded by NSF grant CCF-0917285.

than any currently installed platforms are already being procured. Much more powerful computers are expected to be built and delivered during the coming decade. These future exascale systems may feature a variety of architectural innovations that offer new ways to address the inherent challenges of scale, power, reliability, cost, and packaging.

To attain the desired levels of computational performance while meeting operational constraints, hardware designers have begun to exploit recent developments in mainstream computer architectures. Following hard upon the heels of the industry-wide move from single to multi-core systems – designed to increase the overall computational performance without unduly increasing the amount of energy required to operate it – dominant mainstream computer architectures are now undergoing a second transition from *homogeneous* to *heterogeneous* designs that incorporate components designed for high-throughput, providing massive levels of parallelism. Emerging large-scale platforms will be heavily impacted by these technology shifts. For instance, the Tianhe-1A relies on graphic processing units (GPUs) to achieve 2.5 Petaflops of peak Performance. In addition, Oak Ridge National Laboratory (ORNL) has already announced that its next leadership class machine will be based on a heterogeneous multi-core system with GPUs.

Whereas accelerators on today's multi-core nodes are typically GPUs – massively parallel processors that can be programmed to perform a range of computations including, but not limited to, their original graphics domain – future systems are expected to have much higher core counts and density [1], and potentially will integrate a variety of special-purpose devices, or system on chips, that can provide the highest levels of performance on suitable code regions. Significant challenges face application developers as they learn to efficiently exploit recently installed architectures. The effectiveness of MPI-based applications has been reduced by the effects of multiple levels of architectural parallelism, limited amounts of memory per core and complex sharing of resources such as interconnects, caches and shared memories. The transition to heterogeneous node hardware will significantly exacerbate their difficulties, as they will need to explicitly detect and adapt code to run on the GPUs, potentially using yet another programming model. They will be required to deal with distinct and complex memory systems and the high cost of data transfer between them. As a result, GPU directive-based programming APIs are starting to emerge in an attempt to facilitate the porting process.

In this paper, we aim to identify the challenges involved in exploiting GPUs for non-graphics applications; to evaluate the programming effort and performance that can be obtained via the use of two existing, directive-based programming models, the HMPP and PGI accelerator directives, and compare them with CUDA and OpenMP. The accelerator directives used are a relatively new technology and we expect that the underlying implementation is capable of improvement. Nevertheless, we observe that even with the use of directives, a good deal of program reorganization may be required. The amount of effort depends not only upon the application program in question but also on the desired level of performance improvement.

The paper is organized as follows. The next section below discusses the state of the art with respect to heterogeneous computing. We compare two high-level directive-based programming models in Section 3. This is followed by the description of our efforts to adapt two existing programs, with different program structures and requirements, to a platform that includes GPUs in Section 4. In each case, we have used CUDA and two directive-based programming models to port the code to a multicore node that employs a GPU provided by NVIDIA. We then state our conclusions of our work in Section 5.

2 Programming Models for Heterogeneous Systems

Several vendors have provided programming interfaces for accelerators. Most have adapted the C programming language to fit the strict requirements of applications on their platform. GPUs were originally programmed using OpenGL. Domain-specific languages for graphics programming like GLSL (OpenGL Shading Language), HLSL (high level shader language), and Cg (C for graphics) from NVIDIA are also available. With their growing usefulness for compute-intensive functions in general-purpose applications, a number of programming interfaces (mostly based on C) have been provided to facilitate the development of application kernels for them. Rather than being fully fledged languages, most of them are based upon C. These include StreamIt [6], Sh [11], Brook [7], CUDA [13] and OpenCL. Compared with those programming languages, CUDA in particular has become popular for general-purpose programming on NVIDIA GPUs. The OpenCL [3] specification is the first standard programming API for accelerators released by Khronos [2].

Based heavily on CUDA, it is poised to become the first standard

Moreover, a variety of high level programming directives for accelerators are available or are undergoing development. CAPS HMPP [9], PGI accelerator directives [4] and HiCUDA [10] target CUDA and OpenCL [3]. RapidMind [5] defines C++ extensions that allow its users to describe how data in a C++ application should be mapped between GPUs, Cell processors and cores. However their approach requires redefinition of data types and the creation of kernels with a special syntax language. It is important to note that while these approaches provide portability at the language / directive level, the program optimizations and porting strategies required to apply them depend heavily on the target architecture and the application input set.

3 A Comparison of Directive-Based Programming Approaches

CUDA (and OpenCL) require the application developer to carefully study all the salient details of the target architecture. The process of code adaptation and tuning may be lengthy, involving significant reorganization of code and data, and is moreover error-prone. Porting the resulting code to another GPU (e.g. a successor model) may require non-trivial modification. High-level programming models have the potential to simplify the program creation and maintenance effort of the code. There have been few studies [12] that compare different vendors' GPU directives implementations. However, unlike ours, they do not focus on and the optimization process to achieve good performance, and the benefits of using the directives versus native CUDA or OpenMP.

In this section we provide an overview and compare two of the most popular directive based programming languages for GPUs: PGI Accelerator Directives and the HMPP Workbench.

3.1 Overview of the HMPP and PGI Accelerator Directives

HMPP is a directive-based programming interface for hybrid multicore parallel programming that aims to free the application developer from the need to code in a hardware-dependent manner. It is implemented by a source-to-source compiler designed to extract maximal data parallelism from C and FORTRAN kernels and translate them into NVIDIA CUDA or OpenCL. The main concepts of HMPP are the codelet and the callsite. A function that can be executed remotely on an accelerator is identified by the codelet directive; the callsite is the place the codelet (kernel) function call is launched. HMPP has both synchronous and asynchronous modes for the codelet remote procedure calls (RPCs). The asynchronous mode enables the overlapping of data transfers between the host and accelerators with other work. The programmer specifies targets for the execution of codelets. If the desired accelerator is present and available, it will run there. Otherwise the native host version is run.

PGI's Accelerator directives may be incrementally inserted into a code to designate a portion of C or Fortran code to be run on CUDA-enabled NVIDIA GPUs. They enable the application developer to specify regions for potential acceleration, to manage the necessary data transfers between the host and accelerator, and to initialize and shut down the accelerator. They further provide guidance to the implementation to help it perform data scoping, mapping of loops and transformations for performance. The directives assume that it is the host that handles the memory allocation on the device, initiates data transfers, sends the kernel to the device, waits for completion and transfers the results back from the device. The host is also responsible for queuing kernels for execution on the device.

The PGI directives include the kernel region declaration *#pragma acc* with *copyin*, *local* and *copyout* clauses to specify the input data, local data and output data of the kernel. PGI also supports the autoscoping of these data automatically. Directives *#pragma acc for parallel(M)* and *#pragma acc for vector(N)* are used to help the compiler identify parallel loops and how they should be mapped to the GPU.The compiler also attempts to associate a loop nest's iterations to grid and blocksizes that map to the GPU. The grid sizes will depend on the amount of work launched in the kernel while the threadblock size remains constant. In addition, PGI provides other clauses to optimize the kernels such as *cache* to load data to shared memory, loop *unroll* and *host* to run a loop in the CPU. PGI has been developing new directives such as the *#pragma acc region* and *#pragma acc* declaration directive to scope variables that should be shared among kernels and reside in the GPU or CPU or both.

HMPP uses the concept of groups, *#pragma hmpp group <groupid >, target=CUDA*, to specify codelets that share the same data set and will run in the same accelerator. HMPP provides the codelet *#pragma hmpp <groupid ><codeletid >codelet* and callsite directive *#pragma hmpp <groupid ><codeletid >* to specify codelets and where to invoke them. In addition these directive have clauses to specify the input, and output data and their sizes in the format format *args[A1]=size* and *args[A1].io=in, args[An].io=inout*. HMPP provides the asynchronous advanced load and delegate store directive to control when to move data to and from the accelerator. The directive *#pragma hmppcg parallel* indicates that the following loop is parallel and can be mapped to the GPU, while *#pragma hmppcg noParallel* indicates that a loop must not be parallelized. HMPP allows the user to explicitly define the threadblock size of a loop nest by using the *#pragma hmppcg grid blocksize NxN* directive. The *#pragma hmpp <groupid >resident* specifies data that should be allocated in the GPU and that may be shared among codelets.

4 Adapting Programs for GPUs: Two Case Studies

In this section, we present our experimental results that consists of adapting two applications kernels to run on GPU based platform. For both kernels, we use the PGI and HMPP accelerator directive-based programming to accelerate the kernels and we describe the transformation we used to get good performance when comparing it against CUDA and OpenMP.

Our experiments were run on an NVIDIA Tesla C2070 GPU with 448 cores in 14 Streaming Multiprocessors with frequency of 1.15 GHz. The GPU has 6GB DDR5 global memory shared by all threads. The local memory is 64K in size, and can be split 16K/48K or 48K/16K between L1 cache and shared memory. Shared memory for each Streaming Multiprocessor is accessible only within a thread block. The Tesla C2070 is also equipped with an L2 cache that covers GPU global memory.

4.1 S3D Thermodynamics Kernel

S3D is a parallel combustion flow solver for the direct numerical simulation of turbulent combustion. S3D [8] solves fully compressible Navier-Stokes, total energy, species and mass conservation equations coupled with detailed chemistry. The governing equations are supplemented with additional constitutive relations, such as the ideal gas equation of state, models for chemical reaction rates, molecular transport and thermodynamic properties. These relations and detailed chemical properties are implemented as kernels or community-standard libraries that are amenable to acceleration through GPU computing. For this work, we chose the thermodynamics kernel that evaluates the mixture-specific heat, enthalpy and Gibbs functions as a temperature polynomial.The coefficients of the thermodynamic polynomials and their relevant temperature ranges are obtained from thermodynamic databases following the conventions used in the NASA

```
do i = 1, np
  enth(i) = 0.0
  do m = 1, nslvs
  if(temp(i)<midtemp(m)) then
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coefflow(6, m)+temp(i)*(&
                  . . .
            coefflow(5, m)*rp05))))) )
  else
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coeffhig(6, m)+temp(i)*(&
                  . . .
            coeffhig(5, m)*rp05))))) )
    end if
  end do
end do
```

(a)S3D Thermodynamics Serial

```
!$OMP parallel do private(i, m, enth)
do i = 1, np
  enth(i) = 0.0
  do m = 1, nslvs
    if(temp(i)<midtemp(m))then
     enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coefflow(6,m)+temp(i)* (&
                .  .  .
            coefflow(5,m)*rp05))))))
    else
      enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coeffhig(6,m)+temp(i)*(&
                .  .  .
            coeffhig(4,m)*rp04+temp(i)*(&
            coeffhig(5,m)*rp05))))))
    end if
  end do
end do
!$OMP end parallel do
```

(b)S3D Thermodynamics OpenMP

```
!$OMP parallel private(i,m,flag_hig,flag_low)
do m = 1, nslvs
  !$OMP do
  do i = 1, np
    if(temp(i)<midtemp(m)) then
         flag_low(i, m)=1
         flag_hig(i, m)=0
    else
         flag_low(i, m)=0
         flag_hig(i, m)=1
    endif
  enddo
!$OMP end do nowait
enddo

!$OMP do
do i = 1, np
enth(i) = 0.0
  do m = 1, nslvs
   enth(i)=flag_low(i,m)*(enth(i)+yspec(i,m)*&
           Rsp(m)*(coefflow(6,m)+ temp(i)*(&
                    . . .
           coefflow(5, m)*rp05))))) ))+&
           flag_hig(i,m)*(enth(i)+yspec(i,m)*&
           Rsp(m)*(coeffhig(6,m)+temp(i)*(&
                    . . .
           coeffhig(5, m)*rp05))))) ))
  end do
end do
!$OMP end do nowait
!$OMP end parallel
```

(c)Optimized S3D Thermodynamics with OpenMP

Fig. 1. S3D Thermodynamics Kernel Code snippet

Chemical Equilibrium code. The thermodynamic kernel with small variations is applicable across a wide range of reacting flow applications.

Figure 1(a) shows the most time consuming portion of the serial kernel, where a double nested loop contains an *if* statement. The serial version takes about 22 seconds to execute in a CPU core. We decided to parallelize the outerloop with OpenMP as shown in Figure 1(b). We noticed that the inner loop was not being vectorized because of the *if* conditional. To further optimize the code, we hoisted the *if* conditional by precomputing the branch values in a separate loop that was also parallelized with OpenMP. As a result, we merge the if and else computations into single statements that were masked with the precomputed branch result. Figure 1(c) shows the transformation applied. By doing so, we were able to parallelize and vectorize the computational loop which yielded a good speed up of 3.8x when running the code on four cores. When running the code on twelve cores, the original OpenMP version in Figure 1(b) yielded the

```
!$acc data region copyin(temp,...),&
   copyout(enth)
do j = 1, MR
!$acc region
!$acc do parallel(np)
  do i = 1, np
    enth(i) = 0.0
    do m = 1, nslvs
      if(temp(i)<midtemp(m)) then
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coefflow(5, m)*rp05)))))
      else
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coeffhig(5, m)*rp05))))))
      end if
    end do
  end do
!$acc end region
!$acc region
!$acc do parallel(np)
  do i = 1, np
    cp(i) = 0.0
    do m = 1, nslvs
      ...
    end do
  end do
!$acc end region
end do
!$acc end data region
```

(a) PGI

```
!$hmpp <cudagroup> group, target=CUDA
!$hmpp <cudagroup> resident, args[Rsp].io=in
real,parameter::Rsp(1:nstts)=Ru/molwgt(1:nstts)
!$hmpp <cudagroup> resident, args[midtemp].io=in
real,parameter::midtemp(68)=(/ ... /)
!$hmpp <cudagroup> resident,args[coeffhig].io=in
real,parameter::coeffhig(7,68)=reshape(/.../)

subroutine calc_mixenth(np, ... ,cp)
implicit none
. . .
!$hmpp <cudagroup> allocate
!$hmpp <cudagroup> s3d_mixenth advancedload,&
      args[::Rsp; ...; ::coeffhig]
!$hmpp <cudagroup> s3d_mixenth callsite
call hmpp_kernel1(np, ... , coeffhig)
!$hmpp <cudagroup> s3d_mixcp callsite, &
      arg[::Rsp; ...].advancedload=true
call hmpp_kernel2(np, temp, ... , coeffhig)
!$hmpp <cudagroup> release
end subroutine calc_mixenth

!$hmpp <cudagroup> s3d_mixenth codelet, &
    args[np;...;yspec].io=in,args[enth].io=out
subroutine hmpp_kernel1(np,temp,...,coeffhig)
    ...
end subroutine hmpp_kernel1
!$hmpp <cudagroup> s3d_mixcp codelet, &
      args[np;...;yspec].io=in,args[cp].io=out
subroutine hmpp_kernel2(np,...,coeffhig)
    ...
end subroutine hmpp_kernel2
```

(b) HMPP

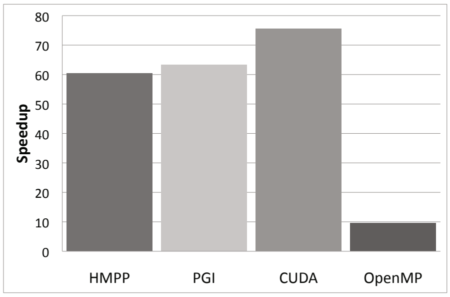
Fig. 2. S3D Thermodynamics Kernel Code snippet

best performance of 9.5x because it does not have any shared memory contention on the masked branch variable.

Our first attempt to accelerate the code with PGI and HMPP directives, by inserting a *!$ acc region* directive and creating a HMPP codelet for the main computational loopnest yielded very little performance for PGI and HMPP directives (2x speedup for PGI and 1.2 speedup for HMPP). By using the CUDA Profiler from NVIDIA, we observed that most of the time was spent in the accelerated kernels on data transfer between the CPU and GPU. Since the kernel contains read-only arrays, we optimized the accelerated kernels by allocating and initializing the read only variables inside the GPU. To do so, we had to inline the main computational kernel loop (loop i) inside the procedure that was invoking it within its loop j. The PGI version of this transformation is shown in Figure 2(a), which uses a data region to define the data that resides in the GPU. For HMPP, we used the *group* and *resident* directive to allocate data in the GPU and share data among the codelets of the same group. Figure 2(b), the HMPP implementation, where codelets *s3d_mixenth* and *s3d_mixcp* belong to the same group named cudagroup. Arrays *Rsp, midtemp, coeffhig* and *coefflow* are declared as resident variables, which makes them accessible to the two codelets

S3D Thermodynamics Timings (Seconds)	
SERIAL	21.926
HMPP	0.363
HMPP Kernel	0.3192948
HMPP Data Transfer	0.042834
PGI	0.346305
PGI Kernel	0.320225
PGI Data Transfer	0.02608
CUDA	0.29
CUDA Kernel	0.269265
CUDA Data Transfer	0.019952
OpenMP 12 Threads (best)	2.274

(a) S3D Thermodynamics Timing Table



(b) S3D Thermodynamics Speedup

Fig. 3. S3D Thermodynamics Kernel Experiment

defined in the HMPP group. In order to optimize the data transfers, we used the *advancedload* directive to initialize the read only data once before the first codelet *calc_mixenth* is executed. We also used the *advancedload* clause of the HMPP *callsite* directive to notify HMPP that the read only data is available in the GPU for the second codelet.

When comparing the results from different parallelization and acceleration strategies, we found that the HMPP and PGI implementations produced 60x and 63x speedup, respectively. The native CUDA implementation produced a speedup of 76 times that amount, while the OpenMP version using twelve threads produced a speedup of 10, as shown in Figure 3(b). The timings of our experiments are shown in Figure 3(a). Our results show that by managing the data correctly we were able to produce good speedups with the PGI and HMPP accelerator directives, within 80% of the native CUDA performance.

4.2 HOMME/SE Application

The High-Order Multi-scale Modeling Environment application, HOMME, is one of the highly promising frameworks for integrating the atmospheric primitive equations in spherical geometry. HOMME applies a spectral element method to conserve both mass and energy using an isotropic hyper-viscosity term. To discretize horizontal dimension, it uses a cubed-sphere grid and in the radial direction a vertical dimension. The HOMME application consists of several hundred Fortran 90 subroutines where the computations are spread evenly across them and whose relevance depends on the input problem.

For each of the spherical elements in the grid, HOMME maintains a global data structure that stores the state of the element, including velocity, temperature, pressure, divergence and geo-potential. Figure 4(a) shows a code fragment of the subroutine *compute_and_apply_rhs* which is one of the routines that computes the divergence for each of the cubed elements. The *ie* loop iterates over the spherical elements, the *q* loop over the advected physics, the *k* loop iterates over

```
  ...
do ie=nets, nete
  do q=1,qsize
    do k=1,nlev
      gradQ5d(:,:,k,q,1)=
      elem(ie)%state%v(:,:,1,k,n0)*&
      elem(ie)%state%Qdp(:,:,k,q,n0)

      gradQ5d(:,:,k,q,2)=
      elem(ie)%state%v(:,:,2,k,n0)* &
      elem(ie)%state%Qdp(:,:,k,q,n0)
    end do
  end do

  divdp4d(:,:,:,:) =
         divergence_sphere5d( &
         gradQ5d(:,:,:,:,:), &
         deriv, elem(ie))
    ...
end do
```

(a) Serial code

```
 ...
!$omp parallel private(ie,j,i,k,q,m,l,&
!$omp& gradQ5d,divdp4d,deriv)
 ...
!$omp do
  do ie=nets, nete
   do q=1,qsize
     do k=1,nlev
       gradQ5d(:,:,k,q,1)=
       elem(ie)%state%v(:,:,1,k,n0)*&
        elem(ie)%state%Qdp(:,:,k,q,n0)
        gradQ5d(:,:,k,q,2)=
        elem(ie)%state%v(:,:,2,k,n0)* &
        elem(ie)%state%Qdp(:,:,k,q,n0)
      end do
    end do

    divdp4d(:,:,:,:) =
           divergence_sphere5d( &
           gradQ5d(:,:,:,:,:), &
           deriv, elem(ie) )
   ...
  end do
!$omp enddo nowait
!$omp end parallel region
```

(b) OpenMP code

Fig. 4. The original and OpenMP divergence_sphere code version

the vertical radial grid points and the j and i loops iterate over the horizontal plane grid points.

In HOMME, coarse grain parallelism is implemented via MPI by distributing the spherical elements across nodes, whereby one or more elements can be assigned to an MPI process (see *ie* loop). In our case we assumed that each node will be assigned twelve elements to provide enough work for all the cores for in-node optimization or acceleration (i.e one element per core). The in-node problem size used was: $ie = 12$, $qsize\_d = 101$, $nlev = 26$ and $nv,np = 4$. We optimized several versions of the kernel for OpenMP, PGI Accelerator directives, and HMPP. We then compared them against the original serial version and the CUDA implementation tuned by NVIDIA and ORNL.

For the OpenMP version, see Figure 4(b). We parallelized the *ie* loop with *OpenMP parallel do* to assign spherical elements to OpenMP threads and take advantage of the node's shared memory. One of the challenges faced, when porting the code to OpenMP is to make sure memory access is consistent, by always accessing the same spherical element with the same thread including the data initialization loops. This improves locality by placing element's data in the core's local memory. We also must determine whether to privatize variables such as *gradQ*, a temporary variable that gathers data that is passed to the procedure or inline the procedure to avoid unnecessary data copies. For the inner loops we need to make sure loops get vectorized, if possible. When running the OpenMP kernel we noticed that using 4 threads gives the best performance with 510 milliseconds and a speedup of 2.67. The sequential version takes 1366 milliseconds.

```
!$acc region
!$acc do parallel(nete)
  do ie=nets, nete
!$acc do parallel(qsize)
   do q=1,qsize
!$acc do vector(32)
   do k=1,nlev
!$acc do vector(nv)
    do j=1,nv
!$acc do vector(nv) private(dudx00,dvdy00i)
    do l=1,nv
      dudx00=0.0d0
      dvdy00i=0.0d0
     do i=1,nv
      dudx00 = dudx00 + Dvv(i,l  ) * &
      (metdet(i,j,ie)*(Dinv(1,1,i,j,ie)* &
      gradQ5da(i,j,k,q,1,ie) + &
      Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))

      dvdy00i = dvdy00i + Dvv(i,j  ) * &
      (metdet(l,i,ie)*(Dinv(2,1,l,i,ie)* &
      gradQ5da(l,i,k,q,1,ie) + &
      Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
     end do
     divdp4da(l,j,k,q,ie)= &
              rmetdetp(l,j,ie) * &
             (rdx(ie))*dudx00+(rdy(ie))*dvdy00i
     end do
    end do
   end do
  end do
 enddo
!$acc end region
```

(a) PGI Accelerator Directives

```
!$hmpp <elements_group> divergence codelet
subroutine hmpp_divergence_sphere(rmetdetp,rdx,
 rdy,Dvv,metdet,Dinv,gradQ5da,divdp4dhmpp)
....
!$hmppcg parallel
!$hmppcg grid blocksize 32x4
  do k1 = 1, (nv*nv*nlev*qsize*(nete-nets+1))
   k2 = k1
   l = mod(k2, nv) + 1
   k2 = k2 / nv
   j =  mod(k2, nv) + 1
   k2 = k2 / nv
   k =  mod(k2, nlev) + 1
   k2 = k2 / nlev
   q =  mod(k2, qsize) + 1
   k2 = k2 / qsize
   ie = mod(k2, (nete-nets+1)) + nets
   k2 = k2 / (nete-nets+1)

   dudx00 =0.0d0
   dvdy00i=0.0d0
   do i=1,nv
    dudx00= dudx00+ Dvv(i,l)*(metdet(i,j,ie) *&
    (Dinv(1,1,i,j,ie)*gradQ5da(i,j,k,q,1,ie) +&
     Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))
    dvdy00i =dvdy00i + Dvv(i,j)*(metdet(l,i,ie)*&
    (Dinv(2,1,l,i,ie)*gradQ5da(l,i,k,q,1,ie)+ &
    Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
   end do
   divdp4dhmpp(l,j,k,q,ie)= rmetdetp(l,j,ie) *
          ((rdx(ie))*dudx00+(rdy(ie))*dvdy00i)
  enddo
end subroutine hmpp_divergence_sphere
```

(b) HMPP Implementation

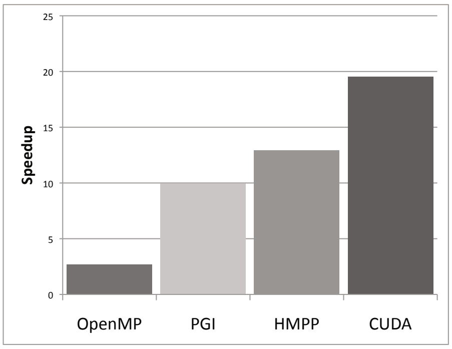
Fig. 5. The inlined and accelerated divergence_sphere code snippet

For the PGI and HMPP implementations of the kernel, it was necessary to inline the procedure *divergence_sphere* to map data parallel loops to a GPU and provide sufficient work. Also, this step is necessary if we want to accelerate the kernel at the *ie* loop, using the same approach adopted by the OpenMP implementation.

With the PGI accelerator directives, we needed to do some code restructuring to achieve good performance. We inlined the procedure *divergence_sphere* and inserted a *!$acc region* to accelerate the *ie* loop. This was necessary for the PGI directives since it cannot handle function calls inside an accelerated region. The next step was to specify how to parallelize the the loop nest iterations across the GPU Symmetric Multi-processors (SM) and within the SMs efficiently. We use the parallel and vector clause to specify the vector size and grid size: in this case we specified a block size of $nv \times nv \times nv$. To avoid non-coalesced memory accesses we eliminated a temporary array *gv* and fused the inner loops. We also allocated and initialized all the data inside the GPU by using the PGI data region directive and copyout directive to obtain the results of the kerne. To achieve good performance, we allocated and initialized the twelve spectral elements state on the GPU. Figure 5(a) shows the implementation of the kernel using the PGI accelerator directives.

HOMME/SE Timings (Miliseconds)	
SERIAL	1366.37
HMPP Kernel	105.72
PGI Kernel	137.43
CUDA	70.00
OpenMP 4 Threads (best)	510.62

(a)HOMME/SE Timing Table



(b)HOMME/SE Divergence Sphere Speedup

Fig. 6. HOMME/SE Kernel Experiments

We used a similar code transformation to implement the kernel with HMPP directives; With HMPP we had to outline the *ie* loop to a separate procedure to create a *codelet*. We also had to transform the loops by collapsing the *ie* and *q* loops with the *l* and the *e* loop respectively to provide enough work for a two-dimensional thread block (At the time of writing, HMPP 2.5.0 only supported two dimensional thread blocks). Figure 5(b) shows the HMPP implementation of the kernel. The OpenMP version (with 4 threads) achieves a speedup of 2.67 (over the serial version). Without counting the data transfer time, the GPU implementations achieve a speed up of 9.9x for the (ACC) PGI directives, 12.92x for HMPP and 19.5x for the CUDA implementation

5 Conclusions

This paper explores GPU programming models and compares the use of two sets of accelerator directives in two real-world application kernel studies. We explain the challenges and limitations encountered and, based on the lessons learned, reached initial conclusions on how to transform code to take advantage of the accelerator directive. In the HOMME/SE kernel, significant restructuring was needed to make sure the compilers generated the correct GPU scheduling (blocksize and grid size) and achieve a comparable performance to CUDA. We also compared the performance of running the codes on the GPU versus the CPU, and found that in all the cases the GPU yielded significantly better performance. In order to use the accelerator directives efficiently, it is necessary to perform code transformations to close the gap in performance to native CUDA implementations.

References

1. International Exascale Software Project Draft Report V 0.93, `http://www.exascale.org/iesp/MainPage`
2. Khronos Group, `http://www.khronos.org/`
3. OpenCL 1.0 Specification, `http://www.khronos.org/opencl/`
4. PGI Fortran & C Accelerator Compilers and Programming Model, `http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf`
5. Rapidmind, `http://www.rapidmind.net/`
6. Amarasinghe, S., Gordon, M.I., Karczmarek, M., Lin, J., Maze, D., Rabbah, R.M., Thies, W.: Language and compiler design for streaming applications. Int. J. Parallel Program. 33(2), 261–278 (2005)
7. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for gpus: stream computing on graphics hardware. In: SIGGRAPH 2004: ACM SIGGRAPH 2004 Papers, pp. 777–786. ACM, New York (2004)
8. Chen, J.H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E.R., Klasky, S., Liao, W.K., Ma, K.L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., Yoo, C.S.: Terascale direct numerical simulations of turbulent combustion using s3d. Computational Science and Discovery 2(1), 15001 (2009)
9. CAPS Enterprise. HMPP: A Hybrid Multicore Parallel Programming Platform, `http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf`
10. Han, T.D., Abdelrahman, T.S.: /hi/cuda: a high-level directive-based language for gpu programming. In: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp. 52–61. ACM, New York (2009)
11. McCool, M., Toit, S.: Metaprogramming GPUs with Sh. A K Peters, Ltd. (2004)
12. Membarth, R., Hannig, F., Teich, J., Korner, M., Eckert, W.: Frameworks for gpu accelerators: A comprehensive evaluation using 2d/3d image registration. In: 2011 IEEE 9th Symposium on Application Specific Processors (SASP), pp. 78–81 (June 2011)
13. NVIDIA. CUDA, `http://www.nvidia.com/object/cuda_home.html`

A GPU Algorithm for Greedy Graph Matching

Bas O. Fagginger Auer and Rob H. Bisseling

Mathematics Institute, Utrecht University,
Budapestlaan 6, 3584 CD Utrecht, The Netherlands
{B.O.FaggingerAuer,R.H.Bisseling}@uu.nl

Abstract. Greedy graph matching provides us with a fast way to coarsen a graph during graph partitioning. Direct algorithms on the CPU which perform such greedy matchings are simple and fast, but offer few handholds for parallelisation. To remedy this, we introduce a fine-grained shared-memory parallel algorithm for maximal greedy matching, together with an implementation on the GPU, which is faster (speedups up to 6.8 for random matching and 5.6 for weighted matching) than the serial CPU algorithms and produces matchings of similar (random matching) or better (weighted matching) quality.

1 Introduction

We propose a fine-grained shared-memory parallel algorithm for generating greedy matchings of undirected graphs. This algorithm was inspired by the parallel graph coarsening algorithm discussed in [8, Sec. 3.2] and follows a paradigm similar to that of the auction algorithm [4] for bipartite graphs (implemented on the GPU in [17]), but applying to general greedy matchings in arbitrary undirected graphs. The GPU has been used to accelerate solving of sparse problems, e.g. in the CUSP library [3], which provides sparse linear algebra and graph computations using CUDA.

Graph matchings have applications in minimising power consumption in dynamic wireless networks [19], heuristics for solving the travelling salesman problem [9], and organ donation [16]. Our primary interest however, will be the coarsening of graphs and hypergraphs (where edges may contain more than two vertices). During coarsening, we first match neighbouring vertices in a (hyper)graph and then merge matched pairs of vertices into a single vertex, which yields a coarser version of the (hyper)graph. Repeatedly coarsening a (hyper)graph gives a multi-level hierarchy of increasingly coarser approximations of the original (hyper)graph, which is useful for (hyper)graph partitioning, e.g. in the context of sparse matrix–vector multiplication [5,18] or LU decomposition of sparse matrices [1,7]. The following definitions have been set up such that they can easily be generalised to hypergraphs.

Graphs will be denoted by $G = (V, E)$, where $V \subseteq \mathbf{N}$ is the set of *vertices* and E the set of *edges* of the graph (all $e \in E$ are of the form $e = \{v, w\}$ for some $v, w \in V$). For $v \in V$, we denote the *collection of neighbours of* v by

$$V_v := \{w \in V \mid \exists e \in E : v, w \in e\} \setminus \{v\}.$$

The graph G is *weighted* if it is provided with a function $\omega : E \to \mathbf{R}_{>0}$ assigning a weight $\omega(e) > 0$ to each edge $e \in E$.

A *matching of* G is a map $\pi : V \to \mathbf{N}$ such that

1. for all $v \in V$ there exists at most one $w \in V \setminus \{v\}$ such that $\pi(v) = \pi(w)$ (we match at most two vertices to each other),
2. for all $v, w \in V$, $v \neq w$, if $\pi(v) = \pi(w)$, then $v \in V_w$ and $w \in V_v$ (we only match neighbouring vertices).

We consider two different vertices $v, w \in V$ to be *matched to each other* if $\pi(v) = \pi(w)$. If we cannot match any more vertices without breaking one of these two conditions, we call π *maximal.* If G is weighted, then the *weight* ω_π *of* π is defined as the sum of the weights of all edges included in the matching:

$$M_\pi := \{\{v, w\} \in E \mid \pi(v) = \pi(w), v \neq w\}, \qquad \omega_\pi := \sum_{e \in M_\pi} \omega(e).$$

2 Serial Matching

We will consider simple greedy random matching, as outlined in Alg. 1. For this algorithm we use $\pi(v) = \infty$ to indicate that the vertex v is unmatched.

Algorithm 1. Serially creates a matching of a graph $G = (V, E)$ with $V \subseteq \mathbf{N}$ by constructing $\pi : V \to \mathbf{N} \cup \{\infty\}$.

```
1: Randomise the order of the vertices in $V$.
2: **for** $v \in V$ **do**
3:     $\pi(v) \leftarrow \infty$;
4: **for** $v \in V$ **do**
5:     **if** $\pi(v) = \infty$ **then**
6:         $w \leftarrow \mathbf{select}(v, V_v \cap \pi^{-1}(\{\infty\}))$;
7:         **if** $w \neq \infty$ **then**
8:             $\pi(v) \leftarrow \min\{v, w\}$;
9:             $\pi(w) \leftarrow \min\{v, w\}$;
```

The function **select**(v, W) is defined for vertices $v \in V$ and collections of neighbours $W \subseteq V_v$. Should W be empty, then **select**$(v, W) = \infty$, otherwise **select**$(v, W) = w$ for some neighbour $w \in W$ of v. Choosing different prescriptions for selecting neighbours gives us different kinds of matchings. Here, we consider two options for **select**: *random matching* and *weighted matching.*

For random matching, **select**(v, W) returns the first available $w \in W$. Because we randomise vertex order, this amounts to matching vertices to random neighbours, while providing an early exit for the selection mechanism.

For weighted matching, **select**(v, W) returns a neighbour $w \in W$ with $\omega(\{v, w\}) = \max_{u \in W} \omega(\{v, u\})$. Here, the selection process takes longer: every neighbour needs to be considered to find the heaviest edge originating from v.

Note that in either case Alg. 1 produces maximal matchings. This ensures that the number of matched vertices is at least half of the maximum possible number of matched vertices when considering all possible matchings.

Other greedy matching strategies such as dynamic minimum degree (vertices with fewest unmatched neighbours are matched first) or Karp–Sipser [10] (vertices with a single unmatched neighbour are matched first) are not considered, because dynamically keeping track of all vertex degrees leads to serialisation. A more in-depth discussion and comparison of such matching strategies can be found in [11]. A distributed-memory parallel implementation of the Karp–Sipser algorithm is presented in [13].

2.1 Matching by Decreasing Edge Weights

We will also compare weighted matchings generated by our GPU algorithm with weighted matchings generated by Alg. 2.

Algorithm 2. Serially creates a weighted matching of a weighted graph $G = (V, E)$ with $V \subseteq \mathbf{N}$ and weights $\omega : E \to \mathbf{R}_{>0}$ by constructing $\pi : V \to \mathbf{N} \cup \{\infty\}$.

```
1: **for** $v \in V$ **do**
2:     $\pi(v) \leftarrow \infty$;
3: **for** $\{v, w\} \in E$ in order of decreasing $\omega(\{v, w\})$ **do**
4:     **if** $\pi(v) = \infty$, $\pi(w) = \infty$, and $v \neq w$ **then**
5:         $\pi(v) \leftarrow \min\{v, w\}$;
6:         $\pi(w) \leftarrow \min\{v, w\}$;
```

Alg. 2 ensures that we always match the vertices belonging to an edge with maximum weight in the entire graph, in contrast to weighted matching by Alg. 1 where the edge with maximum weight originating from a random vertex is matched. Because of this, Alg. 2 is $\frac{1}{2}$-optimal, i.e. the weight ω_π of the matching π generated by Alg. 2 is guaranteed to be at least half of the maximum weight that any matching of this graph can attain. A distributed-memory parallel algorithm for weighted matching is given in [12]; this algorithm is based on locally dominant edges and is also $\frac{1}{2}$-optimal.

3 Parallel Matching

A problem with Alg. 1 is its serial nature: in order to prevent matching more than two vertices to each other, we seemingly have to consider vertices one-by-one. To be able to match vertices simultaneously, while still satisfying the matching criteria, we propose Alg. 3, which permits us to evaluate **select** in parallel for many vertices.

For this algorithm $\pi(v) \in \{\textbf{blue}, \textbf{red}, \textbf{dead}\}$ indicates that v has not been matched. The function **colour**(v) determines for vertices $v \in V$ whether they are put into the **blue** or the **red** group. The **for all . . . parallel do** construct

Algorithm 3. Creates a matching of a graph $G = (V, E)$, with $V \subseteq \mathbf{N}$, in parallel by constructing $\pi : V \to \mathbf{N} \cup \{\mathbf{blue}, \mathbf{red}, \mathbf{dead}\}$.

```
1: for all $v \in V$ parallel do
2:    $\pi(v) \leftarrow$ blue;
3: done $\leftarrow$ false;
4: while not done do
5:    {Assign vertex colours:}
6:    done $\leftarrow$ true;
7:    for all $v \in V$ parallel do
8:       if $\pi(v) \in \{$blue, red$\}$ then
9:          done $\leftarrow$ false;
10:         $\pi(v) \leftarrow$ colour$(v)$;
11:   {Blue vertices propose to red vertices:}
12:   for all $v \in V$ parallel do
13:      if $\pi(v) =$ blue then
14:         if $V_v \cap \pi^{-1}(\{$blue, red$\}) = \emptyset$ then
15:            $\sigma(v) \leftarrow$ dead;
16:         else
17:            $\sigma(v) \leftarrow$ select$(v, V_v \cap \pi^{-1}(\{$red$\}))$;
18:      else
19:         $\sigma(v) \leftarrow \infty$;
20:   {Red vertices respond to blue vertices:}
21:   for all $v \in V$ parallel do
22:      if $\pi(v) =$ red then
23:         if $V_v \cap \pi^{-1}(\{$blue, red$\}) = \emptyset$ then
24:            $\sigma(v) \leftarrow$ dead;
25:         else
26:            $\sigma(v) \leftarrow$ select$(v, V_v \cap \pi^{-1}(\{$blue$\}) \cap \sigma^{-1}(\{v\}))$;
27:   {Match mutual proposals:}
28:   for all $v \in V$ parallel do
29:      if $\sigma(v) =$ dead then
30:         $\pi(v) \leftarrow$ dead;
31:      else if $\sigma(v) \neq \infty$ then
32:         if $\sigma(\sigma(v)) = v$ then
33:            $\pi(v) \leftarrow \min\{v, \sigma(v)\}$;
```

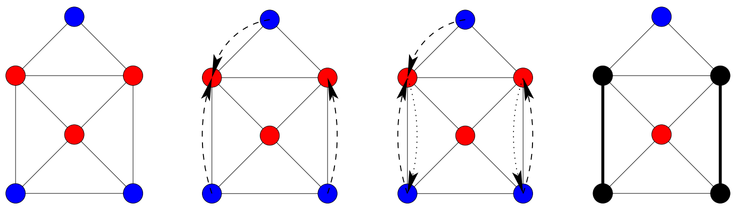

(a) Colour. (b) Propose. (c) Respond. (d) Match.

Fig. 1. Illustration of one iteration of Alg. 3's main loop: (a) we colour all vertices blue or red; (b) let the blue vertices propose to the red vertices; (c) let the red vertices respond to one of these proposals; (d) and match the mutual proposals

indicates a for-loop where each iteration can be executed completely independently. These for-loops make Alg. 3 suitable for a GPU implementation, where each independent loop iteration (corresponding to a vertex) is mapped to a different GPU thread. Furthermore, π and σ can be kept on the GPU during the iterations of Alg. 3, such that communication between the CPU and GPU is limited to only the start and the end of the matching process.

Alg. 3 starts by marking all vertices $v \in V$ as **blue** (line 2), such that they are unmatched. Then, we enter the main loop (line 4) and colour each unmatched vertex **blue** or **red** (line 10, this is irrespective of the current colour of the vertex). All **blue** vertices propose to **red** neighbours, chosen by **select** (line 17). Vertices without unmatched neighbours are flagged as being **dead**. **Red** vertices then consider proposals made to them by their neighbours, and respond to one of them, chosen by **select** (line 26). Here, data thrashing due to parallel reads and writes to σ is avoided by only checking whether $\sigma(w) = v$ for neighbours $w \in V_v$ with $\pi(w) =$ **blue**. After this, we match all vertices that have compatible proposals and responses (line 28). Vertices that were flagged as dead receive a special matching value (**dead**) so that they are no longer considered for matching in subsequent iterations. We restart the main loop and reassign unmatched vertices to either the **blue** or **red** group. The main loop is repeated until we obtain a maximal matching.

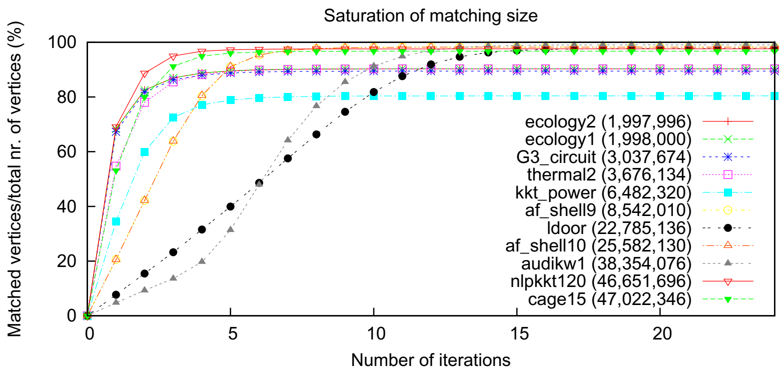


Fig. 2. The ratio between the number of matched vertices and the total number of vertices, as a function of the number of iterations of the while-loop at line 4 of Alg. 3. The number after each graph name indicates the number of edges.

The effect of iterating the main loop of Alg. 3 can be seen in Fig. 2. Here we observe that the number of unmatched vertices decreases rapidly as the number of iterations increases, stabilising when the matching is maximal. Note that the matching is maximal when all vertices are either matched or **dead**. Therefore, we keep track of a 'done' flag in Alg. 3, which becomes **true** when $\pi^{-1}(\{\textbf{blue}, \textbf{red}\}) = \emptyset$. Because we only need to store a fixed value in 'done' at line 9, we can do this directly in parallel without having to resort to atomic operations (an atomic compare-and-swap halved performance during experiments).

3.1 Vertex Labelling

It is important that the function **colour** finds different **blue** and **red** groups every iteration, because otherwise we can get stuck in situations where a non-maximal matching is not enlarged. A direct way to define this function is to determine the **blue** and **red** groups by randomly assigning each vertex to the **blue** group with probability p and to the **red** group with probability $1-p$, i.e.

$$\textbf{colour}(v) = \begin{cases} \textbf{blue} \text{ with probability } p \in [0,1], \\ \textbf{red} \quad \text{otherwise.} \end{cases} \tag{1}$$

Intuitively, we should ensure that the **blue** and **red** groups are approximately of equal size (by picking $p = \frac{1}{2}$, similar to [8, Sec. 3.2]) so that all unmatched vertices have a good chance of possessing a neighbour of a different colour and are therefore able to propose or respond in the current iteration. This leads to a large number of matched or **dead** vertices, which will speed up later iterations.

Let us make this more precise by considering random matching in a random graph G with vertices $V = \{1, \ldots, n\}$, where an edge between two vertices $v, w \in V$ exists with probability $P(\{v,w\} \in E) = d$ for a fixed density parameter $d \in [0,1]$. During a single iteration of Alg. 3 we match a number of vertices equal to twice the number N of **red** vertices that receive a proposal from a **blue** neighbour, i.e.

$$\begin{aligned} N &= \sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red}) \\ &= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V\setminus\{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right) \\ &= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V\setminus\{v\}} \left(1 - \frac{P(\pi(w) = \textbf{blue})\, P(\{v,w\} \in E)}{\text{nr. of } \textbf{red} \text{ neighb. of } w}\right)\right). \end{aligned}$$

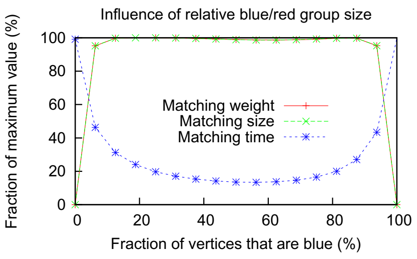
We now approximate the number of **red** neighbours of w by its average $1 + (1 - p)\,(d\,(n-1) - 1)$ (since v is already a **red** neighbour of w). This gives

$$N \approx N^* := n\,(1-p) \left(1 - \left(1 - \frac{p\,d}{1 + (1-p)\,(d\,(n-1) - 1)}\right)^{n-1}\right).$$

The approximate expected fraction of matched vertices in a large random graph for a single iteration of Alg. 3 then equals

$$\lim_{n \to \infty} \frac{2\,N^*}{n} = 2\,(1-p)\,\left(1 - e^{-\frac{p}{1-p}}\right). \tag{2}$$

This function is maximal for $p \in [0,1]$ satisfying $1 - p = e^{-\frac{p}{1-p}}\,(2-p)$, yielding $p \approx 0.53406$, independent of the density d. Therefore, this choice of p yields the largest number of matched vertices per iteration, and hence the shortest running time of Alg. 3, regardless of the edge density of the random graph. Because of this, we expect such a p also to work well for non-random graphs, which we confirmed experimentally for `ecology1` in Fig. 3.



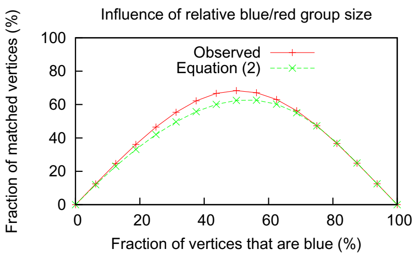


Fig. 3. The effect of the probability p from eq. (1) (left) on the absolute matching size, weight, and time required to generate the matching, rescaled to a range of 100%, and (right) on the observed and theoretical, eq. (2), fraction of matched vertices during the first iteration of Alg. 3 for the matrix `ecology1`

3.2 Random Vertex Assignment

To evaluate eq. (1) in parallel on the GPU we use the MD5 message digest algorithm [15]. This algorithm calculates a 128-bit value, the *MD5 hash*, of a given sequence of bits, such that small changes in this sequence in general result in a completely different MD5 hash. A sequence of bits is converted to an MD5 hash by padding the sequence such that its number of bits is a multiple of 512, and then adding the contribution of each 512-bit chunk of the padded sequence to the hash.

To employ the MD5 algorithm as a random number generator we generate a single random number r on the CPU, which we pass to the GPU as a parameter for all threads (we again create a GPU thread for each vertex $v \in V$). Then, we create for each 32-bit vertex number v a 512-bit array consisting of $\{v, r\,v, \ldots, r^{15}\,v\}$ and calculate the hash of this array, which we normalise to obtain a pseudorandom number in $[0, 1]$ for eq. (1). Changing either v or r will result in a different array and therefore a completely different hash. By generating different values r, we can therefore create completely different assignments of the unmatched vertices of the graph, for each iteration of Alg. 3.

To improve performance, we only use the first quarter of the MD5 algorithm, which did not reduce the quality of the matchings during experiments. This makes Alg. 4 fast and parallel, requires only a small amount of thread-independent storage (we do not store $\{v, r\,v, \ldots, r^{15}\,v\}$ explicitly), and yields reproducible vertex colourings.

4 Results

For graph coarsening, it is important to randomise the ordering of the vertices of the graph to ensure that we do not get stuck in star graphs [14, Sec. 5.3]. Therefore, we randomly permute all vertices on the CPU after the graph has been read from disk (as was also done in the experiments in [11]), where we use the same permutation for benchmarking the serial and parallel algorithms. Randomisation of the vertices will decrease performance, because it prevents

Algorithm 4. Implementation of $\mathbf{colour}_r(v)$ for Alg. 3, based on [15, Sec. 3.4]. Here $v \in \mathbf{N}$ is a vertex, $r \in \mathbf{N}$ is a parameter, $p \in [0, 1]$ (eq. (1)), and K and R are MD5 constants and shift amounts, kept in constant GPU memory.

```
1: Initialise hash as $h_0, h_1, h_2, h_3$.
2: Let $a_0 \leftarrow h_0$, $a_1 \leftarrow h_1$, $a_2 \leftarrow h_2$, $a_3 \leftarrow h_3$.
3: for $i = 0$ to 15 do
4:     $a_4 \leftarrow (a_1$ and $a_2)$ or $(($not $a_1)$ and $a_3)$;
5:     $a_5 \leftarrow a_3$, $a_3 \leftarrow a_2$, $a_2 \leftarrow a_1$;
6:     $a_1 \leftarrow a_1 + \mathbf{rol}(a_0 + a_4 + K(i) + v, R(i))$;   (bitwise rotate left)
7:     $a_0 \leftarrow a_5$;
8:     Add $a_0, \ldots, a_3$ to $h_0, \ldots, h_3$.
9:     $v \leftarrow r\, v$;
10: if $(h_0 + h_1 + h_2 + h_3) \mod 2^{32} < p\, 2^{32}$ then
11:     return blue;
12: else
13:     return red;
```

coalesced reading on the GPU when looping over vertex neighbours. As we are interested in the performance of the greedy matching process itself, permuting the graph and I/O transfer have not been included in the recorded timings. CPU to GPU transfer takes up, on average, 39% of the time.

The actual implementation of Alg. 3 was done with both NVIDIA's Compute Unified Device Architecture (CUDA) library version 3.1.2 and Intel's Threading Building Blocks (TBB) library version 3.0 in C++, compiled with g++ version 4.1.2 using O3 optimisation flags. For the CUDA implementation, static graph data (i.e. neighbour ranges, indices, and edge weights) were placed in one-dimensional textures to improve cache use. Dynamic data (i.e. π and σ) were placed in one-dimensional arrays, such that using a one-dimensional thread distribution (one thread per vertex and a CUDA block size of 256) gives us coalesced data writing everywhere in the algorithm. For more implementation details, we would like to refer the reader to the source code of the discussed algorithms, which is freely available at `http://www.staff.science.uu.nl/~faggi101/`.

For weighted matching, we use 425 symmetric matrices from the University of Florida sparse matrix collection [6], where the edge weights are set to the absolute value of the corresponding matrix entry. For random matching this set is augmented with the graphs from the 10th DIMACS challenge on graph partitioning [2], which do not possess edge weights. This gives us a large, unbiased test set of matrices arising from real-world applications.

The experiments were performed on a computer equipped with two quad-core 2.4 GHz Intel Xeon E5620 processors with hyperthreading, 24 GiB RAM, and an NVIDIA Tesla C2050 with 2687 MiB global memory. We measured the scaling of the TBB implementation of Alg. 3 with respect to the number of threads used by the CPU in Fig. 4 and compared both the CUDA and TBB (using 16 threads) implementations to the serial matching algorithms (Alg. 1 and Alg. 2) in Fig. 5. Fig. 5 shows the ratios of the average (over 32 random permutations of the graph vertices) matching size, time, and weight, together with error bars of one standard deviation. From these results, we observe the following:

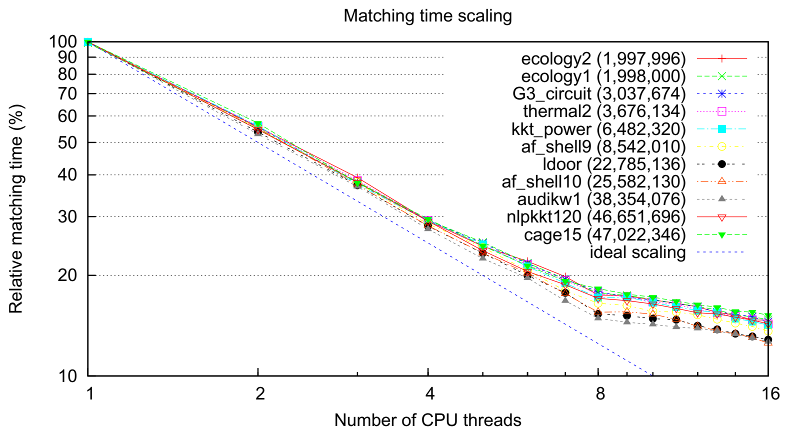
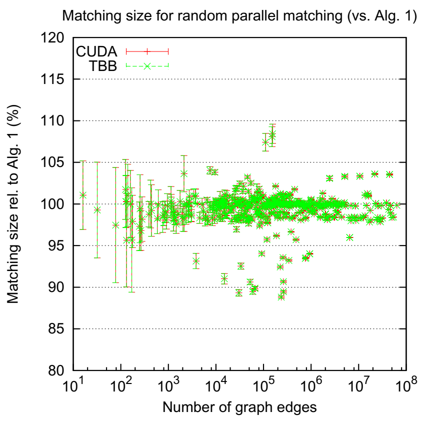


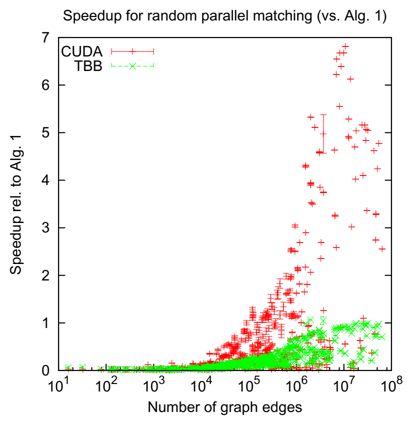
Fig. 4. Scaling of the random matching time of Alg. 3 with the number of TBB threads, on a dual quad-core CPU with hyperthreading (8 physical cores) in a log-log plot. The matching time is relative to the matching time required by Alg. 3 on a single core.

- Alg. 3 scales well as we increase the used number of threads. The test system possesses 8 physical cores, but up to 16 threads with hyperthreading, which explains good speedups up to 8 threads and smaller speedups thereafter.
- The quality of the generated random matchings by Alg. 3 is comparable to that of the serial algorithm: for both CUDA and TBB the average matching size ratio is more than 99%.
- Weighted matching with Alg. 3, for both CUDA and TBB, yields higher quality matchings than Alg. 1 (average matching weight ratio 115%), but lower quality matchings than Alg. 2 (ratio of 85%). This is not surprising, since Alg. 2 always picks the globally heaviest edge, whereas Alg. 1 and Alg. 3 pick heavy edges locally. Furthermore, Alg. 1 does this one-sidedly, whereas Alg. 3 performs a two-sided comparison (both proposers and responders pick the heaviest neighbour), which leads to an increase in matching weight.
- In Fig. 5 we see that Alg. 3 does not obtain the same speedup for all graphs. This is related to the ratio of the maximum and minimum degree of the vertices of the graph, $(\max_{v \in V} |V_v|) / (\min_{w \in V} |V_w|)$. When this ratio is large, vertices with high degree will keep a small number of CUDA threads occupied for a long time, while the other kernels have already finished, leading to a low occupancy of the GPU and decreased performance.
- The speedups increase as the graphs become larger. For CUDA, Alg. 3 reaches speedups up to 6.8, 5.6, and 37 compared to random matching with Alg. 1 and weighted matching with Alg. 1 and Alg. 2. However, for TBB we only reach speedups up to 1.1, 0.7, and 13.

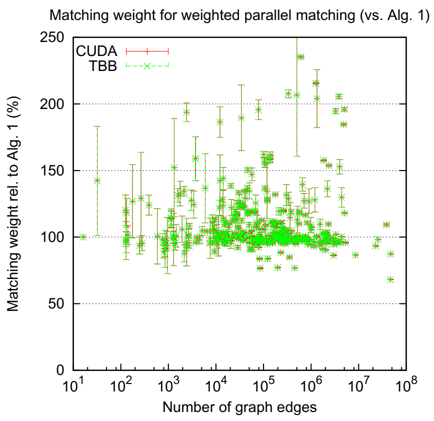
Most of the time in Alg. 3 is spent loading and storing data, instead of performing calculations. This is confirmed by the NVIDIA CUDA profiler for random matching of `ecology1`, where the instruction-to-byte ratios are equal to 2.36 and 1.31 (according to the profiler, they should be close to 4.06) for proposing and responding



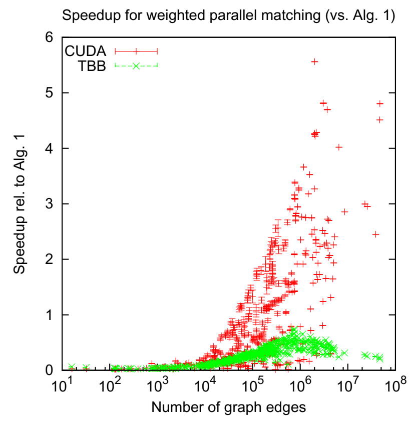
(a)



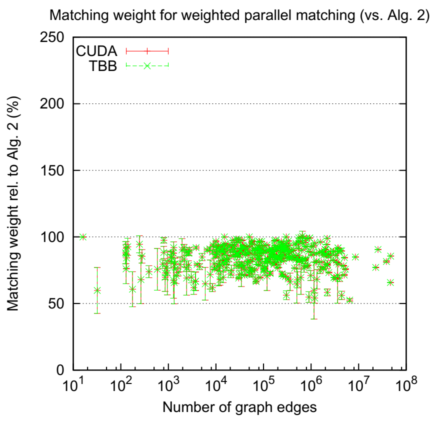
(b)



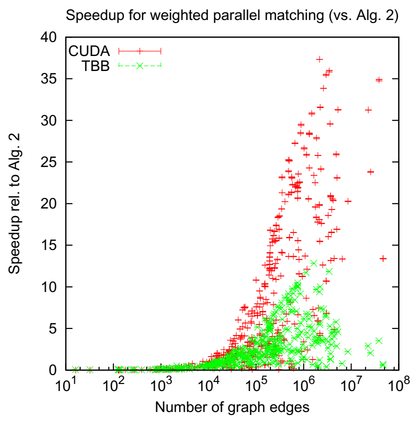
(c)



(d)



(e)



(f)

Fig. 5. Comparison between the serial matching algorithms (Alg. 1 and Alg. 2) and Alg. 3 implemented in CUDA on the GPU and in TBB on a multi-core CPU

to proposals in Alg. 3: this makes the algorithm bandwidth limited. Non-coalesced memory access due to randomisation is reflected in a low texture-cache hit rate, which is 35% for proposing, and 3% for responding, but we do utilize 70% and 82%, respectively, of the maximum available global memory bandwidth. This explains the fact that the GPU, with its much larger bandwidth (144 GB/s for a Tesla C2050's global memory vs. 17 GB/s for DDR3 RAM), performs better than the CPU TBB implementation, and that for weighted matching (where the edge weights also need to be read) the speedups are smaller, because memory traffic is increased. We therefore expect performance to be increased further when **select** involves a more compute-intensive assessment of each of the vertex's neighbours.

5 Conclusion

We have described a fine-grained shared-memory parallel algorithm for greedy graph matching (Alg. 3) and created a GPU implementation of this algorithm to compare it with serial greedy matching on the CPU (Alg. 1 and 2). For random matching, Alg. 3 provides maximal matchings of similar quality as Alg. 1; it is slower for smaller graphs ($< 10^5$ edges), but becomes increasingly faster as the number of edges increases (up to a speedup factor of 6.8). For weighted matching of large graphs, Alg. 3 offers both better performance (speedups up to 5.6) and better quality than Alg. 1, while compared to Alg. 2 we sacrifice matching quality for a much better performance (speedups up to 37). Alg. 3 performs much better on the GPU than on the multi-core CPU because of the GPU's superior memory bandwidth. These results were obtained for a large set of graphs arising from real-world applications.

We are interested in applying this algorithm in the context of (hyper)graph coarsening [18] and anticipate that there, with more complicated ways for vertices to select desired neighbours to be matched to, the ability of Alg. 3 to perform many of these selections in parallel will lead to higher speedups.

Acknowledgements. We would like to thank Albert-Jan Yzelman and Jaap Eldering for their help in refining the algorithm, Vianney Govers for extensive technical support, and the Little Green Machine project, `http://littlegreenmachine.org/`, for enabling us to run our benchmarks on their hardware under project NWO-M 612.071.305.

References

1. Aykanat, C., Pinar, A., Çatalyürek, U.V.: Permuting sparse rectangular matrices into block-diagonal form. SIAM J. Sci. Comput. 25(6), 1860–1879 (2004)
2. Bader, D.A., Sanders, P., Wagner, D., Meyerhenke, H., Hendrickson, B., Johnson, D.S., Walshaw, C., Mattson, T.G.: 10th DIMACS implementation challenge - graph partitioning and graph clustering (2012), `http://www.cc.gatech.edu/dimacs10/index.shtml`

3. Bell, N., Garland, M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations, version 0.1.0 (2010), `http://cusp-library.googlecode.com`
4. Bertsekas, D.P.: A distributed asynchronous relaxation algorithm for the assignment problem. In: 24th IEEE CDC, vol. 24, pp. 1703–1704 (1985)
5. Çatalyürek, U.V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans. Par. Dist. Syst. 10(7), 673–693 (1999)
6. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM TOMS 38(1), 1:1–1:25 (2011)
7. Grigori, L., Boman, E.G., Donfack, S., Davis, T.A.: Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization. SIAM J. Sci. Comput. 32(6), 3426–3446 (2010)
8. Her, J.H., Pellegrini, F.: Efficient and scalable parallel graph partitioning. Parallel Computing (2010)
9. Kahng, A.B., Reda, S.: Match twice and stitch: a new TSP tour construction heuristic. Operations Research Letters 32(6), 499–509 (2004)
10. Karp, R.M., Sipser, M.: Maximum matchings in sparse random graphs. In: Proc. 22nd FOCS, pp. 364–375 (1981)
11. Langguth, J., Manne, F., Sanders, P.: Heuristic initialization for bipartite matching problems. J. Exp. Algorithmics 15(1.3), 1.1–1.22 (2010)
12. Manne, F., Bisseling, R.H.: A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 708–717. Springer, Heidelberg (2008)
13. Patwary, M.A., Bisseling, R.H., Manne, F.: Parallel greedy graph matching using an edge partitioning approach. In: Proc. HLPP 2010, pp. 45–54. ACM (2010)
14. Preis, R.: Analyses and design of efficient graph partitioning methods. HNI-Verlagsschriftenreihe, Heinz Nixdorf Inst. Univ. Paderborn (2001)
15. Rivest, R.L.: The MD5 message-digest algorithm, Internet RFC 1321 (1992)
16. Segev, D.L., Gentry, S.E., Warren, D.S., Reeb, B., Montgomery, R.A.: Kidney paired donation and optimizing the use of live donor organs. JAMA 293(15), 1883–1890 (2005)
17. Vasconcelos, C.N., Rosenhahn, B.: Bipartite Graph Matching Computation on GPU. In: Cremers, D., Boykov, Y., Blake, A., Schmidt, F.R. (eds.) EMMCVPR 2009. LNCS, vol. 5681, pp. 42–55. Springer, Heidelberg (2009)
18. Vastenhouw, B., Bisseling, R.H.: A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. 47(1), 67–95 (2005)
19. Xing, G., Lu, C., Zhang, Y., Huang, Q., Pless, R.: Minimum power configuration for wireless communication in sensor networks. ACM Trans. Sen. Netw. 3(2) (2007)

Hybrid Parallelization of a Large-Scale Heart Model

Dorian Krause[1], Mark Potse[2], Thomas Dickopf[1], Rolf Krause[1], Angelo Auricchio[3], and Frits Prinzen[2]

[1] Institute of Computational Science, University of Lugano, Switzerland
{dorian.krause,thomas.dickopf,rolf.krause}@usi.ch
[2] Cardiovascular Research Institute, Maastricht University, The Netherlands
mark@potse.nl, frits.prinzen@maastrichtuniversity.nl
[3] Fondazione Cardiocentro Ticino, Lugano, Switzerland
angelo.auricchio@cardiocentro.org

Abstract. The simulation of the electrophysiology of the heart is challenging due to its multiscale nature requiring the use of high spatial resolutions. Hence, it is important to efficiently utilize large parallel machines. In this article, we present a code designed to meet these scalability challenges on contemporary multicore-based massively parallel architectures. It is based on a well-established model originally designed for shared-memory systems. To improve scalability and extend support to distributed-memory architectures, we developed a hybrid OpenMP-MPI code. The new code shows excellent scalability up to 8448 cores with both explicit and implicit time discretizations. We present an in-depth analysis of the advantages of hybrid parallelization for this type of application.

1 Introduction

The contraction of the heart is a highly tuned mechanism that is organized by a complex electrical activation system. In each cardiac cell, approximately a million ion channels, pumps, and exchangers work together by allowing or forcing specific ions to cross the cell's inner and outer membranes [5]. They come in dozens of different types, encoded by different genes. Their permeability/activity depends on transmembrane voltage, ion concentrations, and time. Together, the ion channels in a cell membrane generate *action potentials*: temporary changes in transmembrane voltage that serve to open calcium channels, allowing a large amount of calcium to enter the cell, bind to the cell's contractile molecules, and initiate a contraction. Unlike skeletal muscle cells, cardiac muscle cells can trigger action potentials in their neighbors by passing current through the gap junctions that connect their interiors. By this mechanism, the entire cardiac muscle can be activated in less than 100 ms; a prerequisite for an ordered contraction.

Mathematical modeling is essential to understand the dynamics of the interactions between ion channels in the cell membrane [17]. The first numerical models of cardiac cells date from the 1960s. Since then, the models have grown in complexity to capture newly discovered channel types as well as our evolving understanding of the known channels. In addition, it is now possible to couple many such models together to simulate entire hearts.

The high spatial and temporal gradients occurring in the propagation of the action potential require high spatial resolution. The size of whole-heart models therefore ranges from $O(10^6)$ nodes for small mammals [24] to $O(10^8)$ nodes for an adult human heart [6]. The required sizes could increase by more than an order of magnitude when muscle diseases are modeled. Consequently, much work is devoted to improving the performance and scalability of these simulations [1,7,15,16,25].

In this paper, we report on our work to develop a hybrid OpenMP-MPI parallelization for an existing heart model in order to optimize strong and weak scaling, improve performance, and advance the limit of achievable model size. The paper is organized as follows. In Section 2, we describe the mathematical models underlying the numerical simulation of the electrophysiology of the heart. In Section 3, we present the Propag code which is the basis of the work described in the article. In Section 4, we explain the new hybrid parallelization of Propag. Finally, in Section 5, we present and analyze our performance results.

2 Mathematical Model

The human heart contains a few billion muscle cells. Gap junctions allow action potentials to propagate from one cell to another [2,12]. To model this electrophysiological system mathematically, it is customary to treat the intracellular environment with the gap junctions as a continuous domain. Likewise, the extracellular environment, which in reality consists of many different components, is treated as another continuous domain. These domains and the active membrane between them can then be discretized with a spatial step size that is much larger than a single cell. This leads to the *bidomain model* [4,19]

$$\nabla \cdot (\sigma_\mathrm{i} \nabla \phi_\mathrm{i}) = \beta I_\mathrm{m} = -\nabla \cdot (\sigma_\mathrm{e} \nabla \phi_\mathrm{e}) \tag{1}$$

where ϕ_i and ϕ_e denote the intra- and extracellular potential fields, β is the membrane surface-to-volume ratio, σ_i and σ_e denote the conductivity tensors in the intra- and extracellular domain, and the transmembrane current density I_m equals

$$I_\mathrm{m} = C_\mathrm{m} \frac{\partial V_\mathrm{m}}{\partial t} + I_\mathrm{ion} + I_\mathrm{stim} \, , \tag{2}$$

with $V_\mathrm{m} = \phi_\mathrm{i} - \phi_\mathrm{e}$ and $C_\mathrm{m} = 1 \mu\mathrm{F/cm}^2$ the membrane capacitance. Here, I_ion is the ionic current; I_stim denotes a stimulation current. In this study, we simulated I_ion with the Ten Tusscher-Panfilov 2006 model [23]. Free boundary conditions are imposed for ϕ_e, ϕ_i and V_m.

By inserting (2) into (1) and using an operator splitting approach (see also, for example, Vigmond et al. [25]), we obtain the following *bidomain reaction-diffusion model*

$$\frac{\partial V_\mathrm{m}}{\partial t} = \frac{1}{\beta C_\mathrm{m}} \Big[\nabla \cdot (\sigma_\mathrm{i} \nabla (V_\mathrm{m} + \phi_\mathrm{e})) - \beta \left(I_\mathrm{ion} + I_\mathrm{stim} \right) \Big] \tag{3}$$

$$\nabla \cdot ((\sigma_\mathrm{i} + \sigma_\mathrm{e})\nabla\phi_\mathrm{e}) = -\nabla \cdot (\sigma_\mathrm{i}\nabla V_\mathrm{m})\,. \tag{4}$$

Equation (3) is used to integrate V_m and (4) is used to compute ϕ_e from V_m at each time step.

An important simplification of the bidomain model is possible by assuming that σ_i is proportional to σ_e. This allows for lumping the two domains together in the integration. Introducing the monodomain conductivity tensor $\sigma'_{\mu\nu} = (\sigma_{\mathrm{i}\mu\nu}\ \sigma_{\mathrm{e}\mu\nu})/(\sigma_{\mathrm{i}\mu\nu} + \sigma_{\mathrm{e}\mu\nu})$ and eliminating ϕ_e from (3) and (4), we obtain the following *monodomain reaction-diffusion model* [19]:

$$\frac{\partial V_\mathrm{m}}{\partial t} = \frac{1}{\beta C_\mathrm{m}}\Big[\nabla \cdot (\sigma'\nabla V_\mathrm{m}) - \beta\,(I_\mathrm{ion} + I_\mathrm{stim})\Big]\,. \tag{5}$$

Especially in whole-heart simulations, a monodomain model approximates a bidomain model very well [19]. By combining (5) with (4) it is still possible to compute ϕ_e, which is of special importance because, in contrast to V_m, it can be measured clinically (Figure 1). By solving (4) less frequently, this approach is much more efficient than a bidomain reaction-diffusion model. Solution techniques for these equations are a subject of continuing research [1,7,16,25].

3 The Propag Code

The purpose of this work was to improve an existing cardiac simulation code, named Propag, [6,19,20], and to study and remove the bottlenecks that prevented it from running efficiently on contemporary massively-parallel computers.

The original code had been developed to solve both mono- and bidomain models on complicated geometries obtained from CT or MRI images of the heart. It was designed to run efficiently on shared-memory machines such as the SGI Altix family, using 16 to 128 cores. Parallelization had therefore been done with OpenMP directives in a NUMA-aware fashion (taking care of memory placement). In practice, the existing code could run heart models up to 100 million nodes in a reasonable amount of time and with good parallel performance. Strong scaling had a fixed limit of about $4 \cdot 10^5$ model nodes per core.

3.1 Characterization of the Code

Propag works with semi-structured finite-difference meshes, i.e., many of the possible node positions are not occupied. The heart or torso anatomy is input as a Cartesian array storing the cell types (tissue type, blood, or void). We refer to the elements of this Cartesian box as *voxels* whereas non-void voxels are called *cells*. Based on the cell types of surrounding voxels, the vertices of the mesh receive types as well. Vertices that are not completely surrounded by void are referred to as (mesh) *nodes*. In the original code, connectivity was computed on the fly. In the new code, the topology is stored explicitly since we cannot control the shapes of individual subdomains in the domain decomposition.

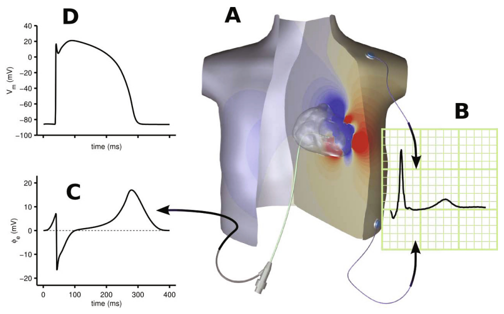


Fig. 1. Visualization of model results. The heart generates a potential field in the torso (A). Electrocardiograms (B) and catheter electrograms (C) can be derived and compared to measured data as well as to the underlying simulated action potentials (D) and dozens of other membrane-related variables.

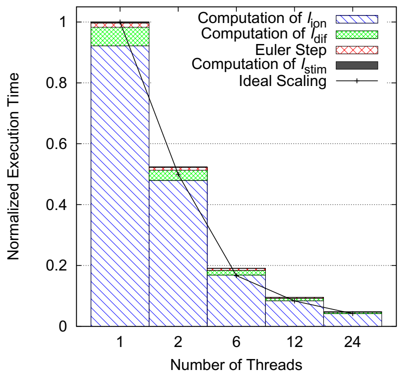


Fig. 2. Scaling of the original PROPAG code in a monodomain run with breakdown of runtime

Algorithm 1. Monodomain Explicit Euler Time Integrator

```
1: Compute $I_{\mathrm{dif}}^{n+1} = \beta^{-1}\nabla \cdot (\sigma' \nabla V_{\mathrm{m}}^{n})$ and $I_{\mathrm{stim}}^{n+1}$
2: Evaluate $I_{\mathrm{ion}}^{n+1} = \texttt{ION\_STEP}(V_{\mathrm{m}}^{n}, I_{\mathrm{dif}}^{n+1}, I_{\mathrm{stim}}^{n+1})$
3: Set $V_{\mathrm{m}}^{n+1} = V_{\mathrm{m}}^{n} + \tau \left[I_{\mathrm{dif}}^{n+1} - I_{\mathrm{stim}}^{n+1} - I_{\mathrm{ion}}^{n+1}\right]$
```

In this article we focus on the monodomain capabilities of the code. Originally, the code used an explicit Euler scheme to solve (5), see Algorithm 1.

In monodomain mode, the computation of I_{ion} in `ION_STEP` dominates the runtime (cf. Figure 2). It consists of a single loop over all mesh nodes and the approximate solution of a set of ordinary differential equations (about 40 in our model) at each node and hence is amenable to parallelization.

In Figure 2, an analysis of the runtime of the original PROPAG is shown. The graph shows a breakdown of the runtime of a monodomain simulation on one 24-core node of a Cray XE6 (equipped with two AMD Opteron 2.1 Ghz "Magny Cours" processors). Due to the NUMA-aware memory allocation and since runtime is distributed over only few scalable tasks of large granularity, the OpenMP parallelization is very efficient and OpenMP management overhead is negligible. The parallel efficiency on 24 cores is 86.9 % for this rather small example (422,091 mesh nodes).

The reaction-diffusion equation (5) contains a stiff diffusion term. With explicit time integration schemes, numerical stability requires a time step size that decreases quadratically with the spatial step size. To enable stable and accurate integration of very large models (with several billion degrees of freedom) we recently implemented an Implicit-Explicit (IMEX) Euler time discretization in PROPAG. Here, the linear diffusion term is treated implicitly while the non-linear ionic current is treated explicitly. In contrast to an explicit integration scheme, the IMEX Euler method requires the solution of a linear system in each time step. In our experience, the matrix in this system is well-conditioned for practical time step sizes so that a few Bi-CGSTAB steps suffice to effectively reduce the (relative) residual norm to the tolerance $\varepsilon = 10^{-8}$.

4 Hybrid Parallelization

The currently largest shared-memory machines are limited to a few thousand cores per machine while the largest distributed-memory architectures scale to hundreds of thousands of cores. To efficiently utilize these resources, we ported Propag to an MPI code that can run on distributed-memory architectures. Such systems usually consist of a large number of multi-socket compute nodes connected by a high-speed interconnect. In recent years, the number of cores per socket has increased significantly. Within a compute node, memory is shared between cores, usually with NUMA architecture. Therefore, we retained the existing OpenMP parallelization, which is efficient for intra-node parallelization, and added an MPI layer for inter-node parallelism. Such a *hybrid* parallelization approach has been used for a variety of codes and has proven beneficial for several reasons:

1. It simplifies adding new levels of concurrency beyond what is easily accomplished with MPI and hence can be used to overcome algorithmic scaling limitations (e.g., GTC [3]).
2. It allows to mitigate efficiency loss in applications that are limited by the scaling of all-to-all communication (e.g., PARATEC [18] and CPMD [8]) or where communication time is a significant part of the runtime.
3. Since the shared memory often renders halo (or overlap) zones unnecessary, hybrid codes can use less memory. If additional work must be performed on the halo, scalability can be enhanced by increasing the number of threads per process (e.g., FISH [10]).
4. It simplifies the load balancing of applications with dynamic or complicated structure since intra-process load balancing is possible using `dynamic` or `guided` loop scheduling (e.g., NPB BT-MZ Benchmark [21]).

It is worth noting, though, that hybrid parallelization is not always beneficial. Mahinthakumar and Saied report no improvement in a hybrid implicit finite element (FE) solver [14]. In general, there are many factors contributing to the performance of hybrid execution and results can vary between simulation setups, cf. [13].

4.1 MPI Parallelization

For the MPI parallelization of the code, we exploited techniques that have proven to be very efficient for the parallelization of general (unstructured) FE applications. Hence, we use a cell-wise distribution of the geometry. The decomposition is computed through an interface to existing graph-partitioning libraries (e.g., ParMETIS [11]). Differently than previous versions of Propag, all arrays range only over cells and nodes and connectivity information is stored explicitly. While this change has a negative impact on single-core performance and the OpenMP scalability of the code (due to additional indirect accessing), it is compensated for by better scalability of the MPI layer.

Since the mesh in Propag is distributed cell-wise, nodes are duplicated on multiple processes. One of these processes is distinguished as the *owner* of the node. For inter-process communication, we use the notion of *communication traces* introduced by Sahni et al. [22]. In Propag a communication trace consists of a set of nodes (located on an inter-process boundary) and the rank of a peer process. On the peer, a matching communication trace is built with a consistent ordering of the interface entities. Hence, by means of a communication trace, inter-process communication is possible without the need for a global numbering of mesh entities. All communication is based on two primitives: The function `SUMUP_AT_OWNER` gathers data on the owner and `COPY_TO_OTHERS` overwrites the data at each copy by the data at the owner (scatter). These communication steps are implemented on top of non-blocking MPI send/receive calls and an extended interface (`START`, `TEST`, `WAIT`) is provided to overlap these operations with computations.

Using these communication primitives, we can rewrite Algorithm 1 as shown in Algorithm 2. The algorithm is written in such a way that it allows for overlapping communication of the diffusion currents with the computation of I_{stim} (to hide the communication in `SUMUP_AT_OWNER`) and with the evaluation of I_{ion} for the interior nodes (to hide `COPY_TO_OTHERS`), assuming the necessary hardware capabilities. In our tests, we have not seen improvements in scalability or runtime due to overlap. Nevertheless, by construction, all receive calls are preposted timely before the `WAIT` call. This is important for good MPI performance on many systems including the targeted Cray XT5.

Algorithm 2. Parallel Monodomain Explicit Euler

```
1: Compute locally $I_{\mathrm{dif}}^{n+1} = \beta^{-1}\nabla \cdot (\sigma' \nabla V_{\mathrm{m}}^{n})$
2: Call SUMUP_AT_OWNER_START($I_{\mathrm{dif}}^{n+1}$)
3: Compute $I_{\mathrm{stim}}^{n+1}$
4: Call SUMUP_AT_OWNER_WAIT($I_{\mathrm{dif}}^{n+1}$)
5: Call COPY_TO_OTHERS_START($I_{\mathrm{dif}}^{n+1}$)
6: Evaluate $I_{\mathrm{ion}}^{n+1}$ = ION_STEP($V_{\mathrm{m}}^{n}, I_{\mathrm{dif}}^{n+1}, I_{\mathrm{stim}}^{n+1}$) for all own nodes
7: Call COPY_TO_OTHERS_WAIT($I_{\mathrm{dif}}^{n+1}$)
8: Evaluate $I_{\mathrm{ion}}^{n+1}$ = ION_STEP($V_{\mathrm{m}}^{n}, I_{\mathrm{dif}}^{n+1}, I_{\mathrm{stim}}^{n+1}$) for all other nodes
9: Set $V_{\mathrm{m}}^{n+1} = V_{\mathrm{m}}^{n} + \tau \left[ I_{\mathrm{dif}}^{n+1} - I_{\mathrm{stim}}^{n+1} - I_{\mathrm{ion}}^{n+1} \right]$
```

4.2 MPI Threading Support

The intra-process parallelization via OpenMP was retained and extended to new code segments. As in the original code, we mostly use `parallel for` worksharing constructs. This approach (in comparison to the use of large parallel sections) incurs some overhead but simplifies the implementation. Experiments with the original code (Figure 2) show that OpenMP overhead does not significantly affect the scalability of the explicit solver.

All MPI calls in Propag are performed outside the parallel sections. Therefore, the minimal level of thread support an MPI implementation must provide is `MPI_THREAD_FUNNELED`. As defined by the standard, this level of thread support suits applications where it is ensured that only the main thread makes MPI calls. In comparison to higher levels of thread support, this does not incur overhead due to locks/mutexes in the MPI implementation.

We do not anticipate savings in communication time by having multiple threads performing communication since the code is limited by latency rather than bandwidth. Using multiple threads for communication can be advantageous if a single thread is incapable of saturating the network interface [21].

5 Performance Analysis

All experiments were performed on a Cray XT5 machine operated by the Swiss National Supercomputing Centre. The system consists of 1844 nodes with two 6-core AMD Opteron 2.4 Ghz "Istanbul" processors per node (22,128 cores in total[1]). The nodes are connected through a Seastar 2+ interconnect.

For our experiments, we consider approximations of a model anatomy (based on CT data of a human heart obtained at autopsy [19]) at different spatial resolutions. We summarize the description of the four considered problem sizes (small, medium, large and extra-large) in Table 1.

Table 1. Problem sizes for experiments

Name	Resolution	#cubes	#nodes
S	0.5 mm	3,024,641	3,200,579
M	0.25 mm	24,197,121	24,900,671
L	0.125 mm	193,576,968	196,390,842
XL	0.0625 mm	1,548,615,744	1,559,870,636

We study strong scaling for the problem sizes **S**, **M**, **L** and **XL**, varying both the number of processes and the number of threads per process, the latter between 1 (one MPI process per core), 6 (one MPI process per socket), and 12 (one MPI process per node). For all setups we start with at least 12 threads. We measure the average time required to perform ten Explicit Euler or IMEX Euler steps, respectively. Every tenth step, an `MPI_ALLREDUCE` is performed to sum up some statistics that have been accumulated locally. For the purpose of our tests, we do not perform significant I/O. For the IMEX runs, we use the Bi-CGSTAB solver with a Jacobi preconditioner and a fixed time step size $\tau = 0.02$ ms.

[1] Due to an interconnect congestion problem, we could not yet perform tests on more than 8448 cores.

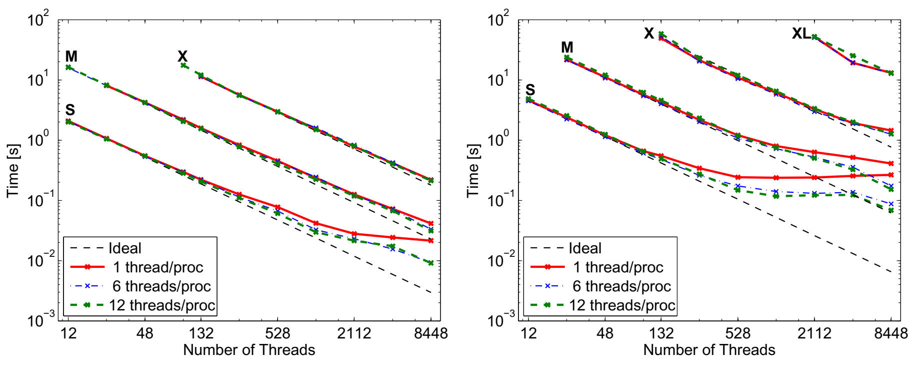


Fig. 3. Scaling of Explicit Euler (left) and IMEX Euler (right) on the Cray XT5. Problem **M** requires at least 24 cores for IMEX Euler or Explicit Euler with one thread per process. **X** requires at least 132 cores for execution (96 when using 12 threads per process). The starting point for the strong scaling study for problem **XL** is 2112 cores.

5.1 Performance of Single-Threaded Execution

In Figure 3, the time per run for the different problem sizes is plotted against the number of threads (i.e., number of processes times threads per process). The code scales well up to 8448 cores for the larger problem sizes. In general, the scaling of the Explicit Euler is much better than the IMEX Euler as the latter requires multiple `MPI_ALLREDUCE` calls per time step and additional point-to-point communication for sparse matrix-vector multiplication.

For **S** on 1056 cores (one thread per process), the IMEX Euler requires $\sim 169\times$ more `MPI_ALLREDUCE` calls than Explicit Euler. At this scale, the code spends 48.0 % of the compute time in the calls to `MPI_ALLREDUCE` (compared to 9.7 % for the Explicit Euler). Hybrid execution can improve this situation, see Section 5.2. Nevertheless, for this small problem size, the code still achieves an efficiency of 56.5 % and 21.9 % on 1056 cores using the Explicit Euler and IMEX Euler, respectively. For larger problems, such as **L**, the parallel efficiency on 8448 cores relative to 132 cores (the minimum required to run the problem) is 81.6 % and 53.2 % for Explicit Euler and IMEX Euler, respectively.

The limits in (strong) scalability of Propag can be linked to two major sources of inefficiency: A relative increase in communication time and a sub-optimal decrease in the degrees of freedom per process.

In Table 2, we report the relative percentage of the average walltime of communication in the main computational loop as reported by the Integrated Performance Monitor (IPM) [9]. The data show that there is an $\sim 4\times$ increase in the relative communication time (both point-to-point and collective) when increasing the number of cores by a factor of 8.

In Table 3 we show the increase in the total number of nodes due to the overlap between subdomains. Due to the cell-based decomposition, nodes on

Table 2. Breakdown of communication time for **S** using Explicit and IMEX integration with one thread per process

#cores	% of walltime in point-to-point communication	% of walltime in collective communication
Explicit Euler		
132	4.91 %	2.31 %
1056	20.10 %	10.07 %

#cores	% of walltime in point-to-point communication	% of walltime in collective communication
IMEX Euler		
132	13.04 %	12.31 %
1056	32.57 %	48.09 %

Table 3. Characteristics of the node distribution during scale-out of **M**

#procs	12	24	528	1056	4224	8448
% Increase in #nodes	1.58	2.33	10.14	13.25	22.55	29.67

inter-process boundaries must be duplicated so that the total number of nodes (where copies are accounted for) grows with the number of processes. As can be seen in Table 3, the number of nodes has grown by almost 30 % on 8448 cores. Using an argument similar to that of Amdahl's law, we can derive an upper bound for the parallel efficiency as the ratio between the total number of nodes in serial and parallel. In our example, the maximum attainable efficiency when scaling from 12 to 8448 cores is 78.3 %. A similar finding was reported by Sahni et al. [22] in the context of an unstructured FE solver.

5.2 Benefits of Hybrid Execution

In Section 5.1, we have identified two major sources of scalability loss in Propag. In this section, we will analyze how hybrid execution, using multiple threads per process, allows to mitigate these inefficiencies.

In Table 4, we present a breakdown of the communication time for the problem size **S**. The results for runs with one thread per process correspond to the results in Table 2. Unlike before, Table 4 contains absolute communication times (for 1010 time steps) to allow for comparing the results from different runs. Our results show that the use of multiple threads per process can significantly reduce the communication time. Using 6 or 12 threads per process reduces the time in `MPI_ALLREDUCE` by 22–52% or up to 61 %, respectively. Similarly, T_{Pt2Pt} is decreased by 22–64 % or 5–72 % for 6 or 12 threads. Interestingly though, a smaller number of processes does not always imply lower communication cost since the T_{Pt2Pt} for 11×12 threads is larger than for 22×6 threads. Using more threads per process leads to larger buffer sizes. This results in an improved bandwidth utilization but also increased latency.

Table 4. Breakdown of communication time for **S** using Explicit and IMEX Euler. T_{Pt2Pt} and T_{Coll} denote point-to-point and collective communication time, respectively.

#cores	procs × threads/proc	T_{Pt2Pt}	T_{Coll}
Explicit Euler			
132	132×1	5.12 s	2.41 s
	22×6	3.99 s	1.37 s
	11×12	4.87 s	2.34 s
1056	1056×1	3.90 s	1.95 s
	176×6	2.43 s	0.95 s
	88×12	2.25 s	0.76 s

#cores	procs × threads/proc	T_{Pt2Pt}	T_{Coll}
IMEX Euler			
132	132×1	35.53 s	33.55 s
	22×6	20.86 s	25.89 s
	11×12	12.18 s	14.99 s
1056	1056×1	38.13 s	56.29 s
	176×6	13.73 s	39.81 s
	88×12	10.52 s	33.46 s

Table 5. Percentage increase in #nodes for **M** with 1, 6, and 12 threads per process

#cores / threads	12	24	528	1056	4224	8448
1	1.58	2.33	10.14	13.25	22.55	29.67
6	0.40	0.84	4.82	6.55	11.31	14.87
12	0.00	0.40	3.33	4.82	8.79	11.31

In Section 5.1, we have noted that a strict upper limit for the parallel efficiency in Propag exists due to the growth of node copies on inter-process boundaries. For the intra-process parallelization based on OpenMP worksharing constructs, no overlap is required. When keeping the total number of threads constant, using more threads per process will result in fewer node copies. In Table 5, we show that this results in a strong reduction of the number of additional nodes. Consequently, the theoretical upper bound for the efficiency improves: When using 12 threads per process, efficiency when going from 12 to 8448 cores is bounded by 89.8 % (rather than 78.3 %, cf. Section 5.1). We measure an efficiency of 74% for the Explicit Euler solver which seems to be practically impossible to achieve with a pure MPI version.

The actual, measured improvement of the hybrid code (running with 6 or 12 threads per process, respectively) is shown in Figure 4. For the case of the Explicit Euler, threaded execution is beneficial starting at 96 cores. The code on 1056 cores with 6 threads per process shows an unexpectedly bad performance that we cannot explain yet. For the IMEX Euler, which is more strongly limited by communication time, execution with 6 threads per process is advantageous already at 24 cores; execution with 12 threads per process is advantageous for 528 cores or more. When 2112 cores or more are used, running with 12 threads per process is faster than running with 6 threads per process.

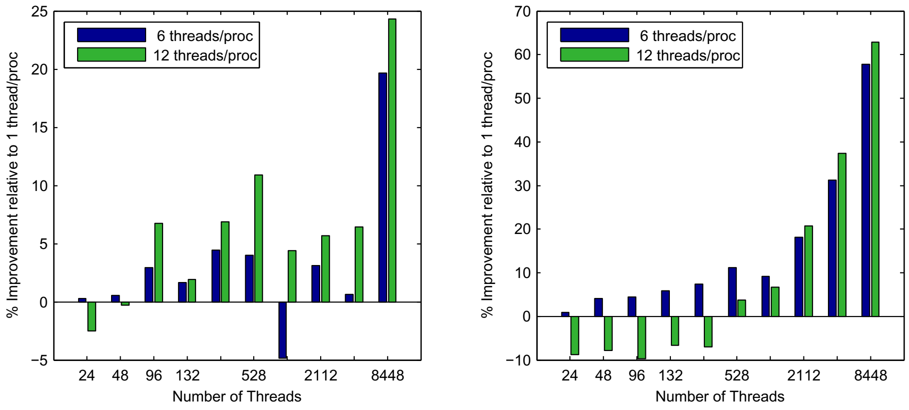


Fig. 4. Improvement through hybrid execution for Explicit (left) and IMEX Euler (right) relative to pure MPI for **M** on the Cray XT5

6 Conclusion

We have presented the successful hybrid parallelization of a large-scale heart model. Performance was measured in monodomain simulations with up to 1.5 billion nodes. These system sizes are among the largest reported in the literature for this scientific problem.

We have shown that hybrid parallelization can improve scalability of this application as it 1) decreases the relative and absolute communication time and 2) reduces the size of the overlap between adjacent subdomains. We have analyzed both effects separately and have demonstrated runtime reductions up to 24 % for an Explicit Euler and up to 62 % for an IMEX Euler time discretization.

Acknowledgments. Computational resources for this work were provided by the Università della Svizzera italiana (USI), the Swiss National Supercomputing Centre (CSCS), and the Réseau québécois de calcul de haute performance (RQCHP). This work was supported by the project "A High Performance Approach to Cardiac Resynchronization Therapy" within the context of the "Iniziativa Ticino in Rete" and the "Swiss High Performance and Productivity Computing" (HP2C) Initiative.

References

1. Bordas, R., Carpentieri, B., Fotia, G., Maggio, F., Nobes, R., Pitt-Francis, J., Southern, J.: Simulation of cardiac electrophysiology on next-generation high-performance computers. Phil. Trans. Roy. Soc. A. 367, 1951–1969 (2009)
2. Desplantez, T., Dupont, E., Severs, N.J., Weingart, R.: Gap junction channels and cardiac impulse propagation. J. Membrane Biol. 218, 13–28 (2007)

3. Ethier, S., Tang, W.M., Lin, Z.: Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. J. Phys. Conf. Ser. 16(1), 1–15 (2005)
4. Henriquez, C.S.: Simulating the electrical behavior of cardiac tissue using the bidomain model. CRC Crit. Rev. Biomed. Eng. 21, 1–77 (1993)
5. Hille, B.: Ion Channels of Excitable Membranes. Sinauer Associates, Inc., Sunderland (2001)
6. Hoogendijk, M.G., et al.: Mechanism of right precordial ST-segment elevation in structural heart disease: Excitation failure by current-to-load mismatch. Heart Rhythm 7, 238–248 (2010)
7. Hooke, N., Henriquez, C.S., Lanzkron, P., Rose, D.: Linear algebraic transformations of the bidomain equations: Implications for numerical methods. Math. Biosci. 120(2), 127–145 (1994)
8. Hutter, J., Curioni, A.: Dual-level parallelism for ab initio molecular dynamics: Reaching teraflop performance with the CPMD code. Parallel Comput. 31(1), 1–17 (2005)
9. IPM Homepage (2009), `http://ipm-hpc.sourceforge.net/`
10. Kaeppeli, R., Whitehouse, S.C., Scheidegger, S., Pen, U.L., Liebendörfer, M.: FISH: A 3D parallel MHD code for astrophysical applications. Technical Report arXiv:0910.2854 (2009)
11. Karypis, G., Kumar, V.: A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In: Parallel Processing for Scientific Computing. SIAM (1997)
12. Kléber, A., Rudy, Y.: Basic mechanisms of cardiac impulse propagation and associated arrhythmias. Physiol. Rev. 84, 431–488 (2004)
13. Loft, R., Thomas, S., Dennis, J.: Terascale spectral element dynamical core for atmospheric general circulation models. In: ACM/IEEE 2001 Conference on Supercomputing (2001)
14. Mahinthakumar, G., Saied, F.: A hybrid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures. Int. J. High Perform. C. 16(4), 371–393 (2002)
15. Mitchell, L., Bishop, M., Hötzl, E., Neic, A., Liebmann, M., Haase, G., Plank, G.: Modeling cardiac electrophysiology at the organ level in the peta flops computing age. In: AIP Conference Proceedings, vol. 1281(1), pp. 407–410 (2010)
16. Niederer, S., Mitchell, L., Smith, N., Plank, G.: Simulating a human heart beat with near-real time performance. Front. Physio. 2, 14 (2011)
17. Noble, D., Rudy, Y.: Models of cardiac ventricular action potentials: Iterative interaction between experiment and simulation. Phil. Trans. Roy. Soc. London; Phys. Sc. 359, 1127–1142 (2001)
18. PARAllel Total Energy Code, `http://www.nersc.gov/projects/paratec`
19. Potse, M., Dubé, B., Richer, J., Vinet, A., Gulrajani, R.M.: A comparison of monodomain and bidomain reaction-diffusion models for action potential propagation in the human heart. IEEE Trans. Biomed. Eng. 53, 2425–2435 (2006)
20. Potse, M., Dubé, B., Vinet, A.: Cardiac anisotropy in boundary-element models for the electrocardiogram. Med. Biol. Eng. Comput. 47, 719–729 (2009)
21. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (2009)
22. Sahni, O., Zhou, M., Shephard, M.S., Jansen, K.E.: Scalable implicit finite element solver for massively parallel processing with demonstration to 160k cores. In: ACM/IEEE 2009 Conference on Supercomputing (2009)

23. ten Tusscher, K.H., Panfilov, A.V.: Alternans and spiral breakup in a human ventricular tissue model. Am. J. Physiol. 291(3), H1088–H1100 (2006)
24. Trayanova, N., Aguel, F.: Computer simulations of cardiac defibrillation: A look inside the heart. Comput. Vis. Sci. 4, 259–270 (2002)
25. Vigmond, E.J., Aguel, F., Trayanova, N.A.: Computational techniques for solving the bidomain equations in three dimensions. IEEE Trans. Biomed. Eng. 49, 1260–1269 (2002)

Efficient AMG on Heterogeneous Systems

Jiri Kraus and Malte Förster

Fraunhofer Institute for Algorithms and Scientific Computing SCAI
Schloss Birlinghoven, 53754 Sankt Augustin, Germany

Abstract. In many numerical simulation codes the backbone of the application covers the solution of linear systems of equations. Often, being created via a discretization of differential equations, the corresponding matrices are very sparse. One popular way to solve these sparse linear systems are multigrid methods - in particular AMG - because of their numerical scalability. As the memory bandwidth is usually the bottleneck of linear solvers for sparse systems they especially benefit from high throughput architectures like GPUs. We will show that this is true even for a rather complex hierarchical method like AMG. The presented benchmarks are all based on the new open source library LAMA and compare the run times on different GPUs to those of an efficient OpenMP parallel CPU implementation. As the memory access pattern is especially crucial for GPUs we have a focus on the performance of different sparse matrix formats.

Keywords: LAMA, AMG, GPU, CUDA.

1 Introduction

In this paper we show that it is possible to gain a significant performance increase when GPUs are used for the solution phase of AMG. To achieve this it is necessary to execute the full AMG cycle on a GPU with massively parallel components. By using Jacobi smoothing the full AMG cycle essentially cuts down to a series of sparse matrix vector multiplications (SpMV). So it is possible to achieve good AMG performance for the solver phase if we have good SpMV performance. We show that the popular CSR format does not lead to acceptable performance on GPUs, at least if the rows are not padded like done by Baskaran and Bordawekar[7]. Instead, more GPU-suitable formats like ELLPACK or JDS are needed. ELLPACK has been successfully used for a GPU AMG implementation by Feng and Zeng[9]. Also Haase, et. al. have been successfully implemented a AMG for GPUs using the interleaved compressed row storage format[11] which is quite similar to our JDS implementation. In contrast to their publications we focus on well known model problems to report comprehensible results. This has been already the approach for our last publication "Scalable parallel AMG on ccNUMA machines with OpenMP"[10], where we have compared our AMG implementation to the open source solver packages PETSc and hypre[2,1]. This CPU implementation also is the baseline for our GPU benchmarks. To make it easier to classify our results this paper follows the structure of our aforementioned paper and uses the same hardware and model problems.

We start with a short introduction of LAMA the library we used for our AMG implementation. In the following section 3 we describe our hard- and software setup and the execution environment we used. After describing the model problems, the used AMG Setup and the used sparse matrix formats we present the obtained benchmark results. The benchmarks evaluate how the performance is influenced by the matrix storage format and the precision of the calculations.

2 LAMA

The Library for Accelerated Math Applications, LAMA, is a new open source project which is available at `http://www.libama.org`. The first of two main design aims of LAMA is to allow easy integration of accelerators like GPGPUs. As a consequence to this the second main design aim is to be extensible with new matrix storage schemes while supporting a natural mathematical syntax without sacrificing performance, like it is also achieved by the C++ Library Blitz++[3]. To achieve both goals LAMA is separated into two parts. A C library which provides BLAS functionality for dense and sparse types and which is used to utilize all types of accelerators and a C++ part which supports the extensibility and provides the natural mathematical syntax. The C library makes our core algorithms of our library usable by a wide range of applications and allows the integration of existing BLAS Libraries. The C++ part uses simplified expression templates [16] to achieve the second design aim. Utilizing this and by formulating solvers only in terms of simple BLAS operations, like they are printed in text books[6], we achieve very comprehensible solver implementations and it is easy to experiment with new accelerators or data structures, e.g. different sparse matrix formats. The results obtained in this paper have been produced with a version of LAMA that is mainly using "compile time polymorphism" through templates. This enables aggressive compiler optimizations while sacrificing some run time flexibility. A very similar approach is taken by the LAToolbox[12] which is part of the HiFlow3 FEM solver package[5].

Using LAMA we had only a minimal implementation effort to make the AMG implementation, that we also used in our previous publication, run with CUDA and OpenCL. It was only necessary to implement SpMV for the tested sparse matrix storage formats within the back ends for CUDA and OpenCL. In addition we also implemented specializations of the Jacobi smoother for the tested sparse matrix storage formats for both back ends. Although this would not have been necessary from a functional point of view, since we also implemented a universal Jacobi based on just SpMV, the specialized version is slightly more efficient. Because we have also used a specialized version of the Jacobi smoother in the OpenMP back end this was necessary to have a fair comparison.

3 Hardware and Software Setup

The CPU results presented in this paper have been computed on the hardware described in table 1. This is one of the systems we have used in our previous publication [10], where it was named BULL. All binaries have been build with gcc version 4.4.3 and the optimization options `-O3` and `-ffast-math`.

3.1 GPUs

The GPU benchmarks have been done with the GPUs that are listed in table 2. We have used CUDA 3.2 for the GPU benchmarks. To compile the GPU kernels we have used the options `-arch=sm_13` and `-use_fast-math` in all cases. If not otherwise mentioned all measurements have been done in double precision with disabled ECC and enabled Texture cache for the access to the input vector in SpMV operations and Jacobi iterations.

Table 1. CPU Hardware

name	CPU
cpu	Xeon X5650
core freq.	2.67 GHz
L3-cache	12 MB
cores/cpu	6
HT	off
sockets	2
cores	12
memory	12 GB
BW (GB/s)	32

Table 2. GPUs used

name	G46	G48	T10	T20
device	GeForce GTX 460	GeForce GTX 480	Tesla C1060	Tesla C2050
compute cap.	2.1	2.0	1.3	2.0
multiprocessors	7	15	30	14
cores	336	480	240	448
core freq	1.43 GHz	1.40 GHz	1.30 GHz	1.2 GHz
memory	1 GB	1.5 GB	4 GB	3 GB
BW (GB/s)	115	177	102	144
HW Cache	yes	yes	no	yes

An introduction to CUDA or GPU programming can be found in [13]. We just want to highlight that in contrast to a CPU a GPU is designed as a high throughput architecture. This has certain implications for the performance characteristics of these devices. Because the algorithm under examination is memory bound the most crucial point for us is the high memory bandwidth of GPUs. This high memory bandwidth comes at the cost of a high access latency and strict coalescing requirements to achieve the full memory bandwidth[8]. To overcome the high latency a GPU can manage a lot more threads concurrently than there are compute cores. If enough threads are available it is possible to hide the high access latency to the GPU memory, by switching between threads that are waiting and threads that are ready to run.

To achieve good performance on a GPU this means that it is necessary to have a high degree of parallelism and regular memory accesses. For the chosen solver the high degree of parallelism is given. The regular memory access however is dependent on the chosen storage format for sparse matrices. The influence of the sparse matrix format on the coalescing is described in section 5.1.

4 Execution

All benchmarks have been computed via the benchmark framework integrated in the LAMA package. This framework running in Python ensures reproducible run times by creating new processes for every test run, eliminating the possibility of a benchmark influencing its followers with respect to memory usage. Within every benchmark process, the reported run time is the minimum of 5 executions. Additionally, every process is started at least 3 times. In case aberrations have to be eliminated here this number will automatically increase. Since GPUs do not support preemption and therefore deliver reproducible results this feature is in general only triggered within CPU benchmarks.

4.1 Model Problems

Our set of test matrices is shown in table 3. It consists of different discretizations of the Laplacian operator on structured grids in up to three dimensions. All matrices have a total of 1 million rows but increase in the number of nonzero entries. Each row corresponds to exactly one grid point and its nonzero values refer to the entries of the differential stencil applied. We have chosen these model problems because they are well known, which makes it more easy to compare our results[8]. Additionally, they are a good measure for real world 1D, 2D and 3D applications because of the basic local access patterns common for matrices based on a wide range of PDE applications.

Table 3. Laplacian discretizations used for solver benchmarks

name	dimensions	diags	entries	CSR mem
1D3P	1,000,000	3	3 Mio.	38 MB
2D5P	1,000x1,000	5	5 Mio.	61 MB
3D7P	100x100x100	7	7 Mio.	83 MB
2D9P	1,000x1,000	9	9 Mio.	107 MB
3D27P	100x100x100	27	27 Mio.	307 MB

To exploit the sparsity all matrices are stored in either Compressed Sparse Row (CSR), Jagged Diagonal Storage (JDS) or ELLPACK format using both, single and double precision. More details about the Matrix formats will be given in Section 5.

To make it easier to compare our work with the results of others table 4 contains the theoretically calculated complexity of 10 AMG preconditioned CG iterations for all evaluated matrix storage schemes and all model problems. The achieved memory bandwidth and compute performance for the different devices and model problems can be easily derived from these numbers and the given execution times. These reference numbers are counted based on the assumptions made in section 6 to allow the comparison of different hardware, even if they differ in the capabilities of the available hardware performance counters.

Table 4. Complexity of 10 CG-AMG Iterations

	CSR			ELL			JDS		
	GB-S	GB-D	GFLOP	GB-S	GB-D	GFLOP	GB-S	GB-D	GFLOP
1D3P	8.24	14.80	1.26	6.96	13.68	1.30	8.12	13.40	1.26
2D5P	13.85	23.99	2.26	13.06	23.53	2.34	14.53	23.61	2.26
2D9P	17.09	29.24	2.91	16.52	28.99	2.98	18.41	29.71	2.91
3D7P	20.62	35.25	3.43	20.96	36.64	3.70	22.02	35.63	3.43
3D27P	36.80	62.01	6.56	38.39	65.27	6.98	40.95	65.43	6.56

4.2 AMG Setup Phase

Due to its complexity and partially sequential nature, the setup phase of AMG in LAMA is computed on the CPU. This includes coarse grid definitions, interpolation and restriction constructions as well as the multiplication of the galerkin operators. As a coarsening strategy we use the classical Ruge-Stüben algorithm[15] (1stage) in combination with standard interpolation.

Table 5 shows the galerkin operator stats of the resulting hierarchies for the corresponding 2D and 3D stencils.

Table 5. Galerkin Operator stats for 2D and 3D stencils

	2D5P		2D9P		3D7P		3D27P	
Lvl	Rows	Entries	Rows	Entries	Rows	Entries	Rows	Entries
0	1000000	4996000	1000000	8988004	1000000	6940000	1000000	26463592
1	500000	4492002	250000	6220036	500000	9320600	125000	13642048
2	125000	3105014	62500	2776594	83331	6321285	14456	2762872
3	31250	1380362	15625	684745	10458	1611064	1317	318939
4	7813	338689	3126	120954	966	171576	181	25235
5	1563	59057	601	21161	133	13507	-	-
6	312	10388	121	3405	-	-	-	-
7	60	1452	-	-	-	-	-	-

The coarsening rates, and therefore also the number of levels constructed, are strongly related to the stencil size as well as the problem dimension. These levels in return will define the amount and shape of the SpMV operations used within each AMG V-cycle in the solution phase later on.

Besides the classical meaning of an AMG setup phase our solver initialization also includes the setup of the coarsest grid inverse as well as the needed data conversions and transfers for the GPU devices. Since this whole process is currently only partially parallelized and optimized, we will not consider it for the benchmarks but focus on the solution phase.

4.3 AMG Solution Phase

Here we describe the implementation of the AMG solution phase which is basically a series of SpMV operations. For the benchmarks we measure 10 iterations of a CG solver preconditioned with AMG. In the solution phase AMG is running a V-cycle performing two pre- and post-smoothing steps with a weighted Jacobi.

Although it has no effect on the performance analysis later on, table 6 exemplary shows the convergence history of the resulting AMG approach applied to the 2D9P stencil.

Table 6. L2-residual reduction for 2D9P

Iter	0	1	2	3	...	8	9	10
LAMA	$1.9E+2$	$2.6E+1$	$2.4E+0$	$1.8E-1$	...	$3.2E-7$	$3.5E-8$	$1.7E-9$

Please keep in mind that we only measure the run times of the solution phase. The transfer of the matrices to GPU memory is considered to be a part of the setup. Theoretically, these transfer costs could also be hidden behind the computation of subsequent level operators. Besides that, also the transfer of the rhs and the solution is not considered in the given run times. This is because they remain constant for a given problem size, independently of the number and complexity of the AMG cycles performed. For our benchmarks the transfer costs for the needed uploads of right hand side and first guess as well as the download of the solution are given in Table 7. The transfer times have been measured with paged locked host memory that enables dma transfers and is necessary to allow asynchronous transfers.

Table 7. Transfer cost of rhs, 1st guess and solution

	float	double
Transfer	$2,4ms$	$4,8ms$
Bandwidth	$4.65GB/s$	$4.65GB/s$

This shows that even with comparably small amounts of data transferred one can utilize a quite satisfying percentage of the maximal available PCI-Express bandwidth of $6GB/s$.

5 Matrix Formats

There are many different storage formats available for sparse matrices. In this paper we will focus on three of them which are quite diverse in their advantages and disadvantages. They will be introduced briefly by showing the main storage vectors for the test matrix in figure 1.

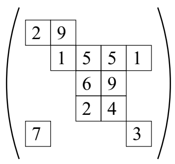


Fig. 1. Example 5x5 matrix

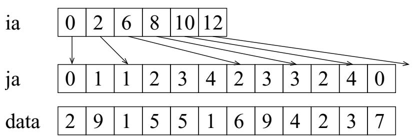


Fig. 2. The CSR storage format

5.1 The CSR Format

The first and probably most commonly used format (at least on CPUs) is the compressed sparse row format (CSR). It keeps all matrix data in two index arrays *ia* and *ja*, as well as the actual matrix values in *data*. The array *ia* keeps track of the start and end of each row in the other two arrays as shown in figure 2, while *ja* and *data* give the column index and value of each nonzero element. To ensure fast access to the diagonal elements of the matrix they are always stored first in each row.[1]

The storage amount is fixed by the number of rows and non zeros and there are no additional requirements for the matrix pattern. Therefore the CSR format is one of the most universal sparse formats available.

To give a baseline of our GPU AMG implementation we compare the run times in single and double precision to the results obtained on the CPU for the popular CSR format. The presented double precision run times on the CPU have been validated against PETSc and boomer AMG from the hypre package[2,1]. This has been done in our aforementioned publication[10]. Table 8 list the run times for all model problems and for all the hardware mentioned in the tables 1 and 2 in single and double precision. The first column shows the identifier of the tested hardware in this column S means a serial run, 1 means a run that uses one socket (6 Cores) of the CPU system and 2 means a run that uses two sockets (12 Cores) of this system. The other identifiers are the GPUs from table 2.

As one can see from table 8 the execution times for the GPUs are dramatically increasing with the complexity of the model problems. To understand that remember the implementation of the AMG solver phase, which is described in section 4.3. As described there it mainly consist of SpMV operations and is therefore memory bound. Given that the dramatic performance drop for more complex problems can be easily explained by the fact that the memory system of the tested GPUs does not run at full speed. This is due to the decreasing memory coalescing with growing numbers of none zeros per row[8].

5.2 The ELLPACK Format

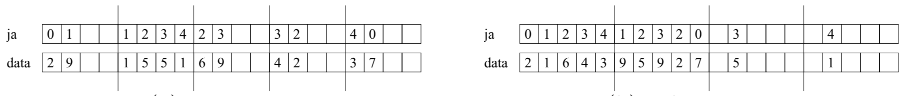
Looking towards GPUs or vector-processors in general one needs to ensure more regularity in memory access in order to achieve good performance. The sparse

[1] The CSR SpMV kernel can be found in our SVN Repository at sourceforge in `trunk/src/lama/lama_CSPBLAS_level2_cuda.cu`

Table 8. Execution Times for the CSR format in seconds

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
S	1.15	1.17	1.78	1.87	2.01	2.13	2.60	2.86	4.09	4.70
1	0.34	0.58	0.53	0.86	0.62	0.98	0.78	1.25	1.30	1.99
2	0.18	0.32	0.27	0.51	0.32	0.53	0.41	0.67	0.67	1.05
G46	0.11	0.17	0.77	0.96	1.40	1.75	1.98	2.39	6.02	6.45
G48	0.08	0.12	0.51	0.62	0.94	1.10	1.28	1.53	3.81	4.06
T10	0.20	0.30	0.74	0.90	1.14	1.29	1.52	1.63	2.93	3.00
T20	0.09	0.13	0.60	0.74	1.08	1.33	1.54	1.84	4.55	4.87

format provided by the ELLPACK package[4] ensures easier storage under the additional requirement of an equally number of non zeroes per matrix row. Because of this assumption it does not need the array *ia* but might artificially increase the number of non zeroes as shown in figure 3(a).



(a) row-wise (b) column-wise

Fig. 3. The ELLPACK storage format

To allow coalesced memory access in terms of multiple threads reading a sequence of matrix rows at once it is also beneficial to store the nonzero Elements of the matrix column-wise as shown in figure 3(b), which is how ELLPACK is stored on GPU devices for the benchmarks in this paper.[2]

While the number of additional artificial elements needed to meet the storage requirements for ELLPACK might not be high for the actual system matrix created from a differential stencil of some specific pattern, this does not need to hold when looking at the whole AMG hierarchy of matrices. Especially Interpolation operators are usually very unbalanced in terms of nonzero entries. Table 9 shows the overall overhead of artificial non zeros throughout the AMG hierarchy of matrices.

The run times of the benchmarks with the ELLPACK format are given in table 10. The table is formatted like table 8 in section 5.1. The run time for the model problems 3D27P in double precision on the GeForce GTX 460 is missing because the available global memory on this device is not large enough to store the whole solver setup in addition to the texture memory reserved by the desktop environment.

[2] The ELLPACK SpMV kernel can be found in our SVN Repository at sourceforge in `trunk/src/lama/lama_CSPBLAS_level2_cuda.cu`

Table 9. Storage overhead of ELLPACK versus CSR for the AMG hierarchy

Stencil	1D3P	2D5P	2D9P	3D7P	3D27P
overhead	14%	14%	11%	17%	13%

Table 10. Execution Times for the ELL format in seconds

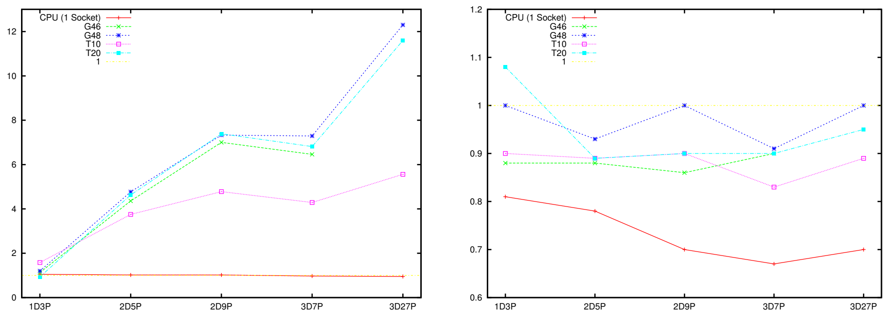
	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
S	0.97	1.12	1.62	1.87	1.90	2.17	2.51	2.95	4.10	4.74
1	0.31	0.55	0.52	0.84	0.61	0.96	0.81	1.29	1.36	2.10
2	0.15	0.27	0.27	0.44	0.31	0.53	0.42	0.66	0.70	1.07
G46	0.10	0.15	0.16	0.22	0.19	0.25	0.26	0.37	0.43	N/A
G48	0.07	0.10	0.10	0.13	0.11	0.15	0.16	0.21	0.25	0.33
T10	0.11	0.19	0.15	0.24	0.17	0.27	0.25	0.38	0.38	0.54
T20	0.08	0.14	0.12	0.16	0.13	0.18	0.19	0.27	0.30	0.42

As one can see from figure 4(a) all tested GPUs have a huge benefit from the ELLPACK format. This can be explained with the same arguments like the bad performance of the CSR format on GPUs. Because we use a column major order layout of the ELLPACK format on the GPU all accesses to the input matrix are perfectly coalesced and therefore no memory bandwidth is wasted[8]. For the CPU version of ELLPACK we are using row major order storage to have a good cache utilization while accessing the input matrix. Although the performance of ELLPACK remains constant its speedup increases because the CSR performance is decreasing for the more complex system like it has been described in section 5.1.

The CPU run times are nearly not affected by the storage format because the additionally stored matrix elements of the ELLPACK format are compensated by fewer indirect memory accesses and the fact that we can save the storage of one integer array. It would be possible to further optimize the ELLPACK format on the CPU, but this has not been done because it was not in the focus of this work.

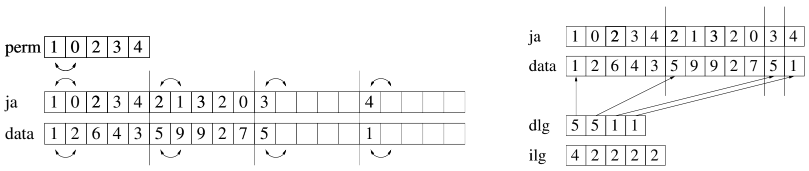
5.3 The JDS Format

For many applications the additional storage of artificial zeroes are a knock-out criterion for the ELLPACK format. For matrices with few long rows this overhead will turn out to be much higher than the percentage given in table 9. To have a more general matrix storage structure we will also look at the Jagged Diagonal Sparse (JDS) format, which has been known to perform well on graphics cards as shown in [14]. The format is a compromise between highly efficient coalesced memory accesses on GPUS from ELLPACK and the flexibility of CSR.



(a) ELLPACK vs. CSR (b) JDS vs. ELLPACK

Fig. 4. Speedup between Storage formats in double precision (colors on line)



(a) permutation of ELLPACK (b) final arrays

Fig. 5. The JDS storage format

The arrays for JDS are very similar to the column-wise ELLPACK storage in figure 3(b). To remove the artificial elements we have to reorder the rows by size and store the permutation in the array *perm*. Now this permutation is also applied to every column in the arrays *ja* and *data* as shown in figure 5(a), sorting all matrix entries to the front of every column.

Given an extra array for each column size *dlg*, it is now safe to remove all artificial values. Additionally we store an array *ilg* for the number of elements in each row which will give us easier access in certain loops. Note that in general only one of the arrays *dlg* and *ilg* is needed, the second one is purely optional. To avoid multiple writes to the output vector and to allow parallel execution of a JDS SpMV operation we are using *dlg* and omiting *ilg*[3]. Figure 5(b) shows the modified arrays for JDS.

Looking at the run times of JDS in table 11 the first thing to notice are the comparably bad timings on the cpu. While there was a row-wise implementation for ELLPACK on the CPU we only support column-wise ordering for JDS. Of course this results in rather bad memory access patterns on a non-vector architecture. In comparison to ELLPACK there are less matrix elements to load

[3] The JDS SpMV kernel can be found in our SVN Repository at sourceforge in `trunk/src/lama/lama_CSPBLAS_level2_cuda.cu`

Table 11. Execution Times for the JDS format in seconds

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
S	1.27	1.41	3.45	3.91	4.40	5.99	7.02	8.06	10.04	12.14
1	0.37	0.62	0.77	1.08	0.98	1.37	1.5	1.92	2.27	3.01
2	0.21	0.33	0.42	0.59	0.55	0.81	0.85	1.07	1.23	1.56
G46	0.10	0.17	0.15	0.25	0.17	0.29	0.26	0.41	0.40	0.62
G48	0.08	0.10	0.10	0.14	0.11	0.15	0.17	0.23	0.24	0.33
T10	0.13	0.21	0.18	0.27	0.21	0.30	0.32	0.46	0.45	0.61
T20	0.09	0.13	0.11	0.18	0.13	0.20	0.20	0.30	0.29	0.44

within the JDS format, but this comes at the cost of two additional vectors *perm* and *dlg*.

Comparing the run times of the JDS directly to ELLPACK we see only slight differences as shown in figure 4(b). Besides that JDS is much more flexible than ELLPACK looking at general matrix patterns with diverse row lengths that would lead to much higher padding overhead.

6 Single Precision vs. Double Precision Performance

The run times with single precision arithmetic are given in tables 8, 10 and 11. A rationale for these numbers can again be derived from the characteristics of a SpMV. If we state that

– the execution time for SpMV is limited by the memory bandwidth
– we ignore the existence of caches.

we can calculate the possible speedup of single precision calculation over a double precision calculation for a SpMV theoretically.

To do that, let A be our input matrix with n rows, n columns and for simplicity k none zero elements per row. To do a SpMV depending on the storage format the following values need to be accessed:

CSR	ELL	JDS	
$n \cdot k$	$n \cdot k$	$n \cdot k$	acc. to the input vector
n	n	n	acc. to the output vector
$n \cdot k$	$n \cdot k$	$n \cdot k$	acc. to the none zero elements of A
$n \cdot k$	$n \cdot k$	$n \cdot k$	acc. to the column index array of A
n	0	0	acc. to the row index array of A
0	0	n	acc. to the permutation array of A
0	0	$n \cdot k$	acc. to the *dlg* index array of A
$n \cdot (2 \cdot k + 1)$	$n \cdot (2 \cdot k + 1)$	$n \cdot (2 \cdot k + 1)$	acc. to floating point values
$n \cdot (k + 1)$	$n \cdot k$	$n \cdot (2 \cdot k + 1)$	acc. to integer values

With the size of a single precision floating point value being $4b$, a double precision value $8b$ and a integer value $4b$ this leads to

CSR	ELL	JDS	
$n \cdot (5 \cdot k + 3) \cdot 4$	$n \cdot (5 \cdot k + 2) \cdot 4$	$n \cdot (6 \cdot k + 3) \cdot 4$	bytes in double precision
$n \cdot (3 \cdot k + 2) \cdot 4$	$n \cdot (3 \cdot k + 1) \cdot 4$	$n \cdot (4 \cdot k + 2) \cdot 4$	bytes in single precision

Because the accesses to the row index array do not grow with the number of none zeros the ratio of double precision to single precision gets bigger with increasing values of k for CSR. For ELL the ratio is getting smaller with increasing values of k and for JDS they remain constant. This leads to the following upper bounds for the theoretical speedup of single precision over double precision.

CSR	ELL	JDS
$\sup_{k>=1} \frac{(5 \cdot k+3)}{(3 \cdot k+2)} = \frac{5}{3}$	$\sup_{k>=1} \frac{(5 \cdot k+2)}{(3 \cdot k+1)} = \frac{7}{4}$	$\sup_{k>=1} \frac{(6 \cdot k+3)}{(4 \cdot k+2)} = \frac{3}{2}$

Taking into account that the peak single precision performance is 8-times of its double precision performance for a Tesla C1060 and 2-times for the other tested GPUs it is obvious that the double precision compute performance is not the bottle neck for a SpMV operation with double precision. That this is not only a theoretical investigation can be seen in figure 6. The fact that the speedup is larger than the stated maximum for the model problem 1D3P can be explained by cache effects (L1/L2/Texture). Because for smaller stencils we have a better cache utilization which makes the theoretical calculation to pessimistic. The more significant single precision performance drop for the CSR format on the GPUs can be explained by the fact that memory coalescing is a lesser issue for double precision, because the doubled size means that only 8 consecutive elements need to be accessed to exploit the available memory bandwidth.

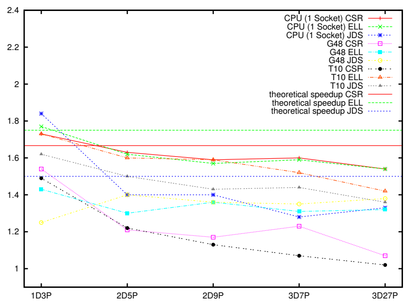


Fig. 6. Speedup of the single precision vs. double precision (colors online)

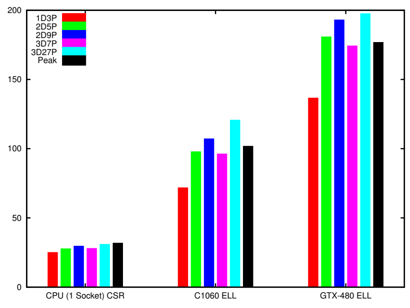


Fig. 7. Memory Bandwidth utilization for double precision (colors online)

7 Conclusion

We have shown that the incorporation of GPUs for AMG can give a performance boost for the solution phase if the right sparse matrix format is chosen. Based on the cache-ignoring memory model from section 6 one can see in figure 7 that the performance - in terms of memory saturation - is nearly optimal. The evaluation of different GPUs shows that this is even true for cheap devices like the GeForce GTX 460. Supplementary we have taken a closer look at the aspect of single precision calculation. Looking at this we want to accentuate that the theoretical disadvantage for double precision calculations of GPUs is really no issue for memory bound algorithms, like AMG.

8 Future Work

Choosing the right sparse matrix format makes the GPU a really good piece of hardware to compute the solution phase of AMG. But if we take a look at the performance of the whole algorithm the setup phase also has big optimization potential. To address this two things should be done. First the proof that it is possible that the transfer of the AMG hierarchy into GPU memory can be almost completely hidden behind its own computation on the CPU. Given this proof also very sophisticated AMG setups can benefit from GPUs during the solution phase, even if the setup process is to complicated to execute effectively on the GPU. The second thing to do is to accelerate the setup with the integration of GPUs, by executing parts or even the whole setup on the GPU.

Besides that more techniques to speedup the solution phase should be explored. These include the effect of mixed precision calculation and the option to choose different matrix storage formats for different parts of the algorithm.

Alongside to these AMG related topics we want to support distributed memory machines with LAMA. This will also address multi GPU aspects and the possibility to effectively utilize CPU and GPU resources in parallel.

References

1. hypre homepage (2010), `https://computation.llnl.gov/casc/hypre/software..html` (last viewed December 2010)
2. Petsc homepage (2010), `http://www.mcs.anl.gov/petsc/petsc-as/` (last viewed December 2010)
3. Blitz++ homepage (2011), `http://www.oonumerics.org/blitz/` (last viewed January 2011)
4. Ellpack homepage (2011), `http://www.cs.purdue.edu/ellpack/` (last viewed April 2011)
5. Hiflow3 homepage (2011), `http://www.hiflow3.org/` (last viewed August 2011)
6. Barrett, R.: Templates for the solution of linear systems: building blocks for iterative methods. Society for Industrial Mathematics (1994)
7. Baskaran, M., Bordawekar, R.: Optimizing sparse matrix-vector multiplication on gpus. IBM Research Report (2008)

8. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. In: Proc. ACM/IEEE Conf. Supercomputing (SC), Portland, OR, USA (2009)
9. Feng, Z., Zeng, Z.: Parallel multigrid preconditioning on graphics processing units (GPUs) for robust power grid analysis. In: Proceedings of the 47th Design Automation Conference, pp. 661–666. ACM (2010)
10. Förster, M., Kraus, J.: Scalable parallel AMG on ccNUMA machines with OpenMP. Computer Science-Research and Development, 1–8 (2011)
11. Haase, G., Liebmann, M., Douglas, C., Plank, G.: A parallel algebraic multigrid solver on graphics processing units. In: High Performance Computing and Applications, pp. 38–47 (2010)
12. Heuveline, V., Subramanian, C., Lukarski, D., Weiss, J.: A multi-platform linear algebra toolbox for finite element solvers on heterogeneous clusters. In: 2010 IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), pp. 1–6. IEEE (2010)
13. Kirk, D., Hwu, W.-M.: Programming massively parallel processors: A Hands-on approach. Morgan Kaufmann Publishers Inc., San Francisco (2010)
14. Klie, H., Sudan, H., Li, R., Saad, Y.: Exploiting capabilities of many core platforms in reservoir simulation. In: SPE Reservoir Simulation Symposium (2011)
15. Ruge, J., Stüben, K.: Algebraic Multigrid (AMG). In: McCormick, S.F. (ed.) Multigrid Methods. Frontiers in Applied Mathematics, vol. 3, pp. 73–130. SIAM, Philadelphia (1987)
16. Vandevoorde, D., Josuttis, N.: C++ templates: the Complete Guide. Addison-Wesley Professional (2003)

A GPU-Accelerated Parallel Preconditioner for the Solution of the Boltzmann Transport Equation for Semiconductors

Karl Rupp[1,2], Ansgar Jüngel[1], and Tibor Grasser[2]

[1] Institute for Analysis and Scientific Computing, TU Wien
Wiedner Hauptstraße 8–10, A-1040 Wien, Austria
[2] Institute for Microelectronics, TU Wien
Gußhausstraße 27-29, A-1040 Wien, Austria
`rupp@iue.tuwien.ac.at`

Abstract. The solution of large systems of linear equations is typically achieved by iterative methods. The rate of convergence of these methods can be substantially improved by the use of preconditioners, which can be either applied in a black-box fashion to the linear system, or exploit properties specific to the underlying problem for maximum efficiency. However, with the shift towards multi- and many-core computing architectures, the design of sufficiently parallel preconditioners is increasingly challenging.

This work presents a parallel preconditioning scheme for a state-of-the-art semiconductor device simulator and allows for the acceleration of the iterative solution process of the resulting system of linear equations. The method is based on physical properties of the underlying system of partial differential equations and results in a block preconditioner scheme, where each block can be computed in parallel by established serial preconditioners. The efficiency of the proposed scheme is confirmed by numerical experiments using a serial incomplete LU factorization preconditioner, which is accelerated by one order of magnitude on both multi-core central processing units and graphics processing units with the proposed scheme.

1 Introduction

With the introduction of multi-core central processing units (CPUs) in average desktop computers as well as the use of graphics processing units (GPUs) for general purpose computations, established serial algorithms need to be adjusted or even replaced by parallel variants. In particular, it can be very challenging to use the massively parallel architecture of GPUs with hundreds of threads even for standard algorithms like matrix-matrix multiplications efficiently [13].

While impressive performance for linear algebra operations can be obtained on GPUs, there are concerns about the use of GPUs from a productivity point of view [2]. While in some cases OpenMP [15] allows for a parallelization of existing code by adding a few lines of code only, GPU programming using OpenCL [11] or CUDA [14] requires a deep understanding of the underlying GPU computing architecture and often a complete redesign of existing CPU-based code. Consequently, the shorter execution times may not balance the increased development effort.

While existing GPU libraries tend to provide only basic LAPACK-style functionality, our C++ library ViennaCL [19] provides high-level access to the vast computing resources of multi-core CPUs and GPUs using OpenCL. The application programming interface is compatible with uBLAS from the peer-reviewed Boost libraries [1] and thus hides the details of the GPU computing hardware from the user, while providing convenient use and high performance computations. Like other GPU libraries such as CUBLAS [14] and MAGMA [12], ViennaCL provides BLAS level 1, 2 and 3 routines for dense linear algebra operations. However, the focus is on sparse matrices and iterative solvers as well as high usability, which is also the case for the CUDA-based Cusp library [3]. ViennaCL targets shared memory systems and can be run on multiple GPUs. An investigation of dense matrix-matrix multiplications on heterogeneous distributed memory architectures using MPI has already been carried out [21] and shown that a distribution of the problem at hand should be accomplished on a higher level in order to keep communication overhead under control.

For the solution of partial differential equations, discretization schemes like the finite element, the finite difference or the finite volume method lead to large systems of linear equations, for which iterative solvers are typically employed [17]. The efficiency of such iterative solution schemes depends on the condition number of the underlying system matrix. Therefore, preconditioners often need to be employed in order to obtain a good convergence rate. However, the design of good parallel preconditioners can be very challenging and problem-specific [18]. Parallel black-box preconditioners for ViennaCL are in preparation, but they typically come at the expense of spectral efficiency compared to – possibly serial – problem-specific techniques. The successful implementation of rather complex preconditioners for GPUs is continuously reported by a number groups in various fields, e.g. [7] for algebraic multigrid methods, [22] for a factored sparse approximate inverse technique or [8] for results on a multi-colored incomplete LU (ILU) factorization.

In this work we present a parallel preconditioner for our state-of-the-art semiconductor device simulator. We demonstrate that only a single additional compute kernel added to the functionality already provided in ViennaCL allows to reduce execution times by about one order of magnitude compared to a single-threaded execution. This readily shows that a library-centric design of GPU-based algorithms allows for short code development times, while leveraging the full power of multi-core CPUs and GPUs.

We give an overview of the simulator in Sec. 2. The block preconditioning scheme we have recently derived from the underlying problem formulation is motivated, detailed and discussed in Sec. 3, 4 and 5, which is the key for the parallelization of the iterative solver. Results are discussed in Sec. 6 and a conclusion is drawn in Sec. 7.

2 A Deterministic Solution Approach for the Boltzmann Transport Equation for Semiconductors

The Boltzmann transport equation (BTE) for semiconductors is given by

$$\frac{\partial f}{\partial t} + \boldsymbol{v} \cdot \nabla_{\boldsymbol{x}} f + \boldsymbol{F} \cdot \nabla_{\boldsymbol{p}} f = Q\{f\} \tag{1}$$

and commonly considered to be the best semi-classical description of carrier transport in semiconductors. Here, $f(\boldsymbol{x}, \boldsymbol{p}, t)$ denotes the distribution function of carriers in the device with respect to the spatial location $\boldsymbol{x}$, momentum $\boldsymbol{p}$ and time t. The velocity $\boldsymbol{v}$ is given by the energy band structure of the material, the force $\boldsymbol{F}$ is obtained from the electrostatic potential, and the scattering operator $Q\{f\}$ is given in low-density approximation as

$$Q\{f\} = \int_{\mathcal{B}} S(\boldsymbol{p}', \boldsymbol{p}) f(\boldsymbol{p}') - S(\boldsymbol{p}, \boldsymbol{p}') f(\boldsymbol{p}) \, \mathrm{d}\boldsymbol{p}' \ , \tag{2}$$

where $\mathcal{B}$ denotes the Brillouin-zone of the material and $\boldsymbol{S}(\cdot, \cdot)$ denotes the scattering rate from one state to another.

The high dimensionality of the problem as well as the integro-differential nature make the solution of the BTE very challenging. While the stochastic Monte Carlo method has been the method of choice for a long time, the spherical harmonics expansion (SHE) method has become an increasingly attractive alternative. Here, the momentum-part of the distribution function is expanded into spherical harmonics $Y_{l,m}$ as

$$f(\boldsymbol{x}, \boldsymbol{p}, t) \cong \sum_{l=0}^{L} \sum_{m=-l}^{l} f_{l,m}(\boldsymbol{x}, H, t) Y_{l,m}(\theta, \varphi) \ , \tag{3}$$

where H denotes total energy. The series is truncated at a finite expansion order L, typically $L \in \{1, 3, 5\}$. In the following, only the steady-state is considered.

While the application of the SHE method has long been restricted to one-dimensional device simulations due to high memory requirements, enough memory is available on modern computers to allow for two-dimensional device simulations [9]. While fully parallel implementations of the Monte Carlo method have already been reported [23], an artificial restriction of the SHE method to a single CPU core would be detrimental to the attractiveness of the method.

The SHE method ultimately leads to the solution of large systems of linear equations for the expansion coefficients $f_{l,m}$ in the $(\boldsymbol{x}, H)$ space. A nonlinear iteration scheme is typically employed to ensure self-consistency of the BTE with the Poisson equation describing the electrostatic potential. As discussed by Jungemann *et al.* [10], the indefinite system of linear equations resulting from the SHE equations requires a good preconditioner in order to obtain convergence of iterative solvers. This is in contrast to many other problem classes, where e.g. the positive definiteness of the system matrix typically ensures convergence, even if the number of iterations required might be large. In recent publications on the

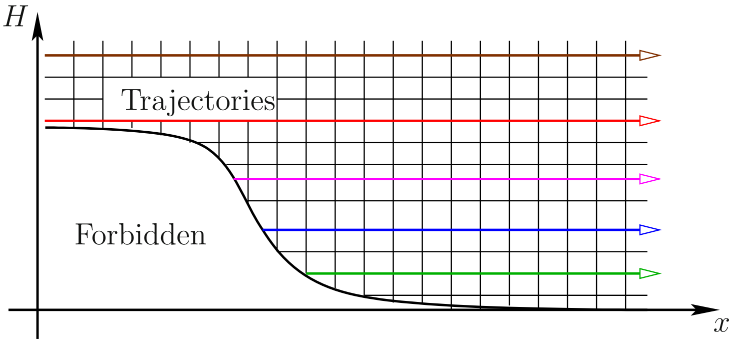


Fig. 1. Trajectories of carriers in free flight within the device are given by constant total energy H. Carriers can change energy only through inelastic scatting, which is instant in time and localized space.

SHE method [9,10], an ILU preconditioner was used for that purpose. ILU is a widely accepted black-box preconditioner [17], but in its pure form restricted to single-threaded execution. Even though parallel block-variants of ILU as well other parallel preconditioning techniques such as sparse approximate inverses [6] or polynomial preconditioners have been developed, their convergence enhancement can be typically considerably lower than for single-threaded variants [17,18].

3 Physics-Based Block-Preconditioning

To obtain a set of equations for the unknown expansion coefficients $f_{l,m}$ in (3), the BTE is formally projected onto the individual spherical harmonics $Y_{l,m}$. In operator form, the SHE equations in steady state can then be written as

$$L_{l,m}\{f\} = Q_{l,m}\{f\} \ , \quad l = 0, \dots, L, \quad m = -l, \dots, l \ ,$$

where $L_{l,m}$ and $Q_{l,m}$ denote the projections of the streaming operator and the scattering operator onto the spherical harmonics $Y_{l,m}$ respectively. Employing the H-transform [9,5], carrier trajectories in free flight are given by hyperplanes of constant total energy H in the simulation domain $(\boldsymbol{x}, H)$, cf. Fig. 1. This is reflected in the model by the fact that $L_{l,m}$ does not couple any of the, say, N_H different energy levels in the simulation domain.

Carriers within the device can change their total energy only by inelastic scattering events, thus the scattering operator $Q_{l,m}\{f\}$ is responsible for coupling different energy levels. However, if only elastic scattering processes are considered, the total energy of the involved particles remains unchanged and the different energy levels do not couple. Therefore, in a SHE simulation using only elastic scattering and N_H different energy levels, the resulting system of linear equations is consequently decoupled into N_H independent problems. Such a decomposition has been observed already in early publications on SHE [4], but it has been of no practical relevance since inelastic scattering processes are essential for predictive device simulation.

Inelastic scattering processes like optical phonon scattering couple different energy levels. As devices are scaled down, the average number of scattering events of a carrier while moving through the device decreases and weakens the coupling between different energy levels. At the algebraic level this can be reasoned as follows: Using a box integration scheme as proposed by Hong *et al.* [9], the volume integral over the free streaming operator $L_{l,m}$ is transformed to a surface integral due to the divergence operator with respect to the spatial variable $\boldsymbol{x}$. Therefore, if the typical device length d is scaled to $d' := \alpha d$ with $0 < \alpha < 1$, the contributions from the free streaming operator scale as α^{n-1}, where n denotes the spatial dimension considered in the simulation. However, the scattering terms are obtained by an integration over the control volume, which scales as α^n. Therefore, in the limit of extremely scaled devices, the coupling between different energy levels is negligible.

While the preconditioner is motivated by physical arguments at the continuous level, a discretization still has to be employed. Following the discretization proposed by Hong *et al.* [9], the even order expansion coefficients are associated with grid nodes and represent densities, while odd order expansion coefficients are associated with edges and represent fluxes. Odd order expansion coefficients can be condensed [10,16], such that one finally obtains a linear system of equations for the even order expansion coefficients at discrete locations in the $(\boldsymbol{x}, H)$ domain. Let $\boldsymbol{S}_{\text{full}}$ denote this condensed system matrix and $\boldsymbol{S}_{\text{elastic}}$ the system matrix of the decoupled problem obtained in the same way by ignoring any contributions from inelastic scattering processes. Then we propose to construct the preconditioner $\boldsymbol{P}_{\text{full}}$ for $\boldsymbol{S}_{\text{full}}$ as

$$\boldsymbol{P}_{\text{full}} \approx (\boldsymbol{S}_{\text{full}})^{-1} \approx (\boldsymbol{S}_{\text{elastic}})^{-1} \approx \boldsymbol{P}_{\text{elastic}} \ . \tag{4}$$

Since the elastic problem is decoupled into N_{H} subproblems, $\boldsymbol{S}^{\text{elastic}}$ decomposes into N_{H} independent blocks. For each of these blocks, a (possibly serial) preconditioner can be efficiently set up as well as applied to the residual vector in parallel. Moreover, due to the decoupling of the system matrix into independent blocks, the proposed scheme is also perfectly suitable for distributed memory architectures.

4 Symmetrization of the Scattering Processes

Due to the exponential decay of the distribution function with respect to carrier energy, the scattering rate from higher energy to lower energy is much higher than vice versa. This asymmetry of inelastic scattering processes for energies H_i and H_j, $i < j$, with respect to energy manifests itself in the system matrix in the form of large values in the block with energy index (H_i, H_j), and small entries in the block (H_j, H_i), cf. Fig. 2. Therefore, the upper triangular part of the system matrix is populated with much larger values than the lower triangular part. It

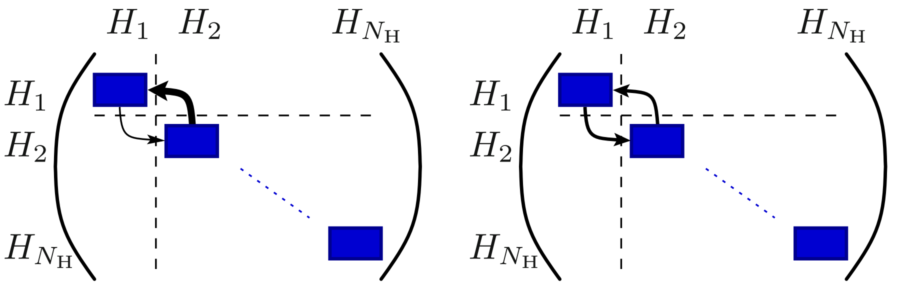


Fig. 2. Structure of the system matrix for total energy levels $H_1 < H_2 < \ldots < H_{N_\mathrm{H}}$ before (left) and after (right) symmetrization. Unknowns at the same total energy H_i are enumerated consecutively, inducing a block-structure of the system matrix. For simplicity, scattering is depicted between energy levels H_1 and H_2 only, using arrows with thickness proportional to the magnitude of the entries. As devices are scaled down, the entries in off-diagonal blocks become small compared to the entries in the diagonal blocks.

should be noted that this asymmetry ensures that the equilibrium solution is a Maxwell (or more generally, a Fermi-Dirac) distribution.

The large values in the upper triangular part of the matrix are a hindrance for the construction of the preconditioner by neglecting off-diagonal blocks. We reduce this asymmetry by rescaling the unknowns of the discrete system according to the expected exponential decay. The new discrete unknowns $f'_{l,m}(\boldsymbol{x}_i, H_i, t)$ are obtained from the old discrete unknowns $f_{l,m}(\boldsymbol{x}_i, H_i, t)$ by

$$f'_{l,m}(\boldsymbol{x}_i, H_i, t) := \exp\left(\frac{\varepsilon_i}{k_\mathrm{B}T}\right) f_{l,m}(\boldsymbol{x}_i, H_i, t) \ , \tag{5}$$

where ε_i denotes the kinetic energy at point $(\boldsymbol{x}_i, H_i)$, k_B is the Boltzmann constant and T denotes a scaling temperature which is either set to room temperature, lattice temperature or can be seen as a numerical parameter. The benefit of this rescaling is that in equilibrium the primed unknowns are then of similar order and show little to no exponential behavior. It should be noted that the proposed rescaling can be written in matrix form equivalently as

$$\boldsymbol{S}\boldsymbol{f} = \boldsymbol{b} \qquad \Leftrightarrow \qquad \boldsymbol{S}\boldsymbol{D}\boldsymbol{f}' = \boldsymbol{b} \ ,$$

where $\boldsymbol{D}$ is a diagonal matrix with the diagonal terms given by the reciprocals of the exponentials in (5). The matrix $\boldsymbol{S}' := \boldsymbol{S}\boldsymbol{D}$ represents the system matrix with rebalanced off-diagonal scattering blocks. Here, symmetrization refers to rescaling the unknowns such that the entries in the off-diagonal blocks (H_i, H_j) and (H_j, H_i) are of similar magnitude – it does not denote symmetry of the system matrix in the strict mathematical sense.

5 Practical Considerations

For the construction of the preconditioner it is not necessary to set up another system matrix $\boldsymbol{S}_{\text{elastic}}$ explicitly. Since the contribution of inelastic scattering operators to the diagonal blocks is positive, it is of advantage to use the block diagonal of $\boldsymbol{S}_{\text{full}}$ for setting up the preconditioner. Thereby, extra memory for a second system matrix is avoided.

It has been observed in numerical experiments that the rescaling of unknowns leads to better results if the temperature T in (5) is set above room temperature. The physical interpretation is that carriers are heated in areas of large electric fields, thus having a lower exponential decay rate, which relates to a higher temperature. Good results are obtained with $T = 400\text{K}$ and only a low sensitivity of the number of iterations on the parameter T is observed.

The rows of the system matrix $\boldsymbol{S}'$ can be normalized prior to the block-factorization. This leads to a matrix $\boldsymbol{S}''$ given as

$$\boldsymbol{S}'' = \boldsymbol{E}\boldsymbol{S}' = \boldsymbol{E}\boldsymbol{S}\boldsymbol{D} \ ,$$

where the diagonal matrix $\boldsymbol{E}$ consists of the inverses of the row norms. Thus, a two-sided diagonal preconditioner is applied to the initial system matrix $\boldsymbol{S}$ before launching the block-preconditioning scheme.

Within ViennaCL, the call to the iterative solver is given in the mnemonic form

```
x = solve(A, b, solver_tag, precond);
```

where `A` denotes the system matrix, `x` and `b` the result and the right hand side vector respectively, the `solver_tag` allows to select the respective solver and `precond` specifies the optional preconditioner. For the case of SHE using the BiCGStab [20] iterative solver and the custom parallel block preconditioner, the respective code lines for the user are

```
x = solve(A, b, bicgstab_tag(), she_block_precond);
```

where `she_block_precond` is a functor that applies the preconditioner to the residual in each iterative solver step. Therefore, the user can focus all development efforts on the preconditioner only, without having to deal with other details of the underlying iterative solver. In particular, comparisons with the built-in ILU factorization with threshold (ILUT) preconditioner in ViennaCL are carried out by

```
//MatrixType denotes the type of the matrix A
ilut_precond<MatrixType> ilut(A, ilut_tag());
x = solve(A, b, bicgstab_tag(), ilut);
```

thus allowing a simple means to switch between different iterative solvers as well as different preconditioners.

6 Results

As a benchmark for the proposed block preconditioning scheme, we consider the spatially two-dimensional simulation of an n^+nn^+ diode with different lengths of the intrinsic region. ILUT is used as a preconditioner for each block. As outlined in Sec. 5, the same preconditioner is used as a single-threaded preconditioner for the full system matrix, since ILU-type preconditioners have been employed in other recent works. It has to be emphasized that the preconditioner used for each block in our scheme can be chosen arbitrarily, thus we aim at confirming the applicability of the physically motivated scheme only, since it then enables the use of any possibly serial preconditioner in a highly parallel fashion. BiCGStab [20] is used as linear solver, since it provides a lower memory footprint than the GMRES method [17] used in [10].

Execution times of the iterative BiCGStab solver are compared for a single CPU core using ILUT for the full system matrix, and for the proposed parallel scheme using multiple CPU cores of a quad-core Intel Core i7 960 CPU with eight logical cores. In addition, comparisons for a NVIDIA Geforce GTX 580 GPU are found in Figs. 3. The parallelization on the CPU is achieved using the Boost.thread library [1], and the same development time was allotted for the OpenCL kernel on the GPU. This allows for a comparison of the results not only in terms of execution speed, but also in terms of productivity.

As can be seen in Figs. 3, the performance increase for each linear solver step is more than one order of magnitude compared to the single-core implementation. This super-linear scaling with respect to the number of cores on the CPU is due to the better caching possibilities obtained by the higher data locality within the block-preconditioner.

The required number of iterations using the block-preconditioner decreases with the device size. For a 25 nm intrinsic region, the number of iterations is only twice than that of an ILUT preconditioner for the full system. At an intrinsic region of 200 nm, four times the number of iterations are required. This is a very small price to pay for the excellent parallelization possibilities.

Overall, the multi-core implementation is by a factor of three to ten faster than the single core-implementation even though a slightly larger number of solver iterations is required. The purely GPU-based solver with hundreds of simultaneous lightweight threads is by up to one order of magnitude faster than the single-core CPU implementation.

The comparison in Fig. 3 further shows that the SHE order does not have a notable influence on the block-preconditioner efficiency compared to the full preconditioner. The slightly larger number of solver iterations for third order expansions is due to the higher number of unknowns in the linear system. The performance gain is almost uniform over the length of the intrinsic region and slightly favors shorter devices, thus making the scheme an ideal candidate for current and future scaled-down devices.

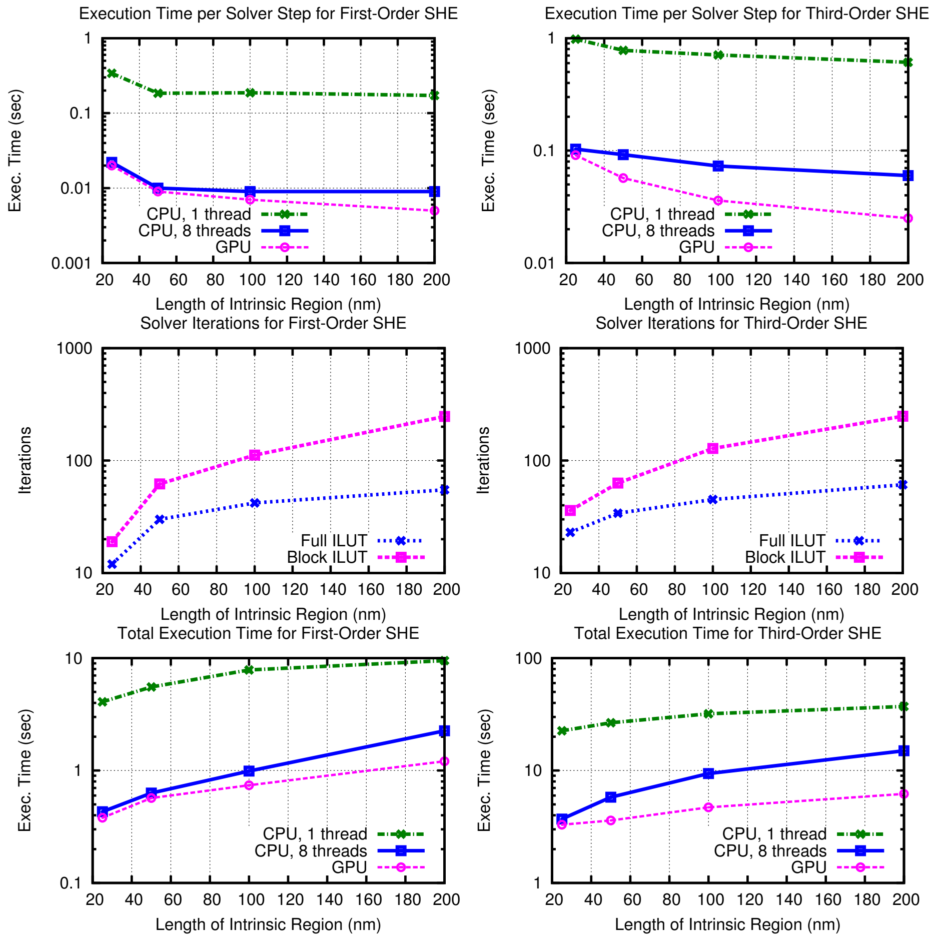


Fig. 3. Execution times per solver iteration, number of solver iterations and total solver execution time for a first-order (left) and a third-order (right) SHE simulation of n^+nn^+ diodes with different lengths of the intrinsic region. As expected from physical arguments, the parallel preconditioner performs the better the smaller the length of the intrinsic region gets. The GPU version performs particularly well for the computationally more challenging third-order SHE. A reduction of total execution times compared to a single-threaded implementation by one order of magnitude is obtained.

7 Conclusions

Our case-study of employing a problem-specific parallel preconditioner within ViennaCL for the acceleration of a semiconductor device simulator readily shows that library-centric design for algorithms on GPUs and multi-core CPUs based on OpenCL allows for high productivity. A development of the full GPU solver from scratch for the particular problem at hand would have resulted in development effort that is at least an order of magnitude larger than a comparable implementation for multi-core CPUs in e.g. C++, while only a performance gain of about an additional factor of two would have been obtained.

The parallel block-preconditioning scheme is proposed and demonstrated to be very efficient especially for scaled-down devices. In contrast to black-box block preconditioners, the proposed scheme is based on a sound physical principle. The number of iterations compared to a single-threaded ILUT preconditioner for the full system matrix is two to four times as large, but this is only a minor price to pay for the huge degree of parallelism provided for the crucial preconditioning step. On the whole, an overall performance improvement of one order of magnitude is obtained for our test case.

Acknowledgment. K. Rupp and A. Jüngel acknowledge partial support from the Austrian Science Fund (FWF), grants P20214, P22108, and I395; the Austrian-Croatian Project HR 01/2010; the Austrian-French Project FR 07/2010; and the Austrian-Spanish Project ES 08/2010 of the Austrian Exchange Service (ÖAD). The authors gratefully acknowledge support by the Graduate School PDEtech at the TU Wien.

References

1. Boost C++ libraries, `http://www.boost.org/`
2. Bordawekar, R., Bondhugula, U., Rao, R.: Can CPUs Match GPUs on Performance with Productivity? Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. Technical report, IBM T. J. Watson Research Center (2010)
3. Cusp Library, `http://code.google.com/p/cusp-library/`
4. Gnudi, A., Ventura, D., Baccarani, G.: One-dimensional Simulation of a Bipolar Transistor by means of Spherical Harmonics Expansion of the Boltzmann Transport Equation. In: Proc. SISDEP, vol. 4, pp. 205–213 (1991)
5. Gnudi, A., Ventura, D., Baccarani, G., Odeh, F.: Two-dimensional MOSFET Simulation by Means of a Multidimensional Spherical Harmonics Expansion of the Boltzmann Transport Equation. Solid-State Electr. 36(4), 575–581 (1993)
6. Grote, M.J., Huckle, T.: Parallel Preconditioning with Sparse Approximate Inverses. SIAM J. Sci. Comput. 18, 838–853 (1997)
7. Haase, G., Liebmann, M., Douglas, C., Plank, G.: A Parallel Algebraic Multigrid Solver on Graphics Processing Units. In: Zhang, W., Chen, Z., Douglas, C.C., Tong, W. (eds.) HPCA 2009. LNCS, vol. 5938, pp. 38–47. Springer, Heidelberg (2010)
8. Heuveline, V., Lukarski, D., Weiss, J.P.: Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method. EMCL Preprint 2011-08, EMCL (2011)
9. Hong, S.M., Jungemann, C.: A Fully Coupled Scheme for a Boltzmann-Poisson Equation Solver based on a Spherical Harmonics Expansion. J. Comp. Electr. 8, 225–241 (2009)
10. Jungemann, C., Pham, A.T., Meinerzhagen, B., Ringhofer, C., Bollhöfer, M.: Stable Discretization of the Boltzmann Equation based on Spherical Harmonics, Box Integration, and a Maximum Entropy Dissipation Principle. J. Appl. Phys. 100(2), 024502–+ (2006)
11. Khronos Group. OpenCL, `http://www.khronos.org/opencl/`

12. MAGMA library, `http://icl.cs.utk.edu/magma/`
13. Nath, R., Tomov, S., Dongarra, J.: An Improved MAGMA GEMM For Fermi Graphics Processing Units. Intl. J. HPC Appl. 24(4), 511–515 (2010)
14. NVIDIA CUDA, `http://www.nvidia.com/`
15. OpenMP, `http://openmp.org/`
16. Rupp, K., Jüngel, A., Grasser, T.: Matrix Compression for Spherical Harmonics Expansions of the Boltzmann Transport Equation for Semiconductors. J. Comp. Phys. 229(23), 8750–8765 (2010)
17. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics (2003)
18. Vassilevski, P.S.: Multilevel Block Factorization Preconditioners. Springer, Heidelberg (2008)
19. ViennaCL, `http://viennacl.sourceforge.net/`
20. van der Vorst, H.A.: Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Non-Symmetric Linear Systems. SIAM J. Sci. and Stat. Comp. 12, 631–644 (1992)
21. Weinbub, J., Rupp, K., Selberherr, S.: Distributed Heterogenous High-Performance Computing with ViennaCL. In: Abstracts Intl. Conf. LSSC, pp. 88–90 (2011)
22. Xu, K., Ding, D.Z., Fan, Z.H., Chen, R.S.: FSAI Preconditioned CG Algorithm combined with GPU Technique for the Finite Element Analysis of Electromagnetic Scattering Problems. Finite Elem. Anal. Des. 47, 387–393 (2011)
23. Zang, W., Du, G., Li, Q., Zhang, A., Mo, Z., Liu, X., Zhang, P.: A 3D Parallel Monte Carlo Simulator for Semiconductor Devices. In: Proc. IWCE, pp. 1–4 (2009)

Parallel Smoothers for Matrix-Based Geometric Multigrid Methods on Locally Refined Meshes Using Multicore CPUs and GPUs

Vincent Heuveline[1], Dimitar Lukarski[1,2], Nico Trost[1], and Jan-Philipp Weiss[1,2]

[1] Engineering Mathematics and Computing Lab (EMCL)
[2] SRG New Frontiers in High Performance Computing
Karlsruhe Institute of Technology, Germany
{vincent.heuveline,dimitar.lukarski,jan-philipp.weiss}@kit.edu,
nico.trost@student.kit.edu

Abstract. Multigrid methods are efficient and fast solvers for problems typically modeled by partial differential equations of elliptic type. We use the approach of matrix-based geometric multigrid that has high flexibility with respect to complex geometries and local singularities. Furthermore, it adapts well to the exigences of modern computing platforms. In this work we investigate multi-colored Gauß-Seidel type smoothers, the *power(q)-pattern enhanced multi-colored ILU(p,q)* smoothers with fill-ins, and *factorized sparse approximate inverse* (FSAI) smoothers. These approaches provide efficient smoothers with a high degree of parallelism. We describe the configuration of our smoothers in the context of the portable *lmpLAtoolbox* and the *HiFlow*^{3} parallel finite element package. In our approach, a single source code can be used across diverse platforms including multicore CPUs and GPUs. Highly optimized implementations are hidden behind a unified user interface. Efficiency and scalability of our multigrid solvers are demonstrated by means of a comprehensive performance analysis on multicore CPUs and GPUs.

Keywords: Parallel smoothers, matrix-based geometric multigrid, multi-coloring, power(q)-pattern method, FSAI, multi-core, GPUs.

1 Introduction

The need for high accuracy and short simulation times relies both on efficient numerical schemes and appropriate scalable parallel implementations. The latter point is crucial in the context of fine-grained parallelism in the prospect of the manycore era. In particular, graphics processing units (GPUs) open new potentials in terms of computing power and internal bandwidth values. These new architectures have a significant impact on the design and implementation of parallel algorithms in numerical simulation. Furthermore, portability of solver concepts, codes and performance across several platforms is a major issue in the era of highly capable multicore and accelerator platforms.

Multigrid methods rely on the decomposition of the error in low- and high-frequency contributions. The smoothers damp out the high frequency contributions of the error at a given level. Adequate prolongation and restriction operators allow to address the considered problem on a hierarchy of discretizations and by this means cover the full spectral range of error contributions only on the basis of these smoothers (see [21] and references therein for further details). For full flexibility of the solvers in the context of complex geometries, stencil-based multigrid methods – typically used on equidistant Cartesian grids and for partial differential equations (PDEs) with constant coefficients – need to be replaced by more flexible concepts. We use the approach of matrix-based geometric multigrid where all operations – i.e. smoothers, grid transfers and residual computation – are represented by sparse matrix-vector multiplications (SpMV). This approach has been shown to work well on modern multicore platforms and GPUs [2]. Moreover, the restriction to basic algorithmic building blocks is the key technique for building portable solvers. The major challenge with respect to parallelism is related to the definition and implementation of adequate parallel smoothers.

Parallel multigrid smoothers and preconditioners on CPUs and GPUs for regularly structured tensor-product meshes are considered in [8]. The author discusses parallel implementation aspects for several platforms and shows integration of accelerators into a finite element package. The references there also give a historic overview of multigrid on GPUs up to 2009. In [7], geometric multigrid on unstructured meshes for higher order finite element methods is investigated. A parallel Jacobi smoother and grid transfer operators are assembled into sparse matrix representations leading to efficient solvers on multicore CPUs and GPUs. In [6] performance results for multigrid solvers based on the *sparse approximate inverse* (SPAI) technique are presented. The corresponding SPAI techniques for parallel multigrid smoothers and preconditioners are described in [9,5,19,4,3]. An implementation and performance results of algebraic multigrid methods on GPUs are discussed in [10].

In this work we evaluate a new smoothing algorithm based on the *power(q)-pattern enhanced multi-colored ILU(p,q) factorization* [16] with respect to its smoothing properties and its parallel speedup on multicore CPUs and GPUs. We compare it to multi-colored splitting-type smoothers and FSAI smoothers. Various hardware platforms can be easily used in the context of these solvers by means of the portable implementation based on the *lmpLAtoolbox* in the parallel finite element software package *HiFlow*^{3} [1]. The capability of the proposed approach is demonstrated in a comprehensive performance analysis.

This paper is organized as follows. In Section 2 we describe the context of matrix-based geometric multigrid methods. Section 3 describes the proposed parallel concepts for efficient and scalable smoothers. The setup and integration of the described multigrid solvers in the HiFlow^3 finite element method package is described in Section 4. The numerical test problem is the Poisson problem on an L-shaped domain where a statically adapted and locally refined mesh is used for the discretizations. The impact of the choice and the configuration of the smoothers on the MG convergence is investigated in Section 5. A performance analysis with respect to solver times and parallel speedups on multicore CPUs and GPUs is the central theme of Section 6. We conclude in Section 7.

2 Matrix-Based Geometric Multigrid Methods

Multigrid (MG) methods are usually used to solve large sparse linear systems of equations arising from finite element discretizations (or related techniques) of PDEs – typically of elliptic type. Due to the ability to achieve asymptotically optimal complexity, MG has been proven to be one of the most efficient solvers for this type of problems. The main idea of MG methods is based on coarse grid corrections and the smoothing properties of classical iterative schemes. High frequency error components can be eliminated efficiently by using elementary iterative solvers – such as Jacobi or Gauß-Seidel. On the other hand, smooth error components cannot be reduced efficiently on fine grids. This effect can be mitigated by performing a sequence of coarsening steps and transferring the smooth errors by restricting the defect to coarser mesh levels. Details on the basic properties of the MG method as well as error analysis and programming remarks can be found in [21,11] and in references provided therein. Aspects of parallel MG on distributed memory systems are e.g. discussed in [18].

The MG cycle $u_h^{(n)}=\mathrm{MG}(u_h^{(n-1)}, L_h, f_h, \nu_1, \nu_2, \gamma)$ is an iterative method with recursive MG iterations where $L_h u_h = f_h$ is the discrete version of the underlying PDE on the refinement level with mesh size h. On each level, pre- and post-smoothing is applied in the form $u_h^{(n)} = \mathrm{RELAX}^{\nu_i}(u_h^{(n-1)}, L_h, f_h)$. The parameters ν_i, $i = 1, 2$, denote the number of pre- and post-smoothing iterations. Grid transfer operators are denoted by I_h^H (restriction) and I_H^h (prolongation) where h is representing the fine grid size and H is the coarse grid size. The transferred residual $r_H = I_h^H r_h = I_h^H(f_h - L_h u_h)$ is the input to the MG iteration $e_H^{(n)}=\mathrm{MG}(0, L_H, r_H^{(n)}, \nu_1, \nu_2, \gamma)$ on the coarser level. The parameter γ specifies the number of two-grid cycle iterations on each level and thus specifies how often the coarsest level is visited. For $\gamma = 1$ we obtain so called V-cycles whereas $\gamma = 2$ results in W-cycles. The coarse grid correction is $u_h := u_h + I_H^h e_H$. The obtained solution after each cycle serves as the input for the successive cycle.

In this work, we consider matrix-based geometric MG methods. In contrast to stencil-based geometric MG methods, all differential operators and grid transfer operators are not expressed by fixed stencils on equidistant grids but have the full flexibility of sparse matrix representations. This approach gives us full flexibility with respect to complex geometries, non-uniform grids resulting from local mesh refinements, and space-dependent coefficients in the underlying PDE. Moreover, the solvers can be built upon standard building blocks of numerical libraries.

3 Matrix-Based Parallel Smoothers

Stencil-based geometric MG methods can be efficiently performed in parallel by domain sub-structuring and halo exchange for the stencils [18]. In contrast, the parallel implementation of a matrix-based geometric MG requires more work. While the grid transfer operators are explicit updates with sufficient locality, parallel smoothers rely on implicit steps where often triangular systems need to be solved, e.g. in Gauß-Seidel or ILU-type smoothers. And as the objective is full flexibility on unstructured meshes, straightforward parallelization strategies like wave-front or simple red-black-coloring cannot be applied. Therefore, the focus of our work is directed to fine-grained parallelism for the smoothing step based on multi-coloring approaches. The considered smoothers are based on the block-wise decomposition into small sub-matrices. In the additive and multiplicative splittings, typically a large amount of forward and backward sweeps in triangular solvers needs to be processed. For the description of the proposed parallel smoothers we point out the link between smoothers and preconditioning techniques. Iterative solvers can generally be interpreted in fixed point form by $x_{k+1} = Gx_k + f$ where the linear system of the type $Ax = b$ is transformed by the additive splitting $A = M + N$ and taking $G = M^{-1}N = M^{-1}(M - A) = I - M^{-1}A$ and $f = M^{-1}b$. This version can be reformulated in an algebraic manner as a preconditioned defect correction scheme

$$x_{k+1} = x_k + \omega M^{-1}(b - Ax_k) \tag{1}$$

where an additional damping parameter ω is introduced. In this context, we apply preconditioners M as smoothers for the linear system $Ax = b$ where the smoothers should reduce highly-ocsillating error contributions. Smoothing properties of our applied schemes are demonstrated in [14]. Additive preconditioners are standard splitting schemes typically based on the block-wise decomposition $A = D+L+R$ where L is a strictly lower triangular matrix, R is a strictly upper triangular matrix, and D is the matrix containing the diagonal blocks of A. We choose $M = D$ (Jacobi), $M = D+L$ (Gauß-Seidel) or $M = (D+L)D^{-1}(D+R)$ (symmetric Gauß-Seidel). For multiplicative splittings we choose $M = LU$ in (1) where L is a lower triangular and U is an upper triangular matrix. For incomplete LU (ILU) factorizations with or without fill-ins we decompose the system matrix A into the product $A = LU + R$ with a remainder matrix R that absorbs unwanted fill-in elements. Typically, diagonal entries of L are taken to be one and both matrices L and U are stored in the same data structure (omitting the ones). The quality of the approximation in this case depends on the number of fill-in elements. The third class of considered parallel smoothers consists of approximate inverse schemes. Here, we are focusing on the *factorized sparse approximate inverse* (FSAI) algorithms [20] that compute a direct approximation of A^{-1}. These schemes are based on the minimization of the Frobenius norm $|I - GA|_F$ where one looks for a symmetric preconditioner in the form $G := G_L^T G_L$. In other words, one builds an approximation of the Cholesky decomposition based on a given sparse matrix structure. FSAI(1) uses the sparsity pattern of A, FSAI(q), $q \geq 2$, uses the sparsity pattern of $|A|^q$ respectively.

In order to harness parallelism within each block of the decomposition we apply multi-coloring techniques [15,16] as a preprocessing step. The applied changes to the sparsity pattern are then kept during the preconditioner factorization.

For splitting-based methods (like Gauß-Seidel and SOR) and ILU(0) decompositions without fill-ins the sparsity pattern of the system matrix is preserved in the additive or multiplicative decompositions. Here, only the original sparsity pattern is populated and no additional matrix elements are inserted. Before applying the smoother, the matrix is re-organized such that diagonal blocks in the block decomposition are diagonal itself. Then, inversion of the diagonal blocks is just an easy vector operation [15]. Furthermore, we allow fill-ins (i.e. additional matrix elements) in the ILU(p) factorization for achieving a higher level of coupling in terms of increased numerical efficiency. The power(q)-pattern method is applied to ILU(p) factorizations with fill-ins [16]. This method is based on an incomplete factorization of the system matrix A subject to a predetermined nonzero pattern derived from a multi-color analysis of the matrix power $|A|^q$ (where $|A| = (|a_{ij}|)_{i,j}$) and its associated multi-color permutation π. It has been proven in [16] that the obtained sparsity pattern is a superset of the modified ILU(p) factorization applied to the re-ordered matrix $\pi A \pi^{-1}$. As a result, for $q = p + 1$ this modified ILU(p,q) scheme applied to the multi-colored system matrix has no fill-ins into its diagonal blocks. This leads to an inherently parallel execution of triangular ILU(p,q) sweeps and hence to a parallel and efficient smoother. The degree of parallelism can be increased by taking $q < p + 1$ at the expense of some fill-ins into the diagonal blocks. In this scenario (e.g. for the ILU(1,1) smoother in our experiments) we use a drop-off technique that erases fill-ins into the diagonal blocks. These techniques have already been successfully tested in the context of parallel preconditioners [16]. See this reference for further details of the algorithms.

4 Integration of the Multigrid Solvers into HiFlow^3

Flexibility of solution techniques and software is a decisive factor in the current computing landscape. We have incorporated the described multigrid solvers and parallel smoothers into the multi-platform and multi-purpose parallel finite element software package HiFlow^3 [12,1]. With the concept of object-oriented programming in C++, HiFlow^3 provides a flexible, generic and modular framework for building efficient parallel solvers and preconditioners for PDEs of various types. HiFlow^3 tackles productivity and performance issues by its conceptual approach. It is structured in several modules where its linear algebra operations are based on two communication and computation layers: the *LAtoolbox* for inter-node operations and the *lmpLAtoolbox* for intra-node operations in a parallel system. The lmpLAtoolbox has backends for multiple platforms with highly optimized implementations – hiding all hardware details from the user while maintaining optimal usage of resources. Numerical solvers in HiFlow^3 are built on the basis of unified interfaces to the different hardware backends and parallel programming approaches – where only a single code base is required for all platforms. In such a way, no specific and platform-adapted optimizations are necessary since we rely on efficient and tested library implementations of basic kernels. By this means, HiFlow^3 is portable across diverse computing systems including multicore CPUs, CUDA-enabled GPUs and OpenCL-capable platforms.

The modular approach also allows an easy extension to emerging systems with heterogeneous configuration. The SpMV implementations for the GPU used in this work are the scalar SpMV kernels in the CSR data format presented in [2]. In this work we are only focusing on single node parallelism. Therefore, we do not consider the global MPI level. Based on the matrix and vector classes HiFlow^3 offers several Krylov subspace solvers and local additive and multiplicative preconditioners. Further information about this library, solvers and preconditioners can be found e.g. in [16,13,15,17].

Starting on an initial mesh, the HiFlow^3 mesh module refines the grid in order to reach the requested number of cells according to the demands for accuracy. The current version supports quadrilateral elements as well as triangles for two dimensional problems. In the three-dimensional case tetrahedrons and hexahedrons are used. On each refinement level, the corresponding stiffness matrix and the right hand side are assembled for the given equation. Our implementation performs a bi-linear interpolation to ascend to a finer level and a full weighting restriction to descend to a coarser level respectively. The multigrid solver can also be applied on unstructured meshes. Our coarse grid solver is the parallel version of the conjugated gradient (CG) solver running on the respective device. The sparsity pattern of the original stiffness matrix and of the FSAI approximation is determined by the enumeration procedure in HiFlow^3. In our scenario, the degrees of freedom are ordered by the local numbering and by the iterator over the finite elements. However, for the additive and multiplicative decomposition we use the multi-color re-ordering that also affects the sparsity pattern and locality of interactions, and hence performance of the solver [13].

5 Numerical Experiments and Smoother Efficiency

As a numerical test problem we are solving the Poisson problem

$$-\Delta u = f \text{ in } \Omega \tag{2}$$

in the two-dimensional L-shaped domain $\Omega := (0,1)^2 \setminus (0,0.5)^2$ with a reentrant corner with homogeneous Dirichlet boundary conditions, i.e. $u = 0$ on $\partial\Omega$. For our test case we choose the right hand side $f = 8\pi^2 \cos(2\pi x)\cos(2\pi y)$. We solve (2) by means of bilinear Q1 elements on quadrilaterals and linear P1 elements on triangles [12,1]. A locally refined mesh for discretization of the L-shaped domain is depicted in Figure 1 (left). The numerical solution of (2) with boundary layers is detailed in Figure 1 (right). Due to the steep gradients at the re-entrant corner we are using a locally refined mesh in order to obtain a proper problem resolution. Figure 2 (left) shows a zoom-in into a uniformly refined coarse mesh. In the mesh in Figure 2 (right) we are using additional triangular and quadrilateral elements for local refinement and avoiding hanging nodes and deformed elements. Our coarsest mesh in the MG hierarchy is this locally refined mesh based on triangular and quadrilateral cells with Q1 and P1 elements summing up to 3,266 degrees of freedom (DoF). By applying six global refinement steps to this coarse mesh we obtain the hierarchy of nested grids where the

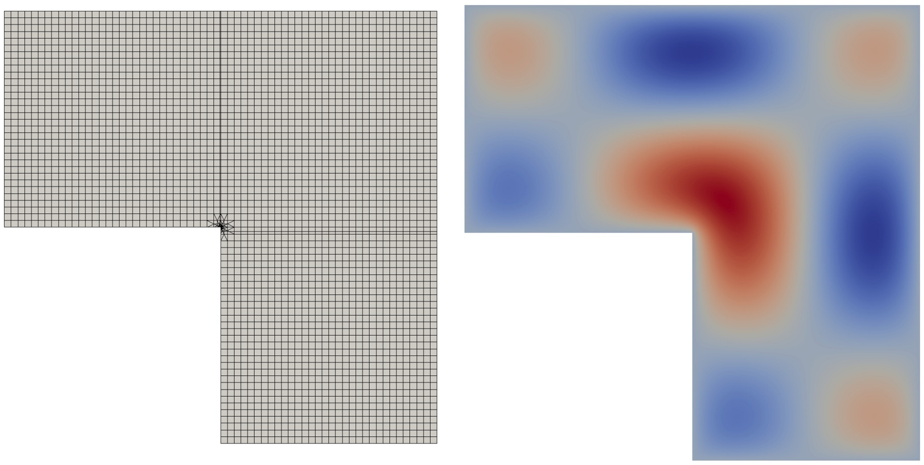


Fig. 1. Locally refined mesh for discretization of the L-shaped domain (left) and the discrete solution of the Poisson problem (2) (right)

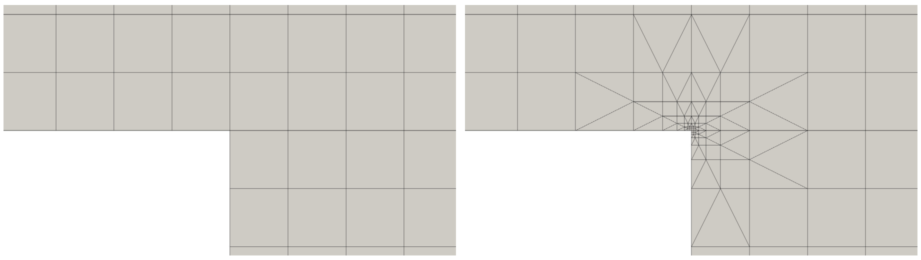


Fig. 2. Zoom-in to a uniform mesh with 3,201 DoF (left) and a locally refined mesh with 3,266 DoF (right) of the L-shaped domain around the reentrant corner – our coarsest MG mesh with level 1

finest mesh has 3,211,425 DoF. In this work we study the behavior of parallel smoothers based on additive splittings (Jacobi, multi-colored Gauß-Seidel and multi-colored symmetric Gauß-Seidel), incomplete factorizations (multi-colored ILU(0) and power(q)-pattern enhanced multi-colored ILU(p,q) with fill-ins) and FSAI smoothers. We investigate their properties with respect to high-frequency error reduction and convergence behavior of the V- and W-cycle parallel MG. In our tests performed here we did not consider damping in order to reduce complexity of the parameter space (except for the Jacobi smoother where damping is necessary).

In particular, we consider the multigrid behavior on different test platforms. Our numerical experiments are performed on a hybrid platform, a dual-socket Intel Xeon Harpertown (E5450) quad-core system (with eight cores in total) that is accelerated by an NVIDIA Tesla S1070 GPU system with four GPUs attached pairwise by PCIe to one socket each where we only use a single GPU in our experiments. The memory capacity of a single CPU and GPU device is 16GB and 4GB respectively. We are only using double precision for the computations.

We perform several tests with different configurations for the pre- and post-smoothing steps. We determine the number of cycles required to achieve a relative residual less than 10^{-6}. In Table 1 the iteration counts are shown for the V-cycle based MG. Note, that the iteration counts in this table do not reflect the total amount of work. Some smoothing steps, e.g. ILU(p,q), need more work than others. From a theoretical point of view, a Gauß-Seidel smoothing step is half as costly as a symmetric Gauß-Seidel step; ILU(0) is cheaper than ILU(1,1) which is cheaper than ILU(1,2). For the MG solver, $\nu_1 + \nu_2$ is the number of smoothing steps on each level and should be kept low in order to reduce the total amount of work (and hence the solver time). For estimation of the corresponding work load, computing times are included for the MG solver on the GPU platform.

Table 1. Number of MG V-cycles for different smoothers and total run times of the V-cycle based MG solver on the GPU for different smoother configurations; ν_1 is the number of pre-smoothing steps, ν_2 is the post-smoothing step count. (Best run time results highlighted in boldface.)

(ν_1, ν_2)	(0,1)	(0,2)	(0,3)	(0,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(2,0)	(2,1)	(2,2)	(2,3)
GS [#it]	30	14	10	8	35	16	10	8	7	20	12	9	7
time [sec]	5.75	3.89	3.62	**3.58**	6.44	4.43	3.63	3.59	3.73	5.33	4.34	4.03	3.74
SGS [#it]	23	13	9	8	28	14	10	8	7	17	11	9	7
time[sec]	4.89	4.17	**3.88**	4.28	5.71	4.49	4.28	4.28	4.51	5.29	4.69	4.80	4.49
ILU(0) [#it]	18	11	8	6	25	11	8	7	6	15	8	7	6
time [sec]	3.80	3.49	3.40	**3.19**	5.00	3.50	3.40	3.72	3.82	4.59	3.42	3.72	3.83
ILU(1,1) [#it]	18	10	7	6	23	11	8	7	6	14	8	7	6
time [sec]	4.30	3.73	**3.56**	3.85	5.26	4.10	4.06	4.49	4.66	5.06	4.06	4.49	4.65
ILU(1,2) [#it]	10	6	5	5	12	7	5	5	5	9	6	5	5
time [sec]	2.93	**2.75**	3.08	3.86	3.36	3.20	3.10	3.87	4.65	3.98	3.69	3.87	4.66
FSAI(1) [#it]	17	9	7	6	22	10	7	6	6	14	8	7	6
time [sec]	4.14	**3.35**	3.46	3.69	5.41	3.72	3.47	3.70	4.39	5.18	3.94	4.27	4.39
FSAI(2) [#it]	14	7	6	5	15	8	6	5	5	10	6	6	5
time [sec]	4.54	**4.08**	4.96	5.36	5.71	4.64	4.96	5.38	6.59	6.30	4.96	6.42	6.60
FSAI(3) [#it]	9	6	5	5	12	6	5	5	5	8	6	5	5
time [sec]	**4.04**	5.15	6.40	8.42	6.58	5.20	6.39	8.45	10.50	7.77	7.63	8.44	10.49

For the damped Jacobi smoother with ($\omega < 1$) the run times are larger than 20 seconds and therefore not competitive. Multi-colored symmetric Gauß-Seidel (SGS) is better than multi-colored Gauß-Seidel (GS) in terms of iteration count. Multi-colored ILU smoothers are better than additive factorizations (GS, SGS). The quality of the ILU(p) schemes in terms of iteration counts gets better with increasing p. The drop-off technique for ILU(1,1) is providing only little improvements compared to ILU(0). Iteration counts for the FSAI smoothers are in the same range as the ILU smoothers. Best candidates with respect to reduced iteration counts are ILU(1,2) and FSAI(2). Minima with respect to the iteration count can be found for configurations where $\nu_1 + \nu_2 = 2, 3$ or 4. For larger values there are no more significant improvements. The run time results show that the benefits for the SGS smoother in terms of smoothing properties and reduced iteration count are canceled with respect to the run time due to the additional overhead in each smoothing step and an additional V-cycle. For the ILU smoothers the additional work complexity still yields improvements in run time. The best performance is obtained for the ILU(1,2) smoother. The drop-off technique for ILU(1,1) has some diminishment in performance. The FSAI smoothers give no particular improvements in run time compared to the other smoothers. All timing considerations exclude the setup times for multi-coloring, factorizations or building FSAI. A more detailed parameter study – also for W-cycle-based multigrid – can be found in [14]. Although the chosen Poisson test problem in an L-shaped domain is not a complex scenario that requires strong smoothers by all means, our investigation for this problem already shows that there are benefits from higher level smoothers such as ILU(p,q). While our focus here is to demonstrate applicability of parallel smoothers and flexible multigrid on GPUs and multicore CPUs, the consideration of more complicated problems in terms of anisotropies are subject of future work.

In Table 2 the run times and iteration counts for the best V-cycle and W-cycle based smoother configurations for the parallel MG solvers on the GPU are summarized. The first column in this table specifies the optimal values for the parameters ν_1 and ν_2 that correspond to the number of pre- and post-smoothing steps on each refinement level. The third column gives the number of necessary MG cycle iterations. The iteration counts for the W-cycles are reduced compared to the V-cycle based MG solver. However, the run times show that the W-cycle based MG solver is by a factor of 2 to 3 slower than the V-cycle based MG solver. The W-cycle based MG solver is slower because more smoothing steps are performed on coarser grids. This involves a larger number of calls to SpMV routines for small matrices with significant overhead (in particular kernel call overheads on the GPU). The best results for W-cycle based MG are obtained

Table 2. Minimal run times and corresponding iteration counts for the V-cycle and W-cycle based parallel MG solvers on the GPU for various smoothers and optimal configurations of the corresponding number of pre- and post-smoothing steps (ν_1, ν_2)

	V-cycle based MG			W-cycle based MG		
Smoother	(ν_1, ν_2)	time [sec]	#its	(ν_1, ν_2)	time [sec]	#its
GS	(0,4)	3.58	8	(2,3)	7.01	6
SGS	(0,3)	3.88	9	(2,3)	8.20	6
ILU(0)	(0,4)	3.19	6	(2,3)	6.84	5
ILU(1,1)	(0,3)	3.56	7	(1,3)	8.01	6
ILU(1,2)	(0,2)	2.75	6	(2,2)	8.15	4
FSAI(1)	(0,2)	3.35	9	(1,3)	9.39	5
FSAI(2)	(0,2)	4.08	7	(1,3)	10.64	4
FSAI(3)	(0,1)	4.04	9	(0,2)	11.73	5

for the ILU(0) smoother with Gauß-Seidel following next. Except of the FSAI smoothers, all other smoothers are on the same performance level.

In some problem scenarios the necessary multigrid hierarchy may not be accessible and only the system matrix is available. In such cases, algebraic multigrid methods or preconditioned Krylow subspace solvers are a good alternative. Since the considered smoothers can also be used as preconditioners, we present a performance comparison for our model problem. In Table 3 the run times are listed for the preconditioned *conjugated gradient* (CG) method on the CPU and GPU test platforms using various preconditioning schemes. The number of CG iterations is given in the first row. We find that all preconditioners significantly reduce the number of iterations. Run times are presented for the sequential CPU version, the eight-core OpenMP parallel CPU version, and the GPU version. We see that the MG solver on the GPU is faster by a factor of up to 60 than the preconditioned CG solver on the GPU – which expresses asymptotic predominance since MG has an h-independent convergence behavior. The best preconditioner for CG on the GPU is ILU(1,2).

Table 3. Run times in seconds and iteration counts for the preconditioned CG solver for various preconditioners on the CPU (sequential and eight-core OpenMP parallel) and on the GPU

Precond	None	Jacobi	SGS	ILU(0)	ILU(1,1)	ILU(1,2)	FSAI(1)	FSAI(2)	FSAI(3)
# iter	5,650	4,167	2,323	2,451	2,066	1,387	2,198	1,493	1,139
Sequential	1492s	1134s	1433s	1472s	1347s	1498s	1271s	950.4s	1082.5s
OpenMP	604.4s	573.4s	662.6s	676.9s	630.0s	589.7s	435.5s	438.5s	500.3s
GPU	273.0s	218.3s	186.6s	195.1s	206.7s	158.0s	204.1s	280.0s	359.7s

6 Performance Analysis on Multicore CPUs and GPUs

In this section we conduct a performance analysis with respect to the different test platforms. We assess the corresponding solver times and the parallel speedups. In Figure 3 we compare run times of the V-cycle (left) and W-cycle (right) MG solver with various smoothers for the Poisson problem on the L-shaped domain. It shows the results for the sequential version and the OpenMP parallel version on eight cores of the CPU as well as for the GPU version. For this test problem, there are no significant differences in performance for the tested smoothers on a specific platform. The multiplicative ILU-type smoothers are slightly faster than the additive splitting-type smoothers. The ILU-based smoothers are more efficient in terms of smoothing properties and reduced iteration counts, but they are more expensive to be executed. In total, both effects interact such that the total execution time is only slightly better. Best performance results are obtained for the ILU(1,2) smoother based MG solver on the GPU. In this 2D Poisson test problem with unstructured grids based on Q1 and P1 elements, the resulting stiffness matrix has only six colors in the multicoloring decomposition. By increasing the sparsity pattern with respect to the structure of $|A|^2$ used for building the ILU(1,2) decomposition, we obtain 16 colors. In this case the triangular sweeps can still be performed with a high degree of parallelism. However, on the CPU the FSAI algorithms perform better due to the utilization of cache effects. The FSAI algorithms are performed by relying on parallel matrix-vector multiplications only. This is in contrast to the multi-coloring technique which re-orders the matrix by grouping the unknowns. This distribution is performing better on the GPU due to the lack of bank conflicts in the sparse matrix-vector multiplications. On the other hand, the FSAI algorithm is performing better on the cache-based architecture due to the large number of elements that can benefit from data reuse.

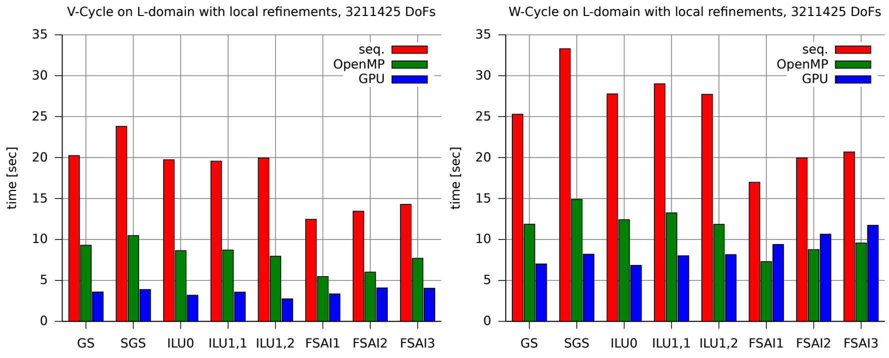


Fig. 3. Run times of V-cycle (left) and W-cyle (right) MG solver with various smoothers for the Poisson problem on the L-shaped domain: sequential version and OpenMP parallel version on eight cores on the CPU and GPU version

The parallel speedups of the V-cycle and W-cycle based MG solvers are detailed in Figure 4. The OpenMP parallel speedup is slightly above two. In the sequential run on a single CPU core, about less than one third of the eight core peak bandwidth can be utilized (see measurements in [15]). The bandwidth is already saturated with three cores. Therefore, the speedup of the eight-core OpenMP parallel version is technically limited by a factor of less than three on this particular test platform. This performance expectations are reflected by our measurements reported here. There is no difference in performance for different core allocation since our platform is a UMA architecture. The GPU version is by a factor of two to three faster than the OpenMP parallel version. These factors are in good conformance with experience for GPU acceleration of bandwidth-bound kernels. For the FSAI smoothers, the speedup on the GPU falls a little bit behind since it is better suited for CPU architectures. The presented parallel speedup results demonstrate the scalability of our parallel approach for efficient multigrid smoothers. Parallel multigrid solvers are typically affected by bad communication/computation ratios and load imbalance on coarser grids. As we are working on shared memory type devices, these influences do not occur.

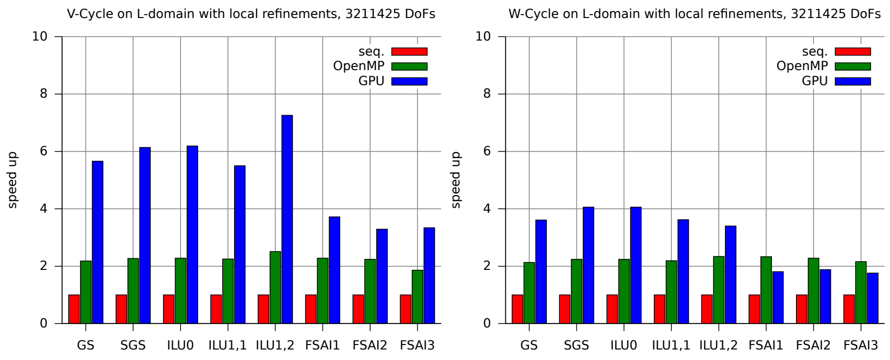


Fig. 4. Parallel speedup of the V-cycle (left) and W-cycle (right) based multigrid solvers for various smoothers

In contrast however, our matrix-based geometric multigrid solver depends by construction on the calls to SpMV kernels. On coarser grids, these kernels have a significant call overhead at the expense of parallel efficiency. This has a direct impact on the speedups of the W-cycle based MG solvers on the GPU.

Similar results are obtained for the performance numbers and speedups of the preconditioned CG solver. In the left part of Figure 5 run time results are listed for various preconditioners. The right figure details corresponding speedups for the OpenMP parallel version and the GPU implementation. Speedups for the CG are larger than those for the MG solver. Although both solvers consist of the same building blocks, CG is more efficient since it fully relies on the finest grid of the MG hierarchy with huge sparse matrices. In contrast, the MG solver is doing work on coarser grids where call overheads for SpMV kernels have a significant influence on the results.

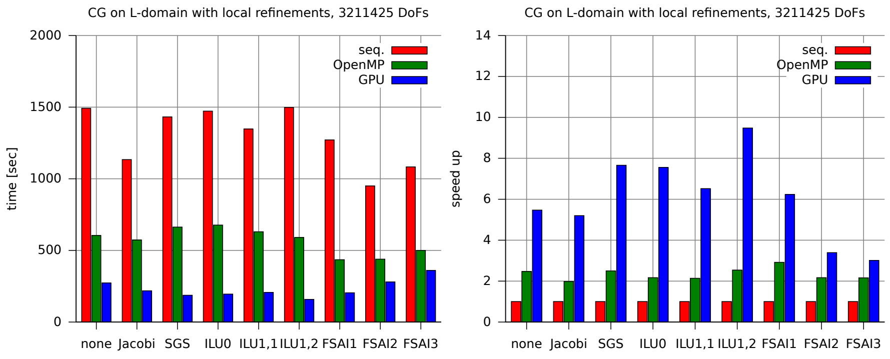


Fig. 5. Run times and parallel speedups for the preconditioned CG solvers on the CPU (sequential and eight-core OpenMP parallel version) and on the GPU

7 Conclusion

Matrix-based geometric multi-grid solvers on locally refined and unstructured meshes are efficient numerical schemes for solving complex and highly relevant problems. The paradigm shift towards manycore devices brings up new challenges in two dimensions: first, the algorithms need to express fine-grained parallelism and need to be designed with respect to scalability to hundreds and thousands of cores. Secondly, software solutions and implementations need to be designed flexible and portable in order to exploit the potential of various platforms in a unified approach. The proposed techniques for parallel smoothers and preconditioners provide both efficient and scalable parallel schemes. We have demonstrated how sophisticated mathematical techniques can be extended with respect to scalable parallelism and how these techniques can harness the computing power of modern manycore platforms. In particular, with the formulation in terms of SpMV kernels the capabilities of GPUs can be exploited. With the described solvers, solution times for realistic problems can be kept at a moderate level. And with the concept of our generic and portable software package, numerical solvers can be used on a variety of platforms on a single code base. The users are freed from specific hardware knowledge and platform-oriented optimizations. We have reported speedups and we have demonstrated that even complex algorithms can be successfully ported to GPUs with additional performance gains. The proposed ILU(1,2) smoother and the FSAI(2) smoother show convincing performance and scalability results for parallel matrix-based geometric MG. The proposed V-cycle based MG solvers with parallel smoothers on the GPU are by a factor of 60 faster than the preconditioned CG solvers on the GPU. But due to the absence of coarse grid operations, the CG solvers have slightly better scalability properties on GPUs.

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 its collaboration partner Hewlett-Packard. The Engineering Mathematics and Computing Lab (EMCL) at KIT has been awarded NVIDIA CUDA Center of Research for its research in GPU computing. The authors acknowledge the support by means of hardware donation from NVIDIA. We would also like to express our gratitude to Felix Riehn and Niels Wegh who have contributed to the implementation of our software.

References

1. Anzt, H., et al.: HiFlow^3 – a flexible and hardware-aware parallel finite element package. Tech. Rep. 2010-06, EMCL, KIT (2010), `http://www.emcl.kit.edu/preprints/emcl-preprint-2010-06.pdf`
2. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: SC 2009: Proc. of the Conf. on High Perf. Computing Networking, Storage and Analysis, pp. 1–11. ACM, New York (2009)
3. Bröker, O.: Parallel multigrid methods using sparse approximate inverses. Ph.D. thesis, Dept. of Computer Science, ETH Zürich (2003)

4. Bröker, O., Grote, M.: Sparse approximate inverse smoothers for geometric and algebraic multigrid. Applied Numerical Mathematics 41 (2002)
5. Bröker, O., Grote, M., Mayer, C., Reusken, A.: Robust parallel smoothing for multigrid via sparse approximate inverses. SIAM J. of Scient. Comput. 23 (2001)
6. Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P., Turek, S.: Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. In: Iványi, P., Topping, B.H. (eds.) Second Int. Conf. on Parallel, Distributed, Grid and Cloud Computing for Engineering (2011), doi:10.4203/ccp.95.22
7. Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P., Turek, S.: Towards a complete FEM-based simulation toolkit on GPUs: Geometric multigrid solvers. In: 23rd Int. Conf. on Parallel Computational Fluid Dynamics, ParCFD 2011 (2011)
8. Göddeke, D.: Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters. Ph.D. thesis, Technische Universität Dortmund (2010)
9. Grote, M., Huckle, T.: Parallel preconditioning with sparse approximate inverses. SIAM J. of Scient. Comput. 18 (1997)
10. Haase, G., Liebmann, M., Douglas, C.C., Plank, G.: A Parallel Algebraic Multigrid Solver on Graphics Processing Units. In: Zhang, W., Chen, Z., Douglas, C.C., Tong, W. (eds.) HPCA 2009. LNCS, vol. 5938, pp. 38–47. Springer, Heidelberg (2010)
11. Hackbusch, W.: Multi-grid methods and applications. Springer, Berlin (2003)
12. Heuveline, V., et al.: HiFlow^3 - parallel finite element software (2011), `http://www.hiflow3.org/`
13. Heuveline, V., Lukarski, D., Subramanian, C., Weiss, J.P.: Parallel preconditioning and modular finite element solvers on hybrid CPU-GPU systems. In: Iványi, P., Topping, B.H. (eds.) Proc. of the 2nd Int. Conf. on Parallel, Distr., Grid and Cloud Comp. for Eng., Paper 36, Civil-Comp Press (2011)
14. Heuveline, V., Lukarski, D., Trost, N., Weiss, J.P.: Parallel smoothers for matrix-based multigrid methods on unstructured meshes using multicore CPUs and GPUs. Tech. Rep. 2011-09, EMCL, KIT (2011), `http://www.emcl.kit.edu/preprints/emcl-preprint-2011-09.pdf`
15. Heuveline, V., Lukarski, D., Weiss, J.-P.: Scalable Multi-coloring Preconditioning for Multi-core CPUs and GPUs. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 389–397. Springer, Heidelberg (2011)
16. Heuveline, V., Lukarski, D., Weiss, J.P.: Enhanced parallel ILU(p)-based preconditioners for multi-core CPUs and GPUs – the power(q)-pattern method. Tech. Rep. 2011-08, EMCL, KIT (2011), `http://www.emcl.kit.edu/preprints/emcl-preprint-2011-08.pdf`
17. Heuveline, V., Subramanian, C., Lukarski, D., Weiss, J.P.: A multi-platform linear algebra toolbox for finite element solvers on heterogeneous clusters. In: PPAAC 2010, IEEE Cluster 2010 Workshops (2010)
18. Hülsemann, F., Kowarschik, M., Mohr, M., Rüde, U.: Parallel geometric multigrid. In: Numerical Solution of Partial Differential Equations on Parallel Computers. LNCSE, ch. 5, vol. 51, pp. 165–208 (2005)
19. Kallischko, A.: Modified sparse approximate inverses (MSPAI) for parallel preconditioning. Ph.D. thesis, Fakultät für Mathematik, Technische Universität München (2008)
20. Kolotilina, L., Yeremin, A.: Factorized sparse approximate inverse preconditionings, I: theory. SIAM J. Matrix Anal. Appl. (1993)
21. Trottenberg, U.: Multigrid. Academic Press, Amsterdam (2001)

Author Index