On the Effective Parallel Programming of Multi-core Processors

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op dinsdag 7 december 2010 om 10.00 uur

door

Ana Lucia VÂRBĂNESCU

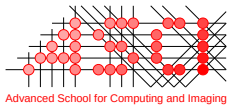
Master of Science in Architecturi Avansate de Calcul,
Universitatea POLITEHNICA Bucureşti, România
geboren te Boekarest, Roemenië.

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. ir. H.J. Sips

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr.ir. H.J. Sips	Technische Universiteit Delft, *promotor*
Prof.dr.ir. A.J.C. van Gemund	Technische Universiteit Delft
Prof.dr.ir. H.E. Bal	Vrije Universiteit Amsterdam, The Netherlands
Prof.dr.ir. P.H.J. Kelly	Imperial College of London, United Kingdom
Prof.dr.ing. N. Ţăpuş	Universitatea Politehnicã Bucureşti, Romania
Dr. R.M. Badia	Barcelona Supercomputing Center, Spain
Dr. M. Perrone	IBM TJ Watson Research Center, USA
Prof.dr. K.G. Langendoen	Technische Universiteit Delft, *reservelid*



This work was carried out in the ASCI graduate school.
ASCI dissertation series number **223**.

The work in this dissertation was performed in the context of the Scalp project, funded by STW-PROGRESS.

Parts of this work have been done in collaboration and with the material support of the IBM T.J. Watson Research Center, USA.

Part of this work has been done in collaboration with the Barcelona Supercomputing Center, and supported by a HPC Europa mobility grant.

ISBN/EAN: 978-90-8570-708-0

Printed in the Netherlands by Wöhrmann Print Service (www.wps.nl).

Acknowledgments

There have been many moments in which my PhD felt a lonely fight against the world. But now, when writing the very last pages of my thesis - the acknowledgements - I realize I've been lucky to have a lot of people by my side, all the way. I try to thank all of them here - in a vague chronological order. As the list can only be limited and therefore prone to omissions - I will probably fail. For those of you that I might have omitted: please accept my apologies, my thanks, and a glass of wine from me, to wash away the sorrow.

I want to thank Prof. Nicolae Tapus, for his guidance during my academic years in Romania, and for accepting to be part of my committee. I am grateful to late Prof. Irina Athanasiu, for her continuous encouragement, and for her role in my first collaboration with TUDelft. And I want to thank Prof. Sorin Cotofana, for making my first encounters with TUDelft, The Netherlands, and the Dutch interesting and comfortable.

Special thanks go to Prof. Henk Sips, my supervisor, for his help and support, for putting up with all my last minute ideas and panic moments (starting from that CPC'06 paper). I owe him thanks for his patience in waiting for my thesis, who was "almost ready" for many months, but also for our enlighting discussions about non-scientific topics such as sailing, traveling, wine, or earrings.

I'd like to thank Prof. Arjan van Gemund for his help on PAM-SoC and the SCALP project, and for the valuable insight in performance analysis, modeling, and prediction. Furhter thanks go to the other SCALP members: Dr. Maik Nijhuis, Dr. Herbert Bos, and Prof. Henri Bal (from Vrije Universiteit Amsterdam) - for our collaboration in building SP@CE. One extra "thank you" to Prof. Henri Bal for his useful comments on the draft of my thesis. Thanks to Paul Stravers and Jos van Eindhoven (currently at VectorFabrics) for their help with the SpaceCAKE architecture itself, and for the valuable insights in the pragmatic thinking of "the industry".

I would like to thank Dr. Michael Perrone, my two-time manager at IBM TJ Watson Research Center - his enthusiasm for the Cell/B.E., and his ability to quickly detect good solutions from bad ones and extract the essence from any (vague) idea played an important role in my work so far. Thank you, Michael, for your help and support, for the subtle ways you found to motivate me, and for the continuous interest you showed in my career. Last, but not least, thank you for the suggestions made for improving this thesis. Additional thanks go to the other members of the Cell Solutions team with whom I worked at TJ Watson - Gordon Braudaway, Daniele Scarpazza, Fabrizio Petrini, and Virat Agarwal - for sharing with me some of the secrets of Cell/B.E. programming.

I owe thanks to Rosa Badia and Xavier Martorell, with whom I worked for six weeks at the Barcelona Supercomputing Center (and continued to collaborate afterwards). Our work on classifying,

comparing, and evaluating a large variety of programming models, has helped me understand more of the subtle effects that programmability and productivity have on platform adoption and performance.

Thanks to Ian Buck and Stephen Jones for my nice summer internship in the CUDA team from NVIDIA, where I had to opportunity to see how difficult it is to build the right software tools and models in a hardware-dominated environment.

I'd like to thank Prof. Paul Kelly for accepting to be part of my committee, for his comments on the draft of my thesis, as well as for our occassional, yet very interesting discussions on parallelism and performance. And thank you, Prof. Joerg Keller, for our fruitful collaboration, which I hope to maintain and improve.

I would like to end my "professional" acknowledgements by thanking two of my most stable scientific partners, and friends: Dr. Rob van Nieuwpoort and Alexander van Amesfoort. Thanks, guys, for the smooth collaboration in our various projects together, and for our entertaining discussions. I surely hope we will keep working together for a while longer.

I am grateful for all the help and support I got from my friends and my family. Thanks, Monel and Mihaitza for always finding the energy to spend some time with me. Thanks, Robert, for keeping me up to date with the pubs in Bucharest. Thanks Nicoleta, Ervin and Anca Vitos, and Michou for your good thoughts from far away. Thank you, Ana Maria O. and Corina for the fun in Amsterdam, and for geek girls nights out. Thank you, Karan, for being so close to me from the other side of the world. Thank you, Ivo, for the wine and cheese evenings, and for being such a great officemate - no wonder I never wanted to work from home! Thanks, Otto, for your help with the Dutch in all its/their forms, and for your visits on the dark side of the corridor. Thanks, Pawel and Alex F., for making weekends at the TUDelft bearable.

These years, basketball has been an important part of my life. I am grateful to my teammates, trainers, and coaches from Punch, for the many fun memories on and off court. Special thanks to Tessa, Marije, Marta, Charlotte, Siri, Bridgit, and Jill for their kind friendship outside the club. Thanks to "my" Heren3 team, for accepting such a shouting coach, and for teaching me the inner beauty of pink. Thank you, Remus, Coco, Patricia, and Rita for the many crazy tournaments we have played together - I enjoyed them a lot. And thank you very much, Maider, for extending our friendship so far beyond basketball.

I would not remember my time in the US with that much pleasure if it wouldn't have been for the friends I made there. Thank you, Janet, for your kindness and your care during the months I spent in your house. Thank you, Nicolas, for the biking trips and for your restless efforts to make me understand the beauty of american football (sorry, I still don't get it). Thank you, Marcio, for the afternoon coffees. Thank you, Virat and Leo, for the tennis matches and the movie evenings in the conference rooms. Thank you, Mesut, for the wine and cheese parties at Janet's. Thank you, Vlad, for showing me some of the beautiful places of California.

Finally, there are four more special people in my life, to whom I am most grateful - without them, all these years would have been a lot tougher. Thank you, Ancutza, for being my best friend all these years, for being there in the cloudiest of days (and those are plenty in The Netherlands!), for listening and for talking, for always being honest, direct, and fair. Thank you, Alexandru, for your love, for putting up with my mood swings and pre-deadline panic, for standing by my side when I needed it, for believing in me and for encouraging me; thank you for your patience and your calm support. And thank you, Mom and Dad, for teaching me to work hard and to never give up, and for inspiring me to be the best in what I do. Thank you for being so proud of me. Thank you, Mom, for teaching me the magic powers of good food. Thank you, Dad, for these last 25 years of basketball and computing machineries. Thank you for everything. I love you very very much.

Ana Lucia.

Contents

Chapter 1

Introduction

The pace of the entire (computational) world is increasing. We all push the limits of (conventional) processors when we require more accurate, more detailed, and quicker results from the multitude of devices we use every day. In fact, we all demand *better performance* from these devices.

Over the last 40 years, the responsibility for increasing application performance (or the blame for the lack there of) has oscillated between hardware and software. Originally, processors had very limited resources (e.g., a few hundreds of kilobytes of main memory, one simple arithmetic-logical unit, and one slow floating point processing unit), so programmers had to invent ingenious solutions to efficiently use them. That was the time when major optimization techniques, in both compilers and programming models, have emerged.

As technology advanced, larger and faster memories, combined with faster and more complex processing units, have significantly increased the hardware performance. When the hardware constraints reduced, application development became easier, with many designers/developers paying less attention to detailed optimizations in favor of shorter time-to-delivery. This trend has generated a very large collection of algorithms and implementations in a multitude of computational fields. For these applications, speed-up came almost for free, as every new processor generation increased clock frequency, allowing for more operations to be solved per time unit. Although parallelism and concurrency were already recognized as potential performance boosters, they remained the "fancy" tool of High Performance Computing (HPC), despite impressive performance gains for compute-intensive scientific applications running on large-scale parallel machines.

In the early 2000s, the frequency increasing trend slowed down, mainly due to power consumption and overheating concerns. Even more, it became obvious that this technique can no longer sustain, by itself, significant performance improvements. Multi-core processors have emerged as an alternative approach: instead of a single, large, very complex, power hungry core per processor, multiple "simplified" cores are tightly coupled in a single chip and, by running concurrently, they yield better overall performance than the equivalent single-cores[1].

In terms of applications, this shift from single-core processors to multi-cores made parallelism ubiquitous. Quite suddenly, most existing applications, as well as applications emerging from new computational fields, have to be parallelized, optimized, tuned, and deployed on multi-core processors. This process is non-trivial: finding the right architecture for a given application, choosing the right parallelization and data distribution strategies, and tuning the code for the specifics of the chosen architecture are all difficult tasks. Also, the variety of available architectures decreases portability, leading to significant differences in algorithm design. The increased number of parallelism layers and the finer granularity imposed by the tightly-coupled cores make "inherited" parallelization techniques (i.e., solutions available for large-scale parallel machines) hard to (re)use. Finally, performance analysis methodologies and prediction tools are practically nonexistent.

[1] It is worth mentioning that, before becoming **the** viable alternative to traditional processors, multi-cores have had other names, such as *multi-processor system-on-chips (MPSoCs)*, or *chip multi-processors (CMPs)*

1.1 The Hardware Push

Multi-cores are now designed and developed by all the important players on the processors market. In only five years, this inherent competition has generated a broad range of multi-core architectures. Major differences include the types and numbers of cores, the memory hierarchy, the communication infrastructure, and the parallelism models. For example, a Cell Broadband Engine (Cell/B.E.) offers eight dedicated RISC accelerator cores managed by a ninth light PowerPC core, in a Direct Memory Access (DMA)-controlled distributed memory model. The latest Intel© Core i7 has eight x86 cores, all hypethreaded [Int09]. A typical NVIDIA Graphical Processing Unit (GPU) has 16 to 30 processing units, each comprising of 16 or 32 "individual" computing units; various shared and distributed memory modules are "scattered" in the chip.

However, the history of parallel computers has already seen so many architectures that it is hard not to wonder what is new with multi-cores. The differences are both fundamental and subtle, and they refer to three main issues: (1) fine granularity, (2) multiple parallelism layers, and (3) very complex memory infrastructure. In the following paragraphs we discuss these key differences in more detail, proving that the complexity level of multi-cores is significantly higher than that of traditional parallel architectures.

Parallelism Layers

Multi-core processors offer multiple layers of parallelism: Core-Level Parallelism (CLP), multithreading, Single Instruction Multiple Data (SIMD), Instruction Level Parallelism (ILP), and communication-computation overlapping.

Core-Level Parallelism (equivalent to Multiple Instruction Multiple Data (MIMD)) is present in processors where cores can run (pseudo-)independent tasks. It is typical for General Purpose Processor (GPP) multi-cores and for heterogeneous platforms. Further, cores may offer multithreading, a technique that allows multiple (hardware) threads to run in parallel. SIMD is a technique that allows multiple threads to run the same operations on different data elements; note that for different architectures, SIMD can be exploited at different levels (core level, thread level, or instruction level). Below multithreading and SIMD, cores use Instruction Level Parallelism to overlap instruction execution. Finally, several architectures can overlap off-chip communication and computation (using techniques similar to prefetching or context switch in case of memory stalls), resulting in parallelism between some memory and compute instructions.

Of course, multiple parallelism layers can also be found in larger scale parallel architectures, like clusters or Massively Parallel Processors (MPPs). However, the impact of successive layers on the overall performance is usually orders of magnitude apart, and quite independent. In the case of multi-cores, each of these layers is critical to overall performance. Note that the theoretical performance of a multi-core is quoted with full parallelism, i.e., benchmarked with a set of applications that fully exploit each of these layers. Unfortunately, as experience shows, very few real-life applications can be "divided" to expose enough parallelism to fill the entire stack. Further, because the layers are not independent and the relative impact on performance is comparable, the performance effects when choosing one layer over another are hard to predict.

Large Scale, Fine Granularity

We define the (computational) granularity of a parallel architecture as the largest "computational unit" that should run sequentially on a node. Note that granularity is directly connected with the parallelism

layers of a platform: a platform that offers low-level parallelism will implicitly have fine granularity. Additionally, the coarser the platform granularity is, the least intrusive application parallelization will be: fine granularity requires more synchronization points and complex data distribution.

Finally, the simplest accepted paradigm for programming parallel machines is for the programmer to define application computational units and their communication patterns, and let the system-level tools deal with concurrent execution (via mapping) and lower-level parallelization (via compiler and hardware optimizations).

When comparing multi-core with conventional parallel machines, the difference in granularity is significant. For most traditional parallel machines, the granularity is logically uniform, and the computational units are, in both the SPMD and MPMD models, fairly coarse. In the case of multi-core processors, one should investigate multiple granularity levels, starting from tasks, but also investigating at thread-level and loop-level parallelism, in search for appropriately sized computational units (i.e., a unit that fits the resources of a node and allows for concurrent execution with other units). Thus, we typically deal with non-uniform, nested granularity layers.

The lower granularity of multi-cores has clear implications for (1) performance, (2) mapping/scheduling, and (3) low-level parallelization tools. A poor choice of the parallelization granularity decreases platform utilization and limits the achievable performance to only a fraction of the theoretical peak. As the computational units become smaller, we deal with a large-scale mapping and scheduling problem. Finally, low-level parallelization tools have to act as very fine tuners/optimizers, a cumbersome task as the number of techniques with performance gain potential is unpredictable (i.e., it depends on the application, on the platform, and most of the time also on the mapping/scheduling decisions).

Memory and Data Locality

For multi-core based systems, we consider a typical memory hierarchy to include both on-chip and off-chip memories, and the I/O subsystem. As expected, on-chip memories are small (and offer low-latency), while off-chip memories typically large (but they also exhibit high-latency). Most architectures opt for a layered on-chip memory hierarchy, with per-core ("local") memories and shared memories per groups of cores. Some architectures, like Intel's Core i7 or the newest NVIDIA GPUs use some of the local memories as caches, while others, such as the Cell/B.E., give the programmer full control over the transfers.

Most on-chip memories have strict restrictions on access alignments, patterns, and sizes in order to minimize the latency and fully utilize the memory bandwidth. Low-latency is essential for decreasing communication overheads (especially for fine-grain parallelism), while increased bandwidth allows the hardware to overlap memory accesses and computation efficiently.

Data locality plays an essential role in the performance of multi-cores, similar to traditional parallel machines: fetching data from a close-by memory is always faster than fetching data from remote memories. Consequently, the communication cost of an application is increased by the number and the latencies of the memory layers, as well as by the frequency of memory operations. A compact memory hierarchy, typical for traditional parallel machines, allows for larger memories per layer, and minimizes the need for inter-layer memory traffic. A "taller" memory hierarchy, typical for multi-cores, has less memory per layer, increasing memory traffic between layers; ultimately, this translates into severe performance penalties for poor data locality.

These considerations can be easily extended to the I/O subsystems, i.e., the connection of the processor with "the outside world". Multi-core systems have two types of I/O traffic: to/from the off-chip memory and to/from the on-chip memory. The bandwidth of both these connections is a

crucial factor in the overall performance of the system. In fact, in many cases, the first estimate of a maximum achievable performance of an application is not given by the platform's computational throughput, but rather by the platform's I/O bandwidth.

1.2 The Programmability Gap

Once multi-core processors have become state-of-the-art in computing systems, applications have to use parallelism and/or concurrency to achieve the performance these chips can offer. The failure to do so will make multi-cores highly inefficient and, ultimately, application performance will stall or even decrease.

Despite the very large collection of (traditional) parallel programming models, languages, compilers, the software community is still lagging behind: the applications, the operating systems, and the programming tools/environments are not yet coping with the parallelism requirements posed by this new generation of hardware. With new platforms being advertised every month, the widening "multi-core programmability gap" has become a major concern shared by both communities.

Attempts to bridge the programmability gap have been quite abundant in the last few years. Application case studies, dedicated programming models, and a few compilers and code generators have enabled, to some extent, a systematic approach to multi-core programming. But most of these initiatives lack generality, infer performance penalties, and/or pose severe usability limitations.

In reality, multi-core programmers still base their application design and implementation on previous experience, hacking, patching, and improvisation. Furthermore, when considering the entire pile of applications that have to be implemented for multi-cores, these solutions can be generically called "needle-in-the-haystack methods".

Nevertheless, even when these methods eventually succeed, they raise three fundamental issues: (1) there is no measure of success (except the functional correctness), (2) productivity and efficiency are unpredictable, and (3) the path to the found solution is in most cases non-reconstructible and non-repeatable for new problems (even for slight variation of the originally solved problem). Any approach towards systematic parallel programming has to tackle the three key issues that are ignored by the needle-in-the-haystack methods: (1) performance, (2) productivity, and (3) portability.

Performance

The ultimate goal of using new hardware platforms is the search for increased *performance*. However, performance in itself is a complex function of multiple parameters: the application, the chosen algorithm and its implementation, the input data set, and the hardware platform. When compared with single-core applications, multi-core processors increase the complexity of this performance function to a new level, both in terms of numbers of parameters and in terms of their interdependence.

Also note that performance metrics are different for various computing fields. For example, HPC applications aim to minimize execution time. In turn, typical embedded systems applications focus on real-time constraints. Streaming applications (think of on-line multimedia streaming, for example) focus on data throughput. Mobile applications target low power consumption.

Not all these performance metrics are easy to measure. Furthermore, their way to relate to hardware complexity and/or application modules/phases is non-trivial. Therefore, performance analysis becomes complicated, while traditional performance estimation loses accuracy.

Nevertheless, application development must be performance-driven. Remember that the increased complexity of multi-core processors effectively means that the number of ways to solve a problem

(i.e., the variations of algorithms, implementations, and optimizations) increases significantly. Consequently, the choices programmers are facing increase and testing all solutions becomes unfeasible. In many cases, implementation and optimization decisions are taken either based on programmer experience or even arbitrarily. We strongly believe that including simple performance estimation (for the application-specific metrics) early in the development process leads to a bounded solution search space and its systematic exploration.

Productivity

Currently, the hardware players lead the effort, releasing new processors with increased theoretical performance every few months. Still, this trend will end very soon, as the critical mass of applications unable to get a decent fraction of the theoretical peak performance is reached. We argue that only productivity and efficiency can significantly alleviate this problem, by enabling a shorter and more effective application development process.

For multi-core application, we loosely define productivity as the achieved performance per time unit spent on programming. One can see this as an efficiency metric, i.e., a ratio between the achieved performance and the invested effort. Either way, to increase productivity, one has to balance the application performance with the time spent in the development process.

Speeding up the development process is only possible by using high-level programming models and performance analyzers, as they offer an increased level of standardized programming, decreasing development time and, more importantly, decreasing maintenance and update costs. Further, we acknowledge that programmers fiddling around with applications might get (slightly) better performance, but this is usually a result of using platform-specific optimizations and tricks, hard to reproduce and estimate for a general system. Therefore, we belive that hand-tuning and platform-specific optimizations should come only as the final parts of the development process.

Portability

The increased variety of multi-core processors brings along a new question: given a company that has to improve the performance of a large application, what platform should be used for achieving best results?

The answer is non-trivial. First, these "best results" have to be qualified and quantified, by defining the metrics and their acceptable values. Next, application and data set characteristics have to be taken into account for establishing the platform components and scale.

When the options for platform components (i.e., one or more types of multi-cores, in our case) are very different, the choice is made by some sort of estimation, be it driven by experimental or analytical evidence, or by programmer expertise. However, once this choice is made, the result is a platform-specific implementation. And if the original platform choice does not meet the set performance goals, the development process is restarted.

In a context where most platforms are not compatible, this is yet another reason for low productivity/efficiency. Therefore, portability must be achieved by the software layers. Of course, such a model is unable to deliver stable good performance for entirely different platforms. However, a high-level programming model that enables portability among various hardware platforms is much more likely to be adopted than a platform-specific one. Such a model can also become an incentive for eliminating "hardware glitches" (i.e., features that are cluttering the hardware while only delivering very little performance) because a (semi-)universal software model will force hardware designers to think if the

model can express and use the novel features they plan to include, thus being forced to evaluate the impact these changes have on software.

1.3 Problem Statement

Both the lack of structure in the field of multi-core programming, as well as the low effectiveness of available parallel programming tools are direct consequences of the high complexity and fine granularity that multi-core architectures exhibit. The multiple parallelism layers, the complicated memory hierarchies, and the delicate performance balance between arithmetic, memory, and I/O operations generate a large variety of solutions to a single problem. Searching for the right solution, especially when using a non-systematic approach, makes multi-core programming difficult, as it usually comes down to an iterative process with an unclear stop-condition.

We believe that the current state-of-the-art in programming multi-cores is no more than a "needle-in-the-haystack" search, being driven by programmer experience and ad-hoc empirical choices. This approach hinders performance, productivity, and portability. In this thesis, we investigate whether a structured approach to parallel programming for multi-cores is possible and useful, as well as what conditions, actions, and tools are necessary to transform this approach into a complete framework that can improve the design and development of parallel applications for multi-core processors.

1.3.1 Approach

To structure multi-core application development, we propose **MAP**[2], a complete framework that guides the user through the phases of application development, from analysis to implementation and optimizations. Essentially, the framework, sketched in Figure 1.1, takes as input an *application specification* and a *set of performance targets*, and allows the programmer to use a step-by-step approach to analyze, design, prototype, implement, and optimize a multi-core application. Furthermore, the framework enables hybrid design, allowing a switch from application-driven to hardware-driven development in any stage, by taking platform specific decisions, at the expense of portability (i.e., the sooner this decision is taken, the lower the application portability will be). Finally, the framework also includes a performance feedback loop, which enables an essential roll-back mechanism, based on performance checkpoints, which allows the user to come back at previous design stages and take a different path without sacrificing the entire design flow.

By the end of this thesis, we show an application development strategy for multi-core applications, we discuss the MAP framework prototype, and we investigate the tools to support our framework.

1.4 Thesis Main Contributions and Outline

The main goal of this thesis is to propose a framework for efficient programming of parallel applications on multi-core processors. To build this framework, we first investigate the state-of-the-art multi-cores and predict a future trend in their development (Chapter 2). Further, we analyze three case-studies (Chapters 3, 4, 5), aiming to determine the major challenges that arise in application development, from design to implementation and optimization. We also give an example of the practical challenges

[2] **MAP** stands for "Multi-cores Application Performance", the three major elements of our framework

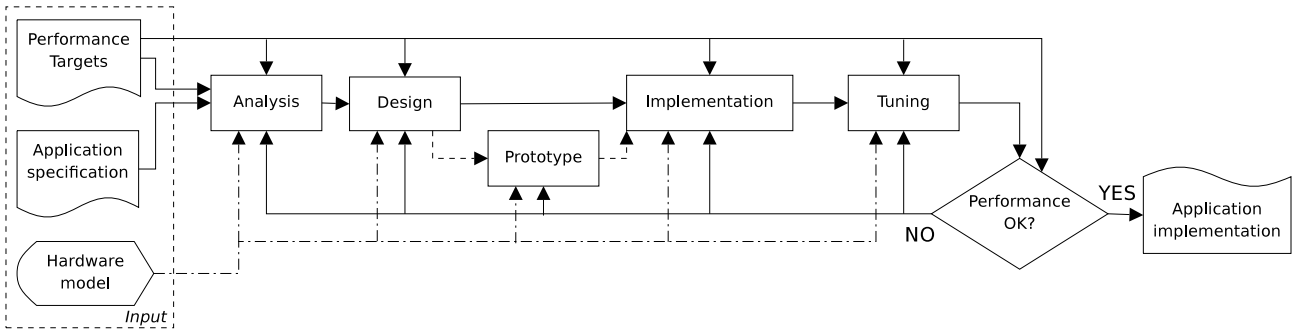


Figure 1.1: *The MAP framework. Application specification and performance targets are user-specified inputs. Platform and application modeling, as well prototyping and programming tools are parts of the framework tool-chain.*

of porting legacy code on multi-cores (Chapter 6). We summarize these challenges and we verify if any of the modern programming tools, mostly designed to support multi-cores, offer the right mix of features to support the user in the whole development process (Chapter 7). We find that the programming models are separated into two disjoint classes: application-centric and hardware-centric. This split mimics, and unfortunately widens, the programmability gap. To bridge this gap, we devise a complete analysis-to-implementation strategy to deal with multi-core applications. Next, we design the framework prototype (Chapter 8) to enable this strategy, and we investigate the availability/effort of building a potential tool-chain. We conclude that despite the elegance and completeness of the solution, the state-of-the-art tool support is by no means sufficient. Among other interesting future research directions (Chapter 9), we recommend more effort to be invested in developing these support tools (eventually at the expense of hardware-centric programming models).

1.4.1 Thesis Outline

Chapter 2 presents the state-of-the-art in multi-core architectures, including the Cell/B.E., graphical processing units, and homogeneous multi-cores (for desktops, servers, and HPC clusters). We summarize by discussing architectural trends and their impact on the application world.

Chapter 3 describes a multi-core specific implementation of a traditional HPC application, and the lessons to be learned from this exercise. This implementation focuses on the compute-intensive kernel, and investigates Cell-specific low-level optimizations that enable the overall execution time to drop more than a factor of 20. This chapter is based on joint work with IBM Research, previously published in IPDPS'07 ([Pet07]).

Chapter 4 describes a multi-core specific implementation of a large, multi-kernel image content-analysis application. Specifically, we investigate both low-level optimizations and the influence of high-level mapping and scheduling of kernels on a heterogeneous multi-core. Finally, we compare the application behavior for task, data, and hybrid parallelization, and we show unintuitive performance results. This chapter is based on joint work with IBM Research, previously published in ICME'07 ([Liu07]) and Concurrency and Computation: Practice and Experience 21(1) ([Var09b]).

Chapter 5 focuses on the implementation of a data-intensive application. The application is part of a radio-astronomy imaging chain, and requires very large, unordered collections of data to be processed. We investigate the challenges it poses for a multi-core processor - data intensive behavior,

random memory access patterns, very large data structures - and we propose potential solutions to address them. This chapter is based on work previously published in Euro-Par 2008 ([Var08b]) and in Scientific Programming 17(1-2), 2009 ([Var09a]).

Chapter 6 presents some of the challenges that one can encounter when porting legacy, object-oriented applications on multi-core processors. Specifically, we discuss a generic, template-based strategy for porting large C++ applications on an accelerator-based multi-core processor. We validate the strategy by showing how MarCell [Liu07] was ported onto the Cell/B.E.. Although not fully aligned with the thesis story-line, our reasons to include this chapter are twofold: we give one more example of the difficulties of multi-core programming, and we provide more technical details for the discussion in Chapter 4.

Chapter 7 summarizes the lessons we have learned from all these case studies. We extract two sets of best-practice guidelines: a platform-agnostic set, mainly addressing parallel application design for multi-core processors, and a platform-specific set, combining guidelines for all three multi-core processor families used for HPC applications. Finally, based on these best practices, we investigate the available multi-core programming models for a best match. The work in this chapter is partially based on work presented at CPC'09 ([A.L09]).

Chapter 8 presents a strategy for systematic multi-core application development. We propose the MAP framework as the right way to support this strategy, and we show how a MAP prototype should look like. We further investigate the available tool support for all its components. Finally, we also include an example to show how the framework works for designing, implementing, and optimizing a real application.

Chapter 9 concludes the thesis and discusses future work directions.

Appendix A presents additional work and results on the data intensive case-study presented in Chapter 5. Specifically, we discuss the design and implementation of the same application on two more hardware platforms, comparing all these with the original Cell/B.E. implementation. We compare the three platforms in terms of performance and programmability, and we recommend a few empirical guidelines for choosing the right platform for a data-intensive application.

Appendix B is a brief discussion on performance prediction for multi-cores. We briefly present the Pamela methodology [van93], which uses application and machine models to "reason" about the potential performance of parallel applications. Further, we present PAM-SoC, a Pamela-based tool for Multi-processor System-on-Chip (MPSoC) application performance prediction, focused on predicting application behavior *before* the application and/or the architecture are (fully) available. Finally, we discuss why such a tool is difficult enough to implement for multi-core processors that its feasibility is questionable.

Chapter 2

Multi-Core Architectures

This chapter presents an overview of the available multi-core processors, focusing on their architectural features.

Multi-core processors come in various shapes and sizes. As they are difficult to compare, there is little consensus on a potential best architecture. Rather, a few types of architectures tend to polarize the applications in just a few hot-spots. General purpose multi-cores cover the consumer market - for both workstations and servers. Typically, these processors have few identical powerful cores, shared memory, and a hierarchy of caches. They mainly focus on increasing the performance of workloads composed of multiple generic consumer applications.

Instead, embedded systems multi-processors (we include here network processors, digital signal processors, and other similar, application-specific processors) have a few different cores (typically, only some of them are programmable), little memory (either shared or distributed), and small caches. They focus on specific application fields (by design), and they are usually unsuitable for large, generic workloads.

With the development of General Processing on GPUs (GPGPU), GPUs claim a category for themselves. With special, massively parallel architectures (i.e., collections of hundreds of simple cores), state-of-the-art GPUs have passed the TFLOP barrier - for the applications that are able to use it.

High-performance computing (HPC) processors are platforms that offer hundreds to thousands of GFLOPS, and are used primarily for number crunching. Traditionally, experimental platforms that use non-traditional architectures to obtain high computational peaks become experimental HPC processors. Therefore, we hardly see any common architectural trends for HPC multi-cores - except for the continuous search for the highest absolute peak performance (counted in GFLOPs).

However, the borders between these categories are quite blurred: architectures like the STI Cell/B.E. [Kah05], NVIDIA's Fermi [NVI10], or IBM's POWER7 [IBM09] are hard to place in a single bin. However, as there is no clear, universally accepted taxonomy of multi-cores (nor a clear terminology, either), our classification is sufficient to structure our processor survey.

To provide a broad overview of the multi-core platforms, we discuss several examples of *representative* multi-core processors from each of these four categories. Our presentation targets mostly the architectures of these platforms, discussing similarities and differences as seen by both the hardware and the software designers. Hence our emphasis on the specific hardware characteristics, the performance potential, and the programming approach for each of these processors. For simplicity, we present the platforms in the chronological order of their appearance, and we only discuss platforms introduced after the year 2000.

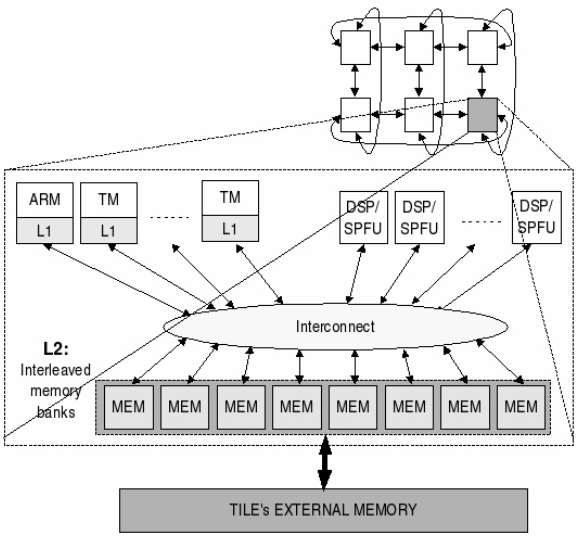


Figure 2.1: *The Wasabi architecture - one tile of the SpaceCAKE architecture.*

2.1 Wasabi/SpaceCAKE: An Embedded Multi-core Processor

Embedded systems have featured multiple cores per chip since microcontrollers have emerged, but those cores were hardly programmable, very rudimentary, and geared towards accelerating very simple operations directly in hardware. As the complexity of embedded-systems increased, the performance requirements for application-specific processors has also increased, and *multi-processor system-on-chips* (MPSoCs) or *chip multi-processors* (CMPs) have emerged as programmable multi-cores for embedded systems. Today, most embedded systems use such processors for a wide range of embedded applications (e.g., automotive systems, multimedia and digital signal processing, and medical devices).

Typical MPSoCs are heterogeneous platforms with multiple types of cores: one master core able to execute an operating system and I/O operations, a set of programmable cores dedicated to generic computations, and a set of non-programmable specialized functional units. In most cases, they are shared-memory machines, with simple memory hierarchies, used as stand-alone processors.

An example is the Wasabi/SpaceCAKE architecture, a tile-based embedded processor [Str01] designed in 2001 by Philips for its high-end multimedia systems. SpaceCAKE is a collection of homogeneous tiles, communicating via a high-speed interconnect. A SpaceCAKE tile, presented in Figure 2.1, is a typical MPSoC: it has one main core (an ARM or a MIPS), eight programmable multimedia processors (Philips Trimedia) [Bor05; Phi04], and several special functional units (non-programmable hardware accelerators). The memory hierarchy has three levels: (1) private L1's for each processor, (2) one shared on-chip L2 cache, available to all processors, and (3) one off-chip memory module. Hardware cache consistency between all L1's and L2 is enforced.

Wasabi/SpaceCAKE Programming[1]

For software support, Wasabi runs eCos [eCo], a modular open-source Real-Time Operating System (RTOS), which has embedded support for multithreading. Programming is done in C/C++, using eCos synchronization system calls and the eCos thread scheduler.

Despite the promising architecture, the processor has never been fully built. In our experiments (see more details in Appendix B) we have only used a cycle-accurate software simulator. However, the estimated performance of a single Wasabi chip, based on the specification of its cores, was around 10 GFLOPs (for all 9 cores combined).

Despite the platform being canceled in 2006, Wasabi/SpaceCAKE was a good experiment. Many of Wasabi's design ideas have been reproduced in later, more successful architectures. Also, the ambitious fully-coherent memory system generated a lot of interesting research on caches, cache partitioning [Mol09], and coherency.

2.2 General-purpose multi-core processors

Starting from 2004, General purpose Multi-cores (GPMCs) are replacing traditional CPUs in both personal computers and servers. Generically called "multi-cores", they are already offered by most of the big players - Intel, Sun, AMD, and IBM. GPMCs are homogeneous platforms with complex cores, based on traditional processor architectures; they are typically shared-memory architectures, with multiple layers of caches, and they are used as stand-alone processors.

In the following paragraphs, we discuss a few very common examples of general-purpose multi-cores, and we propose a brief comparison between their features and peak performance potential. Note that the time-frame in which these processors have emerged and developed is quite similar, so we just group them under the GPMCs label.

2.2.1 Intel Nehalem

Nehalem EE: Quad-core Intel Core i7

Nehalem EE, introduced in 2008, is a four-core processor based on the Intel Core i7 architecture; with Intel's hyperthreading technology [Int09], the processor can use eight hardware threads. Base clock speeds range between 2.66 GHz and 3.2 GHz, while turbo clock speeds range between 2.80 GHz and 3.46 GHz. Note that the turbo mode is supposed to be turned on automatically when more computation power is needed.

The peak performance of Nehalem EE is achieved by the Intel Core i7-975 Extreme Edition: in normal mode, it runs at 3.33 GHz, delivering 53.28 GFLOPS; in turbo mode, the same processor runs at 3.6GHz and it delivers 57.6 GFLOPS.

The on-chip memory hierarchy has three levels of caches: the L1, per core, separated for data and instructions (32 KB each), a low-latency L2, per core, 256 KB for both data and instructions, and a fully shared fully inclusive L3 cache, up to 8 MB. The chip supports 3 channels of external DDR3 memory, with interleaved access, and at most 8 GB per channel. The bandwidth per channel is 8.5 GB/s. The system bus, using Intel®Quick Path Interconnect (QPI), offers a peak bandwidth of 25.6 GB/s. A diagram of a system based on a Core i7-975 is presented in Figure 2.2.

[1] As Wasabi is the processor that has started this work, we decided to include it here as an interesting and original embedded multi-core platform.

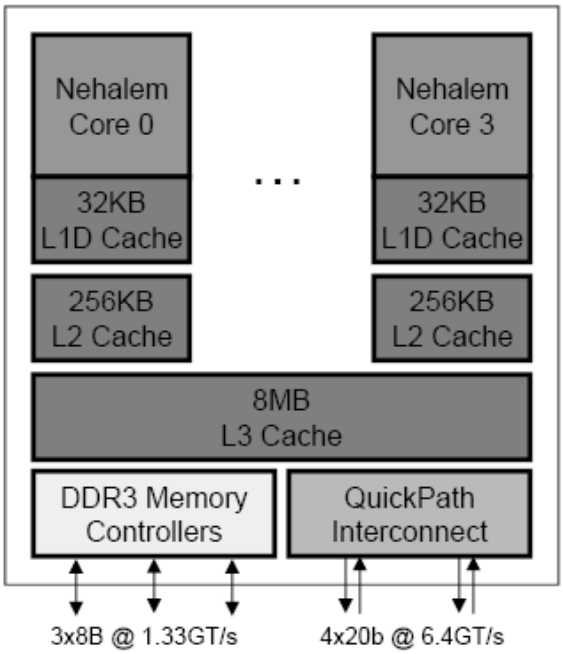


Figure 2.2: *Intel Core i7 (figure courtesy of RealWorld Technologies [Kan08]).*

Nehalem EX: Eight-core Intel Core i7 (Xeon 7500 series)

The Intel Nehalem EX is an eight-core processor (introduced in early 2010), using hyperthreading to raise the number of available hardware threads to sixteen. The chip supports the SSE instruction set, allowing operations on SIMD vectors of 4 single-precision floating point numbers. The cache hierarchy has the same three as Nehalem EE, but the fully shared fully inclusive L3 cache can be increased up to 24 MB. The chip supports 4 channels of external DDR3 memory.

Intel Nehalem Programming

For programming its multi-cores, Intel provides a set of highly optimized function libraries, like Intel®Integrated Performance Primitives (IPP) and Intel®Math Kernel Library (MKL), the Intel®Threading Building Blocks (TBB) C++ template library, and a generalized data parallel programming solution with the Ct programming model. Besides these home-brewed solutions, programmers may use traditional, low-level multi-threading primitives, or opt for higher-level generic solutions such as OpenMP and MPI.

2.2.2 AMD Barcelona, Istanbul, and Magny-Cours: N-Core Opteron Processors

The architecture of AMD multi-cores is similar to the one of Intel Core i7: homogeneous cores, per-core L1 and L2 caches, shared L3 cache and off-chip memory, and separated memory and I/O buses/channels.

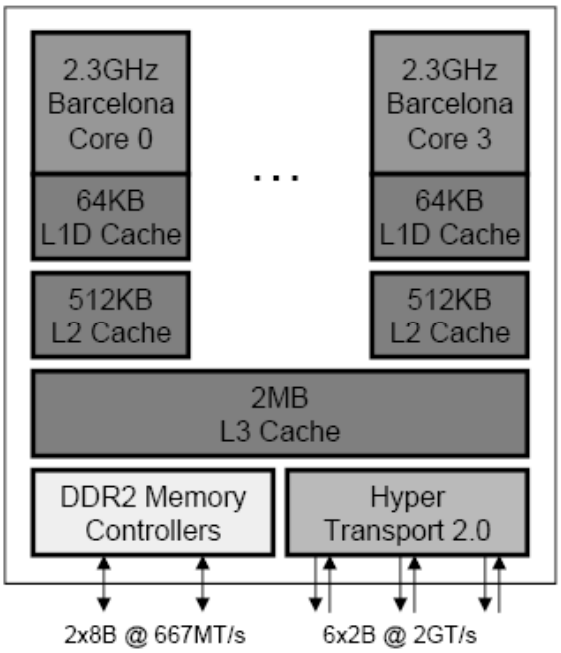


Figure 2.3: *A Multi-Core AMD Opteron Processor(figure courtesy of RealWorld Technologies [Kan08]).*

The first in the series, AMD Barcelona (late 2007), is a four-core processor able to reach a theoretical peak performance of 41.6 GFLOPS. The six-core version, code-named Istanbul (and launched in mid-2009), is clocked at 2.6GHz, delivering a theoretical peak performance of 62.4 GFLOPS. The latest generation, code-named Magny-Cours (launched in early 2010) has 12 cores (although on two separate dies, with very fast inter-die interconnect).

Each core has an L1 cache of 128 KB, half for data and half for instructions, and an L2 cache of 512 KB; all cores share a 6 MB L3 cache. The dual-channel main memory controller is clocked at 2.2GHz. The main memory (off-chip) supports up to DDR2-800 RAM, and can go up to 16 GB. The maximum bandwidth for main memory transfers is 19.2 GB/s. The system bus, using AMD®Hyper Transport 3 (HT3), runs at a maximum bandwidth of 19.2 GB/s (4.8 GT/s). A diagram of the AMD Barcelona is presented in Figure 2.3.

AMD Multi-core Programming

Just like Intel, AMD provides a highly optimized math library, the AMD®Core Math Library (ACML). To compete with Intel®IPP, AMD has invested in the open-source Frame wave project [AMD08], which offers code-level access to a vast array of signal and image processing functions and routines. The Framewave library is API compatible with IPP. Currently, for application development, programmers use low-level multi-threading primitives or higher-level generic solutions such as OpenMP and MPI. AMD is said to be in discussions with Microsoft for developing/supporting additional multi-core programming tools.

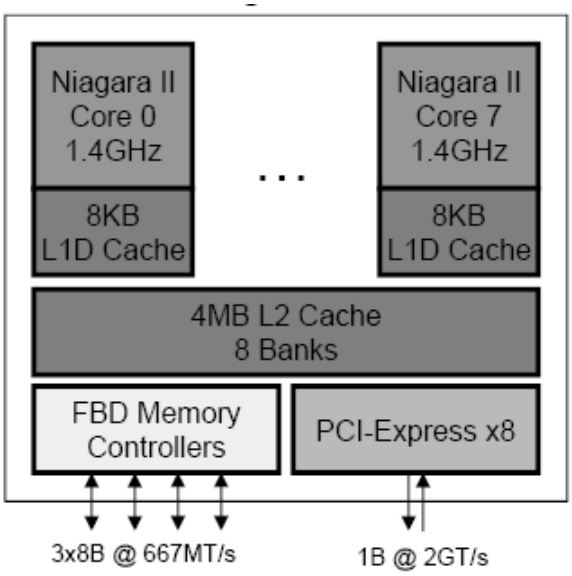


Figure 2.4: *Sun UltraSPARC T2 Plus (figure courtesy of RealWorld Technologies [Kan06]).*

2.2.3 Sun UltraSPARC T2 and T2+

Sun's top multi-core processors are visible in the servers product line. The Sun multi-core family is based on the UltraSPARC Tx processors, and it has three members: Niagara (Sun UltraSPARC T1, launched in 2006), Niagara II (Sun UltraSPARC T2, launched in 2008), and Victoria Falls (Sun UltraSPARC T2+, launched in early 2009). The architectural differences between these chips are not significant. Rather, most differences come from the "numbers", as later platforms have faster clocks, more threads, more memory, and increased numbers of multi-chip configurations.

Victoria Falls [Ora08] , the top of the line, is an Symmetrical Multiprocessor (SMP) architecture with 8 SPARC cores, each core handling 8 concurrent threads. Cores are clocked at 1.4 GHz (T2+) (compared to 1.2 GHz for T2). Consequently, the achieved performance is 11.2 GFLOPS (for T2, we have 9.33 GFLOPS).

The memory system has two levels of cache - an L1 (separated for data and instructions) per core, and a shared L2. The L1 instruction cache is 16 KB large (32 B lines, 8-way set associative), while the The L1 data cache is 8 KB large (16 B lines, 4-way set associative). The L2 is a 4MB, 8 banks, 16-way associative cache. Each core is bidirectionally connected to each L2 bank by a custom, full 8×9 cache crossbar switch. Thus, the interconnect supports 8 Byte writes from any core to any bank, and 16 Byte reads from any bank to any core. An additional port in the crossbar switch allows cores to perform I/O operations. All I/O operations are handled by a System Interface Unit (SIU), which connects networking and I/O to the memory. The crossbar bandwidth (i.e., the aggregated L2 memory bandwidth) reaches 90 GB/s for writes and 180 GB/s for reads per channel. To access the (off-chip) main memory, Victoria Falls uses two Fully Buffered DIMM (FB-DIMM) controllers per socket, with four DIMMs each. The aggregated memory bandwidth (for both sockets) is about GB/s. A diagram of the Sun UltraSPARC T2+ is presented in Figure 2.4.

Sun Niagara II Programming We note that the Sun Niagara II differs significantly from the other GPMC processors, because it is a scalar, in-order processor (with hardware threads) instead of a super-scalar out-of-order processor (with - Intel Nehalem - or without -AMD N-Core Opteron - hardware threads). The Niagara II focuses on high throughput instead of fast completion of tasks. The very high bandwidth, both internal and external, clearly indicates very good support for data intensive workloads, as the processor is dedicated to a large variety of servers (the "Cool Threads" family, by Sun).

In terms of programming, besides the classical OpenMP and MPI solutions, Sun offers the largest variety of tools, including application evaluation/analysis, porting, debugging, optimizing, and testing tools. Sun Studio 12 provides an integrated environment for compilers and tools that cover the application development phases [Sun08].

IBM POWER7

The latest IBM processor, the POWER7, announced in the summer of 2009, is also a large multi-core that focuses on the servers market. Clocked at 3.0 GHz to 4.14 GHz, a POWER7 chip has 4, 6, or 8 cores; each core is 4-ways SMT. Thus, a dual-chip module (available already for server machines) can offer up to 64 concurrent threads. The theoretical peak performance per core is estimated at 33.12 GFLOPS per core, allowing for a maximum of 264.96 GFLOPS per chip. For its memory system, POWER7 relies on 2 layers of private caches - 32 KB L1 instruction and data cache (per core) and 256 KB L2 Cache (per core), and 1 layer of shared cache - 32 MB on-die [2] , shared by all cores (at most 32/die). The chip also features two DDR3 memory controllers, reaching 100GB/s sustained memory bandwidth. In SMP mode, the POWER7 delivers 360GB/s bandwidth/chip.

IBM POWER7 Programming

Overall, POWER7 is a HPC processor aimed at supercomputers and/or high-end application servers. Being yet another symmetric multi-core architecture, POWER7 also allows programmers to use generic parallel programming models such as OpenMP or MPI. While IBM is not spending a lot of effort in the high-level design tool chains (it does, however, provide specially optimized compilers and profilers), there are "imported" solutions to be considered. For example, Partitioned Global Address Space (PGAS) languages like UPC [Car99; Zhe10] and Co-Array FORTRAN [Coa05; MC09], together with Charm++ [Kal96; Kal95b; Kal95a], are currently used for programming Blue Waters, the largest POWER7 supercomputer to date.

2.2.4 A Summary of GPMCs

A summary of the important features of the three general purpose multi-core processors we have discussed is presented in Table 2.2.4. Note that from the computer architecture point of view, the platforms are very similar. A more significant variety appears in the architectural parameters - number of cores and/or threads, clock frequency, memory bandwidth, etc.

In terms of programming, it is safe to say that all GPMC processors target similar applications, namely coarse grain MPMD and/or SPMD workloads. The parallelism model is symmetrical multi-threading (i.e., all the available hardware threads are similar). The mapping/scheduling of processes

[2]The cache is implemented in eDRAM, which does not require as many transistors per cell as a standard SRAM so it allows for a larger cache while using the same area as SRAM.

Platform	Intel Nehalem EE	AMD Istanbul	Sun Niagara	IBM POWER7
Cores				
Number	4	6	8	8
Type	x86	x86	UltraSPARC	PowerPC
SMT	2	-	8	4
Perf[GFLOPS]	57.6	62.4	11.2	264.96
Caches				
L1 (I+D)	private, 32KB+32KB	private, 64KB+64KB	private, 16KB+8KB	private, 32KB+32KB
L2	private, 512KB	private, 512KB	shared, 4MB	private, 256KB
L3	shared, AMBY	shared, AMBY	-	shared, 32MB
Main Memory				
Organization	triple-channel, DDR3	double-channel, DDR2	4 x dual FB-DIMM	triple-channel, DDR3
Size	up to 24GB	up to 16GB	up to 32GB	up to 32GB
Bandwidth	3 x 8.5GB/s	2 x 12.8GB/s	2 x 38GB/s	100GB/s
System Bus				
Type	QPI	HT3	SIU	?
Bandwidth	25.6GB/s	19.2GB/s	-	?

Table 2.1: *A summary of architectural features for general purpose multi-core processors*

on threads is performed by the operating system. Users may influence the process by circumventing/-modifying the running operating system and/or its policies.

2.3 General Processing on GPUs (GPGPU) Computing

Graphical Processing Units (GPUs) have emerged as multi-core processors around 2003, when the General Processing on GPUs (GPGPU) became an interesting branch of computer science (an offspring of graphics, at the time). In the meantime, GPUs have increased their computation abilities by several factors. Nowadays, novel GPU architectures target HPC markets directly, by adding computation-only features next to their graphics pipelines.

GPUs are close to the many-core boundary, as models with 30+ homogeneous programmable cores are already available. They are mostly shared memory machines, with a complex memory hierarchy, combining different types of caches and scratch-pads at each level. GPUs are always used as accelerators, which requires very low flexibility in the hardware; in turn, this allows for architectures that provide high memory throughput and computation capabilities.

In the following paragraphs, we briefly present three different families of GPU architectures: NVIDIA's (with G80, GT200, and GF100), AMD/ATI (with RV7** and Radeon HD5870(Cypress)), and Intel's Larrabee.

2.3.1 NVIDIA G80/GT200

Much like the G80 processor, GT200 (launched in 2008) is a 2-layers hierarchical multi-core focused on high throughput computing for data parallel applications. Programming is based on a direct mapping of the architecture into an Single Instruction Multiple Threads (SIMT)-based parallel programming model, called CUDA. Essentially, an application is split into highly data-parallel regions (called *kernels*) which are sent to be executed by grids of thread blocks. A thread block has up to 512 threads that start at the same address, execute in parallel, and can communicate through shared memory. All blocks in a grid are unordered and can execute in parallel; how many of them will actually execute in parallel depends on the hardware capabilities.

The GT200 architecture uses 10 Thread Processing Clusters (TPCs), each containing 3 Streaming Multiprocessors (SMs) (also known as Thread Processor Arrayss (TPAs)) and a texture pipeline, which serves as a memory pipeline for the SMs triplet. Each SM has 8 Streaming Processors (SPs), and it can be seen as the equivalent of a 8-way SIMD core in a modern microprocessor. Thread blocks are issued on the same SM, and each thread is scheduled on a single SP.

Note that the main differences when compared with the G80 processor (the previous generation), are in the number of SMs (8×2 for G80 vs. 10×3 for GT200), the memory subsystem (wider and higher bandwidth memory in GT200), and the memory controller coalescing (GT200 has less strict requirements on memory access patterns than G80).

The GT200 (same as G80) memory system has three logical layers (mapped on two physical memory spaces), visible both in the hardware architecture and inside CUDA. Each thread has a private register file and a private local memory. For GT200, the register file is 64KB (twice as much as G80). Each SM has 16KB register entries, partitioned among the SPs. Thus, each SP has 2K entries in the register file dynamically partitioned by the driver to be used by up to 128 threads.

The 16KB of shared memory, local to each SM is shared by the entire block of threads (up to 512). Organized in 16 banks to provide parallel access to threads inside a warp, the shared memory is very fast (comparable with the register file) unless bank conflicts occur. Shared memory is dynamically partitioned among different thread blocks.

The data that does not fit in the register file, stored on-chip, spills in the local memory, stored in the DRAM space (and having a worse access time compared to the on-chip register file). Also in the DRAM space, there are two read-only memories - a (small) constant memory (64KB) and a texture memory. As they are both read-only, these memories are cached in the shared memory. For the larger texture memory, there are two caching levels: 24KB L1 cache per TPC (organized as 3×8 KB, or 8 KB/SM), and 32KB L2 cache located with each of the 8 memory controllers (256 KB L2 texture cache in total).

Finally, the global memory (the main DRAM space) is a normal system memory, randomly read-/written by both the CPU and the GPU, it is not cached, and it is accessible to a whole grid of threads. Note that depending on the chosen product, the global memory (known also as device memory) ranges between 1 GB and 4×4 GB.

The diagrams of the two processors, G80 and GT200, are presented in Figures 2.5 and 2.6, respectively.

Each of these architectures features in a wide range of products. A GeFORCE GTX280 (a GT200 consumer card, rather targeted to gamers than programmers), has 1 GB of global memory, accessible at 141.7 GB/s. The raw peak performance for single precision workloads is 933.1 GFLOPs (considering dual-issuing), while the double precision performance is only 77.8 GFLOPs. For the Tesla C1060, a GT-200 board dedicated to high-performance computing, the device memory is 4 GB, accessible at 102.4 GB/s (lower memory frequency than the GeFORCE), and averaging about the same performance (936 SP GFLOPs and 78 DP GFLOPs).

NVIDIA GF100 (Fermi)

The GF100 (launched at the end of 2009) is even more compute-friendly than its predecessors. Fermi is a 2-layer hierarchical many-core focused on high throughput computing for data parallel applications. The chip features 16 SMs (Streaming Multiprocessors), where each SM has 32 cores. So, in total, the GF100 has 512 CUDA cores, where a core executes a floating point or integer instruction per clock. Since a multiply-add instruction is supported, a core can perform 2 FLOPs per clock. Running

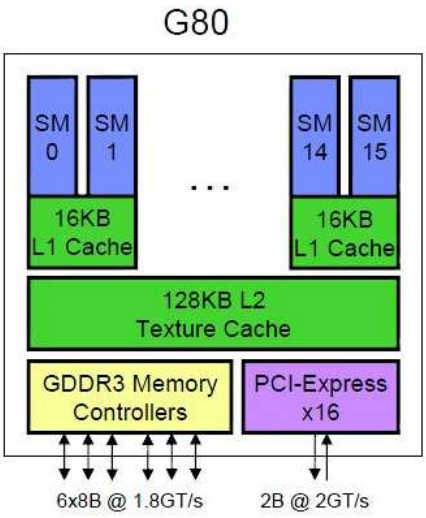


Figure 2.5: *The NVIDIA G80 architecture (image courtesy of RealWorld Technology [Kan09]).*

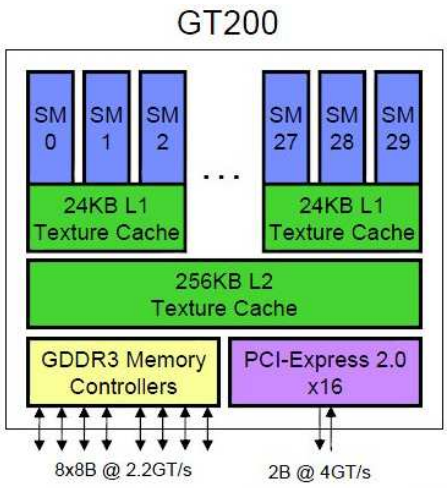


Figure 2.6: *The NVIDIA GT200 architecture (image courtesy of RealWorld Technology [Kan09]).*

applications issue thread blocks; the hardware scheduler distributes blocks on the SMs, and, threads on the cores.

The GF100 has a complex memory architecture. The host system's memory cannot be directly read or written. Instead, data is transferred over the PCI-e bus, using asynchronous DMA. The so called device memory on the GPU is accessed through a 384-bit wide GDDR5 memory interface. The memory system supports up to 6 GB of memory. In addition, the GF100 has 64 KB of fast local memory associated with each SM, which can be used either as a 48 KB cache plus 16 KB of shared memory, or as a 16 KB cache plus 48 KB of shared memory. Finally, there is a 768 KB L2 cache, which is shared by all 16 SMs.

Each SM has 32768 registers. Therefore, there are 1024 registers per core. Within a core, the registers are dynamically partitioned between the threads that are running on that core. Therefore, there is a trade-off between using more registers per thread, or more threads with less local data stored in registers. Fermi is the first GPU to provide ECC (error correcting code) memory. The register files, shared memories, L1 and L2 caches are also ECC protected. A diagram of Fermi's architecture is presented in Figure 2.7.

NVIDIA GPUs Programming

NVIDIA GPUs are typically used for highly data-parallel workloads, where hundreds to thousands of threads can compute concurrently. The most common parallelism models are SIMD/SIMT, with medium granularity - due to the time-consuming offloading of the kernels from the host (CPU) to the GPU, too low-granularity kernels are not suitable for these architectures. Kernel and thread scheduling are done in the hardware.

As mentioned earlier, programming NVIDIA GPUs relies heavily on the CUDA model. CUDA C, the API released by NVIDIA (similar with CUDA FORTRAN, released together with the Portland Group) is a variant of C that includes CUDA-specific extensions for separating device (i.e., GPU) from host (i.e., CPU) code and data, as well as for launching CUDA kernels with correct/suitable grid configurations. While considered a fairly simple programming model, CUDA C is still a low-level tool,

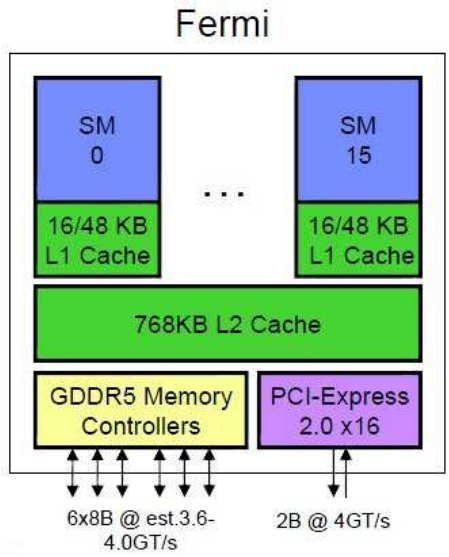


Figure 2.7: *The NVIDIA Fermi architecture (image courtesy of RealWorld Technology [Kan09]).*

and requires a lot of programmer's insight and experience to claim impressive performance results. An alternative to CUDA C is OpenCL, a new standard for multi-core programming. When it comes to NVIDIA GPUs, OpenCL is quite close to the CUDA approach. Thus, it does not offer higher-level abstractions, but it offers code portability. For typical scientific workloads, there are highly optimized kernel libraries available, especially for signal processing or linear algebra kernels. Higher level programming models, novel or adapted, are emerging as well - see RapidMind, PGI Accelerator, or Mars (based on MapReduce).

AMD/ATI RV770 and RV790

The AMD/ATI RV770 GPU (released in 2008 as part of the Radeon 4000 series) has 10 SIMD cores, each with 16 streaming processor units. The SIMD cores are similar to NVIDIA TPC units and the 160 SPs (10×16) are similar to the NVIDIA SP units (GT200 has 240 of these). Each SP has 5 ALUs - 4 symmetrical and a 5th running also as a special function unit [3] , a branch unit, and a 16KB set shared of general purpose registers. Further, each SIMD has an associated Texture Mapping Unit (TMU), with 4 dedicated texture units and an L1 texture cache; the SIMD and its TMU can communicate data through a Local Data Share. All SIMD cores can exchange data using 16 KB of on-chip Global Data Share.

The memory hierarchy is quite complex. Besides the local SP memories (16KB general purpose registers), the SIMD local and global data shares, the chip has a set of caching units: the L2 caches are tied to the four 64-bit memory channels, while the L1 (texture) caches store data for each SIMD; the L1 bandwidth is 480 GB/s, and the L1-to-L2 bandwidth is 384 GB/s. The off-chip memory depends on the board that hosts the GPU, ranging between 512 MB GDDR3 and 2 GB GDDR5.

[3]Note that when RV770 is mentioned to have 800 "shader cores" ("threads" or "streaming processors"), these cores are in fact the 800 ALUs.

The RV770 is used in the desktop Radeon 4850 and 4870 video cards, and evidently the "workstation" FireStream 9250 and FirePro V8700. Radeon 4870, clocked at reference 750MHz, can reach 1.2 TFLOPs single-precision. For double-precision workloads, performance drops to 240 GFLOP/s (plus, if needed, 240 GFLOP/s single precision on the 160 SFUs). The off-chip main memory (ranging between 512 MB and 2 GB) is clocked at 900 MHz/s, which allows for a peak bandwidth of 115GB/s for the 4 channel GDDR5 (or 64GB/s for the 2 channel GDDR3).

AMD/ATI Radeon HD 5870 (Cypress)

Like NVIDIA's GF100 (its direct competitor) the AMD Radeon 5870 is a hierarchical multi-core, aiming at both high-end graphics card market, as well as at general purpose processing. The Cypress has 20 SIMD cores, each of which has 16 thread processors. Each of those thread processors has five ALUs (arithmetic logic units). So, in total, a single Cypress chip has 1600 ALUs, significantly more than NVIDIA's hardware.

Each SIMD core has 8 KB of L1 cache (the chip thus has 160 KB L1 in total), and 32 KB local data store, which can be used for inter-thread communication. All SMs can in turn exchange data using 64 KB of on-chip Global Data Share. In addition, the chip has 512 KB of L2 cache. The device memory is based on GDDR5, with a 256-bit memory bus. Similarly to NVIDIA's hardware, there is a hardware thread scheduler that schedules the threads on the different SIMD cores and thread processors.

ATI/AMD GPUs Programming

In terms of workloads, ATI GPUs are targeting similar applications as NVIDIA's processors: highly data-parallel applications, with medium granularity. Therefore, choosing between the two becomes a matter of performance and ease of programming.

Programming the ATI GPUs was originally based on the Brook+ streaming language. Currently, Brook is one of the components of ATI Stream SDK [AMD09], together with several higher-level performance libraries and tools, and the device drivers for the stream processors. OpenCL is also available for the ATI GPUs.

2.3.2 Intel's Larrabee

No GPU made more waves than the Intel Larrabee, rumored as the first architecture with 32 or even 48 large cores (x86 compatible), and proved (in November 2009) able to cross the 1 TFLOP barrier on its own. Still, in December 2009, the project was partially canceled (no consumer boards will be available), and partially postponed (research platforms for High Performance Computing (HPC) and graphics will be made available late 2010).

Briefly, Intel's intentions were to build a fully-programmable high-performance many-core graphics processor based on the famous x86 instruction set and architecture, including hardware cache coherency, but also featuring very wide SIMD vector units and texture sampling hardware. The expected peak performance was calculated at 2 TFLOPs.

Each of Larrabee's 32 cores [Sei08] is a 512 bits wide vector unit, able to process 16 single precision floating point numbers at a time. Each core supports 4-way interleaved multithreading. The cores are interconnected and communicate with the memories through a 1024-bit wide ring. Cache coherency among cores is enabled in hardware, but direct cache control instructions for minimizing cache trashing

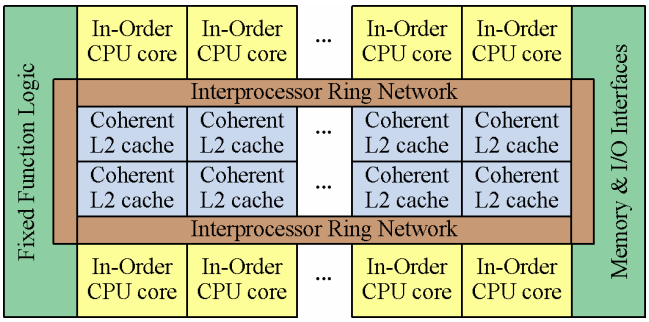


Figure 2.8: *A draft diagram of the Intel Larrabee chip (image courtesy of Intel [Sei08]).*

and prefetching data into L1 and L2 are also provided. A draft diagram (courtesy of Intel [Sei08]) is presented in Figure 2.8.

Larrabee has publicly targeted both high-end graphics applications (games) and high-performance computing. Some believe that the performance delivered by the latest prototypes was disappointing for both, and that has lead to a "stop-and-think or retarget" approach from Intel.

Larrabee Spinoffs

Intel MIC The Knights Ferry [Wol10], also known as Intel MIC (launched in 2010 for research projects only), presented in Figure 2.9) has 32 x86 cores on chip, each with 32KB L1 instruction cache, 32KB L1 data cache, and 256KB L2 cache. Each core has a vector unit, essentially a very wide (512 bits or 16 floats) SSE unit, allowing 16 single precision floating point operations in a single instruction. Double-precision compute throughput is half that of single-precision. The 32 data caches are kept coherent by a ring network, which is also the connection to the on-chip memory interface(s). Each processor supports a multithreading depth of four, enough to keep the processor busy while filling an L1 cache miss. The Knights Ferry is implemented on a PCI card, and has its own memory, connected to the host memory through PCI DMA operations. This interface may change in future editions, but Intel advertises the MIC as "an Intel Co-Processor Architecture". This could be taken as acknowledgement that accelerators can play a legitimate role in the high performance market.

Intel MIC uses a classical two-level cache per core, and has a separate memory from the host and functions as attached processor. However, it is expected that the x86 cores support full virtual memory translation; as the cores run a reasonably complete microkernel operating system, the support software allows a shared address space across the host and the accelerator. The real advantage of this approach is that the same address space can be used on the MIC and on the host. For the rest, the programmer still has to specify when to move what data to which memory, so as to amortize the data movement latency.

MIC programming is based on multi-threading with classical software thread creation, scheduling, and synchronization. Threads are coarse grain and have a long lifetime. Thread creation is relatively inexpensive, but it is likely that parking and recycling threads will be much more efficient than recreating threads each time they are needed. Scheduling is done by the operating system, but a lightweight task scheduling API, such as was provided for Larrabee, will allow user tuning. Synchronization is done in software using memory semaphores, depending on the hardware cache coherence ring.

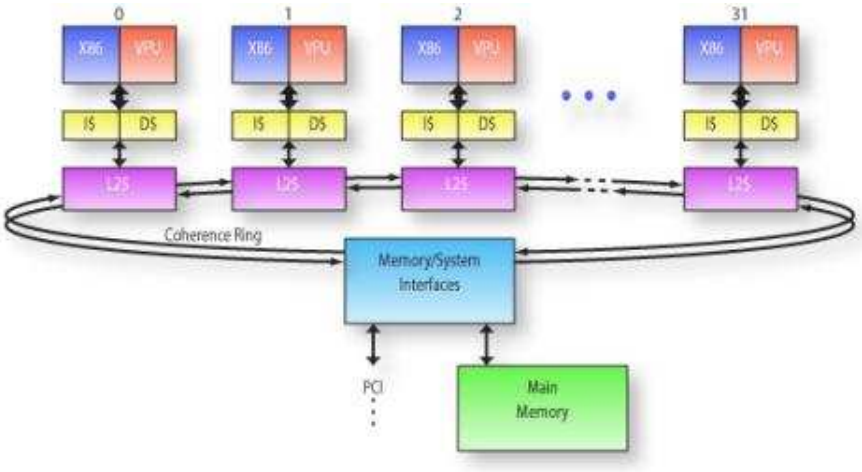


Figure 2.9: *A draft diagram of the Intel Knights Ferry (MIC) chip (image courtesy of HPC Wire [Wol10]).*

The Intel MIC will be programmed using native C/C++ compilers from Intel, and presumably from other sources as well. If the program is already parallelized with threads and the compiler vectorizes the code successfully, then the program may be ported with nothing more than a recompile. Getting high performance requires exploiting the two levels of parallelism - multi-core (MPMD) and wide SSE (SIMD), optimizing for memory strides and locality, and minimizing data communication between the host and MIC coprocessor. First performance results on a MIC/Larrabee-like processor have already been presented for sorting benchmarks [Wol10].

2.4 Architectures for HPC

We include here architectures that are used to increase application performance (thus, higher performance computing), and do not fit in any of the above categories. The ones we focus on are (1) the Cell Broadband Engine (Cell/B.E.) and (2) Intel Single-Chip Cloud Computer (SCC). Note that we do not claim that these processors are only used for traditional HPC workloads; instead, we claim that all workloads that use these multi-cores aim for higher performance.

2.4.1 The Cell/B.E.

If there is any processor to be credited with starting the "multi-core revolution", the Cell/B.E. must be the one. Originally designed by the STI consortium - Sony, IBM and Toshiba - for the PlayStation 3 (PS3) game console, Cell/B.E. was launched in early 2005 and quickly became a target platform for a multitude of HPC applications. Still, being a hybrid processor with a very transparent programming model, in which a lot of architecture-related optimizations require programmer intervention, Cell/B.E. is also the processor that exposed the multi-core programmability gap.

A block diagram of the Cell processor is presented in Figure 2.10. The processor has nine cores: one Power Processing Element (PPE), acting as a main processor, and eight Synergistic Processing Elements (SPEs), acting as computation-oriented co-processors. For the original Cell (the variant from 2005), the theoretical peak performance is 230 single precision GFLOPS [Kis06a] (25.6 GFLOPS per

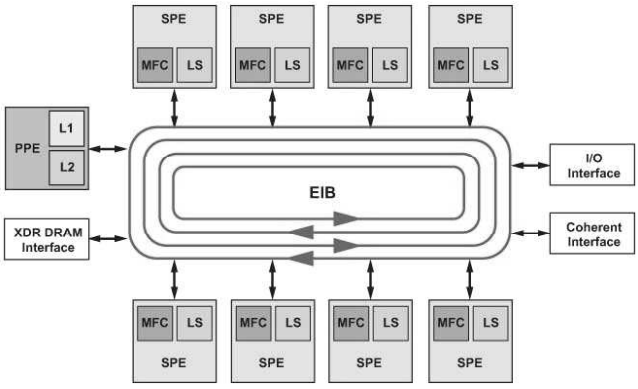


Figure 2.10: *The Cell Broadband Engine architecture.*

each SPE and for the PPE) and 20.8 double precision GFLOPS (1.8 GFLOPS per SPE, 6.4 GFLOPS per PPE). For the latest Cell version, named PowerXCell 8i, the double precision performance has been increased to 102.4 GFLOPS. All cores, the main memory, and the external I/O are connected by a high-bandwidth Element Interconnection Bus (EIB). The maximum data bandwidth of the EIB is 204.8 GB/s.

The PPE contains the Power Processing Unit (PPU), a 64-bit PowerPC core with a VMX/AltiVec unit, separated L1 caches (32KB for data and 32KB for instructions), and 512KB of L2 cache. The PPE's main role is to run the operating system and to coordinate the SPEs.

Each SPE contains a RISC-core (the Synergistic Processing Unit (SPU)), a 256KB Local Storage (LS), and a Memory Flow Controller (MFC). The LS is used as local memory for both code and data and is managed *entirely* by the application. The MFC contains separate modules for DMA, memory management, bus interfacing, and synchronization with other cores. All SPU instructions are 128-bit SIMD instructions, and all 128 registers are 128-bit wide. The integer and single precision floating point operations are issued at a rate of 8, 16, or 32 operations per cycle for 32-bit, 16-bit and 8-bit numbers respectively. The double precision 64-bit floating point operations, however, are issued at the lower rate of two double-precision operations every seven SPU clock cycles.

Cell/B.E. Spinoffs

Toshiba SpursEngine SE1000 is a trimmed Cell/B.E. architecture, released by Toshiba in 2008, and dedicated to streaming and multi-media encoding/decoding. As seen in Figure 2.11, the processor features only four SPEs, replacing the rest with hardware encoding and decoding units: two for H264 and two for MPEG-2.

The chip clock is lowered to 1.5 GHz. The maximum performance of the SE1000 (the current generation) is 48 GFLOPS, and the on-board memory is 128 MB, accessible at a peak bandwidth of 12.8 GB/s.

IBM RoadRunner is a supercomputer installed at Los Alamos National Labs, located in New Mexico, USA. Currently (November 2010), it is the 7th fastest supercomputer in the world, according to Top500[4] (RoadRunner did hold the top spot for about one year, being overpowered in the Fall

[4]Top500 hosts the list of the most powerful supercomputers in the world. The latest edition of this list can be found at `http://www.top500.org/`

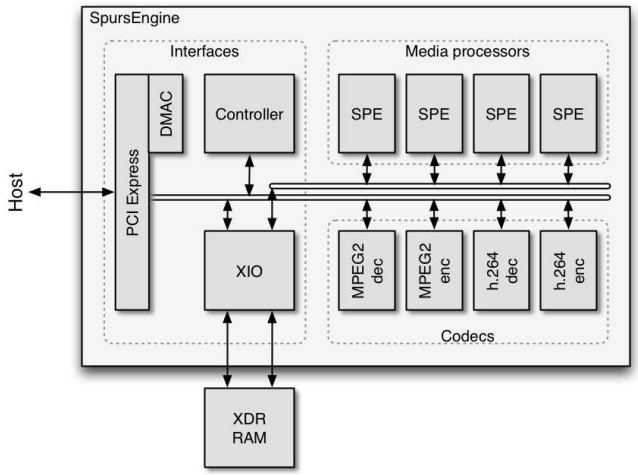


Figure 2.11: *The SpursEngine, a Cell/B.E. based streaming accelerator.*

of 2009 by Jaguar, the current number 2). The machine has a peak performance estimated at 1.7 PFLOPS, and it has been the first Petaflop system for LINPACK (i.e., it was able to sustain a rate of over 1 PFLOP for LINPACK operations). In November 2008, it reached a top performance of 1.456 PFLOPS. It is also the fourth-most energy-efficient supercomputer in the world on the Supermicro Green500 list, with an operational rate of 444.94 MFLOPS per watt of power used [Wik08].

IBM RoadRunner is a heterogeneous system, using two completely different processors: 6912 AMD dual-core Opteron processors (a total of 13824 general purpose cores) and 12960 IBM PowerXCell 8i processors (a total of 116,640 cores - 12,960 PPE cores and 103,680 SPE cores).

Cell/B.E. Programming

The Cell/B.E. cores combine functionality to execute a large spectrum of applications, ranging from scientific kernels [Wil06] to image processing applications [Ben06] and games [D'A05]. The basic Cell/B.E. programming uses a simple multi-threading model: the PPE spawns threads that execute asynchronously on SPEss, until interaction and/or synchronization is required. The SPEs can communicate with the PPE using simple mechanisms like signals and mailboxes for small amounts of data, or using DMA transfers via the main memory for larger data. In this model, all data distribution, task scheduling, communication, and processing optimizations are performed "manually" by the user (i.e., they are not automated). As a result, programming the Cell/B.E., and especially optimizing the code for Cell/B.E., are notoriously difficult.

2.4.2 Intel's Single-Chip Cloud Computer (SCC)

Intel SCC [Bar10; Int10a; How10] is one of the products of Intel's Tera-Scale research. A 48-core chip (launched in late 2009), SCC is currently a research tool for multi-core architecture, networks on chips, and software.

SCC is a 6×4 2D-mesh network of tiled core clusters with high-speed I/Os on the periphery. Each tile has two enhanced IA-32 cores, a router, and a Message Passsing Buffer (MPB). Off-chip memory accesses are distributed over four on-die DDR3 controllers for an aggregate peak memory bandwidth of 21GB/s at 4x burst. The controllers feature support for DDR3-800, 1066 and 1333 speed grades.

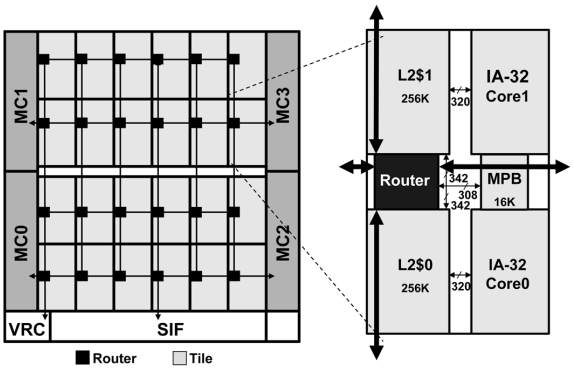


Figure 2.12: *Intel Single-Chip Cloud Computer: the chip and the tile structure (image courtesy of Microprocessor Report [Bar10]).*

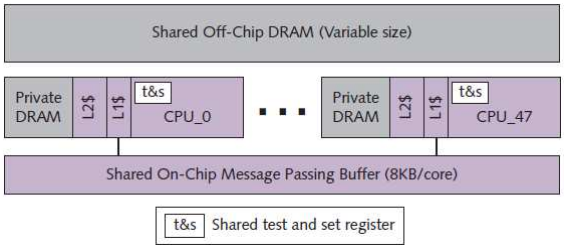


Figure 2.13: *The memory hierarchy as seen by cores(image courtesy of Microprocessor Report [Bar10]).*

For example, by supporting dual rank and two DIMMs per channel, a system memory of 64GB can be realized using 8GB DIMMs. I/O transactions are handled by an 8-byte interface. This interface is double-pumped and can sustain 6.4GB/s bandwidth at 200MHz. A system-level translation circuit converts the interface into the PCIe protocol and communicates over a 4x PCIe link. The platform allows for fine-grain power management, with controllers on all cores and memories. A diagram of the architecture is presented in Figure 2.12.

Cores operate in-order and are two-way superscalar. Each core has its own 32KB 1-cycle-access L1 cache, equally divided between instructions and data. Further, each core has its own 256KB 4-way write-back L2 unified cache. The L2 uses a 32-byte line size, matching the cache line size internal to the core, and has a 10-cycle hit latency; further, it supports several programmable sleep modes for power reduction. The memory hierarchy, as seen by cores, is presented in Figure 2.13. It includes, in addition to the usual L1 and L2 caches, private DRAM, access to shared off-chip DRAM, and a shared on-chip message-passing buffer supporting software cache coherency among cores. Shared areas are protected by test-and-set registers

As mentioned, each core is optimized to support a message-passing-programming model, with cores communicating through shared memory. The 16KB message-passing buffer (MPB) is present in every tile, giving a total of 384KB on-die shared memory. Shared memory coherency is maintained through software protocols, aiming to eliminate the communication and hardware overhead required for a memory coherent 2D-mesh. Messages passed through the MPB see a 15x latency improvement

over messages sent through DDR3.

Inter-core communication is done by the mesh of routers. Each 5-port virtual cut-through router used to create the 2D-mesh network employs a credit-based flow-control protocol. Router ports are packet-switched, have 16B data links, and can operate at 2GHz. Each input port has five 24-entry queues, a route pre-computation unit, and a virtual-channel (VC) allocator. Route pre-computation for the outport of the next router is done on queued packets. An XY dimension ordered routing algorithm is strictly followed. Deadlock free routing is maintained by allocating 8 virtual channels (VCs) between 2 message classes on all outgoing packets. VC0 through VC5 are kept in a free pool, while VC6 and VC7 are reserved for request classes and response classes, respectively. Input port and output port arbitrations are done concurrently using a wrapped wave front arbiter. Crossbar switch allocation is done in a single clock cycle on a packet granularity. No-load router latency is 4 clock cycles, including link traversal. Individual routers offer 64GB/s interconnect bandwidth, enabling the total network to support 256GB/s of bisection bandwidth.

SCC Programming

All 48 IA-cores of SCC boot Linux simultaneously. A software library, similar to MPI, is written to take advantage of both the message-passing features and the power optimizations. An application is loaded on one or more cores and the program runs with operating system support, in a cluster-like fashion.

In the normal usage model, the SCC board is connected to a Management Console PC (MCPC). The MCPC runs some version of an opensource Linux and Intel-provided software to configure the SCC platform, compile the application(s), and load them on the SCC cores. The application I/O is done through the MCPC [Int10b]. This hints to an accelerator model, where computation is offloaded to the SCC from the "host"; however, the host does not seem to participate in the computation.

We reiterate that SCC is a very new chip (released for software prototyping in early 2010), and it is promoted as a research platform. Therefore, Intel does not impose a best programming model for it; rather, it recommends using the proprietary RCCE message passing model [Mat10], but the alternatives and/or performance tradeoffs are yet to be determined.

2.5 Comparing Multi-Core Processors

2.5.1 Architectural characteristics

The architectural details used to characterize (and, implicitly, differentiate) multi-core architectures refer to three subsystems: the cores, the memory system, and the interconnections. Let's take a look at the different variants and/or metrics for each of these subsystems:

Cores

A platform is *homogeneous* or *symmetrical* if all its cores are the same; a *heterogeneous/asymmetric* platform has at least two different types of cores. For each core, the main characteristic is performance, typically quoted as two different values: one for single precision and one for double precision GFLOPs. Additional metrics are: Simultaneous Multithreading (SMT) count, i.e., specifies the number of hardware threads supported by a core, and the vector size, i.e., the number of words that are processed in parallel by the core functional units.

Memory

Multi-core processors have complex memory hierarchies, including at least three layers: per-core memory, per-chip memory, and system memory. Each one of these memories is characterized by several metrics: latency, bandwidth, capacity, granularity, RD/WR access, banking, and ownership. *Memory latency* is the time taken to from a memory request to return a result. Note that depending on how memories and memory controllers are built, access patterns can have a huge impact on latency: unfavorable access patterns can decrease memory performance by a couple of orders of magnitude. *Memory bandwidth* is the rate at which data can be read from or stored into memory. *Capacity* is the size of the memory.

With *granularity* we measure the smallest data unit that can be read/written into the memory in a single operation. The RD/WR access is more of a descriptive characteristic, which specifies if memory operations are explicit or implicit. For example, most caches have implicit RD/WR access, as the user does not initiate a cache RD/WR operation; by contrast, a scratch pad memory is explicit, as the user requests a RD or WR operation at a given address.

Banking specifies the number of banks of the targeted memory. We assume that a memory organized in multiple banks will always provide interleaved access (i.e., consecutive data words are in adjacent banks, such that data locality does not generate bank conflicts).

Ownership specifies which cores own and/or share a memory module.

Interconnections

We include in the interconnection system of a multi-core processor all the mechanisms that facilitate the communication between cores and/or memories, both on- and off-chip. An interconnection mechanism is characterized by: latency, throughput, width, and users. The interconnect *latency* is the time taken by a message "routed" by the interconnect to travel between two entities. For some interconnects (bus-based or NoC-based, for example), latency may vary with the position of the communicating entities, while for others, (crossbar-based, for example) it is fixed. Both types of interconnects are found in today's multi-core processors. The *interconnection throughput* is the total amount of data that can theoretically be transferred on the bus in a given unit of time. The number of simultaneous active transfers that can be facilitated by an interconnection mechanisms is the *width of the interconnect*. Finally, *the interconnect users* are all devices that are connected and can use the device as a communication mechanism - typically, memories or memory banks, cores, and special function units.

2.5.2 A quantitative comparison

We finalize the brief descriptions of some of the most common multi-core processors by a comparative analysis, presented in Tables 2.2 and 2.3 While it is practically impossible to choose a single best multi-core architecture, we make a few comments and recommendations based on the data presented in the table.

Table 2.2 focuses on the computing properties of the multi-core platforms, i.e. how is the parallelism achieved. The increase in the number of parallelism levels is a visible trend. Programming models can handle this explicitly or implicitly, trading performance for programmability. In Table 2.3, we summarize the memory properties of the same multicore processors. The memory subsystems of many-cores are increasing in complexity. This happens because they have to compensate for the inherent decrease in memory bandwidth *per core* with the increase in the number of cores and ALUs. More and more complex memory and caching hierarchies are needed to compensate for this problem. One

Table 2.2: *Computing properties of multi-core hardware.*

Platform	Cores / threads	Vectors	ALUs	Types	Scheduling	Par levels
Intel Nehalem EE	4 / 8	4-wide	64	HO	OS	2
Intel Nehalem EX	8 / 16	4-wide	64	HO	OS	2
AMD Istanbul	4 / 4	4-wide	48	HO	OS	2
AMD Barcelona	6 / 6	4-wide	48	HO	OS	2
AMD Magny Cours	12 / 12	4-wide	48	HO	OS	2
IBM Power 7	8 / 64	4-wide	256	HO	OS	2
Sun Niagara II	8 / 64	no	64	HO	OS	1
NVIDIA G80	(8x2) x 8	no	128	HO + host	Hardware	3
NVIDIA GT200	(10x3) x 8	no	240	HO + host	Hardware	3
NVIDIA GF100	(16x32) x 8	no	512	HO + host	Hardware	3
ATI HD4890	(10x16) x 5	4-wide	800	HO + host	Hardware	4
ATI HD5870	(20x16) x 5	4-wide	1600	HO + host	Hardware	4
Cell/B.E.	9 / 10	4-wide	36	HE	User/App	5

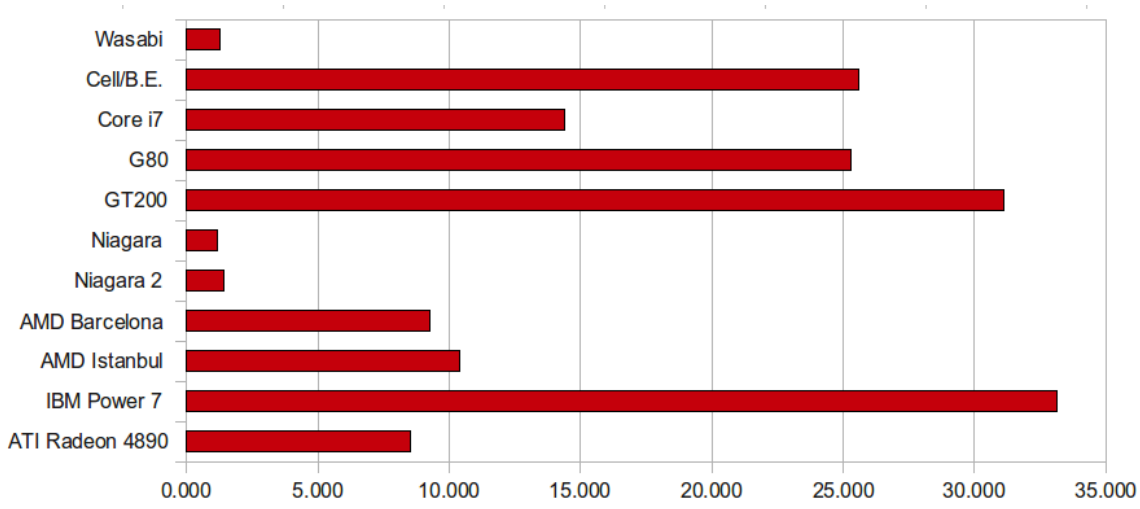
of the key differences between multi-core CPUs on the one hand, and the GPUs and the Cell/B.E. on the other, is that the memory hierarchy is more exposed, and often explicitly handled in the latter. This has a clear impact on the programming effort that is needed to achieve good performance.

Table 2.3: *Memory properties of multi-core hardware.*

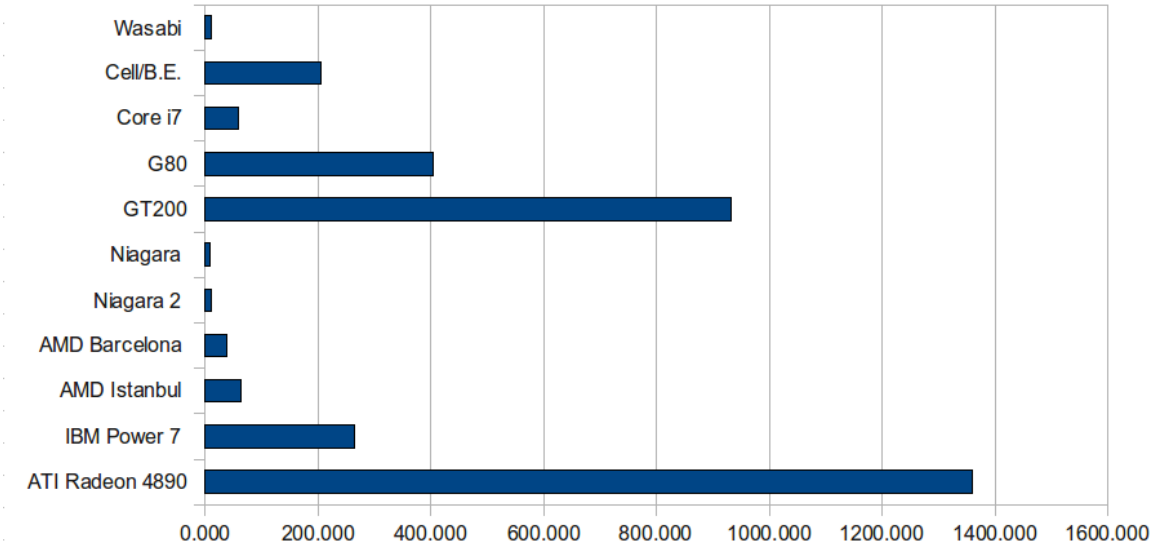
Platform	Space(s)	Access	Cache
Intel Nehalem EE	shared	R/W	transparent L1-3
Intel Nehalem EX	shared	R/W	transparent L1-3
AMD Istanbul	shared	R/W	transparent L1-3
AMD Barcelona	shared	R/W	transparent L1-3
AMD Magny Cours	shared	R/W	transparent L1-3
IBM Power 7	shared	R/W	transparent L1-3
Sun Niagara II	shared	R/W	transparent L1-2
NVIDIA G80	shared; device; host	R/W; R/W; DMA	app-controlled shared store; texture
NVIDIA GTX200	shared; device; host	R/W; R/W; DMA	app-controlled shared store; texture
NVIDIA GF100	shared; device; host	R/W; R/W; DMA	app-controlled shared store; transparent L1-2
ATI HD4890	shared; device; host	R/W; R/W; DMA	app-controlled shared store; transparent L1-2
ATI HD5870	shared; device; host	R/W; R/W; DMA	app-controlled shared store; transparent L1-2
Cell/B.E.	PPU, SPU	R/W; DMA	app-controlled local store

Finally, we propose an absolute performance comparison. Thus, Figure 2.14(a) and (b) present two graphical comparisons of multi-core platforms performance - per processor and per core, respectively. Further, the graphs in Figure 2.14(c) illustrate a comparison between the processor bandwidths. Figure 2.15 shows a comparison of the theoretical peak arithmetic intensity (number of operations per transferred byte) for the same processors.

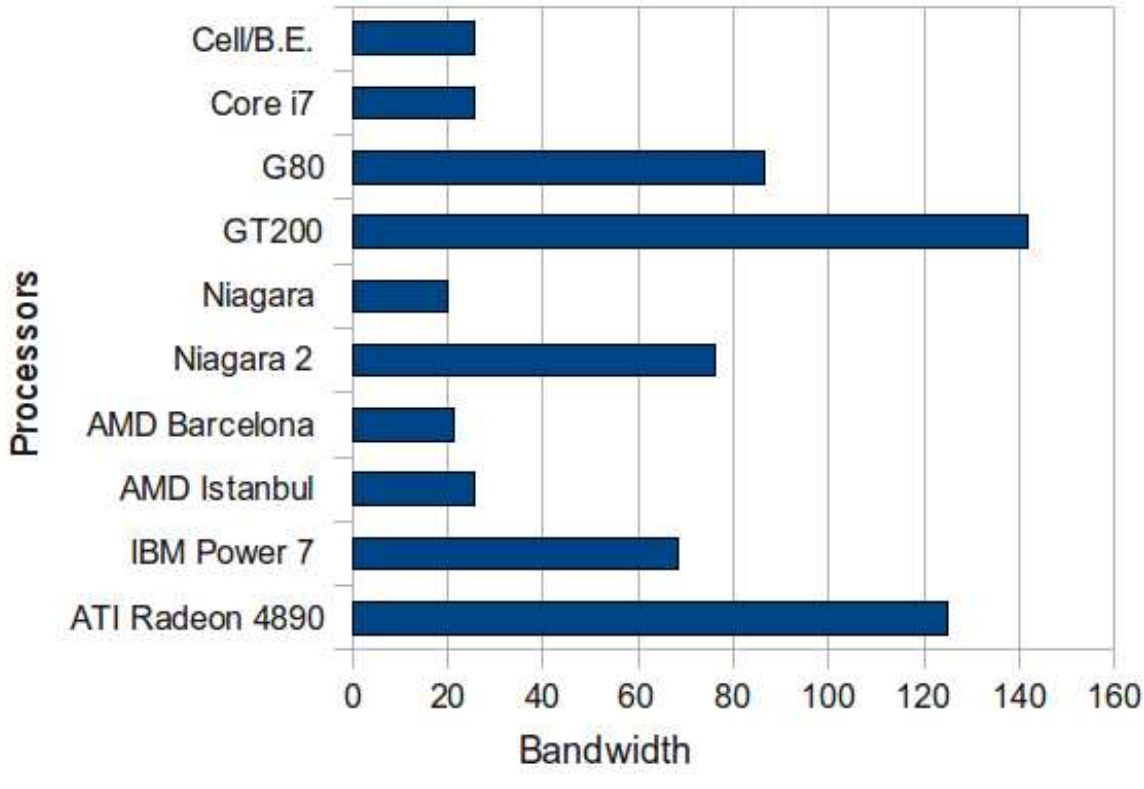
2.5



(a) Processor performance [GFLOPs].



(b) Performance per core [GFLOPS].



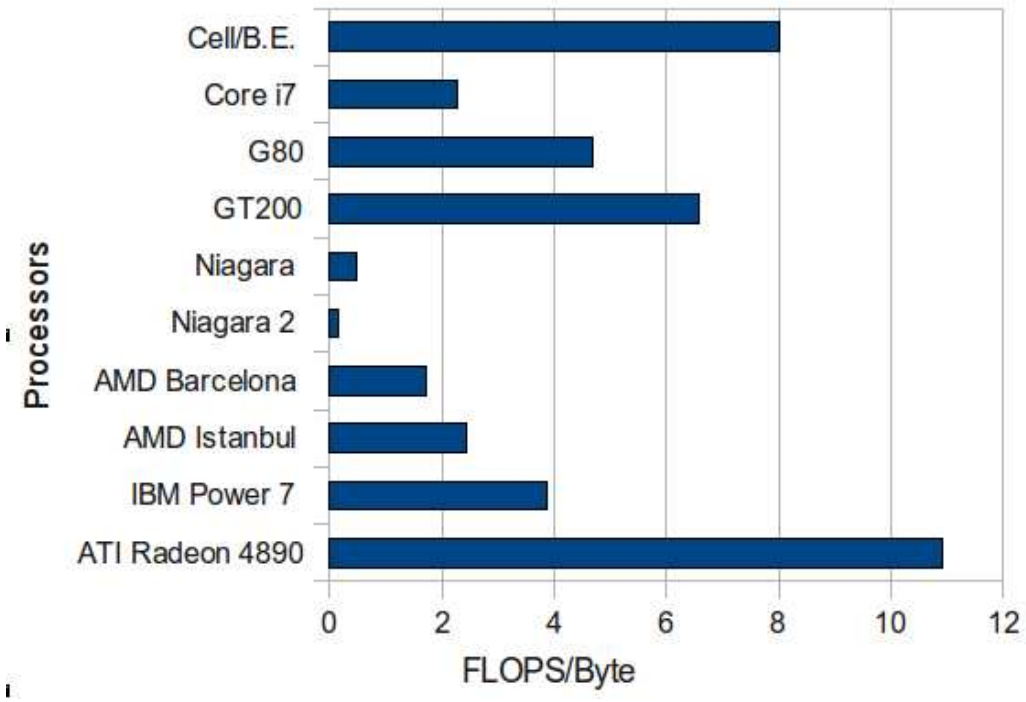


Figure 2.15: *A graphical comparison of the airthmetic intensity of existing multi-core processors.*

2.5.3 Compatibility

So far, we have exposed a number of differences between multi-core platforms: architecture, performance, parallelism and programming models, as well as workloads. However, given their increased availability, as well as the large number of applications that aim to achieve significant performance improvement using multi-cores, one has to also analyze compatibility.

We define *compatibility* between two platforms as the ability of both platforms to run the same workload without significant changes, and obtain similar performance behaviour. It should be immediately obvious that compatibility is in fact a subjective metric of platform similarity. Note that the complementary feature of compatibility (on the application side) is portability (i.e., a workload is portable if it can run on multiple platforms without changes and with similar performance behaviour).

We define three types of platform compatibility:

1. *Inter-family compatibility*, which refers to different types of platforms - for example, GPMCs vs. GPUs vs. The Cell/B.E.;

2. *Intra-family compatibility* refers to different type of platforms from the same class - for example, Intel GPMCs with AMDś GPMCs, or NVIDIA GPUs with AMD GPUs.

3. *Inter-generations compatiblity*(also known as backward-compatibility) refers to the compatiblity of different generations of the same family - for example, NVIDIA G80 and NVIDIA GF100.

We also define two compatibility levels:

1. *Parallelization level* - two platforms are compatible at the parallelization level if the same parallelization gives similar performance behaviour on both platforms;

2. *Code level* - two platforms are compatible at the code level if the code written for one can be compiled and executed for the other; the requirements of similar performance behaviour

Table 2.4 presents a compatibility chart of the platforms discussed so far in this chapter. We note the clustering of the platforms. In other words, we note that inter-family compatibility does not exist (except for specific workloads in certain cases). Further, intra-family compatibility holds at the parallelization level for most GPMCs, but not for GPUs. Cell and SCC are quite unique (no real family of processors), so there is not much to add there. Finally, in terms of generations compatibility, most platforms are both parallel and code compatible. NVIDIA GPUs, however, are the exception, as the platforms changes between generations were quite dramatic, and therefore parallelization decisions have to change for similar performance behaviour.

Table 2.4: *A compatibility chart for multi-core platforms. We assume a workload is designed and implemented for the platforms on rows, and the chart registers the type of compatibility with the platforms in each column. Legend: "–" stands for not compatible, "P" and "C" stand for parallelization and code level compatibility, respectively; "(P)" stands for possible parallelization compatibility, depending on the workload; "(C)(language)" stands for possible code-level compatibility, when the specified language is used for the programming; "NA" stands for not applicable.*

	Nehalem	AMD N-core	Niagara	POWER7	ATI GPUs	G80	GT200	GF100	Cell/B.E.	SCC
Intel Nehalem	NA	P,C	P	P	-	-	-	-	(P)	(P)
AMD N-core	P,C	NA	P	P	-	-	-	-	(P)	(P)
Sun Niagara	P	P	NA	P	-	-	-	-	(P)	(P)
IBM POWER7	P	P	P	NA	-	-	-	-	(P)	(P)
ATI GPUs	-	-	-	-	NA	(P)	(P)	(P)	-	-
NVIDIA G80	-	-	-	-	(P)	NA	P,C(CUDA)	(P),C(CUDA)	-	-
NVIDIA GT200	-	-	-	-	(P)	C(CUDA)	P	(P)	-	-
NVIDIA GF100	-	-	-	-	(P)	C(CUDA)	(P),C(CUDA)	NA	-	-
STI Cell/B.E.	-	-	-	-	-	-	-	-	NA	-
Intel SCC	-	-	-	-	-	-	-	-	-	NA

Some of the compatibility problems can be solved using specific programming models and/or languages (see Chapter 7 for more details). The best example of such a solution is OpenCL, a programming language developed as a standard for multi-cores. OpenCL enables code-level compatibility between all platforms that implement it, and enhances intra-family parallelization compatibility. To do so, OpenCL uses a virtual multi-core platform, presented in Figure 2.16. The platform consists of a host connected to one or more OpenCL compute devices. A compute device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements.

OpenCL applications are designed and programmed for this platform; the task of proper mapping from this virtual architecture to real hardware belongs to the processor manufacturers, and it is deployed as OpenCL drivers. Consequently, the performance of the application depends heavily on the driver performance. Note that the best intuitive match for the OpenCL platform are the GPU architectures, but drivers for the Cell/B.E. and AMD's multi-cores are already available. However, due to its code level compatibility, OpenCL offers a good solution for benchmarking parallel applications on multiple platforms.

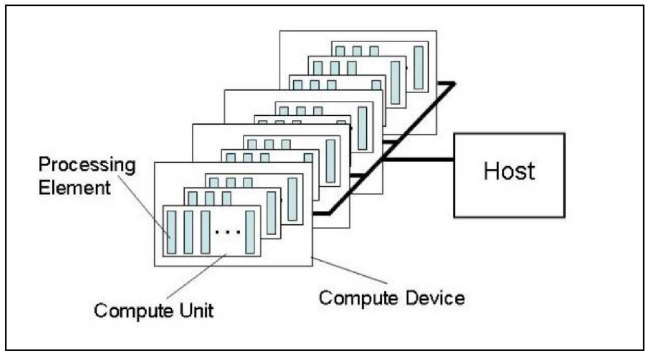


Figure 2.16: *The generic OpenCL platform model (image courtesy of KHRONOS).*

2.6 Summary

The multi-core processors landscape is very diverse, covering different granularities from the architectures with very few, fat cores (like the x86 cores of the Nehalem processor or the Opteron cores in the AMD Magny-Cours) to those with hundreds of very slim cores (typical for GPUs). All these platforms have three main things in common: (1) they offer very high performance (hundreds to thousands of GFLOPs), (2) they have several hierarchical parallelism layers (from two - e.g., for Sun processors, to five - e.g., for the Cell/B.E.), and (3) the only way for applications to get close enough to the theoretical peak performance is to expose enough software concurrency to match each hardware parallelism layer.

We also found that comparing multi-cores at the hardware performance level, besides being quite complicated, makes little sense in the absence of a workload/application. One can compare the different types of cores, the absolute performance numbers, and the memory characteristics, but this characterization leads more to a theoretical platform classification rather than to a specific ranking. To be able to compare the real-life performance of these architectures, one needs a benchmarking strategy, a set of highly-parallel applications to stress various platform characteristics, and a set of metrics that would enable a meaningful comparison. Such a benchmarking strategy should not result in a ranking of platforms per-se, but rather in a matching ranking for various types of workloads.

In our analysis we have identified three major families of multi-cores: general purpose multi-cores (GPMCs), graphical processing units (GPUs), and HPC architectures (Cell and SCC). We claim that the significant differences between the hardware and software models of these platforms make them incompatible: a workload parallelized and optimized to achieve maximum performance on one of these families will not show similar performance on the others. We believe that in the search for absolute best performance for a new workload, programmers have to choose very early in the design process which architecture is the most suitable to maximize the metric of interest, be it execution time, throughput, energy efficiency, or productivity.

This matching is currently done by empirical similarity analysis: given a reference application that behaves best on a certain family of multi-cores, all similar applications should be ported to the same architecture(s). Later in this thesis we analyze several types of workloads and extract empirical guidelines to help programmers to make an educated choice (see Chapters 3, 4, and 5).

Alternatively, one can use high-level programming languages, which abstract away architectural

details - see Chapter 7 for more details. However, as we will see, most of these languages sacrifice too much performance for portability. One interesting exception is OpenCL, a standard programming model that provides a virtual common hardware platform for the parallel application design (i.e., the platform architecture is the same), forcing inter-platform compatibility and, consequently, software portability. However, performance portability is not guaranteed. To increase performance, hardware-specific optimizations can be applied to an OpenCL application in a second phase of the design/implementation cycle, but this sacrifices portability.

2.6.1 Future Trends for Multi-core Architectures

Based on our experience with both applications and workloads, we foresee five trends in the near-future development of multi-core architectures:

First, the number of cores per processor will increase. In fact, many-core architectures (with at least 32 cores) are already available - see Intel's SCC [Bar10] and MIC [Wol10].

Second, platforms tend to base their main compute power on homogeneous collections of cores. Architectures with heterogeneous cores (i.e., more than two types of cores) will be very rare in the scientific/HPC field. One exception might be the use of FPGAs for hardware acceleration of time-critical sections.

Third, we note a trend towards accelerator-based platforms. This prediction was already made by Hill et al. in [Hil08], where they showed why Amdahl's law effects are best countered with a fast, fat master core for sequential code, and a collection of slim worker cores for the parallel code. We note that the Cell/B.E. was the first architecture to empirically discover this model, except that the PPE was too slow for the SPEs capabilities. Later on, offload-based GPGPU programming uses the same principle, only at a higher granularity, separating the master (the "host" machine) from the accelerator(s) (the device multiprocessors/processing elements). Also note that the OpenCL virtual platform adheres to the same principle.

Fourth, we believe that the memory systems will become very complex on the hardware side; still, the users will see three types of memories: core/cluster-of-cores cache hierarchies (up to two levels, with hardware coherency), a large pool of shared memory (be it a cache or a scratch pad) with mechanisms to enable software coherency, and a slow, large (system) memory, for all other storage needs.

Fifth, and last, we shall finally see a shift of interconnects towards network on chip designs, because the bus-based solutions will no longer scale for the high number of cores per chip. For these architectures, task-level parallelism, together with mapping, and scheduling will become even more significant factors in the overall application performance.

Chapter 3

Case-Study 1: A Traditional Scientific Application*

In this chapter we present the transformations needed to port a traditional HPC application, parallelized using MPI, onto the Cell/B.E. processor. We focus on the most important algorithmic changes that have to be performed for the application to suite this accelerator based platform.

High performance computing (HPC) is thought to be the first beneficiary of the huge performance potential of multi-core processors. Currently, the fastest ten supercomputers in the world[1] are large scale machines built using (a mix of) general purpose multi-core processors, GPUs, and Cell/B.E. processors. Therefore, many classical HPC applications have been used as case-studies for multi-cores [Wil06]. In this chapter, we focus on one such application - Sweep3D, and we study its parallelization and performance on the Cell/B.E. processor. Sweep3D (see Section 3.1 for a detailed analysis) is a real-life application, representative for a substantial fraction of the computing cycles executed on some of the most powerful supercomputers in the world. Furthermore, it is a compact application, publicly available[2], and it has been studied in previous publications, allowing for direct apple-to-apples comparisons [Hoi00b].

Cell/B.E. [Kah05; Hof05] is a heterogeneous processor with nine cores: a control core (PPE) coupled with eight lightweight independent processing units (SPEs). Combining several layers of parallelism, the Cell/B.E. is capable of massive floating point processing: at a frequency of 3.2 GHz, performance peaks at 204.8 GFLOPs in single-precision floating point, and 14.63 GFLOPs in double precision. Criticized for Cell's relatively weak double precision performance, IBM improved this parameter in the newest generation of the chip, called PowerXCell 8i, who now reaches 102.4 GFLOPs in double precision. Note, however, that all the experiments in this paper have been performed on the "classical" Cell/B.E..

Despite the initial excitement for Cell's performance [Wil06], the scientific community has quickly expressed two major programmability concerns. First, given the platform complexity, it is unclear what is the fraction of the peak performance that can be *actually* achieved by scientific applications. Second, given the multiple parallelism layers of the processor, a systematic porting strategy for existing legacy software (most notably the applications that have already been written using popular communication libraries such as MPI) is yet to be found.

Using Sweep3D, we provide some answers to these questions. We start from the existing MPI implementation and transform it into an efficient Cell-based implementation. In the process, we aim

*This chapter is based on joint work with IBM's TJ Watson Research Center, previously published in the Proceedings of IPDPS 2007 ([Pet07]). The chapter supercedes the paper: we made minor revisions and added the discussion on architectures other than the Cell/B.E..

[1] Top500:http://www.top500.org

[2] http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/

(1) to determine the peak performance Sweep3D can achieve on the Cell/B.E., and (2) to define a clear migration path from an MPI-like implementation into one that combines five distinct levels of parallelism.

Briefly, the initial, coarse-level MPI implementation is incrementally enhanced to support *thread level parallelism* across the SPEs, *data streaming parallelism* to address the space limitations of SPE local stores, *vector parallelism* to execute multiple floating point operations in parallel, and *pipeline parallelism* to guarantee dual issue of instructions when possible. Section 3.2 gives a "global view" of our multi-dimensional approach to parallelization for the Cell/B.E.. The complexity and the performance impact of each of these parallelization steps, together with platform specific optimizations (described in detail in Section 3.3.1), are quantified and discussed along this chapter.

Our experimental results (presented in detail in Section 3.3) prove that all these platform specific parallelization strategies and optimizations paid off, leading to an overall execution speedup ranging from 4.5x when compared to processors specifically designed for scientific workloads (such as IBM Power5) to over 20x when compared to conventional processors.

However, Cell has also posed unexpected hurdles in the data orchestration, processor synchronization, and memory access algorithms. We summarize our experiences - both the problems and our chosen solutions - in a set of empirical guidelines (see Section 3.6), aiming to help programmers to systematically deal with similar applications.

3.1 Application Analysis

Sweep3D solves a three-dimensional neutron transport problem from a scattering source. In general, "particle transport" (or "radiation transport") analyzes the flux of photons and/or other particles through a space. For the discrete analysis, the space is divided into a finite mesh of cells and the particles are flowing only along a finite number of beams that cross at fixed angles. The particles flowing along these beams occupy fixed energy levels. The analysis computes the evolution of the flux of particles over time, by computing the current state of a cell in a time-step as a function of its state and the states of its neighbours in the previous time-step.

Sweep3D has been chosen as the heart of the Accelerated Strategic Computing Initiative (ASCI)[3] benchmark. For this implementation, the three-dimensional geometry is represented by a logically rectangular grid of cells (with dimensions I, J and K) divided into eight octants by the scattering source. The movement is modeled in terms of six angles (three angles in the forward direction and three angles in the backward direction) for each octant. The equations for each angle can be seen as a wavefront sweeping from one corner of the space (i.e., the octant) to the opposite corner. A complete sweep from a corner to its opposite corner is an iteration. There are several iterations for each time step, until the solution converges. Hence, the solution involves two steps: the streaming operator (i.e., result propagation), solved by sweeps, and the scattering operator, solved iteratively [Bak95].

An S_n sweep for a single octant and a given angle works as follows. Each grid cell has 4 equations with 7 unknowns (6 faces plus 1 central). Boundary conditions complete the system of equations. The solution is reached by a direct ordered solver, i.e., a sweep. Three known inflows allow the cell center and three outflows to be solved. Each cell's solution then provides inflows to 3 adjoining cells (1 each in the I, J, and K dimensions). This represents a wavefront evaluation with a recursion dependence in all three grid directions. Each octant has a different sweep direction through the grid of cells, but all angles in a given octant are independent and sweep in the same way.

[3] http://www.sandia.gov/NNSA/ASC/

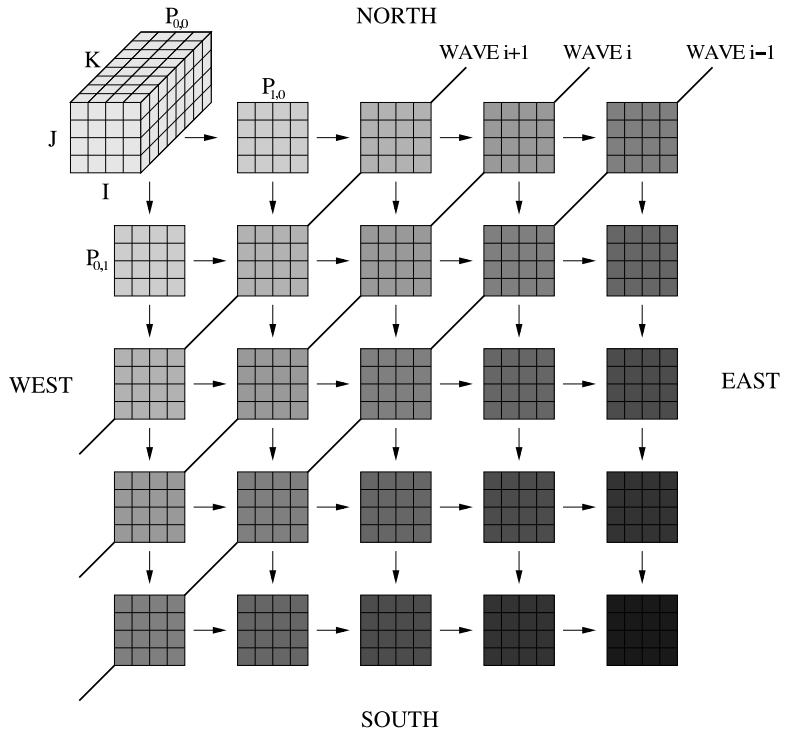


Figure 3.1: *Data mapping and communication pattern of Sweep3D's wavefront algorithm.*

Sweep3D exploits parallelism via a wavefront algorithm [Hoi00a; Hoi00b]. Each grid cell can be only computed when all the previous cells in the sweep direction have been already processed. Grid cells are evenly distributed across a two-dimensional array of processes. In this way, each process owns a three-dimensional tile of cells. The data mapping and the wave propagation during a noth-to-south, west-to-east sweep is described in Figure 3.1. For the sake of simplicity, the K dimension is hidden for all processes other than upper-left corner process. The wave is originated by the process in the upper-left corner, that is, $P_{0,0}$. This process solves the unknowns of the local cells and then propagates the results to its east and south neighbors, that is, $P_{1,0}$ and $P_{0,1}$, respectively. At this point the two adjacent processes can start computing the first wave, while the upper-left corner process starts the second wave. In an ideal system, where computation and communication are perfectly balanced, each diagonal of processes would be computing the same wave at any given time.

The pseudocode of the `sweep()` subroutine, the computational core of Sweep3D, is presented in Listing 3.1. Before each inner iteration (lines 7 to 17), the process issues waits for the I-inflows and J-inflows coming from the west and north neighbors, respectively (lines 5 and 6). Then, it computes the incoming wave through its own tile of cells using a stride-1 line-recursion in the I-direction as the innermost work unit (lines 8 to 15). It is worth noting that this access to the arrays in a sequential fashion, can be exploited to extract further pipeline and data parallelism. Finally, the process sends the I-outflows and J-outflows to the east and south neighbors, respectively (lines 18 and 19).

With this form of the wavefront algorithm, parallelism is limited by the number of simultaneous waves. In the configuration depicted in Figure 3.1, there would always be idle processes if $\sqrt{P}$ is greater than the number of simultaneous waves, an artificial limitation if Sweep3D is executed on a large configuration. To alleviate this problem, `sweep()` is coded to pipeline blocks of MK K-planes

Listing 3.1: *The* `sweep()` *subroutine.*

```
1 SUBROUTINE sweep()
2 DO iq=1,8                          ! Octant loop
3  DO m=1,6/mmi                      ! Angle pipelining loop
4   DO k=1,kt/mk                     ! K-plane pipelining loop
5    RECV W/E                        ! Receive west/east I-inflows
6    RECV N/S                        ! Receive north/south J-inflows
7    DO jkm=1,jt+mk-1+mmi-1          ! JK-diagonals with MMI pipelining
8     DO il=1,ndiag                  ! I-lines on this diagonal
9      IF .NOT. do_fixups
10      DO i=1,it [...]                    ! Solve Sn equation
11     ELSE
12      DO i=1,it [...]               ! Solve Sn equation with fixups
13     ENDIF
14    ENDDO                          ! I-lines on this diagonal
15   ENDDO                           ! JK-diagonals with MMI pipelining
16   SEND W/E                        ! Send west/east I-outflows
17   SEND N/S                        ! Send north/south J-outflows
18  ENDDO                            ! K-plane pipelining loop
19 ENDDO                             ! Angle pipelining loop
20 ENDDO                             ! Octant loop
```

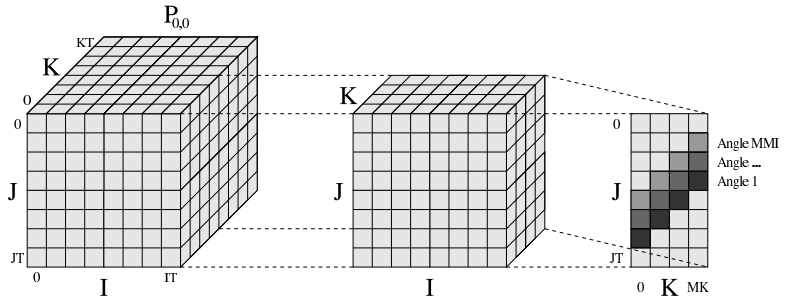


Figure 3.2: *SWEEP3D parallelization through K-plane and angle pipelining.*

(MK must factor KT) and MMI angles (1 or 3) through this two-dimensional process array for each octant. As an example, the first inner iteration for $P_{0,0}$ is shown in Figure 3.2. This iteration considers a block of four K-planes (MK is 4) and eight J-planes (JT is 8) for three different angles (MMI is 3). The depicted `jkm` value is 6 which includes the sixth JK diagonal for angle 1, the fifth diagonal for angle 2 and the fourth diagonal for angle 3, that is, `il` is 12. One important feature of this scheme is that all the I-lines for each `jkm` value can be processed in parallel, without any data dependency. This property plays a central role in our proposed parallelization strategy discussed in the next section.

3.2 Parallelization Strategies

The shift of paradigm in architectural design imposed by the new multi-core processors was expected to determine a comparable, if not bigger, advancement in parallelizing compilers and run-time systems. However, even if early results [Eic05] show that parallelizing compilers can achieve very good results

on many significant application kernels, the efficient and automatic parallelization of full scientific applications is not yet feasible with state-of-the-art technology [Ier01].

In this section we discuss the strategy we have adopted to obtain a parallel, Cell-enabled version of Sweep3D. We aim for a parallel application that closely matches the architecture: all computations are performed by the SPEs, while the PPE is performing dynamic task allocation and message handling only. Our presentation follows step-by-step the parallelization techniques needed to enable the application to exploit the multiple parallelism layers of the Cell processor. Note that although we are specifically working with the Sweep3D application, the strategy presented here can be seen as a general set of empirical guidelines for extracting all the potential parallelism that an application should expose to run efficiently on the Cell/B.E..

One way to parallelize Sweep3D is to extend the wavefront algorithm to the SPEs [Ker06]. The logical grid of processors would be simply refined by the presence of a larger number of communicating computational units. However, this strategy is not able to capture the multiple parallelism levels of the Cell/B.E. processor, which are essential to get close to the peak performance on this architecture. Even further, this solution does not take into account the data orchestration required to tackle the limited local storage available on the SPEs.

Therefore, we chose a different approach, graphically outlined in Figure 3.3. We exploit five levels of parallelism and data streaming in the following way:

1. **Process-level parallelism.** At the highest level, we maintain the wavefront parallelism already implemented in MPI and other messaging layers; this guarantees portability of existing parallel software, that can be directly compiled and executed on the PPE without major changes (no use of the SPEs, yet). Furthermore, this allows for an additional, high-level parallelism layer, as the application can run on a cluster of Cell nodes.

2. **Thread-level parallelism.** We extract thread-level parallelism from the subroutine `sweep()` (lines 7 to 17), taking advantage of the lack of data dependencies for each iteration of the `jkm` loop. In our initial implementation, the I-lines for each `jkm` iteration are assigned to each SPE in a cyclic manner.

3. **Data-streaming parallelism.** Each thread loads the "working set" of the iteration using a double buffering strategy: before executing loop iteration i, the SPE issues a set of DMA

   `get` operations to load the working set of iteration $i+1$ and a set of DMA

   `put` operations to store the data structures modified by loop $i-1$. Note that "double buffering" is the usual Cell-terminology, while the technique itself is widely known as software pipelining.

4. **Vector parallelism** Chunks of loops are vectorized in groups of 2 words (double precision) or 4 words (single precision).

5. **Pipeline parallelism** Given the dual-pipeline architecture of the SPEs [IBM], the application can use multiple logical threads of vectorization, increasing pipeline utilization and masking eventual stalls. Our double precision implementation uses four different logical threads of vectorization.

It is worth noting that our loop parallelization strategy does not require any knowledge of the global, process-level parallelization. And, given that it operates at a very fine computational granularity, it can efficiently support various degrees of communication pipelining. Also, while the process

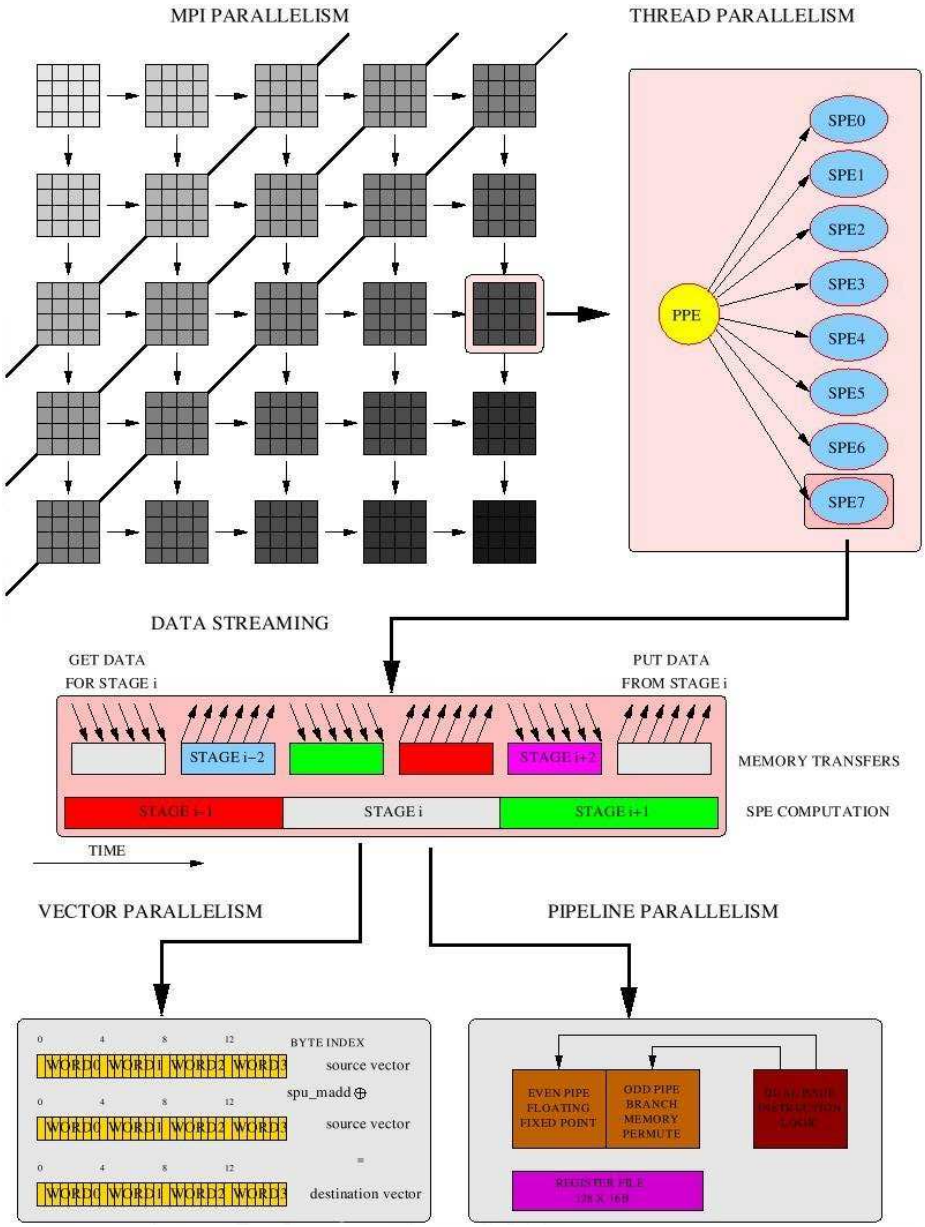


Figure 3.3: *The parallelization process.*

level parallelism can be seen as application dependent, the other parallelism levels are to be exploited in a similar manner for any application to be ported onto the Cell/B.E.. In other words, the same techniques for thread-level, data streaming, vector, and pipeline parallelization apply for optimizing other applications.

3.3 Experiments and Results

As a starting point for the experiments, we have ported Sweep3D onto a platform containing a single Cell BE chip, running at 3.2 GHz. Sweep3D ran on the PPU alone (on a single thread) with a $50 \times 50 \times 50$ input set (50-cubed), without any code changes, in 22.3 seconds.

On top of the layered parallelization, we have also used an arsenal of optimization techniques to fine tune the performance of Sweep3D. In this section, we briefly explain these techniques, and we evaluate them in terms of their impact on the overall Sweep3D performance.

3.3.1 SPE Optimizations

To prepare for efficient execution on SPUs, several additional steps were required:

1. all the arrays were converted to be zero-based,
2. multi-dimensional arrays were flattened (by computing the indices explicitly),
3. cache-line (128 bytes) alignment was enforced (by padding) for the start addresses of each chunk of memory to be loaded into the SPU at run-time, to improve DMA performance [Kis06b],
4. all big arrays were zero-ed (by `memset` calls).

To benefit from more aggressive optimizations, we have replaced Gnu's `gcc 4.1` compiler with the IBM `xlc 8.2` compiler, a mature, production quality compiler with excellent optimization support. After all these changes, the execution time of the code (still running only on the PPE) was 19.9 seconds (10% improvement).

Next, we identified the code to run on the SPUs and relocated routines to group them into one subroutine call. Further, the loop structure was remodeled to split the computation across eight independent loop iterations. Finally, the computation was offloaded to the eight SPEs. Each SPE gets to compute one iteration at a time, and PPE "distributes" these iterations cyclically. The application execution time dropped to 3.55 seconds (less than 16% from the original time). This speed-up (6.3x) is due to a combination of two factors: the speed-up of the SPE versus the PPE, together with the parallelization across 8 cores. Similar drops in execution times are to be expected for any application that is able to fully utilize the eight SPE cores of the Cell. Hence, the comparison with any other 8-core platfrom may be unbalanced if the cores on the other target platform are performance-wise inferior to the SPEs on the Cell.

After we have modified the inner loop to eliminate goto statements, and modified the array allocation to ensure that the "multi-dimensional" arrays started each of their rows on a new cache line, the run time decreased to 3.03 seconds. Double buffering (a Cell-specific technique, which can be seen as a counter-balance to other architectures' caches and cache-prefetching) further reduced the execution time to 2.88 seconds.

By using explicit SPE intrinsics (manual SIMDization, a technique common to all vector machines), we have achieved a new significant improvement, bringing the run time down to 1.68 seconds. A code

Listing 3.2: *Scalar loops (original).*

```
1 for( n = 1; n < nm; n++ )
2   for( i = 0; i < it; i++ )
3     Flux[n][k][j][i] = Flux[n][k][j][i] + pn[iq][n][m]*w[m]*Phi[i];
```

snippet illustrating the SIMDization process is provided in Listings 3.2(before) and 3.3(after) - we will discuss this in more detail in the following section.

Finally, converting the individual DMA commands to DMA lists, and adding offsets to the array allocation to more fairly spread the memory accesses across the 16 main memory banks, further reduced the execution time to 1.48 seconds. Eliminating the use of mailboxes, and using a combination of DMAs and direct local store memory poking from the PPE to implement a PPE-SPE synchronization protocol, the execution time dropped to 1.33 seconds. Note that all these "fine tuning" optimizations, although applicable to any application running on Cell, may have very different impacts on other applications. It is worth noting that the execution time we have obtained is the best result seen so far on a fully functional implementation of Sweep3D.

3.3.2 Loop Vectorization

A closer look to the performance numbers presented in the previous paragraphs reveals the key role that vectorization, together with the SPE thread parallelism and the data orchestration, are playing in reducing the execution time of Sweep3D. Among the three, vectorization has the biggest impact in terms of relative gain. Thus, we have included a short example showing a snippet of code 'before' and 'after' the vectorization - see Listings 3.2 and 3.3, respectively.

The following are a few hints to understand the transformation of the scalar loops into vectorized loops, and, consequently, the way the vectorization has been performed in the case of Sweep3D:

1. The SIMDized version performs operations on four separate threads of data simultaneously (A, B, C and D).
2. `wmVA`, `wmVB`, `wmVC`, `wmVD` represent a vectorized portion of `w[m]`. In this case, the values stored in the four vectors are interleaved because of the four logic threads.
3. The number of elements in any of the `wmV*` vectors is $(16/sizeof(w[m]))$, which is 2 in this case because `w[m]` is a double precision floating point value.
4. `spu_madd` does a double-precision 2-way SIMD `multiply_add`.
5. `spu_splats` replicates a scalar value across a vector variable.
6. the inner loop runs half as many times because each operation is 2-way SIMD.

While the natural choice for vectorizing code is innermost loop unrolling, it requires these loop iterations to be data-independent. The innermost loop of Sweep3D processes one sweep, and iterations are data dependent, while the outer loop deals with time-step iterations, which are data-independent. Thus, we have chosen to vectorize along the time-wise loop, using four simultaneous logical threads, a solution that allows avoiding the data stalls. Therefore, we had changed the initial implementation: instead of having the PPE sending only one iteration to each SPE, cyclically, it now sends chunks

Listing 3.3: *Vectorized loops (optimized).*

```
1  for( n = 1; n < nm; n++) {
2    vector double pnvalA, pnvalB, pnvalC, pnvalD;
3    pnvalA = spu_splats(LS_pn[0][comp][n]);
4    pnvalB = spu_splats(LS_pn[1][comp][n]);
5    pnvalC = spu_splats(LS_pn[2][comp][n]);
6    pnvalD = spu_splats(LS_pn[3][comp][n]);
7
8    pnvalA = spu_mul(pnvalA, wVA);
9    pnvalB = spu_mul(pnvalB, wVB);
10   pnvalC = spu_mul(pnvalC, wVC);
11   pnvalD = spu_mul(pnvalD, wVD);
12
13   FluxVA = (vector double *) LS_Flux[0][comp][n];
14   FluxVB = (vector double *) LS_Flux[1][comp][n];
15   FluxVC = (vector double *) LS_Flux[2][comp][n];
16   FluxVD = (vector double *) LS_Flux[3][comp][n];
17   for( i = 0; i < (it+1)>>1; i++ ) {
18     FluxVA[i] = spu_madd(pnvalA, PhiVA[i], FluxVA[i]);
19     FluxVB[i] = spu_madd(pnvalB, PhiVB[i], FluxVB[i]);
20     FluxVC[i] = spu_madd(pnvalC, PhiVC[i], FluxVC[i]);
21     FluxVD[i] = spu_madd(pnvalD, PhiVD[i], FluxVD[i]);
22   }
23 }
```

of four consecutive iterations in every "dispatch" cycle. The granularity of this parallelization is still relatively fine: four iterations on a 50-cubed input can be executed in as little as 10μs.

3.3.3 Peak and Actual Floating Point Performance

The vectorized version of loop listed in Listing 3.4 (for the sake of simplicity and readibility shown in scalar form) takes 590 cycles ("do_fixup" off) and 1690 cycles ("do_fixup" on) to execute 216 Flops. There are 24 and 85 instances of dual issue, respectively. This means that roughly 5% of the cycles are successfully issuing two commands per cycle. Thus, the theoretical peak performance is 4 Flops every 7 cycles, which is equivalent to 64% of the theoretical peak performance in the "do_fixup off" case.

In single precision, the number of operations jumps to 432, and the number of cycles drops to approximately 200, but the theoretical maximum is now 8 ops/cycle, so our efficiency reaches a still-respectable 25%.

3.3.4 Scalability

So far, we have discussed in detail the optimization process by focusing on a specific input set with a logical IJK grid of 50x50x50 cells. Figure 3.4 summarizes the performance impact of all the optimizations.

Further, we focus on how the application performance varies with the input data sizes. We consider the input domain as a three-dimensional cube of a specified size, and we analyze the grind time (i.e.,

Listing 3.4: *Main loop.*

```
1  for( i = i0; (i0 == 1 && i <= i1) || (i0 ==  it && i >= i1) ; i=i+i2 )
2  {
3    ci =  mu[m]*hi[i];
4    dl = ( Sigt[k][j][i] + ci + cj + ck );
5    dl = 1.0 / dl;
6    ql = ( Phi[i] + ci*Phiir + cj*Phijb[mi][lk][i] + ck*Phikb[mi][j][i]);
7    Phi[i] = ql * dl;
8    Phiir          = 2.0*Phi[i] - Phiir;
9    Phii[i]        = Phiir;
10   Phijb[mi][lk][i] = 2.0*Phi[i] - Phijb[mi][lk][i];
11   Phikb[mi][j][i]  = 2.0*Phi[i] - Phikb[mi][j][i];
12 }
```

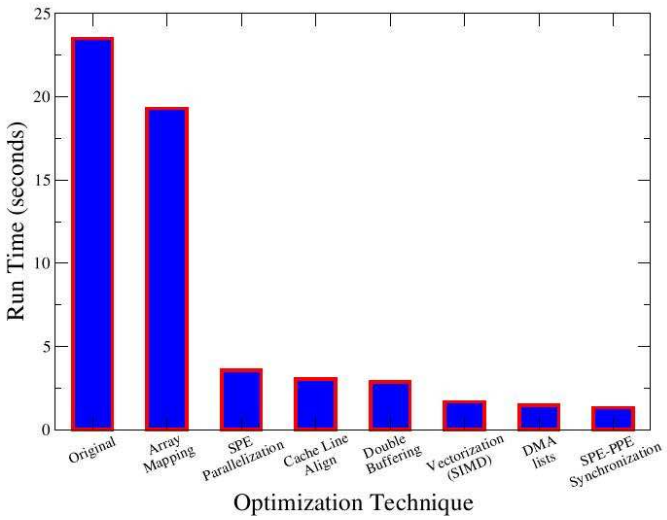


Figure 3.4: *The performance impact of various optimizations.*

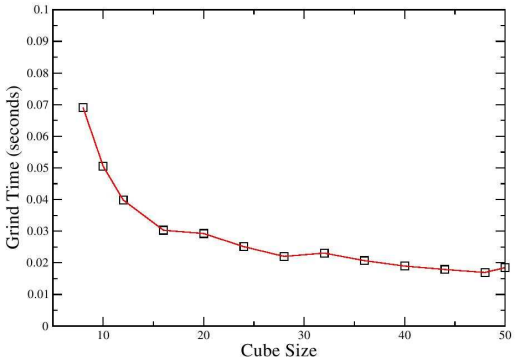


Figure 3.5: *The grind time as a function of the input data "cube" size.*

the normalized processing time per cell) when varying this "cube size". Figure 3.5 shows the grind time as a function of the input data size. We point out that for a cube size larger than $25 \times 25 \times 25$ cells, the grind time is almost constant, indicating that the application scales well with the input data size. Our load balancing algorithm farms chunks of four iterations to each SPE, so perfect load balancing can be achieved when the total number of iterations is an integer multiple of 4×8, as witnessed by the minor dents in Figure 3.5.

With a 50-cubed input size, the SPEs transfer 17.6 GBytes of data. Considering that the peak memory bandwidth is 25.6 GB/s, this sets a lower bound of 0.7 s on the execution time of Sweep3D. By profiling the amount of computation performed by the SPUs we obtain a similar lower bound, 0.68 seconds.

3.3.5 Performance Comparison and Future Optimizations

The gap between the computation bound (0.7 s) and the actual run-time of 1.3 s is mostly caused by the communication and synchronization protocols. We have identified the following directions to further reduce the execution time. The performance impact of these planned optimizations is outlined in Figure 3.6.

- By increasing the communication granularity of the DMA operations, which are currently implemented with lists of 512-byte DMAs (both for `puts` and `gets`), we can further reduce the run time to 1.2 seconds (as predicted by a test implementation, not yet fully integrated in Sweep3D).

- We noticed that the PPE cannot distribute efficiently the chunks of iterations across the SPEs, becoming a bottleneck. By replacing the centralized task distribution algorithm with a distributed algorithm across the SPEs, we expect to reduce the run time to 0.9 seconds.

- Contrary to our expectations, a fully pipelined double precision floating point unit would provide only a marginal improvement, to 0.85 seconds.

- By using single precision floating point, we expect a factor of 2 improvement, with a run time of approximately 0.45 seconds, again determined by the main memory bandwidth.

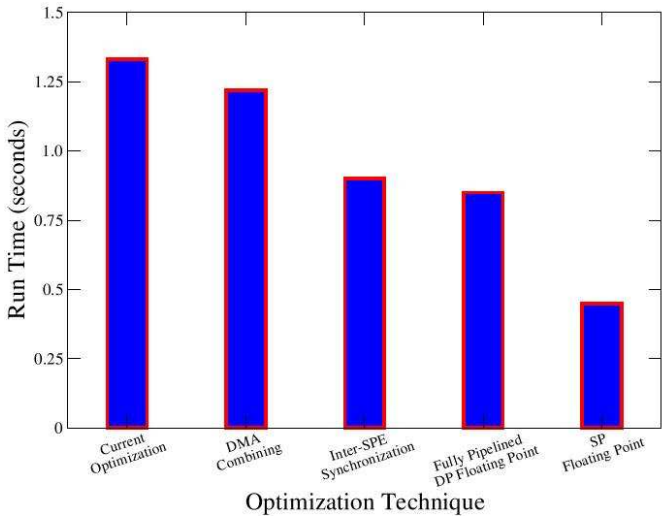


Figure 3.6: *The expected performance impact of optimizations, architectural improvements, and single precision floating point.*

Finally, Figure 3.7 compares the absolute run time with other processors. Cell BE is approximately 4.5 and 5.5 times faster than the Power5 and AMD Opteron, respectively. We expect to improve these values to 6.5 and 8.5 times with the optimizations of the data transfer and synchronization protocols. When compared to the other processors in the same Figure, Cell is about 20 times faster or more.

3.4 Lessons Learned and Guidelines

Our experimental results show that the Cell/B.E. offers high levels of performance for scientific applications *if* they are properly parallelized and optimized. The good performance results we achieved (the optimized Cell/B.E. implementation was up to 20x faster than the original versions running on traditional archtiectures), required a step-by-step parallelization strategy, followed by aggressive core-level optimizations. We express this strategy in a collection of guidelines as follows:

1. Determine and isolate tasks to be managed by the PPE;
2. Detect thread-level parallelism and offload the computation to the SPEs;
3. Hide DMA latencies by pre-fetching and delayed writing to/from the limited SPEs memory;
4. Enable code vectorization and make use of the SPU intrinsics;
5. Enable SPE's pipeline-parallelism by re-shuffling instructions

With all these optimizations, Sweep3D performance increased more than 10 times compared with the original version. Note that this strategy can and should be used for implementing other scientific applications on the Cell (of course, the performance gain factors can vary significantly).

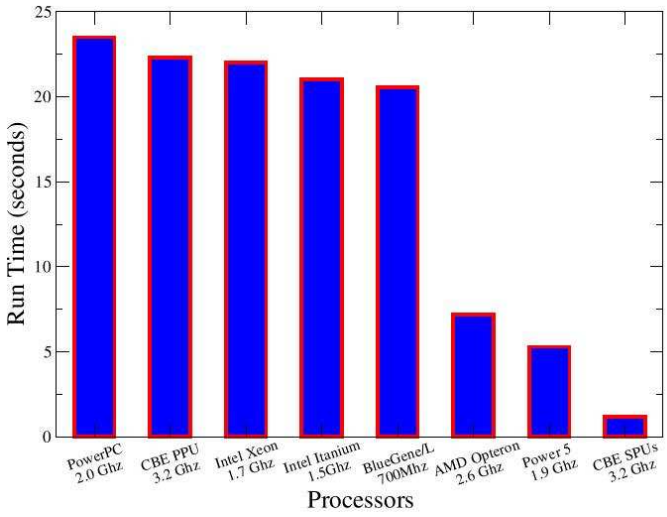


Figure 3.7: *A performance comparison of the Sweep3D execution times on various hardware platforms.*

We believe that the computational requirements of most HPC applications can be met by Cell if properly programmed. Still, the gap between the communication and computation capabilities of the platform will impose additional, complex optimizations for highly data-intensive applications. For these problems, algorithm tuning and data restructuring and/or redistribution become mandatory to avoid large performance penalties due to data starvation.

3.5 Related Work

This work combines a classical research topic, namely the pursuit of the highest achievable performance of scientific workloads, in the context of the state-of-the-art multi-core platforms. Thus, we analyze both the parallelization and optimization methods required to adapt the classical scientific kernels to the complexity of the multi-core platforms, and we also evaluate the potential critical hardware bottlenecks that may hinder these large-scale applications to run on such systems-on-chips.

We note that there are very sparse references to similar efforts. One example is found in [Smi05], where the authors describe a similar analysis, targeted to a slightly different processor type. The authors try to destroy the unfair myth that reconfigurable processors are too small and too slow for being able to cope with the size of scientific applications. Still, the difference in scale for both the applications and the processor itself do not allow such complex parallelization and optimization solutions as our Sweep3D version for Cell BE.

There is a sustained effort spent in keeping scientific applications up to date with modern machines. However, research such as that presented in [Geu94; Guo04; Oli04; Car06] focuses on high performance machines, with rather conventional architectures, where the efforts of adapting the code are significantly different than the unconventional issues that the ones we face in the less standardized field of multi-cores.

The efforts of porting scientific applications onto multi-core systems are less convincing. The

major cause for this low interest is the strong orientation of the multi-core processors towards the embedded systems field, mainly induced by (1) their severe resource constraints, (2) their complex and unpredictable memory hierarchies, and (3) the computation scale difference that seems to completely separate the two fields. However, newer, high-performance systems like the Cell/B.E. show viable solutions to alleviate these problems. Still, to our best knowledge, this work is the first to make a complete study of a successful combination between a *real* multi-core platform and a *complete real-life* scientific application.

As a final remark, we should also signal the interest that the authors of Co-array Fortran [Coa05; Coa06] have shown to the Sweep3D application in their analysis on increasing the productivity of parallel application development. Their study of several CAF implementations of Sweep3D on four modern architectures aims, as our work partially does, at identifying a good platform-implementation match. However, the target platforms are significantly different than the Cell/B.E., and the performed optimizations focus at a higher level of granularity than our approach.

3.6 Summary and Discussion

Together with an unprecedented level of performance, multi-core processors are bringing an unprecedented level of complexity in software development. We see a clear shift of paradigm from classical parallel computing, where parallelism is typically expressed in a single dimension (i.e, local vs. remote communication, or scalar vs. vector code), to the complex, multi-dimensional parallelization space of multi-core processors, where multiple levels of parallelism *must be* exploited in order to gain the expected performance. Our Sweep3D case-study demonstrates very well this shift: we had to change, tweak, and tune the code significantly to be able to extract enough parallelism to "fill" the Cell/B.E. up to a respectable fraction of its potential.

First of all, the Sweep3D implementation has proven that the Cell/B.E. offers good opportunities to achieve high floating point performance for real applications: we reached 9.3 GFLOPs in double precision and about 50 GFLOPs in single precision (64% and 25% of Cell's peak performance, respectively).

However, the Sweep3D performance was not limited by the computation capabilities of the platform. Instead, it was the memory bandwidth that did not match the application data access requirements, leading to a significant bottleneck. This memory-bound behaviour of the Sweep3D application is a direct consequence of the optimizations and changes induced by the parallelization. Furthermore, this means that the Sweep3D performance on the Cell can still increase by reducing memory traffic and/or using a platform with higher core-to-memory bandwidth.

Finally, we conclude that programming complex multi-cores like the Cell/B.E. is significantly simplified if done systematically, by matching the multiple layers of hardware parallelism with corresponding layers of application concurrency. When correctly implemented, this correspondence extends to performance: the more concurrency an application is able to express, the closer its performance will get to the platform peak.

Alternative architectures

Porting Sweep3D onto alternative multi-core architectures requires different parallelizations than the Cell/B.E.. Despite offering multiple parallelism layers, the granularity of concurrent processes required at each layer is different than for the Cell/B.E.

For example, for the general purpose multi-core processors (GPMCs), parallelization is much coarser. The simplest Sweep3D parallelization is based on matching the process-level parallelism of the application (i.e., the MPI processes) with the core-level parallelism. Each MPI process can be ported onto a single core. The MPI communication is replaced by shared memory calls. A slightly finer grain parallelization can mimic the thread-level parallelism of the Cell/B.E., allocating one thread per core, and leaving the MPI processes for a higher level of parallelism (i.e., cluster level). In terms of core-level optimizations, some of the SPE ones are portable (at design-level): the use zero-based arrays, arrays flattening, cache-line alignment (different sizes), code vectorization (using SSE instructions, for example). Others, like DMA double buffering and the explicit dual-pipeline parallelism do not apply.

In the case of GPUs, the situation is reversed: for good performance, we require finer-grain parallelism. Most likely, the best performance would be obtained by refining the thread level parallelism exploited in the Cell/B.E. such that each GPU processing element should compute one loop iteration. Note that this transformation is equivalent with a flattening of the higher level parallelism such that we generate a very large number of threads. Extracting the best parallel performance out of this scheme becomes the task of the platform itself, via the embedded hardware threads scheduler. In terms of core-level optimizations, different GPU architectures may replicate some of the SPE optimizations, but not all. Thus, double buffering might be used between the host and the GPU device. Vectorization is not required for NVIDIA GPUs, but gives additional performance gains on ATI GPUs. Pipeline parallelism is not applicable to either of the two.

Chapter 4

Case-Study 2: A Multi-Kernel Application†

In this chapter we discuss the parallelization of a multi-kernel application for the Cell/B.E.. We focus on the application-level parallelism, analyzing its influence on the overall application behavior.

Evaluating and optimizing the performance of multi-core systems are major concerns for both hardware and software designers. The lack of standard benchmarking suites and the lack of meaningful metrics for platform/application comparison [Int07] has lead to a rather sparse set of experiments, aimed at porting and optimizing individual applications [Pet07; Bla07]. Furthermore, most of these applications are treated as mono-kernel applications, in which a single kernel has to be implemented and optimized for the new architecture; once implemented, this kernel is exploited using an SPMD (Single Process Multiple Data) model.

In general, porting these applications onto multi-core platforms has followed a simple three-steps path: (1) identify the kernel(s) suitable - in terms of size and computation - to the available cores, (2) port the kernel(s) onto the architecture cores, and (3) iteratively optimize each of the kernel(s) for the specific core it is running on. This process results in highly optimized kernels, which show impressive performance gains when compared with the original application [Liu07]. In many cases, however, these kernels have a limited impact in the overall application performance [Var07] (also see Chapter 6) - a simple effect of Amdahl's law.

We believe that the way to tackle this problem is to address mapping and scheduling of the kernels on the target platform as an additional optimization layer: application-level optimization. These optimizations typically address the control flow of the original application, often requiring restructuring data structures and procedure calls.

To verify this hypothesis, this chapter presents a case-study for application-level optimizations. Our main goal is to investigate if and how mapping and scheduling of kernels on cores, typically by combining data- and task-parallelism, influences overall application performance. For this study, we use MARVEL, a digital media indexing application, and the Cell/B.E. processor. The Cell-enabled version of MARVEL is called MarCell.

Cell/B.E. [Kah05; Hof05] is a heterogeneous processor with nine cores: a control core (Power Processing Element (PPE)) coupled with eight lightweight independent processing units (Synergistic Processing Elements (SPEs)) (for more details, see 2.4.1). On the Cell/B.E., application mapping and scheduling are completely under user's control, recommending this architecture as a flexible target for our high-level optimization experiments.

MARVEL is an interesting case-study in itself [Liu07] as it is a complex, real-life application[1].

†This chapter is based on joint work with IBM's TJ Watson Research Center, previously published in ICME'07 ([Liu07]) and Concurrency and Computation: Practice and Experience 21(1) ([Var09b]). This chapter supercedes both papers, making a combination of the two and adding details on the data-parallelism as well as more detailed comments on architectures others than the Cell/B.E..

[1]Porting MARVEL onto the Cell/B.E. while preserving its functionality has required specific techniques, as seen in Chapter 6

Two aspects of MARVEL are highly-computational: the image analysis for feature extraction and the concept detection for classification. There are multiple features to be extracted from each image (we have used four), and the concept detection has to be performed for each feature. Image classification is based on the Support Vector Machine (SVM) method [ST00].

In Section 4.1, we show how all these kernels (five in total) can be implemented on the Cell/B.E. with impressive performance improvements. However, we also observe the disappointing overall performance gain of the application. The root cause of this problem is poor core utilization, a design flaw that needs to be corrected at the platform level.

Therefore, we restructure the kernels and the main application to support both data and task parallelism. Further, we explore the performance of three parallelization solutions: task-parallel (MPMD - see Section 4.2), data-parallel (SPMD - see Section 4.3), and a hybrid solution (see Section 4.4). To prove how critical the high-level application optimizations are on performance, we perform a thorough analysis of *all* static mapping options for the hybrid parallel version of MarCell. We analyze the optimum mappings and we discuss the causes of the large performance gaps between them.

Based on our work with MARVEL/MarCell, we extract a short list of generic guidelines for efficient application parallelization on this multi-core processor (see Section 4.5). We claim that both the application and the platform are very relevant instances in the multi-core application development space, and we show how these rules can be extended to other multi-core platforms and/or applications.

4.1 Application Analysis

By multimedia content analysis, one aims to detect the semantic meanings of a multimedia document [Wan00], be it a picture, a video and/or audio sequence. Given the fast-paced increase in available multimedia content - from personal photo collections to news archives, the traditional manual processes that create metadata for indexing purposes cannot keep up. Furthermore, manual annotation and cataloging are costly, time consuming, and often subjective - leading to incomplete and inconsistent annotations and poor system performance. New technologies and automated mechanisms for content analysis and retrieval must be developed. For these automated classifiers, execution speed and classification accuracy are the most important performance metrics.

An example of an automated system for multimedia analysis and retrieval is MARVEL, developed by IBM Research. MARVEL uses multi-modal machine learning techniques to bridge the semantic gap for multimedia content analysis and retrieval [Nat07; Ami03; Nat04]. MARVEL automatically annotates images and videos by recognizing the semantic entities - scenes, objects, events, people - that are depicted in the content. The MARVEL multimedia analysis engine applies machine learning techniques to model semantic concepts in images and video from automatically extracted visual descriptors. It automatically assigns labels (with associated confidence scores) to unseen content to reduce manual annotation load and improve searching, filtering, and categorization capabilities. The MARVEL multimedia retrieval engine integrates multimedia semantics-based searching with other search techniques (speech, text, metadata, audio-visual features, etc.), and combines content-based, model-based, and text-based retrieval for more effective image and video searching [Ami03; Nat05].

The computational cost of digital media indexing (both feature extraction and concept detection), as well as the need for scalability in various dimensions of the input data, make MARVEL a real stress-case for high-performance platforms. Therefore, we have implemented MarCell (i.e, a MARVEL version ported onto the Cell/B.E. processor). We use MarCell to investigate the potential performance Cell/B.E. can provide to multi-kernel applications.

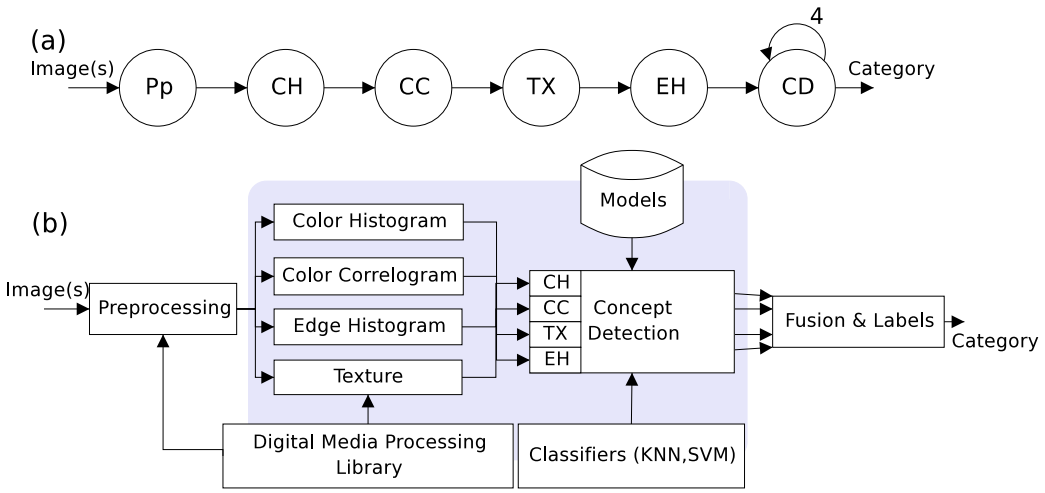


Figure 4.1: *The processing flow of MarCell. (a) The sequential flow of the original application (b) The data flow diagram. The shaded part is executed on the SPEs.*

4.1.1 From MARVEL to MarCell

MarCell is the Cell/B.E. version of MARVEL [Liu07]. In the porting process, the focus was placed on the computation intensive part of the original application, namely the image classification portion. The processing flow of this engine, presented in Figure 4.1, consists of a series of feature extraction filters performed on each image in the available collection, followed by an evaluation of these features against precomputed models.

To develop MarCell, we first adapted MARVEL to run on the PPE - an one-day programming effort, using a Linux version of MARVEL as a starting point. Next, the application was profiled to identify its most time-consuming kernels. These kernels were the best candidates to be offloaded on the SPEs for acceleration. In MARVEL, we identified five such kernels: four feature extraction algorithms and the SVM-based classification. Together, they accounted for 85-90% of the time for an average image size.

Once identified, the SPE kernels were isolated (in terms of functionality), reimplemented in C, and optimized for running on the SPEs. Our first priority was to get the highest possible performance for each kernel, focusing on core-level optimizations.

The Kernels

The preprocessing step (`AppStart`) includes the image reading, decompressing and storing in the main memory (as an RGB image). The four feature extraction techniques that have been entirely ported to run on the SPEs are: color histogram (`CHExtract`) [Smi96] and color correlogram (`CCExtract`) [Hua97b] for color characteristics, edge histogram (`EHExtract`) [Fis96], for contour information, and quadrature mirror filters for texture information (`TXExtract`) [Nap02]. The concept detection is based on an SVM algorithm, which computes a final classification coefficient using a series of weighted dot-products [Nap03]. From the resulting coefficients, the image can be properly categorized according to the given models.

Color Histogram (`CHExtract`) The color histogram of an image is computed by discretizing the colors within the image and counting the number of colors that fall into each category [Smi96]. In MARVEL, the color histogram is computed on the HSV image representation and quantized in 166 bins. It represents 8% of the total execution time per image. The result is a vector of 166 elements (integers).

Since pixels are independent for histogram construction, one can compute an image histogram for an arbitrarily large image by independently processing it in pieces. In MARVEL, the image is read (using DMAs) in fragments, in separate R, G, and B arrays. The algorithm maps each pixel to one of 166 bins based on quantized HSV values derived from the original RGB values. We were able to identify the correct HSV bin *without* doing a full RGB-to-HSV conversion, an important shortcut for saving precious SPE local memory (LS). Further, bin determination uses 8-bit SIMD operations almost exclusively. The code is written without any conditional tests, and all `for`-loops are of constant depth. As a result, the compiler is able to generate very efficient code, avoiding costly branches. Once the (partial) histogram is computed by the SPE, it is written back to global memory using a DMA transfer. The final vector is computed by the PPE with a summing reduction of all SPE-computed vectors.

To recap, the transformations we have applied to optimize the `CHExtract` kernel are: algorithm tuning (no complete RGB-to-HSV conversion), data re-organization and packing (using `char` instead of `int`), SIMD-ization, double buffering (for reading the image), memory footprint minimization, and branch replacement (using arithmetic and shifting operations instead). The speed-up versus the original kernel was in the order of 20×.

Color Correlogram - (`CCExtract`) A color correlogram expresses how the spatial correlation of pairs of colors in an image changes with distance [Hua97a]. An autocorrelogram captures the same spatial correlation between identical colors only. Informally, an autocorrelogram of an image is a list of pixel (quantized) colors and the probabilities of pixels with this same color to be found in the image at the chosen distance. In MARVEL, we extract the autocorrelogram for a single, pre-chosen distance. The correlogram extraction takes 54% of the total execution time. The result is a vector of 166 elements (single precision floats).

The code for `CCExtract` first converts the RGB image into an HSV image, where each pixel is associated with its quantized HSV value (a number between 0 and 165, as described for histograms above). Next, for each pixel P, we count how many pixels at a distance D from P have the same quantized color. The distance between two pixels is computed as

$$D(p_i, p_j) = max(|x_i - x_j|, |y_i - y_j|) \tag{4.1}$$

For MARVEL, we use $D = 8$, a value proven suitable for image indexing by other sources [Hua98]. In practice, this means that for every pixel P of color c_i, we count the pixels from the frame of a square window of size 17×17 centered in P. We add all counts for all pixels of the same color in the correlogram vector; finally, we normalize this vector using the histogram values.

In the SPE correlogram implementation, we focused on enabling SIMD and enforcing data re-use. Thus, we stored the HSV pixels as `char` values, allowing contiguous 8-bit pixel values to be loaded into a single SIMD register. Using SPU intrinsics, we were able to process most comparisons and arithmetic operations on all 8 values simultaneously. Also, we tuned the algorithm (i.e., reordered some of the additions) such that we load each "line" of 8 pixels only once, re-using it for all windows to which it contributes.

To recap, the transformations we have applied to optimize the `CCExtract` kernel are: algorithm tuning (reorder summations/counts), data re-organization and packing (using `char` instead of `int`), SIMD-ization, double buffering (for reading the image), SPU intrinsics (mainly for replacing `if` comparisons with specific SPU-intrinsics). The speed-up versus the original kernel was in the order of 20x.

Edge Histogram (`EHExtract`) The edge histogram descriptor captures the spatial distribution of edges in an image [Man98]. The RGB image is iteratively processed by a sequence of filters, resulting in two new "images" containing the edge angle and edge magnitude for each pixel of the initial image. In MARVEL, edge histogram extraction takes 28% of the total execution time per image. The result is a vector of 64 elements.

The RGB image is read in slices, using DMA. Each slice is converted from RGB to grayscale. While the RGB slice is discarded, the grayscale slice is further processed for edge detection. First, we apply the horizontal and vertical Sobel filters. These are two 3×3 convolution filters (i.e., one pixel's value is computed as a weighted sum of itself and its neighboring pixels) that generate two new images: Sobel-X and Sobel-Y, respectively. In these new images, edges are accentuated by lighter colors. Finally, the Sobel-X and Sobel-Y images are both used to compute the edge magnitude (M) and edge angle (θ). The computation is performed for each pixel 4.2.

$$\forall p \in \text{Image}, |M[p]| = \sqrt{\text{Sobel-X}[p]^2 + \text{Sobel-Y}[p]^2} \tag{4.2}$$

$$\theta[p] = arctan(\frac{\text{Sobel-X}[p]}{\text{Sobel-Y}[p]}) \tag{4.3}$$

For the SPE implementation, the first challenge was the DMA transfer of the RGB chunks. The convolution filters require successive data slices to overlap by one line, and border chunks to be padded. To simplify the address computation for the DMA accesses, we have opted to use single-line slices, only keeping three lines active for the current processing. As soon as one line processing is started, the next line can be loaded by DMA. This double buffering scheme is effective enough (i.e., the DMA latency is fully hidden by the computation), even at this line-level granularity, as the kernel is computation intensive. We have enabled the use of SPU intrinsics for adjacent pixels. Most multiplications and divisions from the original code are implemented using vectorized bit-shifts intrinsics. The code is almost completely vectorized; there are no branches and all loops are of fixed length. Finally, we compute the edge histogram (i.e., using eight "main" angle bins, each with eight magnitude bins), and upate the vectors. As in the case of the color histogram, the final vector is computed by the PPE with a summing reduction of all SPE-computed vectors.

To recap, the transformations we have applied to optimize the `EHExtract` kernel are: heavy SIMD-ization, double buffering (for reading the original image lines), extensive use of SPU intrinsics. The speed-up versus the original kernel was in the order of 18x.

Texture (`TXExtract`) A texture is a visual pattern (or a spatial arrangement) of the pixels in an image. In MARVEL, texture features are derived from the pattern of spatial-frequency energy across image subbands [Nap02]. A texture vector is obtained by computing the variances of the high frequency outputs of a wavelet filter bank in each iteration. For MARVEL, we used four iterations of the Quadrature Mirror Filter (QMF) wavelet decomposition [Aka00; Aka93]. Texture extraction takes 6% of the total execution time per image. The results is a vector of 12 elements.

Texture feature extraction involves color conversion (from RGB to grayscale), 1D QMF decomposition in both the vertical and horizontal directions, and variance computations. The part of the image to be processed by an SPE is DMA-transferred from the main memory to its local store in a sliced fashion (with the slice size equal to the length of the filter bank). Filtering first has to be performed on every column of the image slice and then it has to be performed on every row to get 4 sub-band images (LL, LH, HL, HH). As a 128-bit register can accommodate four single precision floating point numbers, the QMF filtering in the vertical direction was conducted in a 4x4 block-based fashion in order to fully utilize the SIMD feature of the SPE. With the characteristics of the selected filter bank and SIMD operation features, we were able to perform QMF decomposition using a fast implementation which derives the outputs of the high frequency QMF filtering from its low frequency outputs without multiplications. After every iteration, the variances of each of the LH, HL, and HH subband images are computed again as part of the texture vector.

To recap, the transformations we have applied to optimize the `TXExtract` kernel are: algorithm tuning (to remove unneccessary multiplications and recursion), double buffering (for reading the original image slices), extensive use of SPU intrinsics, data reorganization. The speed-up versus the original kernel was in the order of 7x (this somewhat lower speed-up is due to the lower parallelization potential of the kernel - basically, we mainly see the results of SIMD parallelization).

Concept Detection (`CDet`) For concept detection (and, implicitely, image classification), we use SVM testing. To do so, we "compare" each feature vector extracted from an image (the color histogram, the color correlogram, the edge histogram, and the texture descriptor) with a pre-trained model. For each feature, the model contains a number of feature vectors and a list of corresponding coefficients. The image vector is processed against each of the model vectors. The processing is a dotproduct of fixed-length vectors. The result is multiplied by the corresponding model coefficient; these weighted dotproducts are combined in a final similarity score. Based on this score, the image can be classified. The concept detection for one feature vector takes between 1.8% and 2.5% of the application execution time, depending on the model and vector sizes.

The transformations we have applied to optimize concept detection are: data vectorization and code SIMD-ization, and the use of SPU intrinsics. The speed-up versus the original kernel was in the order of 4x. The performance is strictly limited by the very low computation-to-communication ratio of this kernel: all model vectors have to be loaded every time, and they are used only once for each image. The only solution to alleviate this problem is to classify batches of images, but this solution was not explored here, because it is part of a different parallelization strategy.

4.1.2 MarCell: First Performance Measurements

After the application was divided into kernels, and all these kernels were fully optimized, we have performed the first performance experiments.

First, to have an idea of how the optimized kernels perform, we compared their execution times on the SPEs with those obtained by the original kernels running on the PPE and on two other reference commodity systems[2]: a desktop machine with an Intel(R) Pentium(R) D CPU (Dual Core Technology, running at 3.4GHz), from now on called "Desktop", and a laptop with an Intel(R) Pentium(R) M CPU (running at 2GHz), from now on called "Laptop". The performance numbers are presented in Table 4.1.2.

[2]The reference machines have only been chosen as such for convenience reasons; there are *no* optimizations performed for any of the Pentium processors

Table 4.1: *The speed-up factors of the SPE-enabled version of MarCell over the original MARVEL version running on the PPE.*

Kernel	PPE [ms]	Overall contribution	SPE [ms]	Speed-up versus PPE (1thread)	Speed-up versus Desktop	Speed-up versus Laptop
App start	6.81	8%	7.17	0.95	0.67	0.83
CH Extract	42.82	8%	0.82	52.22	21.00	30.17
CC Extract	325.43	54%	5.87	55.44	21.26	22.45
TX Extract	31.28	6%	2.01	15.56	7.08	8.04
EH Extract	225.80	28%	2.48	91.05	18.79	30.85
CDet	2.93	2%	0.41	7.15	3.75	4.88

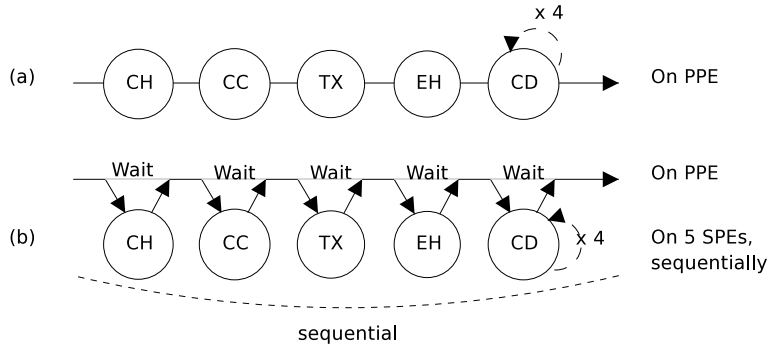


Figure 4.2: *Task parallelism: (a) The original application, with all kernels running on the PPE; (b) All kernels are running on SPEs, but they fired sequentially.*

Further core-level optimization for any of these kernels is beyond the purpose of this work: for our goals, we consider these kernel performance numbers as the reference ones, and we use them to compose and evaluate the scheduling scenarios for the complete application.

Next, we have reintegrated the fully optimized kernels in the MARVEL flow. For this first experiment, we did not alter MARVEL's control flow (for example, by processing more images with one function call). To reintegrate the optimized kernels into the main application, we used an offloading technique that allows the PPE to trigger and control the execution of the SPE code. Thus, on the PPE side, we provided an additional interface class that spawns an SPE thread and manages its execution. On the SPE side we created a template that provides the SPE-to-PPE interface (the same for all kernels), and allows for easy plug-in of the functional part of the kernel. The porting effort, mostly spent in developing the SPE-optimized algorithms, was about 4 person-months[3].

So, in this very first version of MarCell (called `SPU_seq` from now on) we statically assigned each one of the five kernels to a dedicated SPE. Further, we have preserved the control flow of the original MARVEL, as the PPE fires the kernels *sequentially*, one at a time, waiting for its completion before firing the next one. The kernels run in the same order as in the original application. The control flow of `SPU_seq` is schematically presented in Figure 4.2.

[3]For more details on the porting strategy, we redirect the reader to Chapter 6

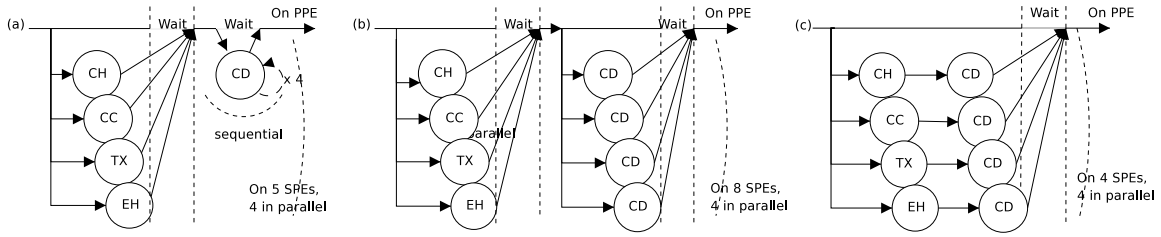


Figure 4.3: *Task parallelism scenarios: (a) SPU_par_5: CD running on one SPE; (b) SPU_par_8: CD running on 4 separate SPEs; (c) SPU_par_4: CD is combined with feature extraction kernels, running on the same 4 SPEs.*

We have run the application on two Cell/B.E. platforms - a PlayStation3 (PS3) and a QS20 blade. The PS3 has 6 SPEs available for programming, while the QS20 has 2 full Cell/B.E. processors, allowing the programmer to use 16 SPEs. For the QS20, the speed-up obtained versus the Desktop implementation was 10.6, comparable with the 10.3x obtained on the PS3. This non-significant difference is as expected: we only need 5 cores from each platform.

We emphasize again that this 10-fold increase in performance is exclusively due to the aggressive core-level optimizations we performed for each kernel. However, we claim we should be able to at least double this gain by adding application-level parallelization. We dedicate the following sections to proving this hypothesis.

4.2 Task parallelism

To exploit task parallelism for MarCell, no change is required on the SPEs side. The PPE can simply enable task parallelism by firing all four feature extraction kernels in parallel, waiting for them to complete, and then running the concept detection. Note that the concept detection for each feature cannot start before the feature extraction is completed. To preserve this data dependency, we consider three mapping options for the concept detection kernel (also seen in Figure 4.3):

1. `SPU_par_5` runs the 4 instances of `CDet` sequentially, using only one additional SPE (5 in total);

2. `SPU_par_8` runs the 4 instances of `CDet` in parallel, using 4 more SPEs (8 in total);

3. `SPU_par_4` combines each feature extraction kernel with its own `CDet`, using no additional SPEs (4 in total).

In general, to implement the task-parallel behavior, the PPE code is modified such that (1) the calls to the SPE kernels become non-blocking, (2) the busy-waiting on an SPE's mailbox is replaced by an event-handler, and (3) eventual data consistency issues are solved by synchronization. Overall, these are minor code modifications.

We implemented all three task parallel versions of MarCell and measured their execution times on both the PS3 and the QS20 blade. To evaluate their performance, we have compared the `SPU_par*` scenarios, with the `SPU_seq` scenario, and with the original sequential application running on the PPE and on both reference systems. We have performed experiments on a set of 100 images, 352 x 240 pixels each, and we report the results as the average over the entire set. The concept detection

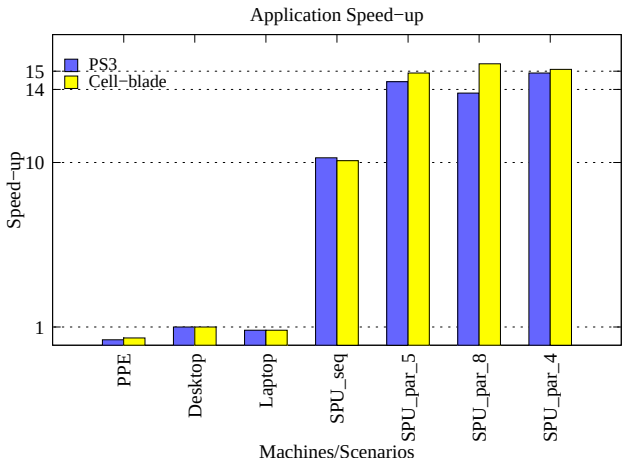


Figure 4.4: *Speed-up results of the task-level scenarios of MarCell*

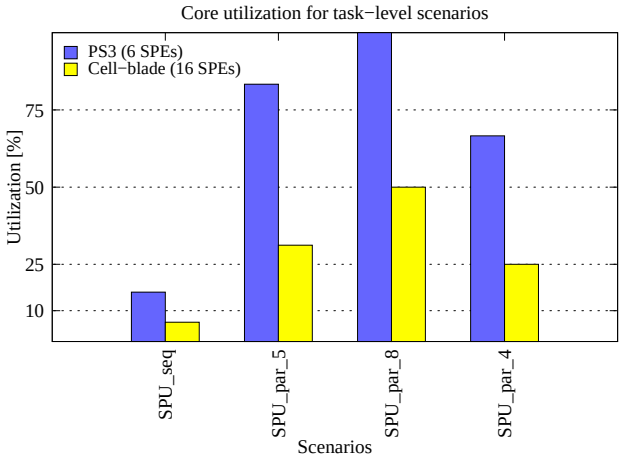


Figure 4.5: *Overall platform utilization of the task-level scenarios of MarCell.*

was performed with a number of models summing up 186 vectors for color histogram, 225 for color correlogram, 210 for edge detection, and 255 for texture.

Figure 4.4 presents the speed-up results of all these scenarios. For reference purposes only, we have normalized the speed-up with the execution time of the Desktop machine (the fastest execution of the original MARVEL code). Figure 4.5 shows the way the platform cores are utilized.

The speed-up of all parallel versions is limited by the speed-up of the slowest kernel (in this case, the color correlogram extraction, which takes more than 50% of the execution time), because the PPE waits for all features extraction to finish before launching the concept detection. The performance differences between the three `SPU_par` scenarios is small due to the low contribution of the concept detection kernel in the complete application.

Note that because the PS3 has only 6 SPEs available, the implementation of the `SPU_par_8` requires thread reconfiguration for at least two of the four SPEs involved. As a result, the application execution time increased with 20% due to thread-switching overhead. Using the Cell-blade, the same scenario runs using dedicated SPEs for each kernel, thus the performance is visibly better. Also note that combining the concept detection with each of the feature extraction kernels resulted in a slight decrease in performance for the extraction kernels. This behaviour is due to the limited amount of LS per SPE. Usually, to maximize kernel performance, programmers use most of the non-code LS space for DMA buffers. Once a new piece of code is added by kernel merging, the buffer space is reduced, and more DMA operations might be required. This was the case of MarCell. And it is a general drawback of merging two kernels on the same SPE: the performance of each is likely to drop.

We note that the overall MarCell speed-up already increased to 15x versus the same application running on the Desktop machine. However, platform utilization is still way below its potential. This is because the application tasks are too coarse-grain, thus not offering enough opportunities for proper load-balancing: if one kernel takes too long (see `CCExtract`, all the other SPEs stall waiting for this "outlier" to finish.

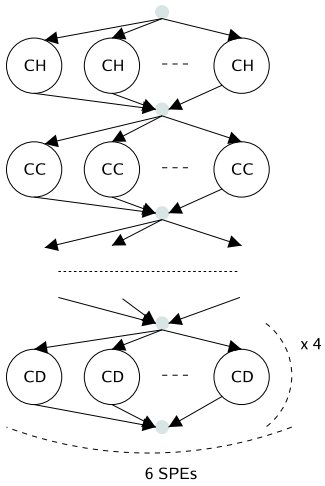


Figure 4.6: *Data parallel MarCell: each kernel is running over multiple SPEs, but no different kernels run in parallel.*

4.3 Data parallelism

As seen in the previous section, task parallelism is too coarse grained to achieve good resource utilization, thus leading to poor overall application performance. Attempting to increase core utilization on the available Cell/B.E. platforms, we have implemented a data parallel version of MarCell. In this case, each one of the kernels was parallelized to run over any given number of SPEs, performing computations on slices of the input image. The merging of the results is done by the PPE, such that the communication overhead induced by the SPE-PPE communication (i.e., the number of DMA operations, in the case of Cell/B.E.) is not increased. We compose the application as a sequence of data parallel kernels, like in Figure 4.6.

First, we evaluated the data-parallel versions of the kernels, each running in isolation on the Cell/B.E. The speed-ups we obtained for each of the data-parallel kernels are presented in Figure 4.7. Both `TXExtract` and `CCEXtract` show almost linear scalability up to 8 SPEs. This limitation is due to the input picture size - a data-set with larger pictures would move this threshold higher. The `CHExtract` kernel fails to provide better performance with the number of SPEs it uses due to its small computation-to-communication ratio. Finally, `EH_Extract` shows slightly super-linear speed-up. This weird behavior is also an artifact of DMA buffers management. When running on image slices, one kernel may execute faster due to less computation *and* smaller communication (significantly less DMA transfers), a combination that can leads to superlinear speed-up. However, even in this case, `EH_Extract` only obtains speed-up on up to 7 SPEs (again, larger images will move the threshold up).

When running the application, scheduling one kernel on all available SPEs requires thread reloading/reconfiguration, which must happen after each kernel execution, on all available SPEs (see Figure 4.6). There are two options to implement this context switch:

- **Thread re-creation**, which can accommodate kernels with bigger memory footprints, at the price of a higher execution time; the solution is not suitable for applications with low computation-

to-communication ratios. In the case of MarCell, for more than 2 SPEs, the thread switching becomes predominant in the application performance.

- **Transparent overlays**, which combine several SPE programs in an overlay scheme; the reconfiguration is done transparently to the user, who invokes the kernel without caring if it is available or not in the SPE memory. When the kernel is not available, it is automatically loaded in the SPE local store via DMA, eventually replacing the previously available kernel(s). The use of overlays leads to a more efficient context switching, but the combination of the SPEs codes in the overlayed scheme requires additional size tuning for data structures and/or code.

Figure 4.8 presents the application speed-up with the number of SPEs for the two proposed context switching mechanisms, measured on both the PS3 and the QS20 blade. We only show the measurements up to 8 SPEs because no significant performance changes appear on more SPEs. Note that despite the data-parallelized kernels having linear scalability with the number of used SPEs, the behaviour does not hold for the overall application, due to the very expensive context switching. Even worse, the performance of the thread re-starting implementation worsens much faster because the number of reconfigurations is linearly growing with the number of used SPEs.

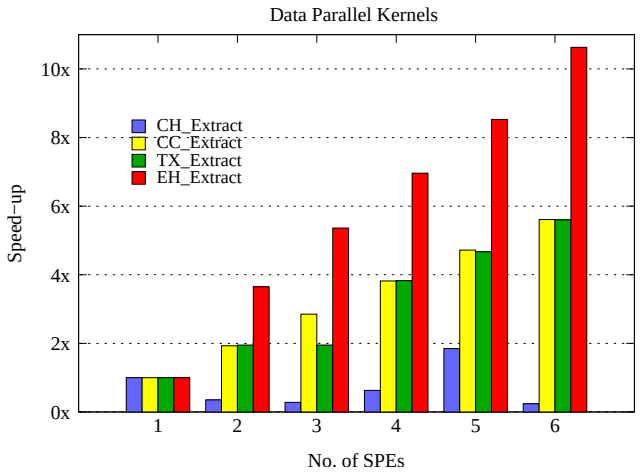


Figure 4.7: *The scalability of the data-parallel kernels.*

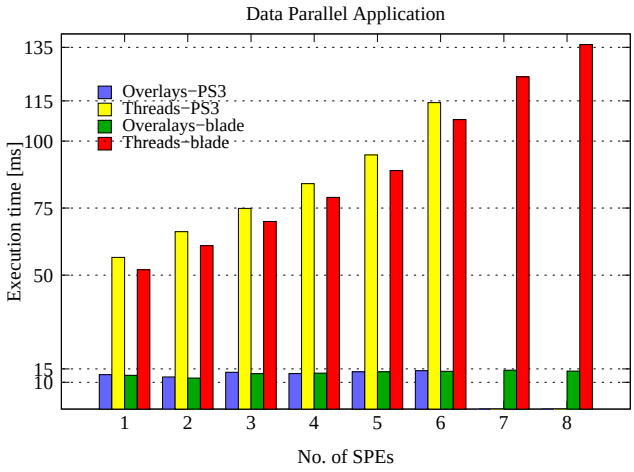


Figure 4.8: *The overall performance of the data parallel application*

From the experimental results, we conclude that data parallelization on Cell/B.E. is not efficient on its own for applications that require the kernels' code to be switched too often. Furthermore, any efficient data parallel implementation for an SPE kernel should allow dynamic adjustment of both its local data structures (i.e., LS memory footprint) and its input data size (i.e., the image slice size): when less local storage memory may be needed for a slice, more space for DMA-transferred data can be available, thus lowering the transfer overheads.

However, using data-parallelized kernels does allow lower-granularity tasks to be scheduled on the SPEs, thus offering a potential to enhance the task-parallel version of MarCell. Therefore, in the next section, we propose a hybrid parallel solution.

4.4 Combining data and task parallelism

To allow a hybrid parallelization of MarCell, we have enabled the application to combine data parallel kernels (i.e., running a single kernel on a single image over multiple SPEs) and task parallelism (i.e., running multiple kernels in parallel on different SPEs), as sketched in the application graph illustrated in Figure 4.9. Note that the mapping and scheduling of tasks on SPEs are static, and configurable at runtime.

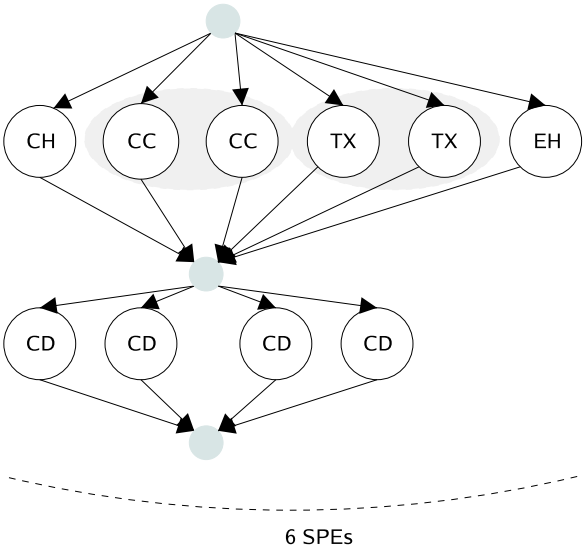


Figure 4.9: *Combining data and task parallelism in MarCell.*

Our goal is to determine the best performance the application can reach using this hybrid parallelization. Therefore, we measured the performance of all possible static mappings for each platform. To avoid expensive SPE reconfiguration, we imposed that any static mapping includes no kernel reconfiguration: one SPE is running one feature extraction kernel only, eventually followed by a concept detection operation for the same kernel.

The results of these measurements on the PS3 platform are presented in Figure 4.10. The notations of the scenarios has the form `x-y-z-w`, coding the number of SPEs used for each feature extraction, in the order CH,CC,TX,EH. For the PS3, the best mapping, with $T_{ex} = 9.58ms$, is `1-3-1-1`, which is an intuitive result given the significant percentage of the overall application time spent on the `CCExtract` kernel.

Next, we performed a similar experiment on the QS20 blade. Figure 4.11 presents the clustering of the execution times for a specific number or used SPEs (from 4 to 16); Figure 4.12 shows the best scenario when considering the number of utilized SPEs varying between 4 and 16. Note that the best mapping, with $T_{ex} = 8.41ms$, is `3-4-3-4`. Also note that this mapping is *not* utilizing all available SPEs, which shows that for the given input set, the application runs most efficiently on 14 SPEs, while a very good ratio performance/utilized cores is already obtained by `2-4-2-2`, only using 10 SPEs.

Given the overall application performance, we conclude that combining data and task parallelism on Cell is the best way to gain maximum performance for applications with multiple kernels, as seen in Figure 4.13. The best performance we have achieved is 24x better than the original MARVEL running

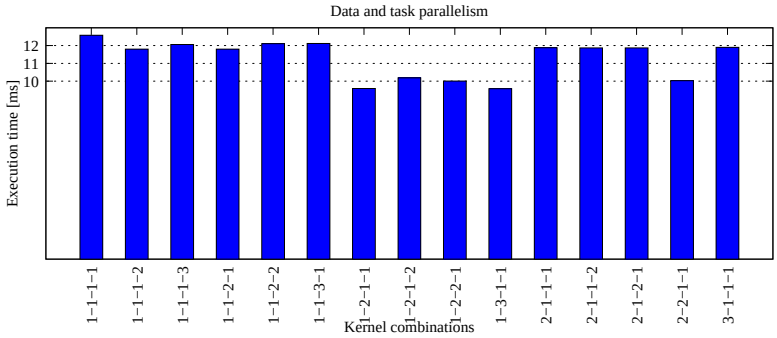


Figure 4.10: *Using combined parallelism for mapping the application on PS3*

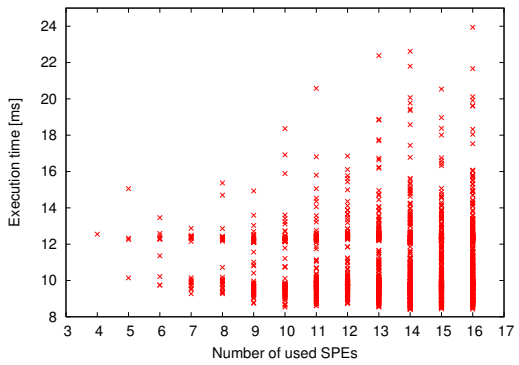


Figure 4.11: *Using combined parallelism for mapping the application on the Cell-blade*

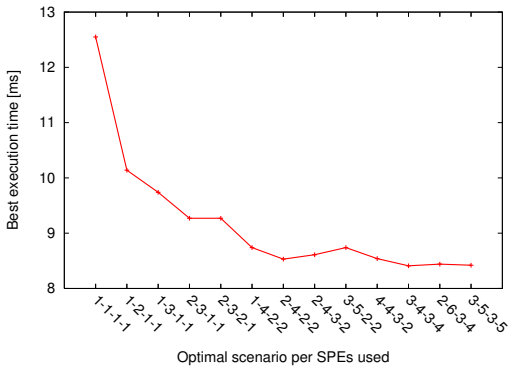


Figure 4.12: *Best scenario for a given number of utilized SPEs*

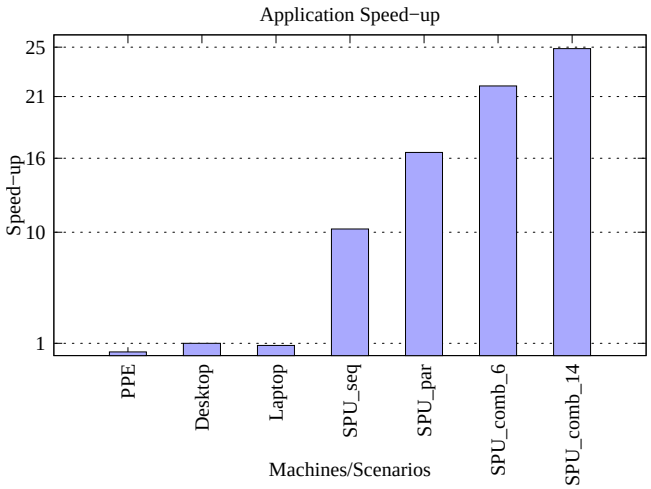


Figure 4.13: *Maximized application speed-up for each of the presented scenarios.*

on the Desktop machine. This significant performance improvement is mainly due to the incremental increase in core utilization for the entire platform. In turn, this increase was possible because we enabled the kernels to have "variable" granularity.

4.5 Lessons Learned and Guidelines

In this section we discuss the generality of our parallelization approach. We present both MarCell's and Cell's relevance in the current multicore lanscape, and we list our generic guidelines for enabling high-level performance tuning on the Cell/B.E..

4.5.1 MarCell Relevance

The code base for Cell/B.E. performance analysis is currently a sum of various ported applications, like RAxML [Bla07], Sweep3D [Pet07], real-time raytracing [Ben06], CellSort [Ged07], digital signal processing [Bad07b], MarCell, etc. Although each one of them has been analyzed individually, they can be thought of as a representatives for larger classes of potential Cell applications. Because there are no benchmark suites for heterogeneous multicores (yet), the performance analysis and results obtained by these applications provide a better understanding of Cell's performance characteristics.

MarCell is the first multimedia content analysis application running on the Cell/B.E.. It is a valid use-case for multicores, because its processing performance influences directly its accuracy. Further, MarCell provides a series of practical advantages for a mapping study on such a platform. First, the task-parallel application has a low number of *non-symmetrical* tasks. Each of these tasks is highly optimized, and the difference between the high individual speed-ups and the relatively low overall application speed-up makes the mapping investigation worthwhile. The low data dependency between the tasks allows the evaluation of several scheduling scenarios.

MarCell is an example of a series-parallel application [GE03a]. On a platform like Cell/B.E., for *any* SP-compliant application, the simplest way to increase core utilization (and, as a result, to gain performance) is to combine data and task parallelism. For more data-dependent applications, there are still two options: to parallelize them in an SP form, or to use more dynamic parallelization strategies, like job-scheduling.

4.5.2 Cell/B.E. Relevance

Combining data and task parallelism is not a new idea in parallel computing - it has been investigated before for large distributed systems (see [Bal98] for a comprehensive overview). However, the scale differences between multi-core processors and such large multi-processors are too big with respect to both the application components and the hardware resources. Thus, previous results cannot be directly applied. A new, multi-core specific investigation is required to establish (1) if such a combination is possible on these platforms, (2) how large the programming effort is, and (3) if it has any significant performance impact.

We have chosen Cell/B.E. as a target platform able to answer all these questions. None of the immediately available alternatives is suitable for such an investigation: GPU architectures are fine-grain massively data-parallel architectures with hardware schedulers, embedded multi-cores are too limited in general purpose compute resources, and generic purpose multi-cores are (so far) only homogeneous.

Despite being moved on a secondary line in terms of new generations, Cell/B.E. remains a relevant target for many types of applications [Wil06]. Further, given the predicted future of multi-core processors [Hil08; Asa09a], as well as the current accelerator-based platforms, heterogeneous machines with user-based scheduling are going to be a constant presence in the HPC world. Our MarCell case-study shows that applications running on such platforms *must* take into account mapping and scheduling as potential performance boosters, and task granularity as an important performance parameter.

4.5.3 Cell/B.E. Parallelization Guidelines

What makes programming on the Cell/B.E. difficult is the complexity of the architecture combined with the five parallelization layers the programmer needs to exploit [Pet07]. In many cases, the choice is guided only by trial-and-error or by experience. As our focus with MarCell is on application-level parallelization, we synthesize our experience in four empirical guidelines. We believe that following these "rules" should enable any application parallelization for the Cell/B.E. to efficiently use, combine, and optimize data and task parallelism.

1. Focus on the problem parallelism, not on platform resources, when analyzing its maximum degree of concurrency. First look for *task parallel kernels*, to be executed in an MPMD fashion. Next, for each task, investigate data parallelism and its minimum granularity (e.g., for an image, it can be a pixel, a line, or a block). The application has to be explicitely decomposed down to the level of its smallest *non-parallel computation units*. Each such unit, characterized by *both* its computation and its data, can be scheduled in parallel or in a sequence with other units, but never further decomposed. The algorithm is then re-built by eventually aggregating and mapping these units on the available cores. For MarCell, the kernels running on slices as small as one image line were the scheduled computation units.

2. For each kernel, apply core-level optimizations. If needed, readjust granularity. Ideally, this step would be performed by the compiler, but for most platforms, current compiler technology is still far from what manual optimizations can achieve. This step typically leads to a significant performance gain, but it is also the most time consuming optimization step. For MarCell, it took 2.5 person-months for the five kernels.

3. Decide on a scheduling policy. This can be a static one, useful when the number kernels is comparable with the number of available cores. However, if the number of (asymmetric) tasks is a lot larger than the number of available cores, a dynamic scheduler is required. Further, make all scheduling decisions to be taken on the PPE, distributing work and data either to maximize a static metric or following a dynamic strategy (FCFS, round-robin, etc.). The PPE should map data-dependent kernels on the same core, to minimize data copying. For MarCell, which has less kernels than Cell/B.E. has cores, we have used static mapping.

4. Evaluate all promising mapping scenarios. Consider, however, using a simple predictor (like Amdahl's law or the Roofline model [Wil09b]) to remove the ones that stand little chance of good performance before implementing and benchmarking them. One can also put additional mapping constraints (like, for example, use the minimum number of SPE cores). Finally, to evaluate and eventually compare the remaining, promising solutions, instantiate and execute them.

4.6 Related Work

Most of performance evaluation studies for Cell/B.E. applications aim to prove the very high level of performance that can be achieved on the architecture, and focus on the algorithmic changes and iterative optimizations on the SPE side [Gsc06].

For example, the first application that emphasized the performance potential of Cell/B.E. has been presented by IBM in [Min05], where a complete implementation of a terrain rendering engine is

shown to perform 50 times faster on a 3.2GHz Cell machine than on a 2.0GHz PowerPC G5 VMX. Besides the impressive performance numbers, the authors also present a very interesting overview of the implementation process and task scheduling, but no alternatives are given. Another interesting performance-tuning experiment is presented in [Ben06], where the authors discuss a Cell-based ray tracing algorithm. In particular, this work looks at more structural changes of the application and evolves towards finding the most suitable ray tracing algorithm on Cell. Finally, the lessons learned from porting Sweep3D (a high-performance scientific kernel) onto the Cell processor, presented in [Pet07], were very useful for developing our strategy, as they pointed out the key optimization points that an application has to exploit for significant performance gains. Again, none of these two performance-oriented papers look at the coarser grain details of resource utilization.

While core-level optimizations lead to very good results [Liu07], the overall application performance is many times hindered by the mapping and/or scheduling of these kernels on the Cell/B.E. platform resources [Var07]. A similar intuition with the one we had when optimizing MarCell forms the main claim of the work presented in [Bla07; Bla08]. Here, the authors take a task-parallel application and change the granularity of the parallelization on-the-fly, based either on previous runs or runtime predictions. The results of this multi-grain parallelization are similar to ours, showing how combining two different approaches significantly increases application performance. Still, due to the specific applications used as case-studies, the granularity at which the scheduling decisions are taken is significantly different.

To the best of our knowledge, this is the first study that focuses on application-level parallelization on the Cell, and proves the high performance impact that task granularity, as well as application mapping and scheduling have on overall performance. For sure, this result will attract more research on dynamic mapping and scheduling for the Cell/B.E., as well as more performance studies on runtime systems and/or programming models such as CellSs [Bel06] or Sequoia [Fat06].

4.7 Summary and Discussion

Most parallel applications that have been successfully ported onto multi-core processors are mono-kernel applications. This allows the focus to be placed on low-level parallelization and core-level optimizations. However, for multi-kernel applications, these optimizations are far from enough.

In this chapter, we have presented a performance case study of running multi-kernel applications on Cell/B.E.. For our experiments, we have used MarCell, a multimedia analysis and retrival application which has five computational kernels, each one of them ported and optimized to run efficiently on the SPE. Our measurements show that by combining these core-level optimizations, MarCell obtained an overall speed-up of more than 10x, even when only using the SPEs sequentially.

To prove that application-level parallelization plays an equally important role in the overall application performance, we have tested both task and data parallelization of the application, by either firing different tasks in parallel (MPMD) or by using all available SPEs to compute each kernel faster (SPMD). The results showed how task parallelization leads to a speed-up of over 15, while the data paralleization worsens performance due to a high SPE reconfiguration rate.

Last but not least, we have shown how various scheduling scenarios for these kernels can significantly increase the overall application speed-up. In order to detect the best scenario, we have performed a large number of experiments, evaluating all possible schedules without SPE reprogramming. For our two Cell-based platforms, we have found the best speed-up factors to be over 21 for the PS3 (6 SPEs available, 6 used) and over 24 for the Cell-blade (16 SPEs available, 14 used).

For parallel programmers, our MarCell case-study is useful for three important directions. First, we have proved how task parallelism can be a good source of speed-up for independent kernels, but we have also shown how a significant imbalance in the execution times of some of these kernels alters overall performance. Second, we have shown how data parallelism can tackle the problem of too coarse task granularity by allowing for more flexibility, but can become inefficient due to the very high overhead of context switching (SPE reprogramming, in the case of the Cell/B.E.). Finally, we have shown how a solution combining the two approaches - data and task parallelism - makes use of the advantages of both, allowing for increased parallelism and flexibility. However, this reconfigurability potential does come at the price of increased programming effort, because all kernels need to be programmed to accept variable I/O data sizes and local structures, local memory management becomes more difficult, and the main thread needs to be able to run various combinations of these kernels (and have the resources to support their compute rate).

Alternative architectures

Parallelizing MARVEL on other multi-core processors is simplified, for both general-purpose multi-cores (GPMCs) and graphical processing units (GPUs), by two architectural features: the shared memory and the already available thread scheduling. Shared memory simplifies the implementation of the data parallelism for all kernels, as well as the concept detection phase. The kernel scheduling and mapping are solved "automatically" by the OS scheduler (for GPMCs) or by serializing the kernel execution (for GPUs).

A port of MARVEL onto GPMCs is, at the highest level, similar to MarCell: images are processed in slices/blocks by threads, which are mapped on cores. However, the size of the blocks is more flexible (i.e., can be a lot larger, given the large memory the cores share), and the parallelization can be done using an arbitrary number of threads. However, for a typical desktop-like GPMC, the programmer has a lot less control over thread mapping and scheduling, which is done by the OS. Therefore, precise experiments like the ones performed for the Cell are quite difficult to perform. We did implement a multi-threaded version of MARVEL, and measured its performance. In terms of kernel scalability, the results were similar: kernels' performance scales with the number of threads allocated for processing them. Furthermore, overall performance scales well with picture size. The overall application performance gain of MARVEL using 8 threads on a 4-core dual-threaded multi-core is about 4x when compared with the single-threaded application running on the same machine. For 16 threads, we observe a performance increase of about 20%, which is due to the OS scheduler and its ability to "fill" stall thread intervals with useful work from other threads. Note that for the GPMC version of MARVEL we have implemented no low-level optimizations - the kernels are the same as in the original code. We expect that applying vectorization, aligning images, and dimensioning picture slices to fit properly into caches would add an extra 30% to the overall application performance gain.

In the case of GPUs, a similar investigation of high-level mapping and scheduling is typically resumed to data parallelism. This is equivalent to our "pure" data-parallel scheduling, in which each kernel is parallelized to use all cores, and the kernels are executed in a sequence. As GPUs are starting to allow task-level parallelism, multi-kernel applications will most certainly benefit from high-level parallelization on these platforms as well. And in this case, the proper dimensioning of multiple kernels to maximize hardware utilization will become essential for GPUs as well. Note, however, that changing the GPU such that the scheduling of different tasks on different multi-processors becomes the concern of the user, then GPUs will become, at the conceptual level, a shared-memory generation of Cells.

For the current GPUs, we expect the performance on the kernels side (especially for large images) to easily exceed that reached by the Cell/B.E. (we consider here a GPU-specific, low-grain parallelization, where each GPU kernel will process 1, 4, or at most 8 pixels from the output). However, the sequential execution of the kernels, will compensate this gain. We expect that the overall performance will be slightly better (within 15-20%) than the best Cell/B.E. performance.

Chapter 5

Case-Study 3: A Data-intensive Application[‡]

In this chapter we present a real-life, data-intensive application, and we show the challenges it imposes on a multi-core platform. We show how these problems can be solved by (1) separating the control flow from the data flow, (2) applying dynamic mapping, and (3) using the I/O data-set characteristics.

We have already shown that application implementation for multi-cores requires multiple layers of parallelism and iterative, machine-specific optimizations in order to come close to peak performance (see Chapters 3 and 4). But this is not always enough. As multi-cores are essentially built for number crunching, compute-intensive applications (i.e., applications that use little I/O data, and run multiple computations) are the typical multi-core workloads [Pet07] to show excellent performance results. For data-intensive applications (i.e., applications that have a low computation-to-communication ratio), things are more complicated: data starvation stalls the processor, leading to rapid performance degradation. For good performance results, data-intensive applications require significant algorithmic changes and data reorganization, aimed at increasing data reuse and core utilization, thus (hopefully) closing the gap between observed and peak performance. In fact, data-intensive workloads are not (yet) a good match for multi-cores.

Yet many data intensive applications emerge from non-traditional high-performance computing (HPC) areas. One example is radioastronomy, where a large part of current research focuses on building larger radio telescopes with better resolutions. Projects like LOFAR [Sch04], ASKAP [Cor04b], or SKA [R.T07] aim to provide highly accurate astronomical measurements by collecting huge streams of radio synthesis data, which are further processed in several stages and transformed into high-resolution sky images. When designing and deploying this data processing chain, radio astronomers have quickly reached the point where computational power and its efficient use is critical. For example, it is estimated that LOFAR can produce over 100TB/day [Sch04]; for SKA, which has about 1500 times more antennas, the number will increase with at least 5 orders of magnitude.

Radio-astronomy applications typically analyze, combine, compress, or filter large streams of data (a radio-astronomy primer is briefly included in Section 5.1). The implementation of these data-intensive kernels on multi-core architectures poses multiple challenges and raises interesting questions on the potential performance. Therefore, in this chapter, we focus on evaluating the match between a radio-astronomy application and a multi-core platform. Specifically, we investigate the implementation and optimization of gridding/degridding, two of the most time-consuming radioastronomy imaging kernels, on the Cell/B.E. processor.

Cell/B.E. [Kah05; Hof05] is a heterogeneous multi-core with nine cores: a control core (PPE) coupled with eight lightweight independent processing units (SPEs) (for more details, see 2.4.1).

Data gridding and degridding, our target kernels, are based on convolutional resampling, a basic

[‡]This chapter is based on work previously published in Euro-Par 2008 ([Var08b]) and in Scientific Programming 17(1-2), 2009 ([Var09a]). This chapter supercedes both previous publications; minor revisions, and additional considerations on different multi-core architectures have been added.

processing block that dominates the workload of image synthesis in radioastronomy. Optimizing this kernel has a direct impact on the performance and hardware requirements of any of modern large radiotelescope [Cor04b]. A thorough analysis of the gridding and degridding kernels, presented in Section 5.2) shows the data-intensive nature of the kernels, as well as their specific computational requirements and memory access patterns.

Starting from the application analysis, we present a model-driven approach on how a memory-bound data-intensive application can be parallelized on the Cell/B.E.. The implementation and subsequent optimizations posed interesting challenges, especially related to proper load balancing and data distribution (details can be found in Section 5.3). Despite these challenges, the performance we have achieved (see Section 5.4 for an overview of the results) proves two important points: (1) our design and implementation strategy are effective in tackling a data-intensive applications on the Cell/B.E. processor, and (2) the solution is suitable, in terms of computation and scalability, for a large, real-life radiotelescope.

Furthermore, our successful solution can be easily generalized to an empirical strategy. We express this strategy as a set of ten guidelines (see Section 5.5) useful for the parallelization of data-intensive applications on multi-core processors.

5.1 Radioastronomy Imaging

One of the radioastronomy goals is to obtain accurate images of the sky. In this section, we introduce the fundamental concepts and terminology required for the reader to understand the background of our application as well as the execution context of our target application.

5.1.1 Radio Interferometry

Radio interferometers are a solution for obtaining high resolutions for the sky images that would otherwise require reflectors of impractically large sizes. Being built as arrays of connected radiotelescopes (with various antenna types and placement geometries), and using the aperture synthesis technique [Tho01], they are able to work as a single large "combine" telescope. However, this solution comes at the price of additional computation, as radio interferometers do not measure the brightness of the sky directly; instead, each pair of antennas measures one component of it. Combining these components into a single image requires significant postprocessing.

Each pair of antennas defines *a baseline*. Increasing the number of different baselines in the array (i.e., varying the antenna numbers and/or placement) increases the quality of the generated sky image. The total number of baselines, B, in an array of A antennas is $B = A \cdot (A-1)/2$, and it is a significant performance parameter of the radiotelescope. For example, the LOFAR radiotelescope [Sch04] has a total of $B = 1830$ baselines, while SKA [R.T07] should have approximately 1500 times as many.

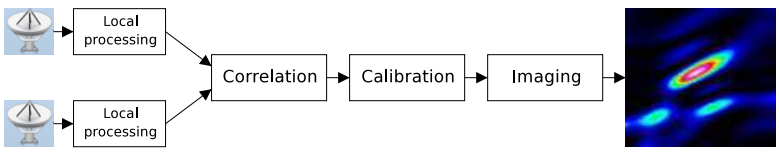


Figure 5.1: *The software pipeline from antennas to sky images*

The simplified path of the signal from each baseline to a sky image is presented in Figure 5.1. The signals coming from any two different antennas have to be correlated before they are combined.

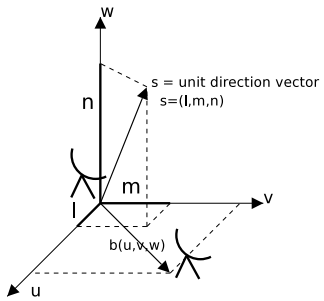


Figure 5.2: *Baseline and measurement direction*

A correlator reads these (sampled) signals and generates a corresponding set of *complex visibilities*, $V_{b,f,t}$, one for each baseline b, frequency channel f, and moment in time t. In other words, taking measurements for one baseline with a sampling rate of one sample/s for 8 straight hours will generate a set of $8 \cdot 3600$ visibilities for each frequency channel (tens to hundreds of them) and each polarization (typically, 2 or 4 in total). For example, using 256 frequency channels (a small number) for LOFAR and its 1830 baselines may lead to about 13.5 GB of data.

Figure 5.2 shows the relationship between the baseline vector, **b**, in the measurement plane defined by its (u, v, w) coordinates, and the direction of the radiation source of interest in the sky, **s**, defined by the (l, m, n) coordinates. Note that (l, m, n) are the projections of the unit direction vector **s**, of the studied object on the (u, v, w) plane of the baseline.

Imaging is the process that transforms the complex visibilities into accurate sky images. In practice, building a sky image translates into reconstructing the sky brightness, I (at a given frequency), as a function of some angular coordinates, (l, m), in the sky. The relationship between the sky brightness, $I(l, m)$, and the visibility $V(u, v, w)$ is given by Equation 5.1. Note that the values (u, v, w) are the components of the baseline vector between the two interferometer elements, and are expressed in units of wavelength of the radiation [Ren97]. For the case when $w = 0$ (e.g., for a *narrow field of view*, i.e., all baselines are in the same (u, v) plane), we obtain the two-dimensional form of the visibility-brightness relationship, as given in Equation 5.2; this equation shows that $V(u, v)$ and $I(l, m)$ are a Fourier pair. Thus, we can compute $I(l, m)$ as the Fourier inverse transform of $V(u, v)$, as shown in Equation 5.3; for the image reconstruction, we use the discrete transform (shown in Equation 5.4) on the observed (sampled) visibilities. For the more general case of non-coplanar baselines (i.e., $w \neq 0$, also known as *wide field of view*), there are two main approaches: (1) use the *faceted approach*, where we approximate the wide field as a sum of a lot of narrow fields for which the simplification $w = 0$ holds, compute the FFT for each one of these *facets*, and combine the results [Cor92], or (2) use the W-projection algorithm [Cor04a], which computes the FFT for projections of the baseline vector and the observed source vector on a (fairly small) number of planes (P_w) parallel to the (u, v) plane, and sums the results. For similar accuracy, P_w is much smaller than the potential number of facets; therefore, the W-projection method is considered more computationally efficient, and it is the one we

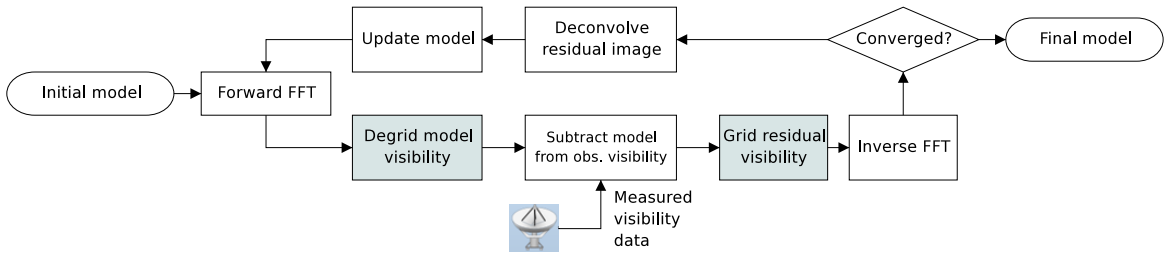


Figure 5.3: *A diagram of the typical deconvolution process in which a model is iteratively refined by multiple passes. The shaded blocks (gridding and degridding) are both performed by convolutional resampling.*

use in this case-study.

$$V(u,v,w) = \int \frac{I(l,m)}{\sqrt{1-l^2-m^2}} e^{-2\pi i\left(u\cdot l+v\cdot m+w\cdot\sqrt{1-l^2-m^2}\right)} dldm \tag{5.1}$$

$$V(u,v) = \int \frac{I(l,m)}{\sqrt{1-l^2-m^2}} e^{-2\pi i(u\cdot l+v\cdot m)} dldm \tag{5.2}$$

$$I(l,m) = \sqrt{1-l^2-m^2} \cdot \int V(u,v) e^{+2\pi i(u\cdot l+v\cdot m)} dudv \tag{5.3}$$

$$I_d(l,m) = \frac{1}{N} \sum_{t=1}^{N} V_d(u_t,v_t) \cdot e^{+2\pi i(u_t\cdot l+v_t\cdot m)} \tag{5.4}$$

5.1.2 Building the Images

The computational process of building a sky image has two phases: imaging and deconvolution. The imaging phase generates a *dirty image* directly from the measured visibilities, using FFT. The deconvolution "cleans" the dirty image into a *sky model*. A snapshot of this iterative process is presented in Figure 5.3. Note that the gridding and degridding operations are the main targets for our parallelization, due to their time-expensive execution (it is estimated that 50% of the time spent in the deconvolution loop is spent on gridding and degridding).

Note that the measured visibilities are sampled along the uv-tracks (i.e., the trajectory that a baseline defined by (u,v) generates due to the Earth's rotation). Before any FFT operations, data has to be placed in a regularly spaced grid. The operation used to interpolate the original visibility data to a regular grid is called *gridding*, and it is performed by using a support kernel that minimizes the aliasing effect in the image plane[1]. *Degridding* is the "reverse" operation used to extract information from the grid and project it "back" to the uv-tracks. Degridding is required when new visibility data is used to refine the current model. A simplified diagram of the gridding/degridding is presented in Figure 5.4. For the remainder of this paper, we focus on gridding and degridding, as they represent the most time-consuming parts of the sky image building software chain.

[1]Explaining how these coefficients are computed is far beyond the purpose of this paper. Therefore, we redirect the interested readers to the work of Schwab in [Sch84]

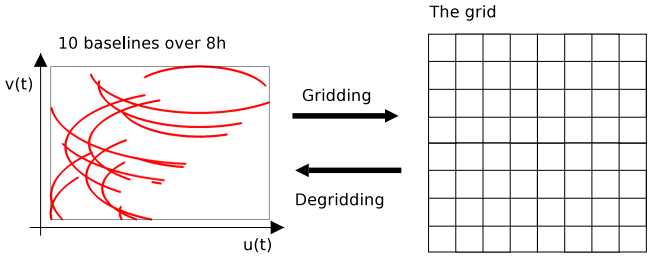


Figure 5.4: *A simplified diagram of the gridding/degridding operations. The data sampled along the (u,v) tracks in the left-hand side is gridded on the regular grid. The data in the grid is degridded along the same (u,v) tracks*

5.2 Gridding and Degridding

In this section we analyze the characteristics of gridding and degridding, focusing on their computation and data access patterns. Further, we build simple models for each of the two applications, models that shall be further used as starting points for developing the Cell/B.E. parallel application.

5.2.1 Application Analysis

For one high-resolution sky image, data is collected from B baselines, each one characterized by $(u, v, w)_{b,t}$, i.e., its spatial coordinates at moment t. During one measurement session, every baseline b collects $N_{samples}$ for each one of the observed N_{freq} frequency channels. These samples, $V[(u, v, w)_{b,t}, f]$, are called "visibilities".

Data and Computation

Gridding is the transformation that interpolates the irregular visibilities $V[(u, v, w)_{b,t}, f]$ on a $2^g \times 2^g$ regular grid, $\mathbf{G}$. Each visibility sample is projected over a subregion SG of $m \times m$ points from this grid. To compute the SG' projection, the visibility value is weighted by a set of $m \times m$ coefficients, called a support kernel SK, which is extracted from a matrix that stores the convolution function, $\mathbf{C}$. Note that this convolution function is *oversampled*, i.e., its resolution is finer than that of the original sampling by a factor of $os \times os$. Equation 5.5 synthesizes this computation. The SG' projection is finally added to the SG region of the grid, updating the image data. All data from all baselines is gridded on the same image grid, $\mathbf{G}$. The size of the projection, $m \times m$, is a parameter calculated based on the radiointerferometer parameters, and it is anywhere between 15×15 and 129×129 elements. Degridding is the operation that reconstructs the $V'(u, v, w)_{b,t}$ samples from the image grid by convolving the corresponding support kernel, SK, and grid subregion, SG. Equation 5.6 synthesizes this computation.

Figure 5.5 illustrates the way both gridding and degridding work, while listing 5.1 presents a simplified pseudo-code implementation.

$$\forall 0 \leq x \leq (m \cdot m), SG'(g_{offset}(u_k, v_k) + x) = \mathbf{C}(c_{offset}(u_k, v_k) + x) \cdot V(u_k, v_k) \tag{5.5}$$

$$V'(u_k, v_k) = \sum_{x=1}^{m \cdot m} ((\mathbf{C}(c_{offset}(u_k, v_k) + x) \times G(g_{offset}(u_k, v_k) + x) \tag{5.6}$$

Listing 5.1: *The computation for the gridding/degridding*

```
1 forall (b=1..N_baselines, t=1..N_samples, f=1..N_freq)  // for all frequency channels
2   compute c_index=C_Offset((u,v,w)[b,t], freq[f]); // the SK offset in C
3   SC(b,t,f) = extract_support_kernel(C, c_index, m*m); // get the mxm coefficients
4   compute g_index=G_Offset((u,v,w)[b,t], freq[f]); //the SG offset in G
5   SG(b,t,f) = extract_subregion_grid(G, g_index, m*m); // get the mxm grid data
6   for (x=0; x<(m*m); x++)   //sweep the convolution kernel
7      if (gridding)    {SG'[x]=C[x]*V[(u,v,w)[b,t],f];SG[x]+=SG'[x]; }
8      if (degridding) {V'[b,t,f]+=SG[x]*C[x];}
```

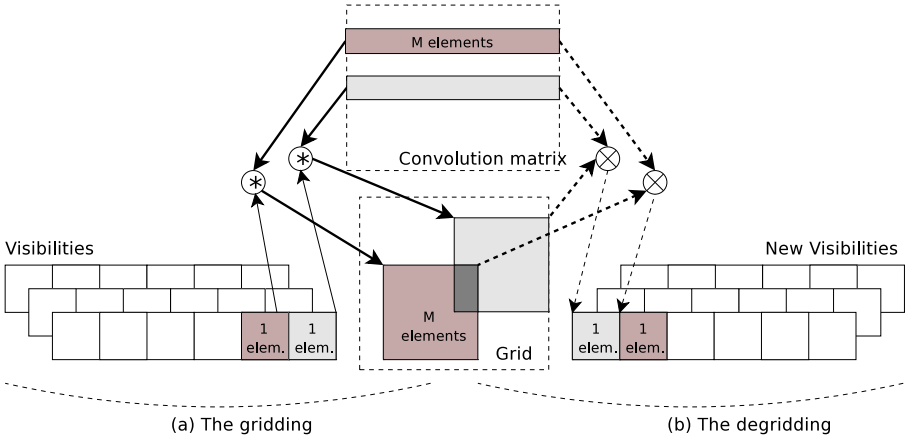


Figure 5.5: *The two kernels and their data access pattern: (a) gridding (continuous lines) and (b) degridding (dotted lines). Similarly shaded regions are for a single visibility sample. For gridding, newly computed data that overlaps existing grid values is added on. For degridding, newly computed visibilities cannot overlap.*

The `c_index` stores the offset inside **C** from where the current convolution kernel, $SK_M(b,t,f)$ is extracted; the `g_index` stores the offset of the corresponding grid subregion, $SG_M(b,t,f)$, in **G**. These two indices are computed for each time sample, baseline and frequency channel, thus corresponding to each visibility $V[(u,v,w)_{b,t},f]$.

The essential application data structures are summarized in Table 5.1. To give an idea of the application memory footprint, consider the case of $B = 1000$ baselines, $SK_M = 45 \times 45$ elements (equivalent of a 2000m maximum baseline), $os = 8$ (a common oversampling rate) and $P_w = 33$ (the number of planes in the W-projection algorithm). Then, **C** has over 4 Melements of data, taking about 34MB. Further, for a sampling rate of 1 sample/s, an 8 hour measurement session generates 28800 samples on each frequency channel for each baseline; for only 16 frequency channels and 1000 baselines, we have over 450M visibilities, which take about 3.5GB of memory/storage.

Data Locality

Low arithmetic intensity and irregular memory accesses are non-desirable features for parallel applications: the first because it does not provide enough computation to hide the communication overhead, and the latter because data locality is very low and it has a high negative impact on the actual bandwidth the application can achieve. Such application features are even worse for multi-core processors,

Table 5.1: *The main data structures of the convolutional resampling and their characteristics. Note that B is the number of baselines.*

Name	Symbol	Cardinality	Type	Access pattern
Coordinates/baseline	(u, v, w)	$B \times N_{samples}$	`Real`	Sequential
Visibility data	V	$B \times N_{samples} \times N_{freq}$	`Complex`	Sequential
Convolution matrix	$\mathbf{C}$	$m^2 \times os^2 \times P_w$	`Complex`	Irregular
Grid	$\mathbf{G}$	512×512	`Complex`	Irregular
Support Kernel	SK_M	$M = m \times m$	`Complex`	Sequential
Subregion Grid	SG_M	$M = m \times m$	`Complex`	sequential

where their negative impact on application performance is increased [But07b].

As both gridding and degridding are memory intensive applications (the arithmetic intensity is slightly less than 1/3), locality plays a significant role in their performance. To analyze data locality and its re-use potential, we trace the memory accesses in both $\mathbf{C}$ and $\mathbf{G}$. The access patterns are generated by the `c_index` and `g_index` offsets. Based on their computation as non-linear functions of the $(u, v, w)_{b,t}$ and f - see Listing 5.1, lines 2 and 4, we can conclude the following: consecutive time samples on the same baseline may not generate consecutive $\mathbf{C}$ and $\mathbf{G}$ offsets: `c_index(b,t,f)` $\neq$ `c_index(b,t+1,f)` and `g_index(b,t,f)` $\neq$ `g_index(b,t+1,f)`, which means that the corresponding support kernels and/or grid subregions may not be adjacent. The same holds for two measurements taken at the same time sample t on different consecutive channels. As a result, we expect a scattered sequence of accesses in both $\mathbf{C}$ and $\mathbf{G}$.

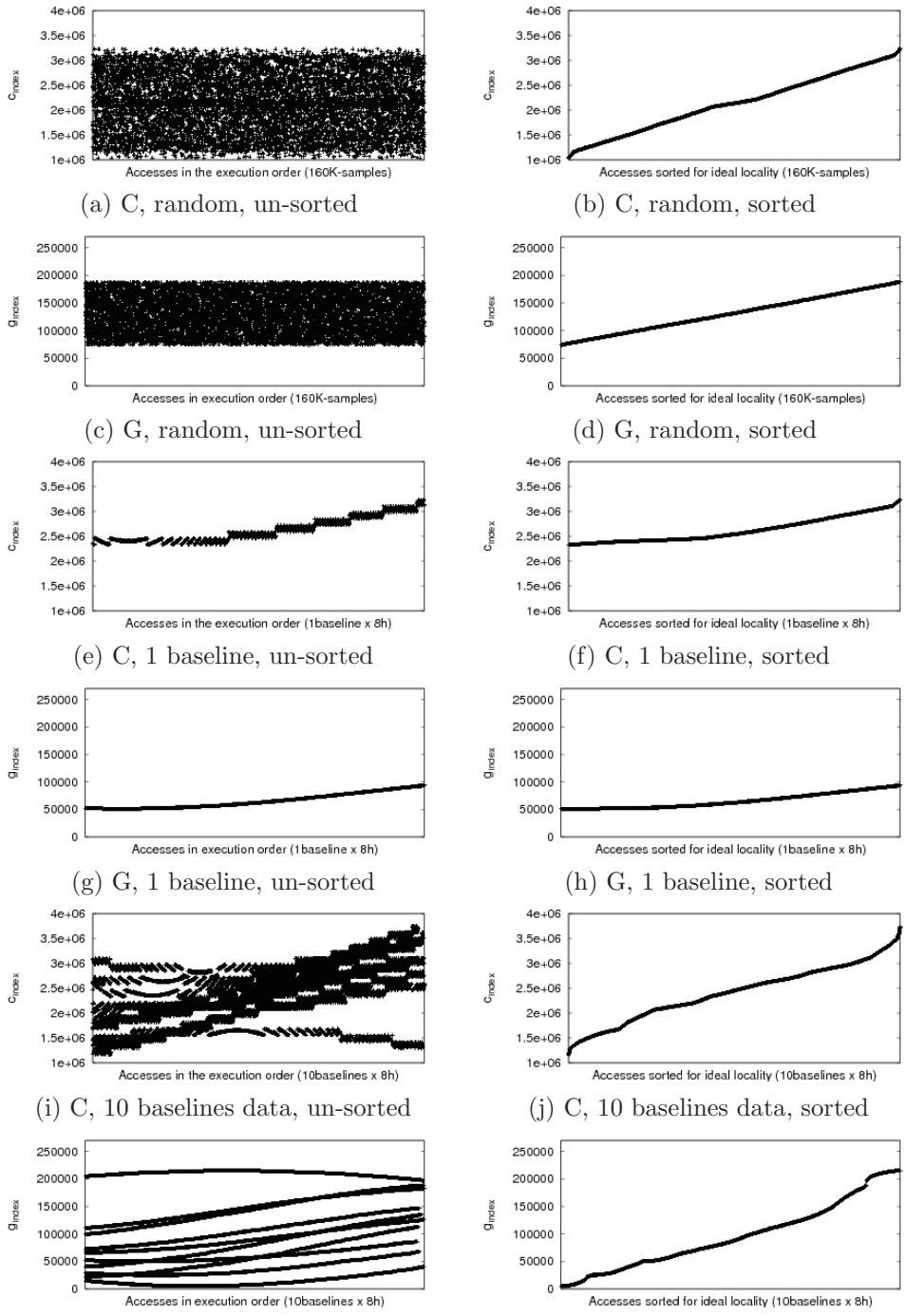
We have plotted these access patterns for three different data sets, and the results can be seen in Figure 5.6. The left-hand side graphs show the accesses in the execution order (i.e., ordered by their time sample). The right-hand side graphs show the results for the same data, only sorted in increasing order of the indices, therefore indicating the *ideal* data locality that the application can exploit. For these "sorted" graphs, note that the flatter the trend is, the better locality is. For random data, i.e., randomly generated (u, v, w) coordinates, the memory accesses are uniformly distributed in both $\mathbf{C}$ and $\mathbf{G}$, as seen in Figures 5.6(a),(c). Note that the more scattered the points are, the worse the data locality is. Next, we have used a set of (u, v, w) coordinates from a real measurement taken from a single baseline. Note, from Figures 5.6(e),(g), the improved locality, which is due to the non-random relationship between $(u, v, w)_{b,t}$ and $(u, v, w)_{b,t+1}$ [2]. Also, accesses from measurements taken from ten baselines are plotted in Figures 5.6(i),(k). The potential for data-reuse is similar for $\mathbf{C}$, and somewhat increased in $\mathbf{G}$. Finally, Figures 5.6(b),(d),(f),(h),(l) (the sorted graphs) indicate that *spatial* data locality does exist in the application, but exploiting it requires significant data pre-processing for *all* (u, v, w) samples.

5.2.2 Application Modeling

The task graphs for both kernels are presented in Figure 5.7. Note the similar structure: a main loop of $N = N_{baselines} \times N_{samples} \times N_{freq}$ iterations, whose core is a sequence of inner tasks. So, we can model the two kernels as follows (we use a simplified version of the PAMELA notation [van93]):

```
Gridding = G1;for(i=1..N){G2;G3;G4;G5}
```

[2] The $(u, v, w)_{b,t+1}$ can be computed from the $(u, v, w)_{b,t}$ coordinates by taking into account the Earth rotation in the interval between the two samples.



(a) C, random, un-sorted (b) C, random, sorted

(c) G, random, un-sorted (d) G, random, sorted

(e) C, 1 baseline, un-sorted (f) C, 1 baseline, sorted

(g) G, 1 baseline, un-sorted (h) G, 1 baseline, sorted

(i) C, 10 baselines data, un-sorted (j) C, 10 baselines data, sorted

(k) G, 10 baselines data, un-sorted (l) G, 10 baselines data, sorted

Figure 5.6: *Access patterns in **C** and **G**, with various data and data-sizes, original and sorted (on the Y axis we plot the indexes of the elements accessed in the target matrix). For the original data, darker regions (i.e., more points) signal good temporal and spatial data locality. For the sorted data, the flatter the curve, the better the spatial locality over the entire data set is.*

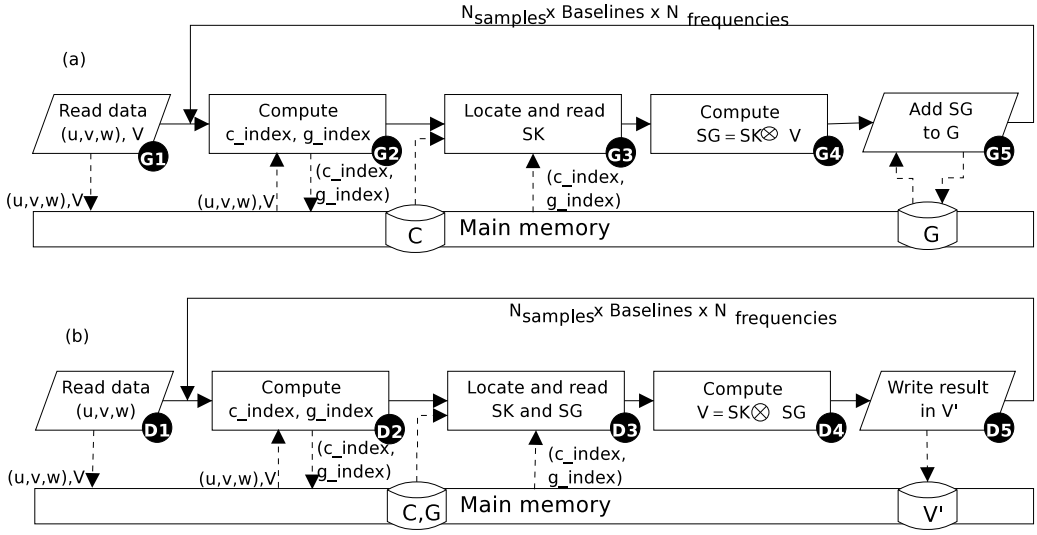


Figure 5.7: *The execution phases of both kernels: (a) gridding, (b) degridding.*

```
Degridding = D1;for(i=1..N){D2;D3;D4;D5}
```

We characterize each of these tasks by a set of parameters that will be used to guide our parallelization decisions. For each task, we focus on the following properties:

- Input data: the type and number of the elements required for one execution;
- Output data: the type and number of the elements produced by one execution;
- Computation:Communication: a qualitative estimation (i.e., from very low to very high) of the ratio between the data communication needs of the kernel (e.g., the time it takes to get the data in/out) and the time spent computing the data. Note that data size can have a significant impact on this parameter.
- I:O ratio: the ratio between the size of the input and output data
- Memory footprint: the size of the data required for a single task execution; as all the significant data structures have either `Real` or `Complex` values, we can measure the memory footprint in number of `Real` elements.
- Parallelism: represents the parallelization potential of a task, which quantifies how easy a task can be parallelized. For example, a loop with independent iterations is highly parallelizable, while a file read is hardly parallelizable.
- Impact: is an estimation of the time taken by a single task execution (i.e., within one iteration of the main loop). Note that this estimation may be sensitive to application-specific parameters. Also note that some tasks are outside the loop and, as such, will not impact the main core computation.

Based on these parameters, the task characterizations for both gridding and degridding are summarized in Table 5.2.

Table 5.2: *The characteristics for the kernels' tasks. Note the following notations:* $N = N_{baselines} \times N_{samples} \times N_{freq}$*; the "Comp:Comm" ratio and the "Parallelism" are qualified from very low to very high, with* $--$, $-$, $\sim$, $+$, $++$*;* c_i *are constant whose value are irrelevant; a task with no impact on the main computation core is denoted with* $-$

Task	Input data	Comp: Comm	Output data	I:O ratio	Memory [Real el.]	Parallelism	Impact
G1	(u,v,w),V	$--$	(u,v,w),V	1:1	$2 \times N$	$--/-$	$-$
G2	(u,v,w),f	$+$	`c_index` `g_index`	1:1	c_1	$\sim$	5%
G3	`c_index`,**C** `g_index`,**G**	$--$	SK,SG	1:m^2	$c_2 \times m^2 + c_3$.	$+$	20%
G4	SK, SG	$+$	SG'	$2m^2$:m^2	$c_4 \times m^2 + c_5$	$++$	65%
G5	SG'	$--$	G	m^2:1	$c_6 \times m^2 + c_7$	$--$	10%
D1	(u,v,w),**G**	$--$	(u,v,w),**G**	1:1	$N + c_8$	$--/-$	$-$
D2	(u,v,w),f	$+$	`c_index` `g_index`	1:1	c_9	$\sim$	5%
D3	`c_index` `g_index`	$--$	SK,SG	1:m^2	$c_{10} \times m^2 + c_3$.	$+$	20%
D4	SK, SG	$+$	V	m^2:1	$c_{11} \times m^2 + c_{12}$	$++$	75%
D5	V	$--$	V'	1:1	$c_{13} \times m^2 + c_{14}$	$--$	$-$

A quick look at Table 5.2 (remember that N is in the order of millions of values and m is at most 129) proves that both gridding and degridding are memory bound (see the I:O ratio, memory footprint), data-intensive (see the computation:communication ratio) kernels. As such applications are not a good match for Cell/B.E.-like machines, where the core-level memory is not shared, a lot of effort will have to be put into investigating two major parallelization aspects: task distribution (i.e., change the sequential patterns of the original application model into parallel ones) and data management (i.e., use data parallelism and pipelining among the tasks to avoid excessive memory traffic).

5.3 Parallelization on the Cell/B.E.

In this section, we describe our model-guided parallelization for the gridding/degridding kernels. Further, we present one implementation solution and we describe the additional Cell-specific and application-specific optimizations that had a significant impact on the kernels performance. As our approach is similar for both kernels, we only use the gridding parallelization for the detailed explanations. Finally, we present the outline of our method as a list of guidelines for enabling other similar applications to run on the Cell/B.E.

5.3.1 A Model-driven Application Parallelization

To have efficient implementations of gridding and degridding on the Cell/B.E., we aim to delegate and optimize most of their computationally intensive parts to the SPEs, thus increasing the overall application performance. To decide what parts of the whole gridding computation chain can be efficiently moved to the SPEs, we start from a simple gridding model, and we investigate the performance

effects each SPE-PPE configuration might have. The model of the sequential gridding chain, depicted in Figure 5.7 is:

```
Gridding=G1;for(i=1..N){G2;G3;G4;G5;}
```

The main loop.

To enable several parallelization options, we need to transform the sequential loop into a parallel one; in turn, this transformation allows the computation inner core, `{G2;G3;G4;G5}`, to be executed in an SPMD (single process, multiple data) pattern over `P` processors. Such a transformation can only be performed if the iteration order is not essential and if the data dependencies between sequential iterations can be removed. To remove the data dependency between G3 and G5 (via **G**, as seen in Table 5.2), we assign each processor its own copy the grid matrix, **G.local**. Further, because the grid is computed by addition, which is both commutative and associative, the iteration order is not important. Thus, after all computation is ready, an additional task (G5.final) is needed to sum all **G**.local's into a final **G** matrix[3]. The new model becomes:

```
Gridding=G1;par(p=1..N){G2;G3;G4;G5};G5.final
```

Note that that the same transformation is simpler for the degridding kernel, as there is no data dependency between D2 and D5.

Data read (G1/D1)

The input task, `G1`, has to perform the I/O operations, and will run on the PPE. Its output buffer, a collection of $(V,(u,v,w))$ tuples, is located in the main memory. For the off-line execution (input data is stored in files), the parallelization among several PPEs is possible, but will only gain marginal application performance by reducing the start-up time. In the case of on-line (streaming) execution, parallelization is only useful in the highly unlikely case of a streaming rate (i.e., G1's input) much larger than the memory rate (i.e., G1's output). Given these considerations, we did not parallelize `G1` (and, similarly, `D1`); further, these tasks are not taken into account when measuring application performance.

The computation core (G2-G5/D2-D5).

Tasks `G2,G3,G4,G5` represent the main computation core of the gridding kernel. Once the parallel loop is in place, we have to map this sequence of tasks and their data on the available SPEs, enabling several instances to run on different data (SPMD). Figure 5.8 shows the task graph of application running on the PPE and one SPE (the shaded part). The bold dotted lines represent off-core memory transfers: because the tasks running on the SPE can only process data in their own local memory, they have to use DMA to fetch data from the main memory, compute, and then use DMA again to store the data in the main memory. Compared with Figure 5.7, these DMA transfers represent additional overhead.

The new model is:

```
Gridding = G1@PPE;par(p=1..P){for(i=1..N/P){G2@SPE;G3@SPE;G4@SPE}};G5@PPE
```

[3]This simple solution is possible for any operation that is both commutative and associative, like addition in this case.

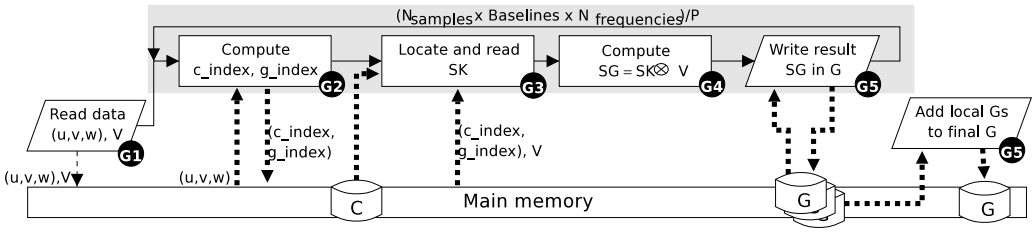


Figure 5.8: *The execution phases of the parallel gridding kernel. The shaded part is a candidate for execution on each SPE. Bold dotted lines represent off-core memory transfers.*

We further decompose the model by exposing the DMA operations (note that `G3@SPE` is replaced by its DMA operations), and we obtain:

```
Gridding = G1@PPE;
  par(p=1..P){
    for(i=1..N/P){
      DMA_RD(u,v,w);G2@SPE;DMA_WR(g_index,c_index);
      DMA_RD(g_index,c_index,SK,V);G4@SPE;DMA_WR(SG);
      DMA_RD(SG,G.local);G5@SPE;DMA_WR(G.local)}
}; G5.final@PPE
```

The final addition (G5.final/D5.final)

The final computation stage, `G5.final`, computes **G** by summing the SPE copies, **G**.local, using, for example, a binary tree model [JaJ92]. However, as this computation is not part of the main core, it represents only a small fixed overhead (depending only on the number of SPEs and the grid size) to the overall application time; thus, it is not parallelized and not included when measuring application performance.

5.3.2 Mapping on the Cell/B.E.

Further, we investigate the mapping of this model onto a Cell/B.E. processor for the streaming data scenario (i.e., the most generic case for data input) and several platform-specific optimizations.

Task mapping.

To avoid expensive context switching, the mapping of the G2, G3, G4, and G5 tasks on the SPEs can be done in two ways:

- Assign one task per SPE, in a task-parallel model. In this case, data is streaming between SPEs, without necessarily going to the main memory. One iteration takes T^T_{unit}, and $N/(N_{SPEs}/4)$ iterations can run simultaneously; t_{stream} is the data transfer time between two SPEs. Equation 5.7 gives a rough estimation of the execution time for this solution.

$$T^T_{unit} = t_{G2} + t_{stream} + t_{G3} + t_{stream} + t_{G4}t_{stream} + t_{G5} \tag{5.7}$$
$$T^T_{total} = \frac{N}{\frac{N_{SPE}}{4}} \times T^T_{unit} = \frac{4 \times N}{N_{SPEs}} \times T^T_{unit}$$

This solution is appealing due to its simplicity, but we have dismissed it as the computation imbalance between G2, G3, G4, and G5 leads to severe core underutilization.

- Assign the entire chain, G2-G3-G4-G5, to one SPE, in a pure SPMD model. The data is shared in temporary buffers in the SPE local store and, when running out of storage space, in the main memory. Note that the number of DMA RD/WR operations is reduced in the new parallel application model, due to data sharing via the local store:

```
Gridding = G1@PPE;
  par(p=1..P){
    for(i=1..N/P){
      DMA_RD(u,v,w);G2@SPE;DMA_RD(SK,V);
      G3@SPE;G4@SPE;DMA_RD(G.local);
      G5@SPE;DMA_WR(G.local)}
    }; G5.final@PPE
```

For this new solution, one iteration takes T^D_{unit}, and N iterations can run simultaneously; equation 5.8 presents a rough estimation of the execution time:

$$T^D_{unit} = t_{G2} + X * t_{mm} + t_{G3} + Y * t_{mm} + t_{G4} + Z * t_{mm} + t_{G5} \tag{5.8}$$
$$T^D_{total} = \frac{N}{N_{SPEs}} \times T^D_{unit}$$

Note that T^D_{unit} includes a potential overhead for accessing the main memory, via the X, Y, and Z coefficients. Even so, this solution has a higher performance potential and enables higher core utilization, so we proceed to its implementation and optimizations.

Data distribution.

There are three main data structures used by the SPEs computation: **C**, **G**, and the $(V,(u,v,w))$ tuples. A symmetrical data distribution is not possible for all three of them, as consecutive visibility measurements will generate non-adjacent regions in **C** and **G**. In other words, a block of data from the visibilities array will not map on a contiguous block in neither **C** nor **G**. Thus, only one array shall be symmetrically distributed to the SPEs. There are two good candidates for this initial distribution: the grid and the visibilities.

G is the smallest array and the one with the highest re-use rate (i.e., more additions are made for each grid point). However, a symmetrical distribution of grid regions to all SPEs will generate a severe load imbalance between SPEs because the grid is not uniformly covered by real data processing - see also Figure 5.6.

Due to its sequential access pattern, the $(V,(u,v,w))$ tuples array is also suitable for symmetrical distribution. For offline processing, a simple *block distribution* can be used to spread the N visibilities on the N_{SPEs} cores. For online processing, some type of *cyclic distribution* is required to keep the load and utilization uniform among the SPEs.

Next, we need to verify that all data structures used by G2-G3-G4-G5 fit in the memory of a single SPE. The required data size for all three tasks, as computed from Table 5.2, is:

$$\begin{aligned} Size &= sizeof(SK) + sizeof(SG) + sizeof(SG') + sizeof((u,v,w)_{b,t}, g_{index}, c_{index}, V_{b,t}) \\ &= 3 \times m^2 \times sizeof(complex) + (4 \times sizeof(complex)) \sim 3 \times m^2 \times sizeof(complex) \end{aligned} \tag{5.9}$$

Given the local store size of 256KB (=32K complex numbers), we can estimate the maximum size of the convolution kernel to be $SK_M = m \times m = 100 \times 100$, which is smaller than the 129×129 maximum requested by the astronomers. Thus, the G2-G3-G4-G5 chain cannot run without communicating with the main memory via DMA (i.e., in equation 5.8, $X, Y, Z \neq 0$). To minimize the DMA overhead influence on the T^D_{unit}, we resort to double buffering and pipelining.

Double buffering and pipelining Because the Cell/B.E. DMA operations can be asynchronous, we overlap computation and communication to increase the application parallelism. The new application model (note that `G3@SPE` has been replaced by its implementation, the two DMA_RD operations for SK and SG) is:

```
Gridding = G1@PPE;
   par(p=1..P){
   for(i=1..N/P){
      par ( DMA_RD(u,v,w) , G2@SPE(u,v,w) , DMA_RD(V) ,
 DMA_RD(SK), G4@SPE(V,SK,SG'.local),
        DMA_RD(SG.local), G5@SPE(SG'.local,G.local),
 DMA_WR(SG.local) );
      }; G5.final@PPE
```

For a simpler implementation, the task chain is "broken" into: the indices computation (`G2@SPE`), which only depends on scalar data, like $((u, v, w), V)$, and the grid computation, (`G4@SPE-G5@SPE`), which requires the array data sets. For the `G2@SPE`, the computation itself is overlapped with the DMA transfer of the visibility data, V. For the rest of the chain, we need to solve two more DMA-related constraints: (1) a single DMA operation is limited to 16KB of data, and (2) one transfer works only on contiguous data regions. For SK_M, for which the convolution matrix construction guarantees that all $m \times m$ elements are in a contiguous memory region, we need $dma_{SK_M} = sizeof(SK_M)/16KB = m^2 \times 8$ consecutive transfers. For SG_M, which is a rectangular subregion in a matrix, we need to read/write each line separately, thus implementing $dma_{SG_M} = m$ consecutive DMA transfers. As the grid is read one line at a time, the computation of the current line can be overlapped with the reading/writing of the next line. This pipeline effect is obtained by double buffering. The detailed model is:

```
Gridding = G1@PPE;
  par(p=1..P){
  for(i=1..N/P){
    DMA_RD(u,v,w); G2@SPE(u,v,w) || DMA_RD(V));
    pipeline (k=1..m-1) {
      DMA_RD(SK[k,*]);
      DMA_RD(SK[k+1,*]) || G4@SPE(V,SK[k,*],SG'.local[k,*]) ;
      DMA_RD(SG.local[k,*]) ;
      par { DMA_RD(SG.local[k+1,*]),
        G5@SPE(SG'.local[k,*],SG.local), DMA_WR(SG'.local[k-1,*]));
      }
    }
}; G5.final@PPE
```

We estimate the new T^D_{unit} in equation 5.10.

$$T^D_{unit} = t_{DMA\_RD} + max(t_{DMA\_RD}, t_{G2}) + \\ + m \times (t_{DMA\_RD} + max(t_{DMA\_RD}, t_{G4}) + t_{DMA\_RD} + max(t_{DMA\_RD}, t_{G5}, t_{DMA\_WR})) \quad (5.10)$$

SIMD-ization Finally, we need to optimize the computation code itself, by applying core specific optimizations: loop unrolling, SIMD-ization, branch removal, etc. However, due to the small amount of computation, as well as its overlap with communication, these optimizations may not have a significant impact in the overall execution time. Nevertheless, we have implemented the core computation of both `G4` and `G5` using the Cell/B.E. SIMD intrinsics. Although a theoretical speed-up factor of 4 can be obtained for perfect code SIMD-ization, the observed improvement has been not larger than 40%.

5.3.3 Application Specific Optimizations

Note that all the optimizations presented so far are data independent, addressing only the structure of the application and not its runtime behavior. In this final parallelization step, we discuss a few additional, data-dependent optimizations, which aim to increase data locality and re-use between consecutive data samples. The experimental results presented in Section 5.4 show how important these data optimizations can be for the overall application performance.

SPE-pull vs. PPE-push

Online data distribution from the PPE to the SPEs can be performed either by the PPE pushing the data (i.e., sending the address of the next elements) or by the SPE pulling the data (i.e., the SPE knows how to compute the address of its next assigned element). Although the SPE-pull model performs better, it is less flexible as it requires the data distribution to be known. The PPE-push model allows the implementation of a dynamic master-worker model, but the PPE can easily become the performance bottleneck. The choice or combination between these two models is highly application dependent; for example, in the case of gridding/degridding, all data-dependent optimizations are based on a PPE-push implementation.

Input data compression

A first observation: if a set of n visibility samples generate the same `(c_index,g_index)` pair, thus using the same SK for updating the same $\mathbf{G}$ region, the projection can be executed only once for the entire set, using the sum of all n data samples:

$$SG' = \sum_i^n SK \times V[i] = SK \times \sum_i^n V[i]$$

We have only applied this optimization for consecutive samples. We have noticed that this data compression gives excellent results for (1) a single baseline, when the frequency channels are very narrow, or (2) for a large number of baselines, where equal `(c_index,g_index)` pairs appear by coincidence on different baselines. Further, there is a higher probability of such sequences to form when the visibility data is block-distributed rather than cyclic-distributed.

However, compressing these additions is limited by an upper bound, due to concerns related to accuracy. This upper limit, i.e., a maximum numbers of additions that preserve accuracy, is computed

such that the overall error is not larger than the accepted level of noise. Lower noise levels will decrease the impact of this optimization.

PPE dynamic scheduling

To increase the probability of sequences of visibilities with overlapping SK and/or SG regions, the PPE can act as a dynamic data scheduler: instead of assigning the visibility samples in the order of their arrival or in consecutive blocks, the PPE can compute the `(c_index,g_index)` pair for each $V(u,v,w)_{b,t}$ and distribute the corresponding data to an SPE that already has more data in that region. To balance the PPE scheduling with the SPE utilization, a hybrid push-pull model is implemented using a system of shared queues. Each queue is filled by the PPE, by clustering visibility data with adjacent SK's and SG's *while* preserving load balancing, and consumed by one SPE. As the PPE requires no intermediate feedback from the SPEs, there is virtually no synchronization overhead. The PPE may become a bottleneck only if the visibilities clustering criteria are too restrictive and/or too computationally-intensive; implementing multiple threads on the PPE side may partially alleviate this problem.

Grid lines clustering

Each SPE has in its own queue a list of N_q visibilities that need to be gridded, so it computes a $m \times m$ matrix for each one of them. However, as the computation granularity is lower (i.e., we compute one line at a time, not the entire subregion), and because a DMA operation can fetch one complete grid line, the SPE can compute the influence of several visibilities on the complete grid line. Thus, the SPE iterates over the **G** lines, searches its local queue for visibilities that need to be gridded on the current **G** line, and computes all of them before moving to the next grid line. This approach increases the grid re-use, but it also increases the computation for each visibility by additional branches.

Online and offline sorting

Finally, the best way to increase data re-use is to be able to sort the data samples by their `(g_index)` coordinate. In the offline computation mode, sorting all visibilities may enable a simple and balanced data block distribution among the SPEs. However, as the data volume is very large, such an overall sort may be too expensive. The other option (also valid in the case of online data processing) is for the SPEs to sort their own queues before processing them. Such an operation is very fast (at most a few thousands of elements need to be sorted), and it increases a lot the efficiency of the grid-lines clustering trick.

5.4 Experiments and Results on the Cell/B.E.

This section presents our experimental study on the performance and scalability of the gridding/degridding implementations on the Cell/B.E.. We include a detailed performance analysis which, together with the novel experimental results for processing multiple baselines on a single Cell/B.E. processor complement our previous work [Var08b].

Our performance study focuses on high-level application optimizations. Therefore, although we have implemented the typical SPE-level optimizations (including DMA double buffering, SIMDization, and loop unrolling), we do not present in detail the improvements they provide over non-optimized code (details can be found in [Var08a]). Rather, we discuss the custom optimizations at the

job scheduling level, i.e., at the administration of the PPE-SPE job queues. Further, we analyze the effect of data-dependent optimizations. Finally, we present the performance and scalability analysis of the fully optimized application, and discuss the impact of results on the on-line application scenario.

5.4.1 The Experimental Set-up

All the Cell/B.E. results presented in this paper have been obtained on a QS21 blade, provided for remote access by IBM. We have validated the blade results by running the same set of experiments (up to 6 SPEs) on a local PlayStation 3 (PS3). The results are consistent, excepting the cases when the overall PS3 memory is too small for the application footprint, leading to a rapid decrease in performance. For example, gridding with a support kernel larger than 100×100 elements is twice as slow as the sequential application, while several attempts to test the application on more than 500 baselines have crashed the PS3 multiple times.

We compare these results with a reference time, measured for the execution of the original, sequential application on an Intel Core2-Duo 6600 processor, running at 2.4GHz[4].

The application is implemented using SDK3.0. We have used three different test data sets, $(V,(u,v,w))_{b,t}$: real data, i.e., a collection of real data gathered by 45 antennas in an 8-hours observation session (990 baselines, with 2880 data samples on 16 frequency channels each); randomly generated data, and single-value data, i.e., all $(u,v,w)_{b,t}$ coordinates are equal.

The main performance metric for the gridding and degridding is the execution time: ultimately, the astronomers are interested in finding out how fast the problem can be solved. Therefore, we present our performance results using *time per gridding*, a normalized metric defined as the execution time of any of the two kernels divided by the total number of operations that were performed. The smaller the time per gridding is, the better the performance is. For the purpose of estimating how good the performance we achieve is in absolute terms, we also report the application throughput, which is computed by estimating the useful computation FLOPs for each gridding operation, and dividing by the measured time per gridding.

Finally, as both gridding and degridding involve a similar computation pattern and the measured results follow the same trends, we only plot and discuss the gridding results (for specific results on degridding, we refer the reader to [Var08a]).

5.4.2 A First Cell/B.E. Implementation

The first Cell/B.E. implementation is based on a PPE scheduler that uses separate job queues for each SPE. The PPE fills each queue with pairs of the `(c_index,g_index)` indices - i.e., the coordinates from where the SPE has to read the support kernel, SK, and the grid subregion, SG. The PPE fills the queues in chunks, with a fixed size of (at most) 1024 pairs per queue.

On the SPE side we have implemented a "standard set" of optimizations. Because the convolution matrix was reorganized such that any usable support kernel is contiguous in memory, SK is fetched (after proper padding for alignment and/or size) either using a single DMA operation or a DMA-list, depending on its size. As the grid subregion SG is a rectangular section from the $\mathbf{G}$ matrix, it requires reading/updating line by line. Therefore, for SG, we have implemented multi-buffering: while the current grid line is being processed, the next one is being read and the previous one is being

[4]The choice for this particular machine was only due to availability, and we use the execution time on Core2-Duo only as a measure of the performance of the reference sequential code, not as a measure of the potential performance of this particular processor.

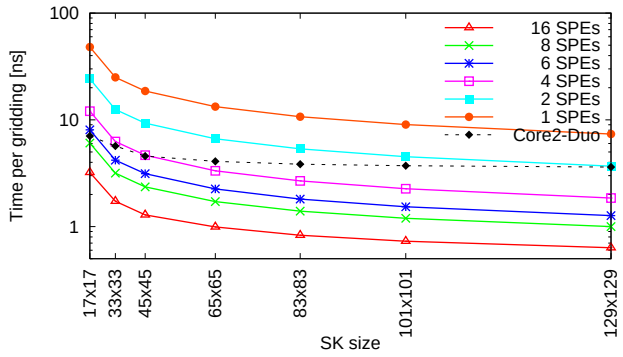


Figure 5.9: *Time per gridding results for seven different kernel sizes. The best performance is achieved for the largest kernel size, which allows for the best computation-to-communication ratio. Note the logarithmic Y scale.*

written. As each SPE works on a "proprietary" grid, no inter-SPE synchronization for reads and/or writes is necessary. For each (SK,SG) pair, the visibility computation, i.e., the loop that computes the $m \times m$ complex multiplications, is SIMD-ized. Because complex numbers are represented as `(real,imaginary)` pairs, the loop is unrolled by a factor of 2, and each iteration computes two new grid points in parallel. Finally, the queue is polled regularly: the SPE performs one DMA read to fetch the current region of the queue it is working on. All new `(c_index,g_index)` pairs are processed before a next poll is performed.

The performance of one visibility computation (the compute intensive part of the gridding) reaches 9 GFLOPS for the largest kernel, and 2.5 GFLOPS for the smallest one. Therefore, more aggressive optimizations can still be performed to increase the SPE performance. However, this path requires significant SPE code changes (which are interesting for future work), without solving the overall memory bottleneck of the application. In the remainder of this section, we focus on high-level optimizations and their ability to efficiently tackle the memory-intensive behavior of the application.

We have tested our first Cell/B.E. implementation on real data collected from 1, 10, and 100 baselines, using seven different kernel sizes. For each kernel size, we have measured the time per gridding using the platform at 100%, 50%, 37.5%, 25%, 12.5%, and 6.25% of its capacity (16, 8, 6, 4, 2, and 1 SPEs, respectively). Figure 5.9 presents the performance results for gridding real data collected from 100 baselines, using four representative support kernel SK sizes (between 17×17 and 129×129). The results show that the performance increases with the size of the support kernel. This behavior is due to the better computation-to-communication ratio that larger kernels provide: a longer kernel line requires more computation, which hides more of the communication overhead induced by its DMA-in operation. We also point out that for this initial implementation, the performance gain is not significant when using more than 4 SPEs, a clear sign that the platform is underutilized.

When running the gridding for 10 baselines, a small performance loss - less than 5% is observable versus the 100-baselines data set. Further, the performance obtained for the 10-baselines data set is up to 20% better than the performance obtained for a single baseline. This improvement is due to the larger amount of work, which in turn leads to better utilization and load balancing among SPEs.

Figure 5.10(a) presents the application throughput for the same reference Cell/B.E. implementation, using the same 100-baselines data set. We notice the throughput increases for larger support kernels, but it still remains below 10% of the theoretical peak performance (25.6 GFLOPS per SPE). Figure 5.10(b) shows the application throughput per core, where the original implementation, run-

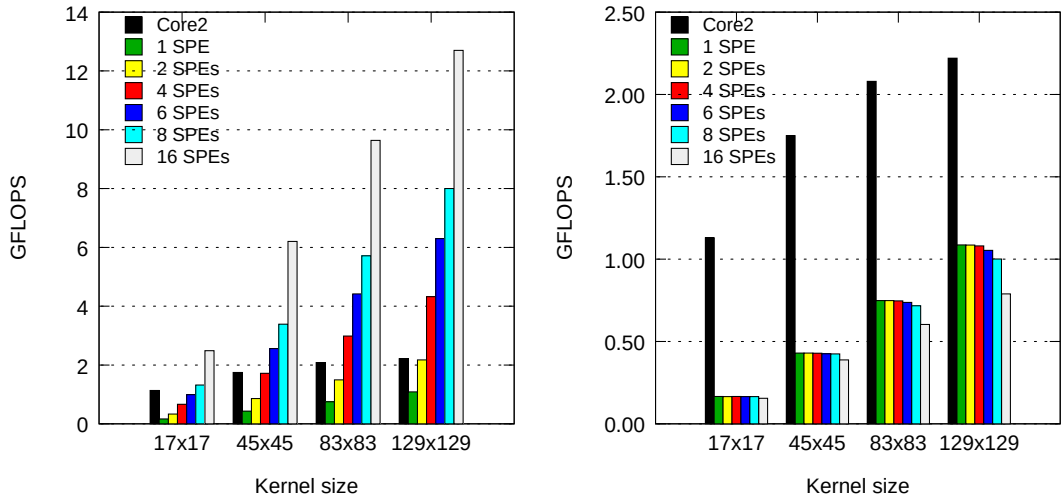


Figure 5.10: *Application throughput for the first Cell/B.E. implementation (a) per platform and (b) per core for 4 different SK sizes. Note that the best throughput is obtained for the largest kernel,* 129×129.

ning on the Core2-Duo, outperforms the Cell/B.E. version. Finally, Figure 5.11 shows the application throughput for the largest support kernel (129×129) on all three data sets: 1, 10, and 100 baselines. The results show that the througput increases with the number of baselines.

5.4.3 Improving the Data Locality

We tackle data locality problem at two levels: at the PPE level, by controlling the way the queues are filled, and on the SPE level, by detecting overlapping **C** and/or **G** regions to be grouped, thus skipping some DMA operations. Remember that the PPE distributes data to the SPE queues in "chunks", such that each queue receives a sequence of `(c_index,g_index)` pairs (not a single one) before the next queue is filled. To optimize this distribution, the PPE checks on-the-fly if multiple consecutive `(c_index,g_index)` pairs overlap, case in which they are all added into the same queue. When the SPEs read their queues, they can exploit the data locality these sequences of pairs provide. Additionally, we have added a local SPE sorting algorithm for the queues. As a result, multi-baseline data locality can be exploited to further increase performance.

Figure 5.12 shows the performance of the application with the same fixed 1024 queue chunks on the PPE side, but including the data sorting on the SPE side.

We have also measured the performance effect of the chunk size variations for queue filling. Figure 5.13 shows the performance effect of six such sizes (64, 128, 256, 512, 1024, and 2048) for the same 100 baselines. We have only plotted the time per gridding of the 8SPEs and 16SPEs configurations with a support kernel of 129×129. The difference in performance can reach even 25% between the best and the worst tested cases. Furthermore, the best chunk sizes are not the same for both configurations, which indicates that a further study needs to address a way to compute this parameter as a function of the kernel size and the number of SPEs.

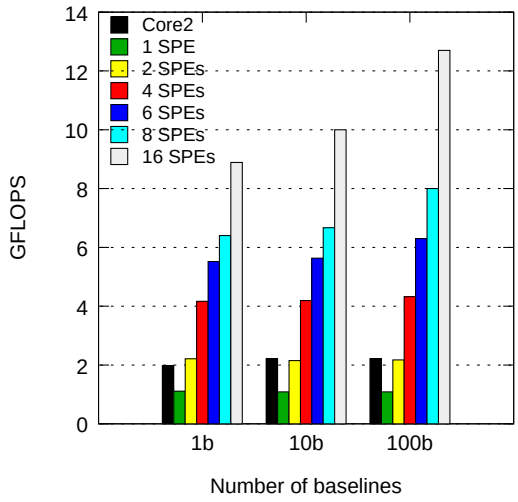


Figure 5.11: *Application throughput for SK=129 × 129 for the first Cell/B.E. implementation running on data from 1, 10 and 100 baselines.*

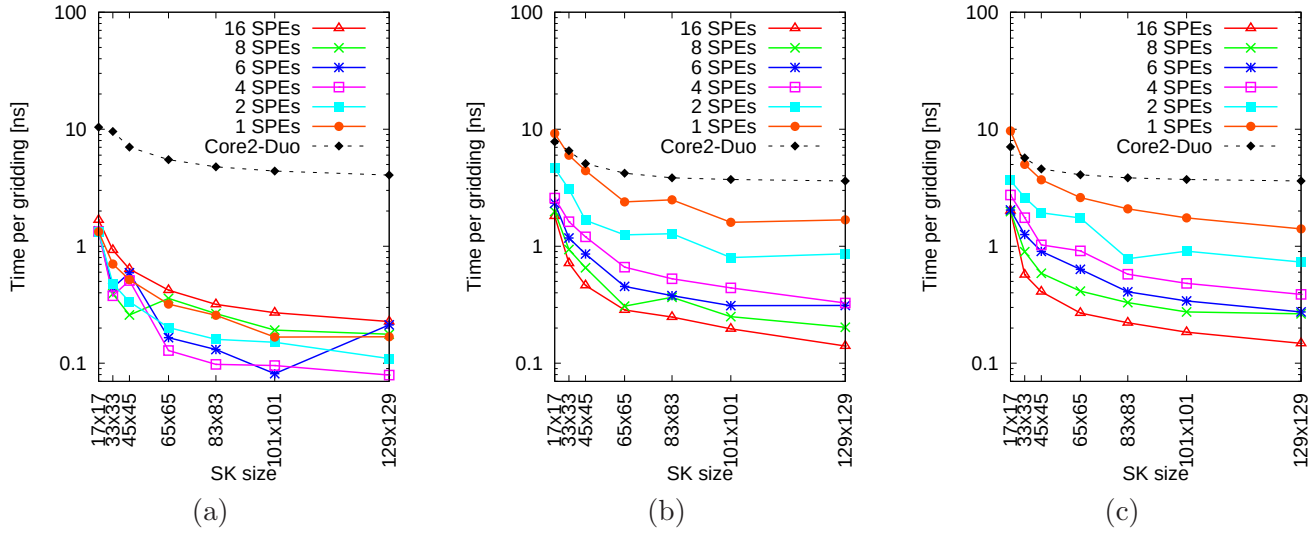


Figure 5.12: *The performance of the Cell/B.E. gridding with sorted SPE queues, running on data from (a) 1, (b) 10, and (c) 100 baselines. Due to the increased data locality, the application performance is significantly better*

5.4.4 PPE Multithreading

An important performance question is whether the PPE acting as a dynamic scheduler for the visibility data in the SPE queues can lead to a performance bottleneck. In other words, is there any danger of the PPE queues filling rate to be too small for the SPE processing rate? To answer this question, we have implemented a multi-threaded PPE version. Given that the QS21 blade we have used for experiments has 2 Cell/B.E. cores, we can run up to 4 PPE threads in parallel. We have measured the application performance when using 1, 2 and 4 PPE threads that fill the queues in parallel. The

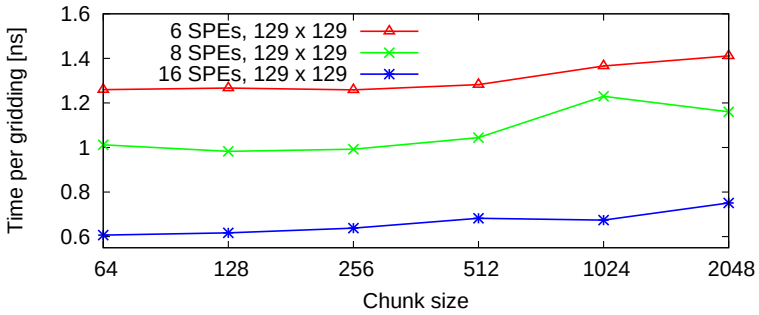


Figure 5.13: *The influence of the chunk sizes to the overall performance of the application. For example, the difference in performance reaches 23.5% for a 16SPEs configuration. Note the logarithmic scale for the x axis*

performance variations - within 5% of the PPE multi-threaded version when compared with the single-threaded version, are not a conclusive indication for the initial question. Therefore, we do not use the additional PPE threads, and we intend to conduct further investigation to determine if there is any other way to increase performance by PPE multi-threading.

5.4.5 Data-dependent Optimizations

Finally, we have also applied the "data set compression" optimization, by avoiding replacing complex multiplication with a complex addition any time a sequence of identical pairs, `c_index,g_index`, have to be scheduled. This optimization increases the execution time on the PPE side (and its utilization as well), but can lead to good performance by minimizing the DMA transfers on the SPE side. Despite the increase in performance, the SPE utilization for this solution is not significantly increased - the performance comes from reducing the number of computations the SPE has to solve and slightly increasing the computation on the PPE side. Note that this optimization is more efficient on the PPE side, as branches are more expensive on the SPEs.

Figure 5.14 shows the performance of the application with these data optimizations enabled.

Because a computation of the application throughput is no longer useful here, we give an estimate on our improvement by comparing the gained performance with the real measurements against two particular data sets: a random one, and one with all (u, v, w) values identical. Figure 5.15 shows these results. Note that, as expected, the data-compression on real measurements "fits" the real curve between the best - the one-value set - and the worst - the random set.

5.4.6 Summary

Best overall performance

To conclude our performance quest, we present a comparison between the performance obtained by the reference code and our Cell/B.E. implementation when running on data from the complete set of 990 baselines. Figure 5.16 present these results.

The speed-up factor on the 16 SPEs of the QS21 blade versus the original implementation is between 3 and 8, while the data-dependent optimizations can increase this factor to more than 20.

Further, we present a comparison of the performance evolution for four different kernel sizes. The graph presents the performance of two sequential implementations, with and without data-dependent optimizations, as well as the performance of the Cell/B.E. implementation, using 1 or 2 processors (8

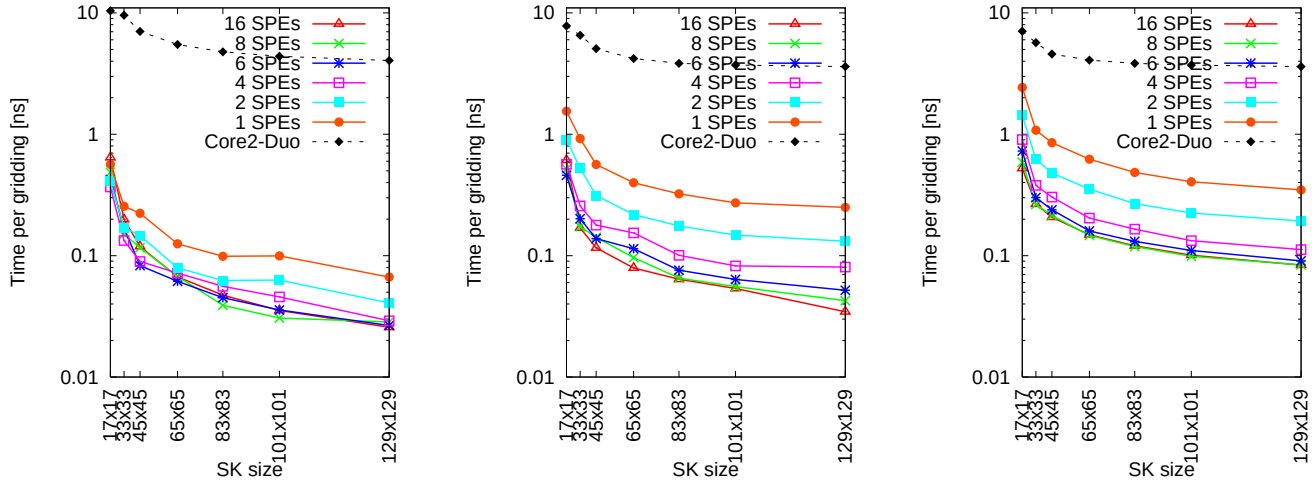


Figure 5.14: *The performance of application with the aggressive data-dependent optimizations, running on data from 1, 10, and 100 baselines. Note that the Core2-Duo application has been also optimized using similar data-dependent optimizations.*

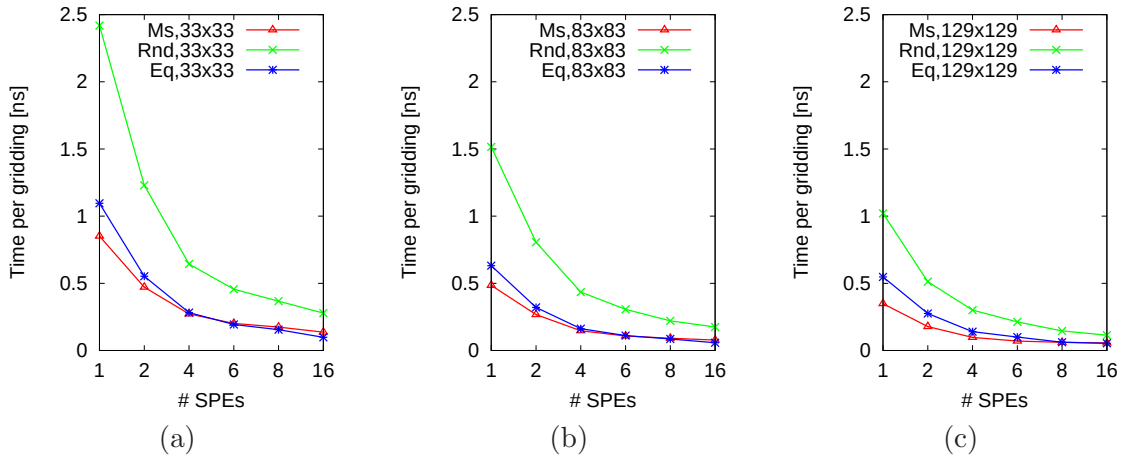


Figure 5.15: *The overall application performance for different kernel sizes. The number of baselines is 100. 'Ms' stands for the real measurements data set, while 'Rnd' and 'Eq' stand for the random and one-value sets, respectively.*

and 16 SPEs, respectively) from the QS21 blade, without any optimizations, with the queue sorting, and with the same data dependent optimizations. Figure 5.17 presents all these results. Note that for small kernel sizes, the performance difference between the Cell/B.E. and the original implementation running on an Intel Core2-Duo is not very large. However, as soon as the support kernel size increases, the gap becomes significantly larger. Finally, note that the impact of the data-dependent optimizations is also significant for the sequential application.

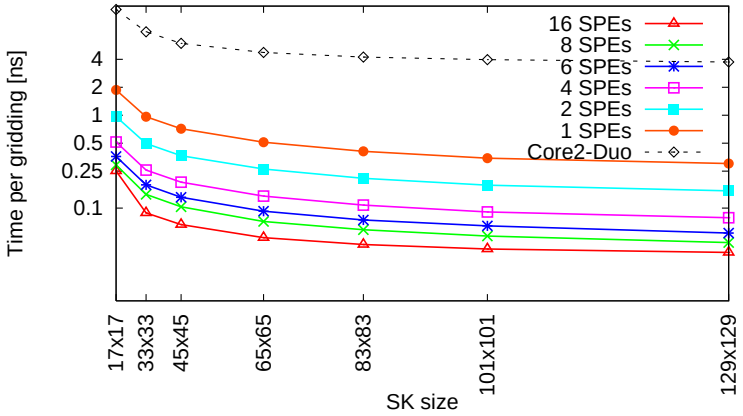


Figure 5.16: *The overall performance of the best implementation versus the original, reference code. The number of baselines is 990. Note the logarithmic scale.*

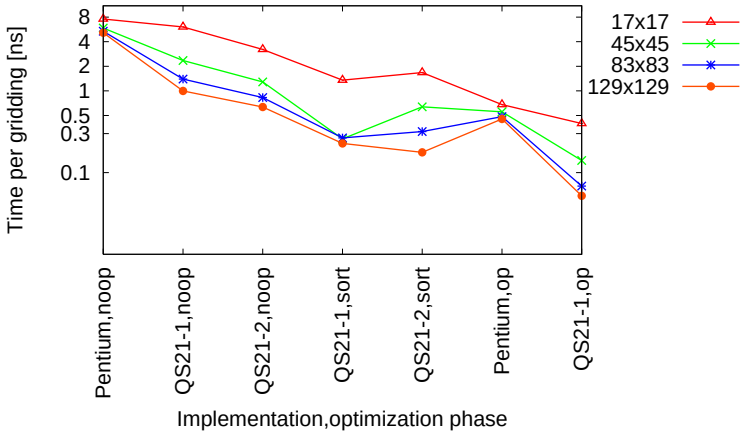


Figure 5.17: *The performance evolution for both the original implementation ("Pentium") and the Cell/B.E. implementations (1 and 2 processors). "noop" stands for no optimizations, "op" stands for data-dependent optimizations, "sort" refers to sorted SPE queues, and "all" is the Cell/B.E. version with all optimizations. Note the logarithmic scale.*

Scalability

We estimate the scalability of this implementation by studying its behavior for various numbers of baselines. Figure 5.18(a,b,c) shows how the fully optimized application behaves for 1, 10, 100, 500, and 990 baselines. Note that for one baseline, the execution time is small, mainly because the work volume is small. For 10+ baselines, we notice a significant execution time increase (20%) - this happens because of a larger amount of data, exhibiting low data re-use. For 100 baselines or more, data re-use is increasing, thus the aggressive data-dependent optimizations become more important in the overall execution time. Overall, increasing the baseline numbers between 100 and 990 does not affect performance significantly, which in turn signals good application scalability with the number of baselines (i.e., the size of the data set). Note the difference in scale (more than a factor of 5) between the performance for small kernels and large kernels.

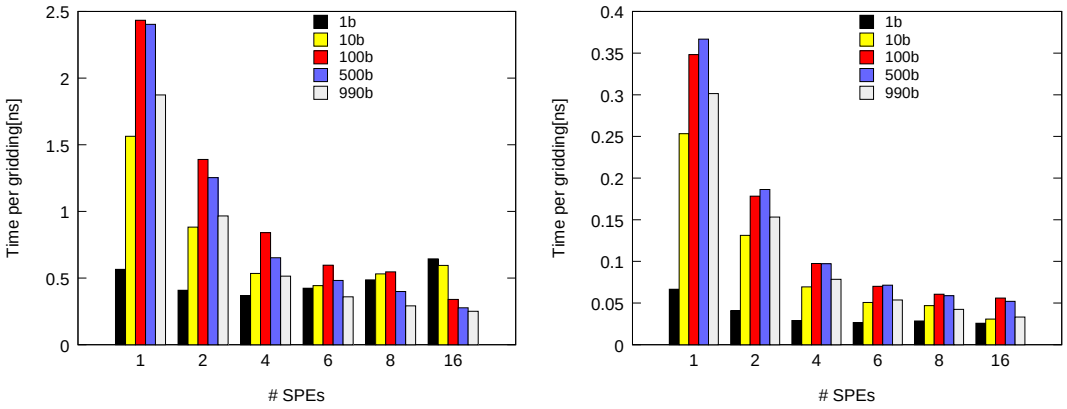


Figure 5.18: *Application scalability for 1, 10, 100, 500 and 990 baselines, and two different kernel sizes: (a) the smallest,* 17×17*, and (b) the largest,* 129×129*. The performance decreases between 1 and 10 baselines (due to the data-dependent optimizations, which reduce the input data set), but increases from 10 to 100+ baselines, a positive sign for the real application scale.*

Streaming

Finally, based on the overall performance, we evaluate if the application can be used for online processing. In this scenario, data is streaming in at a given rate, and the current samples have to be processed *before* the new set of samples is read. We have simulated data samples from a large number of baselines by random numbers, and we have measured the time per gridding for $SK_M = 129 \times 129$ to be about $0.22ns$, or a time per spectral sample of $3.66us$. For a streaming rate of 1 measurement/second, we have to process $B \times N_{freq}$ spectral samples every second. The current performance level allows about 260 baselines with 1024 frequency channels, or more than 530 baselines with 512 frequency channels. Above these limits, i.e., if the application generates more samples per second, the streaming rate is too high for the application performance, and the online processing will either require temporary buffering space, or multiple processors to split the load.

Given all these results, we conclude that the application can reach about 20-30% of the the Cell/B.E. peak performance, depending on the particular problem size. Furthermore, our approach is scalable and optimized such that it manages to increase the radioastronomers' productivity by a factor of 20. Even further, these results indicate that using this implementation, on-line processing is possible using a single Cell/B.E. for systems of up to 500 baselines and 500 frequency channels. Future work will focus on increasing the SPE code performance, as well as on even finer tuning of the queuing system (automatic adjustments of the queue sizes, comparisons with other dynamic scheduling options, etc.). Finally, we note that similar applications can be found also in other HPC fields, such as medical imaging [Sch95; Ros98] or oil reservoir characterization [Pra07].

5.5 Lessons Learned and Guidelines

Our gridding implementation for the Cell/B.E. can be considered a successful experiment in running a data-intensive application on a multi-core processor. Our parallelization approach is generic enough to be useful for other applications and, we believe, can be extended to other platforms (see Chapter A

for mode details). Therefore, we synthesize this step-by-step strategy in a collection of ten guidelines:

1. Identify the main computation core of the application, and recognize a data-intensive behavior by evaluating its computation:communication ratio

2. Split the application into tasks and draw the sequential task graph. Evaluate the data I/O and memory footprint of each task.

3. Evaluate the potential task mappings on the available SPEs (depending on the number of tasks and their load); tasks on separated SPEs communicate by data streaming, while tasks running on the same SPE share data in the local store and/or main memory. Choose the solutions that offer the highest number of parallel instances.

4. Evaluate data distribution options. For static, symmetrical distributions, use either data that is sequentially accessed or data that is frequently reused. For irregularly accessed data, introduce a PPE-based scheduler to dynamically distribute data among the SPEs.

5. When using a PPE-scheduler, implement a queue-based model for the SPEs, to increase SPE utilization and reduce unnecessary synchronization overhead. Further, implement a multi-threaded version of the PPE computation and reserve at least one thread for the queue filling, such that this operation is only marginally affected by other PPE tasks.

6. Analyze and implement any potential data-dependent optimization on the PPE scheduler; note that such a transformation may require modifications to the SPE code as well.

7. Expand the model to include the data transfers, including the DMA operations. Remove the I/O dependencies between tasks on the same SPE and evaluate the data streaming performance for tasks running on different SPEs.

8. Verify the memory footprint for each task or combination of tasks. If too large, reduce task granularity to fit, and recompose up to the original computation size using a pipeline model.

9. Analyze communication and computation overlapping by either double-buffering of instruction reshuffling.

10. Implement core-specific optimizations and eventual application specific optimizations.

5.6 Related Work

In this section we present a brief overview of previous work related to both high-performance radioastronomy applications and significant Cell/B.E. case-studies. In this context, our work proves to be relevant for both fields, quite disjoint until very recent times.

On the Cell/B.E. side, applications like RAxML [Bla07] and Sweep3D [Pet07], have also shown the challenges and results of efficient parallelization of HPC applications on the Cell/B.E. Although we used some of their techniques to optimize the SPE code performance, the higher-level parallelization was too application specific to be reused in our case.

Typical ports on the Cell, like MarCell [Liu07], or real-time ray tracing [Ben06] are very computation intensive, so they do not exhibit the unpredictable data access patterns and the low number of

compute operations per data byte we have seen in this application. Irregular memory access applications like list-ranking have been studied in [Bad07a]. Although based on similar tasks decompositions, the scheme proposed in our work is simpler, being more suitable for data-intensive applications.

Despite all these excellent case-studies for applications and application types, Cell/B.E. has only been used recently for radioastronomy applications, mainly due to I/O and streaming concerns. In fact, astronomers mainly use shared memory and MPI to parallelize their applications on large supercomputers. To use MPI on Cell/B.E., a very lightweight implementation is needed and tasks must be stripped down to fit in the remaining space of the local store. The only MPI-based programming model for Cell is the MPI microtask model [Oha06a], but the prototype is not available for evaluation. OpenMP [O'B07] is not an option as it relies on shared-memory.

Currently, efforts are underway to implement other parts of the radio-astronomy software pipeline, such as the correlation and calibration algorithms on Cell and GPUs. Correlation may be very compute-intensive, but it is much easier to parallelize - it has already been implemented very efficiently on Cell and FPGAs [Sou07]. Apart from the general-purpose GPU frameworks like CUDA [nVi07] and RapidMind [McC07], we are following with interest the work in progress on real-time imaging and calibration [Way07], which deals with similar applications.

5.7 Summary

Data-intensive applications, even from the HPC world, are not naturally suitable for multi-core processors: their focus on data rather than computation defeats the number crunching purpose of these architectures. This mismatch leads to poor platform utilization and, in turn, to low absolute performance. Even so, multi-cores can still offer better performance and/or energy consumption than traditional machines. Therefore, an additional parallelization effort might still be worth a try.

In this chapter, we presented two data-intensive kernels used extensively in radioastronomy imaging: gridding and degridding. Because these two kernels have a big influence on the overall performance of the imaging chain, we focused on their effective parallelization on the Cell/B.E. We addressed the specific algorithmic changes and optimizations required to compensate the gap between the data requirements of the application and the communication capabilities of the platform. We conclude that we indeed require a specific parallelization approach to obtain an efficient and scalable implementation of gridding and degridding on the Cell/B.E..

To systematically build our parallel solution, we started by modeling the two kernels. We used a top-down approach, starting from the high-level application parallelization, (i.e., the task and data distribution between cores), and decomposing the model until (some of) the low-level parallelization layers could be exposed (i.e., double buffering, SIMD-ization). Our model can capture enough information to allow for a quick, yet correct implementation. Both the modeling and the implementation techniques can be easily adapted and reused for similar applications.

Further, we have evaluated a series of data-dependent optimizations. Although such optimizations are hard to re-use directly, the performance improvements they have produced show that data-dependent application behavior has to be studied and, if possible, exploited.

The performance results are very good: after standard optimizations, the parallel gridding running on a Cell blade (2 Cell/B.E. processors, 16 SPEs) was 5 to 10 times faster than the sequential application running on a commodity machine, depending on the volume of data to process (the higher the volume, the better the performance). After applying application specific optimizations, the speed-up factor exceeded 20x. The overall programming effort for this solution was 2.5 men-months.

Translating these speed-up numbers back to radioastronomy, we showed that a single Cell/B.E. can handle on-line processing for over 500 baselines and 500 frequency channels, when data is streamed at a rate of 1 sample/second. Our scalability studies indicate that higher data rates and/or more streams of data would require (1) multiple Cell/B.E. processors, (2) a different type of architecture, or (3) a different, more compute-intensive algorithm.

Alternative architectures

Given the huge interest in the performance of the gridding/degridding kernels, we had implemented the same application for both general purpose multi-core processors (GPMCs), and for graphical processing units (GPUs). We summarize here our findings, and we give more details on our solutions, experiments, and conclusions in Appendix A and cite [Ame09].

For all platforms - the Cell/B.E., and both the GPMCs and GPUs, the same factors influence performance. Thus, on the hardware side, the main factors are the available memory bandwidth and the ability to efficiently exploit core memory; on the application side, arithmetic intensity and memory access patterns are limiting the overall application performance.

The application exposes two significant parallelization problems. First, the possibility of write conflicts in the grid requires replication for concurrent computation, thus limiting the maximum grid size. All platforms suffer from this problem, but with limited memory and the need for a large number of thread blocks, the GPUs suffer the most: only some sort of pre-processing to guarantee conflict-free writes can help, but it remains to be seen if spending a lot of cycles on the host CPU pays off in real HPC application environments. Second, data locality can be increased by optimizing data ordering. While this is possible on the CPU and Cell/B.E., using powerful general purpose cores, it is not trivial to do on the GPU, where using the "remote" host processor adds significant performance overheads because core-local memories cannot be leveraged, and all requests must pass through global memory.

Overall, for GPMCs, the (large) shared memory makes it easier to program the irregular access patterns, but it does not alleviate their performance impact. The platform is heavily under-utilized, as none of the data structures fit in the cache and irregular memory accesses make cache usage extremely inefficient.

In the case of the GPUs, programmed using CUDA, the large amount of fine-grain parallelism leads to very good results when input data are not correlated. However, when data-dependent optimizations are enabled, the other platforms (GPMCs and the Cell/B.E.) come close.

Chapter 6

Porting C++ Applications on the Cell/B.E.**

In this chapter we discuss a generic solution to port object-oriented C++ applications on an accelerator-based platform. We further show how the performance of the ported application can be estimated. We use the Cell/B.E. as an example of a target platform, and MARVEL (i.e., a multi-kernel, image content analysis application) as a case-study application.

With the performance promises of multi-core processors, there is a lot of interest in refurbishing all types of applications for these platforms. For most scientific applications, which are usually written in C and FORTRAN, the challenge is to correctly extract the algorithm, isolate and partition the data structures, and apply proper parallelization.

Still, many consumer-oriented applications are written using object-oriented programming, with a preference towards C++. Typically, they make heavy use of objects, classes, inheritance, and polymorphism, encapsulating the computation and communication details in various layers of their class hierarchies. These applications are more difficult to migrate to multi-cores exactly because the parts that can be accelerated (the computation intensive ones) are so well encapsulated in objects.

In this chapter, we present a solution to tackle the problem of porting sequential C++ applications onto a multi-core processor. Specifically, we target accelerator-based platforms, which have a "master" core that executes the control flow, and a set of "worker" cores that perform the computation. Our parallelization strategy is based on offloading: the computation intensive parts of an application are isolated and detached from the main application, offloaded and optimized for the worker cores, and re-integrated in the control flow.

The parallelization strategy is application independent, and can be applied to arbitrarily large applications. While it might not generate the best possible parallel implementation of a given application, it quickly leads to a first task-parallel prototype, already able to run on the target multi-core. Due to the very modular implementation, further optimizations are enabled. An overview of the strategy, with a special focus on generality, is presented in Section 6.1.

To design and implement our strategy, we choose the Cell/B.E. processor as the target hardware platform. An application using our parallelization offload-model will run its main thread (the control flow) on the PPE, and offload the computation kernels on the SPEs. All explanations and examples included in this chapter use the Cell/B.E. terminology and, when needed, specific source code. However, the strategy does apply to any platform with a similar, accelerator-based architecture.

To support the strategy, we designed a template-based mechanism to enable accelerator code to be unplugged and re-plugged in the control flow of the original application. The code templates for this mechanism are presented in Section 6.2. Note that this modular solution enables the SPE tasks to be independently enhanced towards near-optimal performance.

The strategy itself cannot guarantee any performance gain - the eventual speed-up comes from

**This chapter is base on work published in the Proceedings of ICPP 2007 ([Var07]). The chapter only brings minor textual revisions to the original paper.

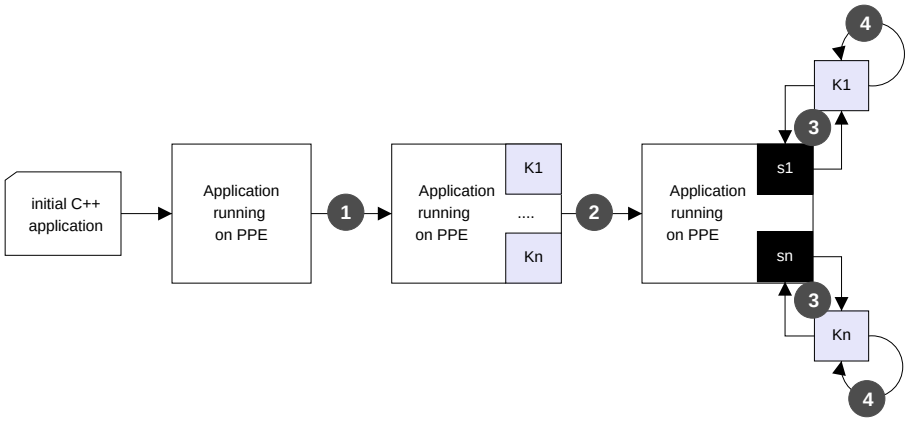


Figure 6.1: *The porting strategy of a sequential C++ application onto the Cell/B.E.*

the accelerated codes only. In most cases, however, when the kernels performance does increase, it immediately shows in the overall application performance. To estimate if the parallelization is worth the effort, we propose a simple sanity-check equation (see Section 6.3) that, given the tasks' performance gains and scheduling, allows a quick prediction of the overall application performance gain. Note that this prediction does not require code to be written: estimates of the SPE performance can be used to evaluate and rank different offloading alternatives.

Because the effective level of achieved performance is highly application dependent, we acknowledge that our generic scheme can limit the potential of specific applications, but we argue that, due to its simplicity and effectiveness, it provides a good starting point for more sophisticated techniques.

As a proof-of-concept, we have chosen to apply our strategy on a multimedia content analysis and retrieval application, named MARVEL (see Section 6.4). In a nutshell, the application analyzes a series of images and classifies them by comparison against precomputed models. MARVEL is a good candidate for a case-study, because it is a large sequential C++ application (more than 50 classes, over 20.000 lines of code) and it performs computation intensive operations, like image features extraction and image classification. The experiments we have conducted by running MARVEL on Cell/B.E. show that our strategy works for large applications, and it leads to good overall speed-up, gained with a moderate programming effort.

6.1 The Porting Strategy

Our strategy for porting C++ applications onto accelerator-based multi-cores focuses on efficiency: we aim to balance the porting effort and the performance gain. Thus, while we aim for the application to be able to use all the available parallelism layers provided by the hardware, we make use of as much code as possible from the initial application. Furthermore, we need a mechanism that allows application partitioning and code unplug/re-plug while preserving the application functionality all the way through the porting process (i.e., the porting does not destroy the overall data and control flow of the entire application).

The general strategy is presented in Figure 6.1. The application is first ported onto the PPE. Next, its most compute-intensive kernels have to be identified, and further detached from the main application as potential candidates for execution on the SPEs. Each kernel will become a new SPE

thread. In the main application, each kernel is replaced by a stub that will invoke the SPE-code and gather its results (step 2 in Figure 6.1). As a result of the separation, all former shared data between the application and kernel has to be replaced by DMA transfers (step 3 in Figure 6.1). Finally, the kernel code itself has to ported to C (to execute on the SPE) and further optimized (step 4 in Figure 6.1).

6.1.1 The PPE Version

A sequential C++ application running on Linux is, in most cases, directly compilable for execution on the PPE processor. The problems that appear in this phase are rather related to the application compile and build infrastructure than to the code itself; notable exceptions are the architecture specific optimizations (like, for example, i386 assembly instructions) which may need to be ported onto the PowerPC architecture.

Once running on the PPE, the application performance is not impressive, being two to three times slower than the same application running on a current generation commodity machine. This result is not a surprise, as the PPE processing power is quite modest[Kis06a].

6.1.2 Kernels Identification

First level of parallelization requires an application partitioning such that the most time-consuming kernels are migrated for execution on the SPEs (step 1 in Figure 6.1). To make efficient use of the SPEs processing power, the kernels have to be small enough to fit in the local store, but large enough to provide some meaningful computation. Kernels are not necessarily one single method in the initial C++ application, but rather a cluster of methods that perform together one or more processing operations on the same data. However, this grouping should not cross class boundaries, due to potential data accessibility complications. Furthermore, kernels should avoid computation patterns that do not balance computation and communication, thus accessing the main memory (via the DMA engines) too often. Typically, SPEs use small compute kernels on large amounts of data, allowing the compute code to be "fixed" in the SPE memory, while the data is DMA-ed in and out.

To identify the kernels, the PPE application running is profiled (using standard tools like `gprof` or more advanced solutions like Xprofiler[1]), and the most "expensive" methods are extracted as candidate kernels. Based on the execution coverage numbers, we find the computation core of each kernel. Based on the application call graph, each kernel may be enriched with additional methods that should be clustered around its computation core for reasons of convenient data communication and easy porting. Performance-wise, we state that the bigger the kernel is, the more significant the performance gain should be, but also the more difficult the porting becomes.

6.1.3 The `SPEInterface` stub

In order to preserve application functionality at all times, we have opted to implement the stub interface as a new class, named `SPEInterface`. The class manages the interface between the code running on the SPE and the main application running on the PPE, as presented in Figure 6.2. Basically, for each kernel, we instantiate an object of type `SPEInterface` and configure its parameters according to the kernel-specific requirements. Every such object will manage the communication between the application and one SPE kernel.

[1] `http://domino.research.ibm.com/comm/research_projects.nsf/pages/actc.xprofiler.html`

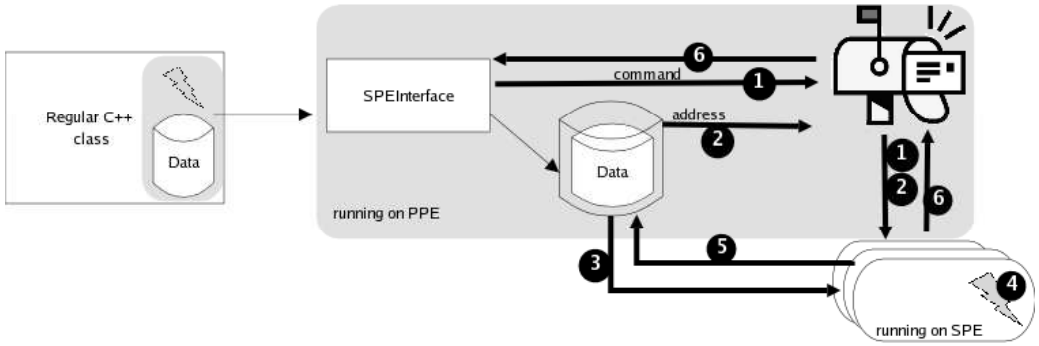


Figure 6.2: *The SPEInterface. From the regular C++ code, the shaded data and methods are replaced by calls to an SPEInterface object*

Thus, the changes required on the main application side are:

- Instantiate the `SPEInterface` class to a new object, say `Kernel1Interface`.
- Wrap *all* the required member data of the original class into a common data structure, and preserve/enforce data alignment for future DMA operations[Bro06].
- Allocate the output buffers for kernel results; typically, for simplicity, these buffers are also included in the data wrapper structure
- Communicate the address of this data structure to the kernel, which will fetch its required data via DMA.
- Replace the call to the code implemented by the kernel with a call to either `Kernel1Interface->Send` or `Kernel1Interface->SendAndWait` methods, thus invoking the SPE code and waiting for the results.
- Use the results put by the kernel in the output buffers to fill in the corresponding class member structure

In order to avoid the high penalty of thread creation and destruction at every kernel invocation, our approach statically schedules the kernels to SPEs. Thus, once the PPE application instantiates the kernel interfaces, the SPEs are activated and kept in an idle state. When calling `Kernel1Interface->Send` or `Kernel1Interface->SendAndWait`, the application sends a command to the kernel, which starts execution. When execution is completed, the kernel resumes its waiting state for a new command.

6.1.4 Kernel migration

The algorithm to migrate the kernels on the SPEs is:

- Define data interfaces - the data required by the kernel from the main application must be identified and prepared for the DMA transfer, which may require (minor) data allocation changes in the main allocation, to enforce alignment.

- Establish a protocol and medium for short message communication between the kernel and the application - typically, this channel is based on the use of mailboxes or signals.
- Program the DMA transfers - the SPE local data structures must be "filled" with the data from the main application. Data is fetched by DMA transfers, programmed inside the kernel. For data structures larger than the SPE available memory (256KB for both data and code), iterative DMA transfers have to be interleaved with processing
- Port the code - the C++ code has to be transformed in C code; all the class member data references have to be replaced by local data structures.
- For large data structures, the kernel code must be adapted to run correctly in an incremental manner, i.e., being provided with slices of data instead of the complete structures.
- Implement the function dispatcher (i.e., the kernel idle mode) in the `main` function of each kernel, allowing the kernel component functions to be executed independently, as dictated by the main application commands

For example (more to come in Section 6.4), consider an image filter running on an 1600x1200 RGB image, which does not fit in the SPE memory, so the DMA transfer must be done in slices. This means that the filter only has access to a slice of data, not to the entire image. For a color conversion filter, when the new pixel is a function of the old pixel only, the processing requires no changes. However, for a convolution filter[2], the data slices or the processing must take care of the new border conditions at the data slice edges.

6.1.5 The PPE-SPE Communication

After performing the correct changes in the main application and in the SPEkernel, the communication protocol between the two has to be implemented. The steps to be performed, indicated by the numbers in Figure 6.2, are:

1. The `Kernel1Interface` writes a command to the mailbox of the target SPE. The `main()` function of the SPE kernel processes the command by choosing the kernel function to execute
2. The `Kernel1Interface` writes the address of the data structure required by the kernel to the mailbox. The kernel reads this address from the mailbox, and it uses it to transfer the wrapper structure via DMA.
3. The kernel code uses DMA transfers to get the "real" data in its LS.
4. Once the kernel gets the data via DMA, it performs the operation.
5. The kernel programs the DMA transfer to put results in the designated output buffer
6. The kernel signals its termination, putting a message in its output mailbox. The `Kernel1Interface` object gets the signal, either by polling or by an interrupt, and copies the results from the buffer back to the class data.

[2] http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Convolut-2.html

6.2 The Code Templates

To support our porting strategy, we have designed and implemented generic code templates for (1) the SPE controller, (2) the PPE-to-SPE interface class, (3) the PPE-to-SPE communication, and (4) the main application plug-in. To describe each of these templates, we assume the main application is already running on the PPE and kernels have been identfied and isolated as single methods (i.e., each kernel is only one method).

6.2.1 The SPE Controller

Each kernel becomes an SPE thread. As a generic application is likely to include multiple kernels, each SPE is configured to be able to execute all potential kernels. Therefore, the SPE controller has to include a selection mechanism that, under the command of the PPE (see the following sections), chooses, configures, and executes the right function.

The SPE code template is presented in the Listing 6.1. Note that for each `Kernel_n`, the `main` function enables both blocking (i.e., PPE polling for a new message) and non-blocking (i.e., the PPE is interrupted by the new message) behavior of the PPE-SPE messaging protocol.

Listing 6.1: *The SPE controller. Kernel_n are the kernels to be accelerated.*

```
1 // Kernel code :
2 int Kernel_1(unsigned int address) { ... }
3 int Kernel_2(unsigned int address) { ... }
4 //more functions ...
5 int main(unsigned long long spu_id, unsigned long long argv) {
6   unsigned int addr_in, opcode;
7   int result;
8  while(1) {
9   opcode = (unsigned int) spu_read_in_mbox();  // read operation code from the mailbox
10  switch (opcode) {
11    case SPU_EXIT: return 0;
12    case SPU_Run_1: {
13       // read the address from the mailbox
14       addr_in = (unsigned int)spu_read_in_mbox();
15       // run function
16       result = Function1(addr_in);
17       //write result in the (interrupt) mailbox
18       if (POLLING)
19          spu_write_out_mbox(result);
20       if (INTERRUPT)
21          spu_write_out_intr_mbox(result);
22       break;
23    }
24    case SPU_Run_2: {...}
25  ...
26 }
```

The major limitation of this SPE controller is the memory footprint of all kernels: if it exceeds the SPE memory, additional communication is required to fetch the right code and data from the main memory. To address this issue, Cell/B.E. provides overlays. Although very practical to use, they

add a small overhead which, for very short kernels, can lead to visible performance degradation. An alternative to overlays would be a more restrictive pre-assignment of functions on the SPEs.

6.2.2 The SPEInterface class

The SPEInterface class (presented in Listing 6.2) is the mechanism that allows a generic SPE kernel to be plugged in a C++ application.

Listing 6.2: *The SPEInterface class snippet*

```
1 class SPEInterface {
2 private:
3 // more fields
4 //the kernel code
5   spe_program_handle_t  spe_module;
6 // is the kernel running at all?
7   int started;
8 // how many SPEs run the same kernel
9   int                   mySPEs;
10 // which SPEs run this kernel
11   speid_t               whichSPEs[MAX_SPES];
12 // which SPEs running this kernel are active
13   speid_t               activeSPEs[MAX_SPES];
14
15 public:
16   SPEInterface(spe_program_handle_t module);
17   ~SPEInterface();
18   int thread_open(spe_program_handle_t module);
19   int thread_close(int cmd);
20   int SendAndWait(int functionCall, unsigned int value);
21   int Send(int functionCall, unsigned int value);
22   int Wait(int timeout);
23 // more code if needed
24 };
```

Typically, the class is instantiated for each type of kernel, and used by the main application to manage the kernel execution on the SPEs. To allow the same kernel to be executed by multiple SPEs, each kernel class has a list of active SPEs that execute it. They can be managed individually or collectively. The class could be extended to also allow subgroups.

6.2.3 The PPE-SPE communication

To implement the SendAndWait method, we have used a simple 2-way communication protocol based on mailboxes, as seen in Listing 6.3.

Listing 6.3: *The SendAndWait protocol*

```
1 int SPEInterface::SendAndWait(int functionCall, unsigned int addr, int spuID) {
2   int retVal;
3 // send command and address to all SPEs
4   if(spuID<0) {
5 // first send all
6    for (i=0; i<mySPEs; i++) {
```

```
7      spe_write_in_mbox(whichSPEs[i],functionCall);
8      spe_write_in_mbox(whichSPEs[i],addr);
9    }
10 // then wait for all results (i.e., completion signal)
11 // do it smartly: avoid busy waiting for each SPE, and iterate over the SPEs
12 // without waiting for one that is not ready yet
13   int count = mySPEs;
14
15   while (count>0) {
16        for (i=0; i<mySPEs; i++) {
17          if (activeSPEs[i]) > 0) {
18             if (spe_stat_out_mbox(whichSPEs[i])>0 ) {
19                int tmp = spe_read_out_mbox(mySPEs[i]);
20                 retval |= (tmp << i);
21                activeSPEs[i]=0;
22                count--;
23             }
24          }
25        }
26   }
27  else {
28 // send command and address to an individual SPE
29 // first check SPE is active and then send
30   if (activeSPEs[spuID]>0) {
31    spe_write_in_mbox(whichSPEs[spuID],functionCall);
32    spe_write_in_mbox(whichSPEs[spuID],addr);
33 // wait for result (i.e., completion signal)
34    while(spe_stat_out_mbox(whichSPEs[spuID])==0);
35 // read result to clean-up the mailbox
36    retVal=spe_read_out_mbox(whichSPEs[spuID]);
37  }
38  return retVal;
39 }
```

Again, note that the SPEs can be manged individually or collectively. We note that enabling collective operations efficiently requires nested loops, to avoid waiting for too long for a "lazy" SPE. Thus, instead of synchronous waiting at each mailbox, we just test if the kernel is ready and, if not, we move on. This strategy minimizes the processor' idle time.

However, the `SendAndWait` protocol forces the PPE to wait for the completion of each kernel before launching the next one. To allow multiple kernels to be launched in parallel, we allowed an interrupt based protocol: the PPE sends the commands to the SPEs and returns to its own execution; an SPE that has finished processing generates an interrupt which will be caught and handled by the PPE. The two methods that implement this protocol, `Send` and `Wait`, are presented in Listing 6.4.

Listing 6.4: *The Send method*

```
1 void SPEInterface::Send(int functionCall, unsigned int addr, int spuID) {
2   int retVal;
3
4   if (spuID < 0) {
5     for (i=0; i<mySPEs; i++) {
6         spe_write_in_mbox(whichSPEs[i],functionCall);
```

```
7     }
8 // write to the mailbox the address of the msg_buffer
9     for (i=0; i<nSPEs; i++) {
10        spe_write_in_mbox(whichSPEs[i],addr);
11 // signal the kernel is ON
12        activeSPEs[i] = 1;
13    }
14 }
15
16 int SPEInterface::CheckInterrupt(int timeout, int previous) {
17   spe_event evt[MAX_SPES];
18   int events, intrs = 0;
19   int enabled = previous;
20
21   if (!started) return -1;
22 // we are only interested in interrupts from the mailbox
23   for (i=0; i<mySPEs; i++) evt[i].events = SPE_EVENT_MAILBOX;
24
25   events = spe_get_event(evt, mySPEs, timeout);
26 __asm__ __volatile__ ("sync" : : : "memory");
27
28   if (events>0) {
29       for (j=0; j<mySPEs; j++) {
30            if (evt[j].revents > 0)
31               for (k=0; k<nSPEs; k++)
32                    if (evt[j].speid==mySPEs[k]) {
33                         enabled ^= (1<<k);
34                         break;
35                 }
36        }
37   }
38
39   return enabled;
40 }
41
42 int CellInterface::Wait(int remaining)
43 {
44        int retVal = remaining;
45        if (!started) return -1;
46
47        int x = CheckInterrupt(0, retVal);
48        retVal = min(x, retVal);
49        return retVal;
50 }
```

These four methods are all that is necessary to implement both the synchronous and the asynchronous PPE to SPE communication needed in our offloading model. In the following section we show the way they are used.

6.2.4 Changes in the main application

As we mentioned multiple times, our goal is to keep the control flow of the main application unchanged. Therefore, we implemented the accelerated kernels as alternative paths. The changes required to allow a kernel to execute either on the SPE or on the PPE are presented in Listing 6.5.

Listing 6.5: *The changes in the main application.*

```
1 // interface initialization
2 #ifdef RUN_ON_SPU
3 // SPE_Kernel1 is the SPE implementation of the original code
4   extern spe_program_handle_t SPE_Kernel1;
5   SPEInterface *Kernel1Interface = new CellInterface(SPE_Kernel1);
6 #endif
7 /* ... more code ... */
8 #ifdef RUN_ON_SPU
9 int& Kernel1::run(SomeObjectType *someObject) {
10 // data wrapping
11   DATA_WRAPPING(aligned_data_structure, someObject);
12 // call SPE kernel
13   int val=Kernel1Interface->SendAndWait(SPU_Run, aligned_data_structure);
14 // put data back
15   memset(result, (float *)aligned_data_structure.result, RESULT_SIZE);
16 // free buffer from wrapped data
17   FREE_DATA_WRAPPER((void *)aligned_data_structure);
18 /* ... more code ... */
19 }
20 #else
21 int& Kernel1::run(/* parameters */)
22 {
23    // the original code
24 }
25 #endif
```

Data-wrapping is necessary for allowing each SPE to only wait (and receive) a single address - the one of the data wrapper. Thus, a data wrapper is nothing else than a collection of pointers to the member data of the class used by the SPE kernel. There is only one copy of the data in the main memory, and it is the one that is transferred by DMA to the kernel itself.

6.3 Overall Application Performance

In this Section we briefly discuss the SPE-specific optimizations and their potential influence in the overall application performance gain.

6.3.1 The Kernel Optimizations

After the initial porting, based as much as possible on reusing the C++ code, the kernels have to be further optimized. Among the optimizations that can be performed [Bla06b] we quote:

- optimize the data transfer - either by DMA multi-buffering, or by using DMA lists
- vectorize the code - to make use of the SIMD abilities of the SPEs; use the data type that minimally suffices the required precision, to increase the SIMDization ways.

- use local SPE data structures that are contiguous (i.e., use arrays rather then linked lists).
- remove/replace branches or, if not possible, use explicit branch hints (using the `__builtin_expect` function) to provide the compiler with branch prediction information
- change the algorithm for better vectorization - interchange loops, replace multiplications and divisions by shift operations, etc.

All these optimizations are highly dependent on the kernel processing itself, so there are no precise rules or guidelines to be followed for guaranteed success. Even though a large part of the overall application performance gain is obtained by these optimizations, they are not the subject of this study, so we do not explain them further, but refer the reader to [Bla06a; Pet07]. Still, we note that due to the inherent modularity of our strategy, which allows subsequent optimizations to be performed iteratively, as different kernel versions that adhere to the same interface can be easily plugged in via the `SPEInterface` stub.

6.3.2 Evaluate application performance

In the case of large C++ applications, the overall performance gain is highly dependent on the application structure, in terms of (1) how representative the kernels are in the overall computation, and (2) how can they be scheduled to compute in parallel. Thus, although one can assume that the speed-up obtained from an optimized kernel running on one SPE can be between one and two orders of magnitude (of course, depending on the kernel) compared to its PPE version, the overall application speed-up may be much lower if all these kernels are only covering a small part of the entire application. In terms of scheduling, if the application preserves the initial execution style, the PPE stalls during the SPE execution, which actually translates to a sequential execution on multiple cores, as see Figure 6.3(b). If the structure of the application allows it, the execution model should increase concurrency by using several SPEs and the PPE in parallel, as seen in Figure 6.3(c). We shall analyze both these scheduling solutions in the following paragraphs, and present results for both scenarios.

To estimate the performance gain of kernel optimizations, we use Amdhal's law as a first order approximation. Given K_{fr} as the fraction of the execution time represented by a kernel, and $K_{speed-up}$ as the speed-up of the kernel over the PPE, the resulting application speed-up, S_{app} is:

$$S_{app} = \frac{1}{(1 - K_{fr}) + \frac{K_{fr}}{K_{speed-up}}} \tag{6.1}$$

For example, for a kernel with K_{fr}=10% of an application, a speed-up $K_{speed-up} = 10$ gives an overall speed-up $S_{app} = 1.0989$, while the same kernel optimized to $K_{speed-up} = 100$ gives an overall speed-up $S_{app} = 1.1098$. Thus, the effort of kernel optimization from $K_{speed-up} = 10$ to $K_{speed-up} = 100$ is not worth in this case.

Considering that n SPE kernels $(K^i, i = 1..n)$ run by the main application sequentially, like in Figure 6.3 (b), the overall performance can be estimated by:

$$S_{app} = \frac{1}{(1 - \sum_{i=1}^{n} K_{fr}^i) + \sum_{i=1}^{n} \frac{K_{fr}^i}{K_{speed-up}^i}} \tag{6.2}$$

Application performance can be improved in case the PPE application can schedule groups of SPE kernels in parallel, like in Figure 6.3 (c). Consider the n kernels split in G groups, each group having its kernels running in parallel. Note that the groups (due to data dependencies, for example) are

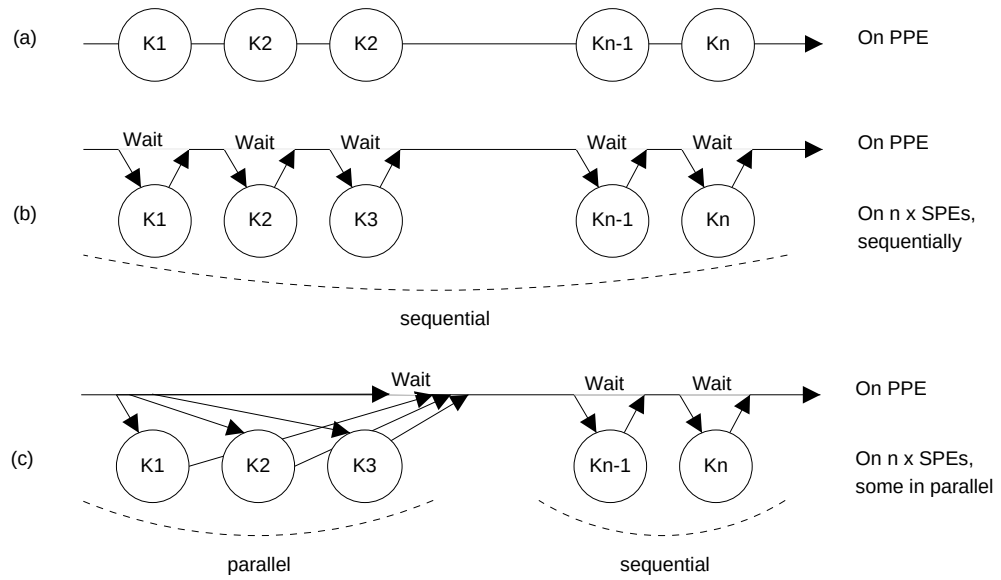


Figure 6.3: *Application scheduling on the Cell/B.E. cores. (a) The original application, with embedded kernels (K1...Kn) running on the PPE; (b) Kernels run on SPEs, sequentially; the PPE waits kernel termination before continuing; (c) Kernels can execute in parallel; PPE continues after each kernel launch and continues working until interrupted by kernel finish or it had no more work*

still executed sequentially. If each group has g_j items, such that $n = \sum_{j=1}^{G} g_j$, the speed-up can be estimated by:

$$S_{app} = \frac{1}{(1 - \sum_{i=1}^{n} K_{fr}^{i}) + \sum_{j=1}^{g} \max_{k=1}^{p_j} (\frac{K_{fr}^{l}}{K_{speed-up}^{l}})} \tag{6.3}$$

By evaluating equations 6.2 and/or 6.3, one can determine (1) what is the weight of kernel optimizations in the overall application speed-up, and (2) what can be the gain of running the SPEs in parallel. Comparing these numbers (examples will follow in Section 6.4) with the estimated effort for the required changes for these different scenarios, we can evaluate the efficiency of the porting process.

6.4 MarCell: A Case-Study

MARVEL[3], is a multimedia content analysis designed to help organizing large and growing amounts of multimedia content much more efficiently. MARVEL uses multi-modal machine learning techniques for bridging the semantic gap for multimedia content analysis and retrieval. After a short training phase, MARVEL is able to automatically annotate multimedia content, thus allowing further searching for and retrieval of content of interest. For more details on the application, we refer the user to Chapter 5.

MARVEL consists of two engines: (1) the multimedia analysis engine, which applies machine learning techniques to model semantic concepts in video from automatically extracted audio, speech, and visual content [Ami03], and (2) the multimedia retrieval engine, which integrates multimedia

[3]MARVEL stands for Multimedia Analysis and Retrieval

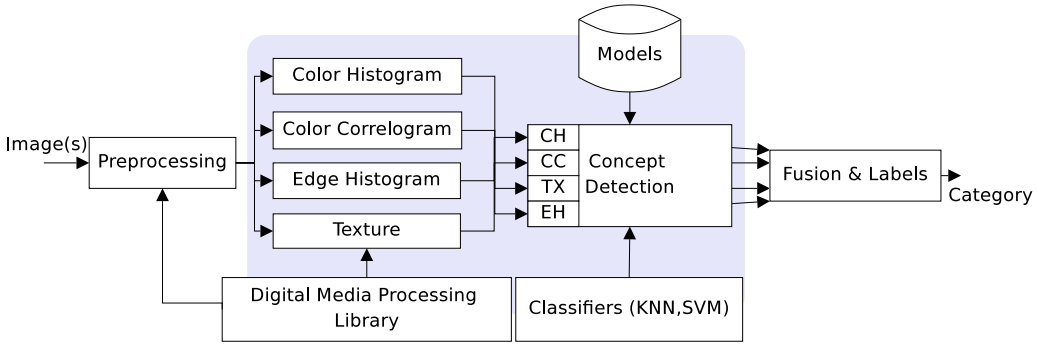


Figure 6.4: *The processing flow of Marvel. The shaded part is ported onto the SPEs.*

semantics-based searching with other techniques for image and/or video searching [Nat04]. In this work, we focus on MARVEL's multimedia analysis engine. Our goal is to increase the execution speed of semantic concept detection in images by using a multi-core processor as an accelerator. This is a two-phase process: first, visual features (e.g., color, edge, and texture information) are extracted from the image, and then statistical concept models (e.g., Support Vector Machines models) are evaluated on these features. The processing flow of this simplified MARVEL version is presented in Figure 6.4. The preprocessing step includes (1) image reading, decompressing and storing it in the main memory as an RGB image, and (2) models reading and storing in memory. From the RGB representation, we extract some visual features of the image as vectors. The extracted features go through the concept detection phase, based on a collection of precomputed models and using one of the several available statistical classification methods like Support Vector Machines (SVMs), k-nearest neighbor search (kNN), etc. For our case-study, we have chosen four feature extraction techniques, for color (two), edge, and texture information, and we have opted for an SVM-based classification.

6.4.1 Porting MARVEL on Cell

It took roughly one day to compile and execute the Linux version of Marvel on the PPE. Next, we have profiled its execution for one 352x240 pixels color image, and for a set of fifty such images. For one image, the feature extraction and concept detection represented 87% of the execution time (the rest being application preprocessing); for the 50-images set, feature extraction and concept detection increased to 96% of the execution time (a large part of the preprocessing is a one-time overhead). Further, based on the profiling information, we have identified the core methods (in terms of execution time) for each of the feature extraction algorithms, as well as for the concept detections.

Around these core methods we have clustered the additional methods required for easy data wrapping, processing and transfer, and we defined five major kernels, briefly listed below together with their coverage from the per-image application execution time (i.e., *without* the one-time overhead):

1. Color histogram extraction (`CHExtract`) - 8% from the application execution time.
   The color histogram of an image is computed by discretizing the colors within an image and counting the number of colors that fall into each bin [Smi96]. In MARVEL, the color histogram is computed on the HSV image representation, and quantized in 166 bins.

2. Color correlogram extraction (`CCExtract`) - 54% from the application execution time.

The color correlogram feature in MARVEL quantifies, over the whole image, the degree of clustering among pixels with the same quantized color value. For each pixel P, it counts how many pixels there are within a square window of size 17x17 around P belonging to the same histogram bin as P [Hua97b].

3. Texture extraction (`TXExtract`) - 6% from the application execution time.
   Texture refers to a visual pattern or spatial arrangement of the pixels in an image. In MARVEL, texture features are derived from the pattern of spatial-frequency energy across image subbands [Nap02].

4. Edge histogram extraction (`EHExtract`) - 28% from the application execution time.
   The edge histogram extraction is a sequence of filters applied in succession on the image: color conversion RGB to Gray, image edge detection with the Sobel operators, edge angle and magnitude computation per pixel, plus the quantization and normalization operations specific to histogram-like functions.

5. Concept Detection - 2% from the application execution time for the classification of all four features: `CHDetect`, `CCDetect`, `TXDetect`, `EHDetect`.

The remainder of the application execution time for one image (2%) is the image reading, decoding and rescaling. In our tests, we used all images of the same size, such that rescaling (otherwise a costly operation) is not required.

6.4.2 Building MarCell

Once the five kernels have been identified, they are ported onto the SPEs. Every kernel becomes an SPE thread, structured as presented in Listing 6.1.

For three of the five chosen kernels, we have first measured the behavior before SPE-specific optimizations. Compared with the PPE version, the speed-ups of CHExtract, CCExtract and EHExtract were 26.41, 0.43 and 3.85, respectively. The significant difference in these results are mainly due to the specific computation structure of each kernel. Further on, we have proceeded with the SPE code optimizations, tuning the "reference" algorithm to allow better opportunities for a broad range of manual optimizations. like double and triple buffering of DMA transfers, loop unrolling, 16-ways and/or 4 ways SIMD-ization, branch removal, etc. Table 6.4.2 shows the final speed-up of each SPE kernel over its initial PPE version.

Once the kernels have been completely implemented, they have to be reintegrated in the application itself. We took the following steps:

1. We instantiated an instance of the `SPEInterface`(Listing 6.2) class for each kernel; each kernel is assigned on a single SPE.

2. We changed the kernel codes according to the template presented in Listing 6.1.

3. We used synchronous communication (Listing 6.3) between the PPE and the SPEs.

4. We modified the main application classes to include both PPE and SPE execution branches (see Listing 6.5).

Table 6.1: *SPE vs. PPE kernel speed-ups*

Kernel	PPE [ms]	Overall contribution	SPE [ms]	Speed-up vs. PPE
CH Extract	42.82	8%	0.82	52.22
CC Extract	325.43	54%	5.87	55.44
TX Extract	31.28	6%	2.01	15.56
EH Extract	225.80	28%	2.48	91.05
CDet	2.93	2%	0.41	7.15

The first version of MarCell, following this procedure, uses 5 SPEs (one per kernel), executed sequentially - i.e., only one SPE is active at a time. Note that by tuning and/or replacing `SendandWait` to implement asynchronous communication, one can exploit the parallelism *between* the kernels as well. The following section is dedicated to the MarCell's performance.

6.4.3 Experiments and results

For MarCell, we have performed experiments on sets of one, ten and fifty images. For each image, we have performed the four feature extractions and the corresponding concept detection. The concept detection was performed with a number of models summing up 186 vectors for color histogram, 225 for color correlogram, 210 for edge detection and 255 for texture.

Next, we have calculated the application estimated performance, using equations 6.1 and 6.2 from Section 6.3, and using the measured speed-up numbers for each kernel. In all scenarios, we map at most one kernel per one SPE. We have evaluated three scenarios (the same as the ones presented in Figure 6.3:

1. All kernels execute sequentially (i.e., no task parallelism between SPEs). This scenario is equivalent with the use of a single SPE, but it avoids the dynamic code switching:
   $S_{app}^{SingleSPE}$(SPE:Desktop)=10.90

2. All feature extractions run in parallel; the concept detection is still running on a single SPE, thus all concept detection operations run sequentially:
   $S_{app}^{Multi-SPE}$(SPE:Desktop)=15.28

3. All feature extractions run in parallel; the concept detection code is replicated on the other 4 SPEs, to allow parallel execution, and each extraction is immediately followed by its own detection:
   $S_{app}^{Multi-SPE2}$(SPE:Desktop)=15.64. The very small difference between the two parallelization solutions can be explained by (1) the high impact of the color correlogram extraction, which dominates the feature extraction operations, (2) the small impact of the concept detection (only 0.5%, on average), and (3) the image preprocessing part, which runs on the PPE.

We have performed experiments for the same scenarios. We have compared the time measurements with the application running on PPE, Desktop, and Laptop. The results, matching the estimates with an error of less than 8%, are presented in Table 6.2.

Table 6.2: *Predicted versus measured execution time for MarCell.*

Scenario[ms]	Predicted speed-up	Measured speed-up	Error
SingleSPE	10.90	10.10	+7.7%
MultiSPE	15.28	14.90	+2.5%
MultiSPE2	15.64	15.40	+1.5%
Hybrid	23.87	22.80	+4.7%

The experimental results show that, for these three task-parallel scenarios, MarCell achieves about 15x speed-up over MARVEL running on a commodity (sequential) machine. Furthermore, following our application porting strategy, we were able to have a first MarCell implementation in a couple of weeks (excluding the 2.5 months spent on optimizing the five kernels). Finally, using the prediction equations, we can not only determine the performance the application gets, but we can also estimate if new kernel optimizations (that would take more development time) would pay off in the overall application performance.

We have applied the same methodology to calculate the potential gain of a hybrid parallelization of MarCell - the best performing one, according to Section 4.4. We have chosen the 3-4-3-4 scenarios for this exercise, aiming to determine if the 24x speed-up is correctly predicted. We used the same Equation 6.3, only by replacing the execution time of the entire kernel with the execution time of a slice of the kernel. For this scenario, the prediction was still correct, and the observed error was below 5% (see Table 6.2). Note that the lack of linear scalbility of the data parallel kernel executions does complicate things by not allowin a simple estimation of the execution of a kernel when split in P slices as $\frac{T_{kernel}}{P}$; instead, one should measure and tabulate the performance values for various P in order for the predictions to be fairly accurate.

6.5 Related Work

Most of the work done so far on the Cell/B.E. processor has focused either on (1) application development and performance evaluation, aiming to prove the level of performance that the architecture can actually achieve, or on (2) high level programming models, proving that the programmer can be abstracted away (to some degree) from the low-level architectural details. In this respect, our approach falls somewhat in between, as we present a generic strategy for fast porting of sequential C++ applications onto the Cell, and we prove its applicability by developing and evaluating a case-study application.

On the programming models side, the MPI microtask model, proposed in [Oha06b], aims to infer the LS administration and DMA transfers from the message passing communication between microtasks that are defined by the user. Once this infrastructure is defined, a preprocessor splits the microtasks in basic tasks, free of internal communication, and schedules them dynamically on the existing SPEs. While the notions of kernels and microtasks/basic tasks are quite similar, we note that the MPI microtasks work on C only. For using this model on C++ applications, the programmer will have to first apply our strategy: kernel identification, isolation, data wrapping. While the clustering and scheduling of the microtask model are quite efficient, it will also require more transformations on

the kernel side, thus decreasing the efficiency of a proof-of-concept porting.

CellSs, a programming model proposed in [Bel06], provides a user annotation scheme to allow the programmer to specify what parts of the code shall be migrated on the SPE. A source-to-source compiler separates the annotated sequential code in two: a main program (that will be running on the PPE) and a pool of functions to be run on the SPEs. For any potential call to an SPE-function, a runtime library generates a potential SPE task which can be executed when all its data dependencies are satisfied, if there is an SPE available. However, CellSs cannot directly deal with C++ applications.

On the applications side, the first applications running on the Cell/B.E. have been presented by IBM in [Min05; D'A05]. In [Min05], a complete implementation of a terrain rendering engine is shown to perform 50 times faster on a 3.2GHz Cell processor than on a 2.0GHz PowerPC G5 VMX. Besides the impressive performance numbers, the authors present a very interesting overview of the implementation process and task scheduling. However, the entire implementation is completely Cell oriented and tuned, allowing optimal task separation and mapping, but too low a flexibility to be used as a generic methodology for porting existing applications.

An interesting application porting experiment is presented in [Ben06], where the authors discuss a Cell-based ray tracing algorithm. In particular, this work looks at more structural changes of the application and evolves towards finding the most suitable (i.e., different than, say, a reference C++ implementation) algorithm to implement ray tracing on Cell. Finally, the lessons learned from porting Sweep3D (a high-performance scientific application) on the Cell processor, presented in [Pet07], were very useful for developing our strategy, as they pointed out the key optimization points that an application has to exploit for significant performance gains.

6.6 Summary and Discussion

To the best of our knowledge, this work is the first to tackle the systematic porting of a large C++ application onto the Cell/B.E.. Our approach is based on detecting the compute intensive kernels from the initial application, isolate them from the C++ class structure and migrate them for execution on the SPEs, while the rest of the C++ application runs on the PPE.

The method is generic in its approach, being applicable for any C++ application. The porting process allows the application to be functional at all times, thus permitting easy performance checkpoints of intermediate versions. The optimization of the SPE kernels code is still the responsibility of the programmer, but given that the resulting application is modular, it allows for iterative improvements. Finally, to support effective optimization of the SPEs, we provide a simple method to evaluate the performance gain of a potential optimization in the overall performance of the application.

We have validated our methodology by presenting our experience with porting a multimedia retrieval application with more than 20000 lines of code and over 50 classes. The performance of the ported application - namely, the overall application speed-up of the Cell-based application - is one order of magnitude above the one of a commodity machine, running the reference sequential code.

We conclude that our strategy is well-suited for a first attempt of enabling an existing C++ code-base to run on Cell. It is recommended for large applications, when rewriting from scratch in a Cell-aware manner is not really an option. Also, the results of our strategy can be used for proof-of-concept approaches evaluating the performance potential of Cell/B.E. for given applications.

Chapter 7

Multi-core Applications Analysis and Design[§]

.

In this chapter we provide several empirical guidelines for analyzing, designing, and implementing multi-core applications, as identified in our case-study work. Further, we present a survey of the available multi-core programming tools, in search for one that offers enough support for adopting these guidelines along the development process of new applications.

As multi-cores evolve towards increased complexity for cores, memories, and interconnects, the fundamental challenge they pose to applications programmers - the multiple layers of parallelism that need to be exploited to achieve performance - will also increase. Our analysis of three different types of workloads showed that these multiple layers of parallelism have a huge influence on application design, development, and deployment. In this chapter, we generalize our findings and propose several rules for efficient multi-core applications design and development.

We first summarize the lessons we have learned from our case studies (see Section 7.1), and we build a comprehensive list of multi-core application development guidelines. We separate these findings in two categories: platform-agnostic and platform-specific.

Further, we focus on understanding how does the programmability gap (see Section 7.2) affect the adoption of multi-cores and multi-core programming models. We show how the gap can be bridged by programming models that adhere to simple set of programmability requirements and constraints.

Finally, we analyze an extensive set of multi-core specific programming models, languages, and tools (see Section 7.3) that aim to increase platform programmability. The outcome of our survey is not very encouraging: programming models are either application-centric (too high-level) or hardware-centric (too low-level), and the gap between these categories is widening.

We did find one promising approach in the OpenCL standard, which boosts both portability and productivity by proposing a generic platform model, used as middleware on the hardware side, and as a virtual development platform on the software side. Still, the potential of this approach is limited to both platforms and parallel applications that map "naturally" on the OpenCL platform model.

We conclude that the principles of OpenCL need to be extended to a strategy that enforces a disciplined approach to application design, implementation, and optimization, all guided by performance constraints and feedback. This strategy is the foundation of our MAP framework, described in Chapter 8.

7.1 Generic Findings and Empirical Guidelines

In this section, we present our findings and guidelines for application design, development, and analysis, as we have extracted them from our application studies.

[§]This chapter is partially based on joint work with Barcelona Supercomputing Center, previously published in the proceedings of CPC 2009([A.L09])

7.1.1 Case Studies Overview

We have extensively discussed three different applications and their parallelization for multi-core processors. To have a clear picture of the important elements of each problem-solution pair, as well as a handy comparison between these applications, Table 7.1 presents a brief overview of the case-studies.

Table 7.1: *An overview of the case-studies and their characteristics from the multi-core design and implementation challenges*

	1. Sweep3D	2. MarCell	3. Gridding/degridding
Application type	HPC mono-kernel Compute-bound	Image-processing multi-kernel Mixed	Scientific mono-kernel Data-intensive
Parallelism	SPMD, SIMD	SIMD, SPMD, MPMD	SIMD
Platforms	Cell, MC-CPU	Cell, MC-CPU	Cell, MC-CPU, GPU
Focus	Kernel-tuning	Mapping/scheduling, Kernel-tuning	Mapping/scheduling
Algorithm changes	Kernel: High Overall: Minimal	Kernels: High Overall: Minimal	Kernel: Minimal Overall: Med
Data reorg.	High	Minimal	High
Highest impact on performance	low-level optimizations	high-level parallelization	data-reorganization, (dynamic) scheduling
Highest impact on portability	granularity, scheduling	granularity, MPMD, scheduling	scheduling
Highest impact on productivity	algorithm changes, low-level optimizations	static scheduling	data rearrangement locality increase

7.1.2 Problems and empirical solutions

In our application analysis research, we have identified a set of problems with high impact on application and/or platform programmability, and we found a set of working solutions for solving them. We coarsely classify these lessons in two disjoint classes: application-centric (also known as platform-agnostic) and platform-specific. Platform-agnostic lessons and guidelines apply mostly to the algorithm design and parallelization, while the platform-specific ones are mostly useful for implementation and tuning.

Application-centric lessons and guidelines

We have identified twelve empirical rules that are very useful for application specification, analysis, and design:

1. *Kernels* are the computational units of an application. Parallelizing an application starts by "splitting" it into independent kernels, which will be further distributed on the (worker) cores. The number and granularity of these kernels are essential for choosing the right target platform. For example, an application with a lot of independent kernels should first consider targeting a GPMC platform, and not a GPU platform; an application with very few independent task-level kernels will have to make use of data-parallelism to utilize a multi-core platform, so it might as well target GPUs first.

2. *Concurrency and granularity* are two interconnected elements with high impact on application performance and portability. Mapping an application on a given platform usually requires matching (1) the number of its concurrency levels with the hardware layers of the platform, and (2) the granularity of the concurrent elements (kernels) with the functional unit(s) that will execute them.

3. *The minimal granularity* of a kernel is the smallest granularity a kernel can reach in terms of parallelism (and, consequently, the higher degree a concurrency an application can achieve). Reaching this minimal granularity typically requires kernels to enable data-parallelism. Note that slim-core multi-cores (like the GPUs) need this level of granularity to get a reference implementation. Further, having the minimal granularity decomposition of an application enables flexible granularity (as kernels can be composed to create coarser compute units), and therefore increases parallelization portability.

4. *The application graph* is a Directed Acyclic Graph (DAG) that summarizes the available knowledge on functional and data dependencies between kernels. Built at the minimal kernel granularities, this graph exposes the maximum application concurrency. Thus, mapping this graph on a target platform gives the (multiple) potential parallel implementations of the given application on the platform.

5. *Data locality and memory footprint* are two more indicators of proper application-platform match. Given the memory constraints per core, data locality is essential for multi-core performance: off-chip transfers are very expensive, especially when compared with the very fast computation that the cores themselves can do. Optimizing for data locality translates into bringing all data required by a kernel running on a particular core in the core's local memory. Memory footprint is a simple metric used for balancing the kernel data locality requirements with the resources available on the target core.

6. *Operational intensity* is a measure of the computation intensity of an application. Computed as the number or arithmetic-like operations per accessed memory byte, it is the simplest indicator to determine compute- or memory-bound behavior. Bringing the operational intensity of a kernel/an application closer to the peak FLOP:byte ratio of the target platform will always result in a performance increase. Note that while the operational intensity is a characteristic of the application, its value is relevant *only* in the context of a certain platform.

7. The *parallelization model* of multi-core applications is typically a combination of MPMD and SPMD: multiple independent tasks, each further data-parallelized. Therefore, implementing a multi-core application should be based on a variant of the master-workers paradigm, where the "master" execute(s) the control-flow, and the workers perform bulk computations. Ideally, the master code should run on a dedicated (master) core.

8. The *control flow and data flow* have to be isolated as much as possible, especially in the context of multi-core platforms, where the working cores are very slow at taking decisions and/or their diverging paths usually lead to serialization (especially on GPU-like architectures). Separating the two is not trivial, especially when starting from sequential code, where the interleaving of the two can be quite complex. A good approach is to first implement (as much as possible from) the master's control flow, using "dummy" stubs for the kernels, and only afterwards implement the computation itself for the workers.

9. *Mapping and scheduling* are very different among the families of multi-core processors. Unless the platform allows for fully user-controlled scheduling (like the Cell/B.E.), matching a multi-task application with a platform also requires investigating what the scheduling policies of the platform are.

10. *Low-level optimizations* are kernel optimizations, platform dependent in their implementation, but based on general methods, usually stemming from compiler research. Examples are: vectorization, data transfer latency hiding, data structure alignment, use of special intrinsics, etc. However, these optimizations often require detailed code restructuring, and they are tedious to implement; further, they target improvements in the computation only. Therefore, implementing these optimizations should be left for the final stages of the development process, as (1) they typically impact performance *only* when the application is compute-bound, and (2) they clutter the code, potentially hiding higher-level optimizations.

11. *Data dependent optimizations* are opportunistic optimizations, which take into account additional properties (such as ordering, statistical, or mathematical correlations) of the input data set(s). For example, knowing that the input data is sorted can have a big impact on the performance of a searching algorithm. Using these correlations is recommended: although they may limit program generality, they lead to significant performance improvements, typically by improving operational intensity and/or increasing data locality.

12. *Performance goals* are typically specified in terms of execution time, throughput, or platform utilization; however, efficiency (ranging from energy consumption to programming effort) can also be used. Setting performance bounds is essential in limiting the application development time, thus impacting productivity and efficiency.

Platform-specific lessons and guidelines

We list here lessons and guidelines for the three families of multi-core platforms discussed throughout this thesis. Note that these guidelines are based on empirical observations coming from application porting research; with these, we aim to complete the processors' handbooks, not to replace them.

- Cell/B.E.

  What makes programming on the Cell/B.E. difficult is the complexity of the architecture combined with the five parallelization layers the programmer needs to exploit. Thus, it is essential for programmers to choose the right parallelization strategy, as well as an effective way to apply mapping/scheduling and SPE-level optimizations. The following are a number of recommendations on mapping and scheduling, data movement, and effective ways to apply the SPE optimizations:

  1. Use a parallelization model with a simple, *centralized* control flow. Preferred solutions are master-workers or a pipeline model. Keep the control flow as much as possible on the PPE.
  2. Use the PPE for all scheduling *decisions*. Favor dynamic scheduling, as it typically leads to higher utilization.
  3. Analyze and evaluate the possible mapping scenarios, keeping the core utilization high and the data transfers low.
  4. Assign independent, regular work to the SPEs. Ideally, the SPEs should fetch work from a common/private pool and execute the same (set of) kernel(s) for each item.

5. Avoid SPE kernels with multiple branches. If branches are inevitable, consider decomposing further into smaller kernels (i.e., different kernels for the various branches) and moving the decision on the PPE.
6. Avoid SPE context switches, because thread killing and (re-)creation has high overheads; furthermore, as these are both sequential operations done by the PPE, the overall application penalty is a sum of these overheads.
7. Avoid SPE synchronization, especially if the PPE is involved (barrier like, for example).
8. Maximize kernel granularity to fit the memory footprint to the SPEs available memory; typically, this requires composing (by mapping) multiple minimal granularity kernels to run on the same SPE.
9. Consider dual-threading on the PPE, to speed-up the scheduling.
10. Use the SPEs idle time to pre-process data (i.e., if needed, by sorting or shuffling), as long as they only use the local memory in the process. In many cases, these operations enable data-reuse and speed-up the application.
11. Apply SPE optimizations if the application is compute bound. Then, focus first on vectorization.
12. Increase data locality, by intelligent mapping - e.g., program the PPE to map data-dependent kernels on the same SPE minimize the DMA transfers.
13. Investigate potential data-dependent optimizations if the application is memory bound. Avoid applying compute-centric optimizations before analyzing the potential to increase data re-use.

- GPMC

  Compared with the Cell/B.E., GPMCs are a lot easier to program, featuring the well-known shared memory model. While this enables easier communication between cores, it also brings synchronization and consistency problems. Furthermore, due to the perceived ease-of-use of the platform, some of the main hardware performance boosters are rarely used, leading to unfair performance comparisons with other platforms. Some important, hands-on experience guidelines include:

  1. Use symmetrical parallelization with coarse granularity kernels - it produces the best results, given the typical complex cores GPMCs have.
  2. Consider caches and line sizes when reading/writing the memory. Align data to cache lines boundaries.
  3. Do not count on thread affinity - assume context switches will appear, even if under-utilizing the number of cores.
  4. Consider benchmarking the application with more threads than the hardware available cores/threads - the operating system might interleave the execution and make use of the stall times, leading to an unexpected increase in performance.
  5. Avoid writing conflicts by using data replicas (followed by a reduction stage for the final result) rather than synchronization.
  6. Use SSE instructions for lower-level optimizations.

7. Apply data-dependent optimizations, including (local) sorting and shuffling.
8. Be fair when comparing performance - apply similar optimizations on all platforms before choosing a "winner".

- (NVIDIA) GPUs

  Graphical processing units have a complex memory architecture and a large collection of slim cores. Unsuitable task granularity and inefficient data distribution are usually responsible for high performance penalties. We recommend the following approach:

  1. Use a symmetrical parallelization with minimal granularity kernels (i.e., an SPMD model, with fine-grain processes);
  2. Maximize the number of threads, as the mapping and scheduling, provided by the hardware, require large collections of processes for optimal execution performance.
  3. Serialize application-level tasks, as MPMD is not possible as such on the GPUs; decompose each task in minimal grain kernels and use these as work-items for the GPU cores.
  4. Use native/fast math operations when floating point precision is not an issue - they are usually an order of magnitude faster, and their accuracy is typically enough for most applications.
  5. Avoid branches on the device side - especially those that tend to generate high divergence between threads. As solutions, consider (1) computing both branches and postpone the decision or (2) sorting data according to the potential branch the thread might take.
  6. Minimize data transfers between the host and the device.
  7. Force data re-use in the kernels, typically by reshuffling computations; this usually alters the order in which computations are performed in the algorithm.
  8. Optimize shared memory and register utilization; do not avoid slight algorithmic changes if they enable the use of local memory.
  9. Use registers, and shared and constant memories as much as possible; leave the (device) global memory as the last solution.
  10. Coalesce the memory reads and writes, as it is essential for performance.
  11. Resort to data dependent optimization only if it reduces branches or data transfers; if it only reduces the number of threads, this will have little impact on the overall performance.

Although the platform specific guidelines are quite different for the three platform families, we note that all of them can be broadly put in three categories: (1) platform-specific computation, (2) memory alignment and traffic, and (3) task management.

We conclude that for overall application performance, there are three essential application classes of features that need to be taken into account: low-level optimizations, mapping and scheduling (partially platform specific), and memory operations (including data locality).

7.2 The Programmability Gap

The programmability gap has been defined (shortly after multi-cores have been introduced) as the (large) gap in performance between the theoretical hardware peaks and real-life applications. In the

meantime, platform programmability has become a key factor in the quick and painless adoption of multi-core processors.

For application developers, platform programmability is reflected by a combination of three major issues: performance, portability, and productivity. Furthermore, programming models aim to increase application portability and productivity without sacrificing performance [Cha07].

In this section, we detail our definitions for all these features - programmability on the platform side, and performance, portability, and productivity on the programming side. Further, we presents a list of features that we believe are essential for a programming tool to properly address the essential challenges of multi-core programming.

7.2.1 Platform Programmability

Programmability is qualitative metric of a platform, usually associated with the flexibility in accepting various types of workloads/applications. Platforms that allow most applications to be quickly implemented and executed show high programmability. In contrast, platforms that are restricted to specific application types show low programmability. As an example, think of desktop multi-cores (General purpose Multi-cores (GPMCs)) as highly programmable, and of network processors as "barely" programmable.

Based on the evolution of multi-core processors today, high programmability is the shortest path to quick adoption; the lack of programmability does make certain platforms appealing for reduced communities of very specialized users, but the rest of the potential (paying) users will be very precautious.

In most cases, platform programmability depends on the complexity and flexibility of the hardware, and the way it is exposed to the programmer. Luckily, the programmability of a platform can be significantly influenced by both the type of workloads and the programming tools used. Therefore, the number of programming models that emerge to "address" a given architecture is probably the best metric for platform programmability: the higher the number of platform-centric programming models, the lower the native programmability is.

Huge efforts are spent by both the industrial and the academic communities to increase platform programmability. We identify two orthogonal approaches to address programmability concerns: low-level strategies (platform-centric) and high-level strategies (workload-centric).

Low-level strategies are based on using specialized libraries for optimizing various application kernels. Specialized libraries are used to effectively reduce development time, using an "offload-model". Essentially, one should identify the application kernels that are available as optimized library functions and invoke these platform-specific variants for increased performance. Most vendors do provide highly optimized, platform-specific libraries for typical application classes, such as linear algebra, signal processing, or image processing kernels. This model is very easy to use, but it is mostly successful for small applications (a limited number of large kernels, running in a sequence). For more complex applications, individual parallelization of a few kernels sacrifices too much concurrency, and leads to platform underutilization.

High-level strategies use high-level parallel programming models, addressing application parallelization in a much more platform-agnostic manner. Most high-level programming models increase productivity by using a mix of high-level platform abstractions embedded into already familiar tools or environments. Further, the higher the abstraction (i.e., hiding more of the low-level platform details),

the more portable the implementation is. However, too high-level languages suffer from performance penalties, as the programmers are unable to exploit low-level parallelism layers.

For the rest of this chapter, we focus on high-level strategies and their potential impact on platform programmability.

7.2.2 Performance, productivity, and portability

High-level programming models have the ability to improve platform programmability. In doing so, they impact three interconnected characteristics of an application: performance, productivity, and portability.

The basic measure for application *performance* is *execution time*. From performance, we derive application throughput as the number of operations per time unit (normally, floating point operations per second, or FLOPS/s). Further, using the application throughput, we relate the application performance to the machine performance by *efficiency*, defined as the percentage of the peak performance achieved by the application on the target architecture.

Portability is the ability of an application to be ran on multiple platforms without (major) changes. Code portability requires the code to work "as-is" on a different platform than the original target, while algorithm portability implies that the parallel algorithm is working as-is on a different platform, while some coding has to be changed (for compatibility reasons, in most cases).

As main *productivity* metric we use the application *development time*, defined as the time required for designing, implementing, and optimizing an application to a certain level of performance. We call *productivity* the ratio between achieved performance and the development time. Note that for most applications, the development time for reaching peak hardware performance is asymptotically infinite. Accordingly, reaching for the absolute performance of an application often pushes productivity to zero.

Interconnections

As multi-cores are sold as the solution for all our performance problems, it is only normal to expect that an application runs faster on a multi-core processor than on a mono-core one - after all, the multi-core peak performance is, in theory, (much) higher than the performance of a similar single-core processor.

But multi-cores are highly-parallel machines designed and built to increase application performance by means of parallelism and concurrency. Thus, in practice, the peak performance of a multi-core is only achieved for those applications able to match *all* the hardware parallelism with concurrency. Unfortunately, most (current) algorithms only show fractions of the required concurrency, often leading to disappointing performance results.

Consequently, for many applications, quite a few platforms remain only *theoretical* solutions to the performance problems. In many cases, programmers are able to design a new algorithm that offers enough concurrency to keep the hardware parallel units busy. But designing a new algorithm takes a lot of time, thus decreasing productivity and increasing time-to-market. Being a platform-specific algorithm, it also lowers application portability. Alternatively, using an readily available algorithm and accepting lower levels of concurrency for a given application on a target platform decreases performance (and lowers platform efficiency, as only parts of its resources are properly utilized).

When some degree of application concurrency can be sacrificed for the sake of portability, multiple platforms can make better use of the existing application algorithm (better application-platform fit); a major disadvantage is, however, the limited performance and efficiency on all platforms.

To increase application portability at the code level, one needs to use standard parallel programming models (MPI or OpenMP are good examples). The broad acceptance of these languages also comes at the expense of some performance loss, but this is usually a small penalty. In this case, the portability on a variety of platforms is given by compilers having a common front-end and different back-ends.

To date, increasing application portability at the algorithm level is a matter of programmer expertise and knowledge. Without any exception, this approach leads to an unpredictable productivity and, possibly, an overall drop in performance - going for the common case will remove platform-aware algorithm optimizations.

To maximize productivity, one needs to decrease development time. One way to do so is to count on portability, with the expected performance penalties. However, if the portable algorithm does not meet the performance requirements, the productivity will drop significantly as platform specific optimizations and/or a new algorithmic approaches are necessary.

Development time also depends on the platform complexity: the more parallelism layers a platform has, the higher the development time tends to be. Despite the potential performance gain due to more platform resources, it is usually the case that (1) the loss in development time generates a drop in the overall productivity, and (2) the portability of the solution is reduced.

7.2.3 Required features

We define here three categories of features that programming models must have to impact platform programmability: design-level, implementation-level, and optimizations. Further, we define three criteria for evaluating them.

Design level features

This category includes the programming model features that help the user with the application design. From our experience with multi-core application implementation [Pet07; Var07; Liu07; Ame09; Var09a], we extract four types of essential design features[1].

1. **Algorithmic specification and global view.**
   To help with the application design, a programming model has to provide ways for *algorithm specification*, i.e., for expressing the solution at the algorithmic level. Ideally, from the programmers' point of view, programming models should also preserve a global application view, avoiding fragmentation. Note that fragmented-view models interleave the computation/algorithmic parts of the application with communication/concurrency/synchronization constructs, while global-view models allow the programmer to keep the algorithm "compact". An example of a fragmented view model is MPI, while OpenMP is a global-view model.

2. **Tasks and granularity.**
   A high-level programming model has to provide means to define and/or control *tasks* and *task granularity*. A task is a work entity, characterized by computation and input/output data. Task granularity is the (typical) size of a task in a programming model; ideally, task granularity for a parallel application has to match closely the core granularity of the target platform. Models that impose (fine) task granularity restrictions will typically be more portable than those that

[1] We do not claim this list is complete; rather, we focus on features that target the design phase of parallel applications.

impose no restrictions at all. Furthermore, a model should allow for *task resizing* - by merging and/or splitting tasks.

3. **Task and data parallelism.**
   Ideally, a multi-core programming model has to support both *data parallelism* (SPMD) and *task parallelism* (MPMD). Although some platforms are data-parallel by nature (see GPUs, for example), a generic, high-level model should really allow both, especially given the generalization trend of GPUs. Note that by *support* we mean here that the model should allow users to indicate concurrent tasks to be run as MPMD, as well as tasks that should be replicated to execute in an SPMD manner.

4. **Data layout and locality.**
   A high-level programming model has to allow users to define a *data layout* - typically, a data distribution/partition that specifies what data each task is using.

Implementation level features

We refer here to features that simplify several aspects of the implementation itself. The features we cover here cover mapping, scheduling, data transfers, and communication.

For multi-cores, *mapping an application* refers to the way the application tasks are distributed on the platform cores. Typically, high-level programming models automate static and/or dynamic mapping, such that the user does not have to implement the mapping separately. In the same context, **scheduling** refers to the way the tasks are scheduled for execution on the cores. Ideally, programming models should employ various scheduling policies, depending on the application domain and on the platform they target. Note that these features are extremely useful for the Cell/B.E. (which has no hardware nor software to support that), but maybe less appealing for platforms like GPUs, which deal with these issues directly in hardware.

Due to their complex memory hierarchies, multi-cores have a lot of *data transfers* between various types of memory. If the programming model allows application data layout specification, it can also generate many of these transfers automatically.

Inter-core *communication* is the transfer of messages - synchronization, acknowledgements, etc. - between different cores of the platform. These messages are either explicit (included by the programmer in the code) or implicit (needed for data consistency/coherency, for example). To simplify communication, programming models (1) automate some of these messages and include (or remove) them, and (2) generate the implementation of the messaging itself.

Optimization features

Programming models can simplify certain types of optimizations, typically platform specific. Examples are the use of intrinsics or the double buffering for the Cell/B.E. [Eic05], or memory coalescing for NVIDIA GPUs. These optimizations are typically facilitated by the programming model (which imposes certain restrictions or performs certain source-code transformations), and then handled by the compiler. Note that it is important for programming models to *always* allow users a form of access to the low-level code for refining these optimizations: with the increasing complexity of platforms, compilers are slow to react to new hardware features, and they hardly (if ever) reach the point when users' experise can be fully automated.

Evaluation criteria

We evaluate programming models by three generic (and rather subjective) criteria: (1) usability, (2) re-targetability, and (3) impact. *Usability* qualifies the entire easy-to-use experience, from installation to application execution. *Re-targetability* estimates the ability of a tool to be retargeted for another platform. In most cases, that means that the front-end of a tool is high-level enough to support generic application parallelization, while the back-end can be retargeted for a different platform. Finally, we judge the *impact* of a programming model by its ability to affect application performance, productivity, and portability, which in turn affect programmability.

7.3 Are we there yet? A Survey of Programming Tools

We focus on models developed for a given platform or class of platforms. For most of them, the main goal is to replace platform-specific low-level programming details with higher-level, user-transparent (or at least user-friendly) solutions. Depending on the platform these models target, we discuss four categories: Cell/B.E. models, GPMC models, GPU models, and generic models.

7.3.1 Cell/B.E. Programming Models

Programmability issues

Cell/B.E. programming is based on a simple multi-threading model: the PPE spawns threads that execute asynchronously on SPEs, until interaction and/or synchronization is required. The SPEs can communicate with the PPE using simple mechanisms like signals and mailboxes for small amounts of data, or using DMA transfers via the main memory for larger data. Cell/B.E. is mostly suitable for coarsely parallel workloads, showing both SIMD and MIMD parallelism.

For the Cell/B.E., the features with the highest impact on both performance and programmability are: the heterogeneous cores, the inter-core communication and synchronization, the manual work- and data-distribution, the task scheduling (including thread management), the SIMD intrinsics, and the DMA operations. Overall, the difficulty of programming the Cell/B.E. resides in its multiple layers of parallelism, which have to be properly managed by the application in order to gain good performance. Note that even a first, "naive" Cell implementation of a new application requires a lot of hardware specific tuning to show speed-up over traditional processors.

Programming Models

We briefly present five different programming models for the Cell/B.E., which target different levels of abstraction.

IDL and the Function-Offload Model This model facilitates the porting of a sequential application to Cell/B.E. by offloading its computation intensive functions on the SPEs. The Interface Definition Language (IDL) [IBM06] facilitates the isolation of computation entities and their mapping. When using IDL, the programmer provides: (1) the sequential code that will run on the PPE side, (2) the SPE implementations of the offloaded functions (hand-optimized, if desired), and (3) IDL-specifications to describe the execution behavior of the offloaded functions. An IDL compiler generates the new application, where each offloaded function is replaced by a stub on the PPE side and the new code on the SPE side (and eventually generates code to share the common data structures). Further,

the IDL specifications determine for each offloaded function: if it is executed synchronously/asynchronously with respect to the PPE (i.e., if the PPE will wait or not, respectively), the types and sizes of the parameters, and the return type. For the asynchronous functions, the PPE needs to specify a results gathering policy. For execution, an RPC runtime library is provided to manage the tasks and the size-limited SPE task queues (one per SPE). Application scheduling is based on distributed queues: a call from PPE to an offloaded function over one or several SPEs translates into tasks being added into the proper SPE queues. A full queue on an SPE does not allow addition of new tasks until a slot becomes available. A slot become available after an SPE function has finished execution.

ALF ALF stands for the Accelerated Library Framework [IBM06; But07a], and it provides a set of functions/APIs to support the development of data parallel applications following an SPMD model on a hierarchical host-accelerator system. Note that the SPEs are the accelerators for a single Cell/B.E. processor, but in the same time the Cell itself can become an accelerator to another host processor. ALF's SPMD model requires that the same program runs on all accelerators at one level. Among its features, ALF provides data transfer management, task management, double buffering support, and data partitioning. Using ALF implies the existence of two types of units: the host, which will run the control task, and the accelerator, which will run the compute task. There are three types of code that can be developed independently in ALF: (1) the accelerator optimized code, which provides the optimized compute task on a given accelerator architecture, (2) the accelerated libraries, which include the kernels and provide interfaces for their usage, including the data management and buffering, and (3) applications, which use the accelerated libraries for the high-level aggregation of the complete application. It is expected that a regular user will only work on application development. Major advantages of ALF over IDL are: increased generality, partial solving of SPE-level optimizations, and the use of SPMD as high-level design pattern. In turn, these elements make ALF suitable for designing new applications on the Cell.

Cell SuperScalar Cell SuperScalar [2] (CellSS) is a very pragmatic programming model, suited for quick porting of existing sequential applications to the Cell/B.E. processor [Bel06]. The CellSS programming framework provides a compiler and a runtime system. The compiler separates the annotated application in two: a PPE part, which is the main application thread, and the SPEs part, which is a collection of functions to be offloaded as SPE tasks. When the PPE reaches a task invocation, it generates an asynchronous, non-blocking request for the CellSS runtime to add a new task to the execution list. The runtime system maintains a dynamic data dependency graph with all these active tasks. When a task is ready for execution (i.e., when all its data dependencies are satisfied and there is an SPE available), the required DMA data transfers are transparently started (and optimized), and the task itself is started on the available SPE. Various scheduling optimizations are performed to limit the communication overhead. Additionally, CellSS provides execution tracing, a mechanism included in the runtime system to allow performance debugging by inspecting the collected traces.

Overall, CellSS is a very handy model for generating first-order implementations of Cell applications starting from suitable sequential code. Still, because tasks are generated from functions in sequential code, the core-level optimizations need to be performed by hand, and task resizing may be often required to adjust task imbalance.

[2]The model is based on the principles of Grid Superscalar, a successful grid programming model.

CHARM++ and the Offload API CHARM++ is an existing parallel programming model adapted to run on accelerator-based systems[Kun06; Kun09]. A CHARM++ program consists of a number of *chares* (i.e., the equivalents of tasks) distributed across the processors in the parallel machine. These chares can be dynamically created and destroyed at run-time, and can communicate with each other using messages. To adapt CHARM++ to run on the Cell/B.E., a chare is allowed to offload a number of *work requests*, which are independent computation-intensive kernels, to the SPEs for faster execution. On the PPE side, the Offload API manages each work request, coordinating data movement to and from the chosen SPE, the SPE execution, and the job completion notification. Each SPE runs a *SPE Runtime*, which handles the work requests it receives for execution; this local runtime has the ability to optimize the execution order of the work requests. The model uses various DMA optimizations, including a specific technique that looks ahead into the work requests queues and allows DMA prefetch. SPE code is treated like a non-changeable computation unit, and its optimizations are to be done by the programmer.

Sequoia Sequoia allows the programmer to reason about the application focusing on memory locality [Fat06]. A Sequoia application is expressed as tree-like hierarchy of parametrized tasks; running tasks concurrently leads to parallelism. The tree has two different types of tasks: inner-nodes, which spawn children threads, and leaf-nodes, which run the computation itself. The task hierarchy has to be mapped on the memory hierarchy of the target machine (not only Cell/B.E.) by the programmer. Tasks run in isolation, using only their local memory, while data movement is exclusively done by passing arguments (no shared variables). One task can have multiple implementation versions, which can be used interchangeably; each implementation uses both application and platform parameters, whose values shall be fixed during the mapping phase.

For the Cell/B.E., all inner nodes run on the PPE, while the leaf nodes are executed by the SPEs. The PPE runs the main application thread and handles the SPE thread scheduling. Each SPE uses a single thread for the entire lifespan of the application, and it continuously waits, idle, for the PPE to asynchronously request the execution of a computation task.

The Sequoia model suits any symmetric application task graph, but it is harder to use for applications that are not expressed in a divide-and-conquer form. Once both the machine model and the decomposition of the application into inner- and leaf-tasks are finalized, the application-to-machine mapping offers a very flexible environment for tuning and testing application performance. Also note that based on the clear tasks hierarchy of the application, application performance can be precomputed (although the Sequoia model does not offer it).

SP@CE SP@CE is a programming model for streaming applications[Var06a], built as a component-based extension of the Series-Parallel Contention (SPC) model. The SP@CE framework is composed from an application design front-end, an intermediate representation of the expanded application dependency graph, and a runtime system with various back-ends. A streaming application is designed in SP@CE by providing a graph of computation kernels, connected by data streams (the only data communication mechanism). Kernels are sensitive to events, and the overall graph can be reconfigured at run-time. This graph is expanded into a complete application dependency graph, optimized, and dynamically scheduled (using a job-queue) on the hardware platform by the runtime system. To accommodate the Cell/B.E. back-end, SPE computation is programmed as a kernel in a special component type. Each SPE runs a tiny runtime system that follows the kernel execution step by step. Being based on a state machine, this SPE runtime system overlaps communication and computation in a deterministic way. Further, it schedules/supervises a queue of active tasks, optimizing both the

memory transfers and the overall execution by allowing data prefetching and out of order execution of jobs. The low-level code optimizations are left to the programmer, but optimized kernels can be easily added to the already provided library of functions and/or reused. So far, performance results for running SP@CE on the Cell/B.E. prove very little performance loss over hand-tuned applications [Nij07; Nij09].

Mercury's MultiCore Framework SDK The MultiCore Framework (MCF) is another programming model that focuses on data parallelism. MCF focuses on optimized data distribution as the most complex task a Cell programmer needs to do. An MFC application has two types of computation entities: the manager and the workers. The manager runs on the PPE; the workers run on the SPEs, and can be independent or organized in teams that work on the same task (SPMD). For efficient data distribution, MFC uses *channels* and *data objects*. Essentially, all data is moving through virtual channels that connect (some of) the workers with the manager. On the manager side, a channel read-/write unit is a *frame*, while on the worker side a unit is a *tile*; a frame is composed from multiple tiles. Therefore, any channel must use a distribution object to specify how exactly are the tiles arranged in the frame and how many of these tiles are to be consumed by a worker in a single iteration. The manager and the workers connect to these channels; the execution of a worker is triggered when all the data it requires is available. Based on the distribution objects descriptions, the required DMA transfers can be generated and optimized (using multi-buffering and/or tile prefetching). Note that all channel operations are expressed using special API calls, while the computation itself is written in C. There are no details on the inside of the preprocessor and/or compiler. The MFC creators recommend it to be used for programming n-dimensional matrix operations on the Cell/B.E.. We find this applicability domain quite restrictive, and the support for different application classes minimal - no parallelization support, no core optimizations, no application-level design.

Cell/B.E. Models Summary

In terms of aiding design, the Cell/B.E. programming tools are not very successful - not because of lack of opportunity, but rather because of lack of interest. For example, Sequoia is the only model that preserves global parallelization, but it is limited to symmetrical trees. SP@CE, with its combination of streaming and data-parallelism, also provides a global-view of the application, but only limited to a single application type. ALF provides a high-level model for master-accelerators applications, more systematic than other offload models, but with less support for implementation. Finally, CellSS uses the sequential code to structure the offloading, but does not provide mechanisms to task definition or granularity.

As predicted, most programming models for the Cell/B.E. address mainly the implementation phases, with some additional focus on low-level (SPE) optimizations. Most high-level models adopt the following pragmatic simplifications: (1) address mapping and scheduling (with eventual pre-fetching and/or overlapping optimizations), (2) automate the cumbersome DMA transfers, with more or less optimizations, and (3) automate the required PPE-SPE and SPE-SPE communication. Low-level models require all these operations to be explicitly programmed by the user.

In terms of low-level optimizations, these are mostly left to the compiler; therefore, the IBM's XLC compiler does perform partial vectorization, applies some intrinsics, and facilitates double buffering. Other optimizations have to be implemented by the user; note that most these models preserve some form of separation between the codes for PPE and SPE, therefore allowing local optimizations to be performed in C or assembly.

7.3.2 GPMC Programming Models

Programmability

In terms of programming, the parallelism model is symmetrical multithreading (as we deal with homogeneous architectures). GPMCs (should) target coarse grain MPMD and/or SPMD workloads. The mapping/scheduling of processes on threads is by default done by the operating system; users may somewhat influence the process by circumventing/modifying the running operating system and/or its policies. Programmers typically use traditional, low-level multi-threading primitives (like pthreads), or opt for slightly higher-level generic solutions such as OpenMP and MPI.

For GPMCs, the features with the highest impact on both performance and programmability are: the number and complexity of the cores, their vector units, the cache configuration, and the shared memory consistency.

Programming models

Shared memory multi-threading (pthreads) Pthreads is a common API to program shared memory architectures. The model is a classical one: threads are spawned on the fly and each thread is given functionality at creation. Threads communicate through shared memory variables. Pthreads provides multiple mechanisms for thread synchronization - barriers, semaphores, and critical regions are all included in the API. The model is successfully used for both homogeneous and heterogeneous multi-core processors. However, it is a low-level explicit model, where the programmer has to split the application functionality in fairly simple, yet independent functions, allocate them to threads, synchronize the threads, and recompose the end result. Task-parallelism is supported by default (i.e., different threads execute different functions); data parallelism is seen as an instance of task parallelism. Scheduling is done, usually, by the operating system - threads might opt to specify an affinity (to a certain core), but that is not common practice. Overall, for simple applications and for programmers that want to manually fine-tune/control the granularity and specificity of their parallel computation threads, the pthreads model offers all the needed mechanisms. However, when it comes to code correctness and/or debugging, errors race conditions or unnecessary synchronization are difficult to detect and fix. Therefore, we consider pthreads a too-low level model to be interesting for our high-level models.

Message passing (MPI) MPI, a representative message passing library, is the most popular parallel programming model. Originally used for programming distributed memory machines, MPI is also a low-level, explicit model, that requires the used to identify the application independent tasks and assign them to processes, which in turn run on different compute nodes. MPI focuses on task-parallelism; data parallelism is implemented by deploying multiple similar process instances on multiple data, in an SPMD fashion. An MPI application is a combination of processing operations and message creation/transfer operations. MPI processes communicate using messages. The message send/receive operations, blocking or not blocking, insure not only data transfer, but also synchronization. The popularity of MPI is due to the clear separation between processing and communication. This model is currently used on multiple types of platforms, including shared memory machines. It does not suit fine-grain processes, and it is difficult to use for applications that have complex communication patterns. Further, the scheduling is static - one process per node - and it cannot be changed at runtime. Similar to pthreads, we consider MPI too low-level for productive implementation of parallel applications on multi-cores, and we will not study it further.

OpenMP OpenMP is a higher level parallel programming model, that uses a sequential implementation of an application to extract concurrent entities, which are then scheduled to be ran in parallel. The parallel entities are specified by the programmer using pragmas. OpenMP is built from a compiler and a runtime library. The compiler (a C compiler enhanced with a pragma preprocessor) "expands" the OpenMP pragmas and code into intermediate code, which in turn uses call functions from the OpenMP runtime library. The runtime library contains thread management and scheduling utilities. Using OpenMP as a front-end for multi-core application development has the advantage that a large code base can be (partially) reused, and a large community of users can approach multi-core programming using a familiar model. However, using a sequential application to generate a parallel one is usually limiting the potential parallelism. Therefore, OpenMP can be considered a productive approach (in cases where a sequential version of the target application exists), but satisfactory performance should not be taken for granted.

Ct Ct (C for throughput) is a data-parallel programming language proposed by Intel as a tool for data parallel programming of homogeneous multi-cores. Ct focuses on nested data parallelism, which can express a wide range of applications (as proven in literature by previous work on NESL [Ble96a] and ZPL [Cha00]). By the virtue of its data parallel model, Ct supports deterministic task-parallelism, relying on the compiler/runtime to create and destroy the parallel tasks, and thus guaranteeing that program behavior is preserved when deployed on one or several cores. Further, this approach essentially eliminates an entire class of programmer errors: data races.

Ct's main parallelism construct is TVEC, a generic write-once vector type. TVECs are allocated in a dedicated memory space, and they are only accessible through specific operators. At creation time, vectors have a type, a shape, and a data manipulation directive (copying, streaming, or no-copying). Note that Ct supports both regular and irregular nested TVECs.

Ct operators are functionally pure (i.e., free of side effects): TVECs are passed around by value, and each Ct operator logically returns a new TVEC. operators include: (memory) copying, element-wise operations, collective operations, and permutations.

Any Ct application makes use of Ct data structures and algorithms. Additionally, Ct Application Libraries provide a set of optimized high-level APIs that target problem domains such as image processing, linear algebra, or physics simulations. Both the application and the libraries can be compiled with a C/C++ compiler, generating a binary that can run on any IA architecture (32 or 64 bits, as requested). To enable parallelism, the binary also includes instructions for the Ct Dynamic Engine, which provides three main services: a threading runtime, a memory manager, and a just-in time compiler. The threading runtime system adapts dynamically to the architectural characteristics of the target platform. Using a fine-grained threading model, the system fits the application threads and their synchronization to the hardware capabilities. The memory manager manages the segregated vector space, where the TVEC collections reside. Finally, the JIT compiler generates and stores an intermediate representation of the Ct API calls, which are executed only when needed (for example, when results are needed back in the C/C++ space). Compilation has three layers of optimizations, which take care of data distribution, operator parallelizations, and SSE instruction selections for the specified target architecture. Due to its ability to dynamically adapt to various architectures, Ct is said to be a "future-proof" programming model, with high scalability and productivity [Ghu07].

Intel threading building blocks Intel Threading Building Blocks (TBB) is a c++ library with a strong focus on data parallelism. It centers around the concept of tasks instead of threads where tasks

are performed on data in parallel. The library provides scalability by means of recursive subdivision of data and tasks that perform work-stealing. It depends heavily on templates in C++.

Programmers can obtain scalability in different ways. They can specify a grain size, which is the number of iterations on data that should be performed on a single core in this context. They can also use an automatic partitioner that adaptively partitions the data. Programmers can specify their own "splittable range" which defines how their data should be split up by a partitioner.

The library has three types of "building blocks". It contains built in constructs that can be composed recursively (nested data parallelism). The constructs are `parallel_for` for independent loops, `parallel_reduce`, `parallel_scan` and a `parallel_sort`. The `parallel_while` is executed serially, but the work per iteration can be overlapped, for example operations on the elements of a linked list. The last construct is a `pipeline` where different stages can be run in parallel or sequential.

Another type of building block is a collection of container classes that are thread-safe. The containers use lock-free algorithms or fine-grained locking. Examples are a `concurrent_queue` and a `concurrent_vector`. Finally, the library provides locking classes. However, the usage of these mechanisms is discouraged. For example there is `scoped_lock` that releases the lock automatically at the end of the scope in which it is constructed.

If these constructs are not powerful enough, programmers can revert to defining their own tasks and let them be scheduled by the library's `task_scheduler`. Functions can be spawned to the `task_scheduler` to obtain for example a divide and conquer pattern.

The task based programming model that TBB offers, is a good abstraction, but the notation of tasks in separate classes is rather verbose. This might improve with the advent of lambda expressions in the new C++0x standard. TBB helps programmers, but they still have to be very careful when they write their programs. The provided parallel constructs are rather limited and writing your own algorithms is not trivial for example when dealing with loop dependencies. The concurrent container classes can still give many problems. It is unclear which operations can be run in parallel with other operations and it is not unusual that programmers have to introduce their own locking when they compose operations on the containers.

GPMC Models Summary

Note that for the chosen GPMC models, most of the differences are in the high-level implementation features. At the design level, given the high complexity of the cores and the shared memory, there are very little features to aid the programmer for task design or data distribution; for low-level optimizations, the models rely on the compilers. The main differences in the implementation are the way the SPMD / MPMD are specified, as well as in the mapping and scheduling of the threads.

7.3.3 GPU Programming Models

Programmability issues

Programming GPUs is based on offloading and fine-grained mass-parallelism: the host (CPU) offloads the data-parallel *kernels* of an application as large collections (blocks) of threads on the GPU. A thread block has hundreds of threads that start at the same address, execute in parallel, and can communicate through (local) shared memory. All blocks in a grid are unordered and can execute in parallel; how many of them will actually execute in parallel depends on the hardware capabilities.

A mapping of this model for NVIDIA GPUs is CUDA, an SIMT-based parallel programming model (SIMT stands for single instruction multiple threads - a model in which groups of threads execute the same instruction stream in lock-step). ATI GPUs use the Brook+ language, currently a component of ATI Stream SDK [AMD09] (together with several higher-level performance libraries and tools). The CUDA-like alternative for AMD is in fact OpenCL, to be described in more detail later.

GPUs are typically used for highly data-parallel workloads, where hundreds to thousands of threads can compute concurrently. The most common parallelism models are SIMD/SIMT, with medium and low granularity. Note that due to the time-consuming offloading of the kernels from the host (CPU) to the GPU, too low-granularity kernels are only suitable for these architectures in large numbers. Kernel and thread scheduling are done in hardware.

For GPUs, the features with the highest impact on both performance and programmability are: the very large number of threads, the (uncontrollable) hardware-based mapping and scheduling, the limited amount of fast memory, the different types of memories and their large performance imbalance (orders of magnitude), the different memory spaces (host and device) and the communication between the host and the device. Finally, since registers are dynamically partitioned between running threads, there is a trade-off between using more registers per thread, or more threads with less local data stored in registers.

Programming models

CUDA As general purpose computation on Graphical Processing Units (GPUs) became popular, the need for a structured, compute-oriented model to replace the graphics oriented OpenGL became obvious. To respond to this need, the two market leaders at the time - NVIDIA and ATI - have both proposed more generic models: CUDA (for NVIDIA) and Brook (for ATI). From the two, Compute Unified Device Architecture (CUDA) become quickly the popular choice, leading to a very large number of test applications to be ported and/or benchmarked on GPUs.

CUDA is a proprietary hardware/software architecture, used (only) for general purpose computations on a GPU. Using CUDA to implement an application eliminates the need to first translate it into the computer graphics domain. A compiler, runtime libraries and driver support are provided to develop and run GPGPU applications.

From CUDA's perspective, a compute system is composed from a host (the main processor) and a device (the GPU accelerator). The two have separate address spaces in both hardware and software. The execution model is based on offloading: time-consuming parts of the application are isolated into *kernels* and offloaded into the accelerator memory. The kernels should be heavily parallel. Once finished computing, the GPU results are transferred back to the host. The host and device can run in parallel, executing different paths of the application.

CUDA extends the C/C++ programming languages with a few declaration qualifiers to specify that a variable or routine resides on the host, device or, in case of a routine, on both. It also introduces a notation to express a kernel launch with its execution environment. A kernel is launched by specifying the routine name and parameters, as well as the threads configuration that will execute the routine. The effective scheduling is done by the hardware itself - once the user specifies how many threads are running a certain kernel, it's the GPU that decides which ones, when and where.

Overall, due to its C/C++ origins, CUDA is a friendly solution. Once the user understands the clear separation between the two memory spaces, programming to a first working solution is fairly quick. However, all the optimizations required to increase parallelism, on both the main application side and the kernels side, are the responsibility of the programmer. Further, the complex, multi-

layered GPU architecture provides multiple design and implementation that lead to correct problem solutions, and CUDA does not help the users in the construction and/or search of the best performing one.

Stanford Brook / AMD Brook+ In terms of workloads, ATI GPUs are targeting similar applications as NVIDIA's processors: highly data-parallel applications, with medium and low granularity. Therefore, choosing between the two becomes a matter of performance and ease of programming. For high-level programming, ATI adopted *Brook*, which was originally developed at Stanford [Buc04]. ATI's extended version is called *Brook+* [Adv08]. In contrast to CUDA, Brook+ offers a programming model that is based on streaming. Therefore, a paradigm shift is needed to port CPU applications to Brook+, making this a more difficult task than porting applications to CUDA.

With Brook+, the programmer has to do the vectorization, unlike with NVIDIA GPUs. Brook+ provides a feature called *swizzling*, which is used to select parts of vector registers in arithmetic operations, improving readability. Studies on the performance of the high-level Brook+ model apply to AMD/ATI GPUs are hard to find; loose reports[3] and our own small experiments show performance way lower than expected. The low-level CAL model that AMD also provides does, but it is difficult to use. Recently, AMD adopted OpenCL as a high-level programming model for their GPUs, but also for the CPUs.

ENZO (from Pathscale) The PathScale compiler company[4] is working on a GPU software suite called ENZO. Although the programming model is device-independent, only the NVIDIA GPUs backend is available for now. ENZO comes with its own hardware device drivers, which focus on computing only, and do not support graphics. This way, PathScale expects to achieve better performance. With ENZO, programmers annotate their code with directives to indicate which code regions should be parallelized on the GPU.

This approach is similar to the one OpenMP proposes. The system is heavily based on the device drivers and the compiler to generate parallel code, and uses sequential code as a starting point. Therefore, similar to OpenMP, it does not address the high-level parallelization of an application, but only provides a quick solution for kernel offloading; for the offloaded kernels, the low-level optimizations are still performed by the programmer.

PGI Fortran & C Accelerator Programming Model Using PGI's Accelerator compilers[Por10], programmers can accelerate applications by adding OpenMP-like compiler directives to existing high-level Fortran and C programs. In this respect, PGI's programming model is similar to PathScale's. Compute intensive kernels are offloaded to the GPU, using two levels of explicit parallelism. There is an outer *forall* loop, and an inner synchronous SIMD loop level. The manual states that *The execution model on the device side exposes these two levels of parallelism and the programmer is required to understand the difference between, for example, a fully parallel loop and a loop that is vectorizable but requires synchronization across iterations.* Thus, it is clear that this programming model is directly inspired by the GPU hardware.

Overall, the PGI Accelerator model is quite similar with the one Pathscale proposes, but focuses more on fine-grained data parallelism (such as loops). Again, the compiler plays an essential role

[3] Several small application case-studies and their performance problems are reported on the AMD/ATI Developer Forum http://devforums.amd.com/devforum

[4] See http://www.pathscale.com.

in the parallelization. However, PGI adopts a philosophy in which the users sees why the compiler fails in a certain, required parallelization, and can therefore change the source code to expose more parallelization opportunities. The advantage of this approach is that these "recommended" changes can be made in the original, sequential code, rather than in the parallel, platform-specific code. However, as the compiler abilities are currently fairly limited, the effort required for changing the sequential code to suit the compiler is comparable to the effort required to parallelize the targetted sections by hand.

GPU Models Summary

The GPUs programming models are very close to the architecture. Most optimizations and design decision are completely left to the user, with very little help from the model/tool itself. PGI's Accelerator can be considered a mild exception, as it generates kernels (i.e., tasks) depending on users' pragmas.

7.3.4 Generic Programming Models

In this section we discuss three models that focus on portability across platforms, proposing solutions for all three categories of multi-core processors. To reach this portability level, these models rely on a common front-end, supported by various back-ends.

RapidMind Originally designed to enable graphical units programming, RapidMind was extended to also support general purpose multi-cores and the Cell/B.E. as back-ends, thus becoming the first programming model to uniformly address all three major multi-core architectures. RapidMind uses a so-called "SPMD streaming" model, based on *programs* (its main computation units) that are executed over parallelized streams of data. Therefore, RapidMind parallelism uses three major components: vector values, multidimensional arrays (streams) of vector values, and programs. Much like the "classical" streaming kernels, a program is a computation of vector values; when invoked, the program executes for each element of the stream. In theory, these computations can all execute in parallel; in practice, the degree of parallelism is bounded by the platform resources (or by other user specified restrictions). Additionally, more advanced features like array data selection, SPMD control flow, and reduction operations are all included in the RapidMind model.

RapidMind is entirely embedded with C++. Therefore, an application is a combination of regular code and RapidMind code, translated by the compiler PPE and SPE programs, respectively. Because the model targets data parallelism, there is no application task graph. However, some degree of ad-hoc task-level parallelism can be obtained due to programs that execute asynchronously: if the platform offers enough resources to accommodate multiple running programs, their computation will naturally overlap even if they have been launched sequentially. The execution model of RapidMind is based on heavily-optimized platform-specific back ends. The experiments published so far [McC08] show the performance of the RapidMind and the hand-tuned versions of the same application to be very close, implying that the backed is indeed very efficient.

Pattern-based approaches Pattern languages for parallel programming have been around for quite some time [Bec87; Cop95] (and based, legends say, on ideas borrowed from architecture [Ale77]), but they have never been broadly visible in their history. However, in the past few years, they are

considered a promising and effective approach to multi-core parallel programming, especially due to their systematic generality [Keu10].

Pattern languages are composed from different classes of patterns, applied at different stages of application design [Mat04]. Consider an application specified as a diagram with nodes and edges, where nodes represent computation and edges represent temporal and/or data dependencies.

First, the high-level application architecture is described in terms of structural patterns (its task graph is mapped on a collection of task-graph patterns) and computational patterns (the nodes are mapped on existing computational patterns). This high-level design phase is entirely user-driven and several iterations are expected.

Next, the algorithmic patterns are require, to identify and exploit the application concurrency, thus enabling it to run on a parallel machine. The way the program and data are organized is specified by the implementation strategies patterns, which are, in essence, (multiple) ways of implementing each algorithmic pattern. Finally, the low-level parallelism support, matching both the application and the target architecture, is also included in the so-called parallel execution patterns.

Overall, pattern-based models are very elegant, generic, systematic, and allow for back-tracking and incremental re-design. However, the first and last categories of patterns - the structural ones and the parallel execution ones are not trivial to apply, and structural mistakes can significantly affect both productivity and performance. Furthermore, there are no implementations of such a pattern-based approach in a complete and coherent programming model or language, so we cannot judge its practical usability.

OpenCL OpenCL has emerged as a standard in 2008, aiming to tackle the platform diversity problem by proposing a common hardware model for all multi-core platforms. The user programs this "virtual" platform [Mun10], and the resulting source code is portable on any other OpenCL compliant platform. The compliance translates to available hardware drivers and compiler back-ends, currently[5] available for ATI's and NVIDIA's GPUs, AMD's multi-cores, and the Cell/B.E..

The OpenCL platform model consists of a host connected to one or more OpenCL compute devices. A compute device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. Each processing element can either behave as a SIMD (single instruction multiple data) or as a SPMD (single program multiple data) unit.

The core difference between SIMD and SPMD relies in whether a kernel is executed concurrently on multiple processing elements, each with its own data and a shared program counter or each with its own data but its program counter. In the SIMD case all processing elements execute a strictly identical set of instructions which cannot be always true for the SPMD case due to possible branching in a kernel. Each OpenCL application runs on a host according to the hosting platform models, and submits commands from the host to be executed on the processing elements within a device.

An OpenCL program has two parts: a kernel (the basic unit of executable code), which will be executed on one or more OpenCL devices, and a host program (a collection of compute kernels and internal functions), which is executed on the host. A host program defines the context for the kernels and manages their execution. Since OpenCL is meant to target not only GPUs but also other accelerators, such as multi-core CPUs, flexibility is given in the type of compute kernel that is specified. Compute kernels can be thought of either as data-parallel, which is well-matched to the architecture of GPUs, or task-parallel, which is well-matched to the architecture of CPUs.

[5] July 2010.

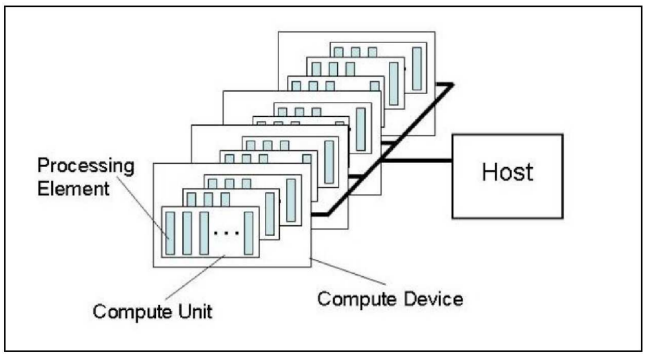


Figure 7.1: *The generic OpenCL platform model (image courtesy of KHRONOS).*

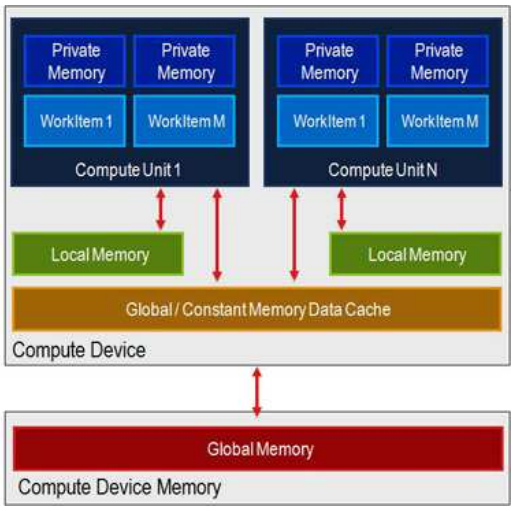


Figure 7.2: *The generic OpenCL memory model (image courtesy of KHRONOS).*

A compute kernel is the basic unit of executable code and can be thought of as similar to a C function. Execution of such kernels can proceed either in-order or out-of-order depending on the parameters passed to the system when queuing up the kernel for execution. Events are provided so that the developer can check on the status of outstanding kernel execution requests and other runtime requests. In terms of organization, the execution domain of a kernel is defined by an N-dimensional computation domain. This lets the system know how large of a problem the user would like the kernel to be applied to. Each element in the execution domain is a work-item and OpenCL provides the ability to group together work-items into work-groups for synchronization and communication purposes. In terms of behavior, the memory model of OpenCL is a shared memory model with relaxed consistency. OpenCL defines a multi-level memory model, featuring four distinct memory spaces: private, local, constant and global. The hierarchy is presented in Figure 7.2. Note that, depending on the actual memory subsystem, different memory spaces are allowed to be collapsed together. Private memory is memory that can only be used by a single compute unit. This is similar to registers in a single compute unit or a single CPU core. Local memory is memory that can be used by the work-items in a work-group. This is similar to the local data share that is available on the current generation of AMD GPUs. Constant memory is memory that can be used to store constant data for read-only access by all of the compute units in the device during the execution of a kernel. The host processor is responsible for allocating and initializing the memory objects that reside in this memory space. This is similar to the constant caches that are available on AMD GPUs. Finally, global memory is memory that can be used by all the compute units on the device. This is similar to the off-chip GPU memory that is available on AMD GPUs.

OpenCL supports two main programming models: the data parallel programming model and the task parallel programming model. Hybrids of the two models are allowed, though the driving one remains the data-parallel. In a data-parallel programming model, sequences of instructions are applied to multiple elements of memory objects. OpenCL maps data to work-items and work-items to work-groups. The data-parallel model is implemented in two possible ways. The first or explicit model lets the programmer define both the number of work-items to execute in parallel and how work-items

are divided among work-groups. The second or implicit model lets the programmer specify the number of work-items but OpenCL to manages the division into work-groups.

Generic Models Summary

While the features that generic models offer their users are quite similar, the approaches these models take are quite different, as they tackle portability at very different abstraction levels.

First, looking at pattern-based approaches, we note that they start from a systematic application design, which leads to a solution based on a composition of nested patterns. Once all these patterns are available for each platform, their composition (of course, also a pattern) leads to a complete application. Pattern-based solutions are therefore the highest-level solutions, which allow users to focus entirely on application analysis and, once that is completed (according to the rules), it generates a running application. To extend to new platforms, pattern-based languages have to implement/re-target the platform-specific implementations to the new platform. Based on the assumption that there is a limited number of patterns that will end up being highly used, this effort might not be very large. As we are not aware of a larger set of implemented patterns, we believe the practical side of the solution is yet to be proven before predicting the future of pattern-based languages.

RapidMind uses a C++-based front-end, and a source-to-source compiler and a runtime system that transforms the SPMD constructs from the code in platform-specific constructs, generating a platform-specific implementation. By restricting the potential parallelism to designated SPMD kernels (marked as special functions in the code) to be executed on data collections, the number of constructs that the parallelizing compiler needs to address is quite limited. Therefore, the C++ structure of the input is preserved. Extending RapidMind to more platforms requires a new back-end to be written - both the compiler and the run-time platform. In the meantime, RapidMind's development seems to have stopped, and it is therefore difficult to predict its future.

OpenCL uses yet another approach: it introduces a *common machine model* for which the programmer design its application. This is another way to separate the design and implementation concerns: OpenCL's back-end targets one machine type only, while it is the responsibility of the hardware vendors to provide the OpenCL drivers. On the other hand, the front-end is more constraint, requiring programmers to write very disciplined and fairly low-level code. Extending OpenCL can be approached by both its ends: extend the features the front-end supports, and extending the machine model.

7.3.5 A Comparison of Multi-Core Programming Models

The hardware-induced approach on programming multi-cores has generated a large collection of *pragmatic* models, focused on hiding complex details from the programmer. These models are built as quick solutions for programming new hardware platforms, and not to solve the fundamental problems of parallel programming. Therefore, they often show a combination of new, hardware-specific features and classical, parallel programming features that lead to unclear semantics and, therefore, cancel any opportunity for analytical reasoning about correctness or performance.

Table 7.2 presents a qualitative comparison between the multi-core models we have described, focusing on the requirements described in Section 7.2.

Only a couple of Cell/B.E. programming models (Sequoia and SPCE) aim to also tackle parallel design; most Cell/B.E. programming models provide the following simplifications: (1) they (dynamically) solve mapping and scheduling (with eventual prefetching and/or overlapping optimizations),

	algo spec view	algo spec generality	type of parallelism	task control	data layout	mapping	scheduling
threads	F	Low-level	any	no	no	no	no
MPI	F	N	task	creation, implicit	explicit	no	no
OpenMP	G	Y	task/data	creation, implicit sizing	no	no	no
Ct	G	Y, nested data par constructs	nested data	no	nested vectors	no	no
TBB	G, costructs	Y	task/data	creation, explicit sizing,resizing	no	no	possible
Cuda	G, Offload	Y, kernel-based	data	no	explicit placement	no	no
Brook+	G, Offload	Y, kernel-based	data	no	explicit placement	no	no
ALF	F	Y, hierarchical architectures	task/data	creation, explicit sizing	explicit placement	dynamic	dynamic
CellSS	G, pragmas	Y	task	extraction, implicit, merging	Data distribution, explicit	dynamic	dynamic, runtime
Charm++	G, offload	N, offload	task	extraction, explicit, merging	explicit placement	dynamic	dynamic, distributed q's
Sequoia	F	Y, div-and-conquer	Div-and-conquer	creation, explicit, merging	explicit placement	explicit	generated, static
SPCE	G	Y, streaming	task/data, streaming	creation, implicit	data distribution, components, implicit	dynamic	dynamic, distributed q's
Mercury	F	N	data	creation, explicit	data distribution, channels, implicit	dynamic	generated, static
PGI	G	Y, pragmas	data	no	no	no	dynamic
PathScale	G	Y, pragmas	data	no	no	no	dynamic
RapidMind	G, constructs	Y, pragmas/kernels	data/task	no	no	no	dynamic
Pattern	G	Y	any	?	?	?	?
OpenCL	G	platform	task/data	creation, implicit sizing, no resizing	explicit placement	dynamic	controlled queue

	Data transfer	Comm/Sync	Optimizations	Productivity	Portability	Performance	Starts from	Abstraction
threads	no	all/explicit	allows	−−	−−	+	NA	threads
MPI	explicit	explicit/explicit	allows	−−	+	+	NA	processes
OpenMP	no	implicit/implicit	obstructs	++	++	+	seq	loops
Ct	no	no	obstructs	+	∼	+	NA	TVEC
TBB	no	possible/possible	supports	∼	∼	+	seq/NA	tasks
Cuda	All explicit, Host-device, intra-device	host-device/explicit	supports	+	−	++	seq/NA	kernels/threads
Brook+	All explicit, Host-device, intra-device	host-device/explicit	obstructs	+	∼	∼	seq/NA	kernels/threads
ALF	explicit, host-dev	implicit/implicit	supports	+	∼	+	NA	tasks (SPMD)
CellSS	all implicit	implicit/implicit	obstructs	++	+	∼	seq	functions
Charm++	implicit, offload	explicit/implicit	supports	∼	∼	+	par	chares
Sequoia	implicit, explicit	implicit/implicit	obstructs	∼	+	+	D-and-c	trees/leafs
SPCE	all implicit	implicit/implicit	allows	+	∼	+	NA	streams
Mercury	explicit, implicit	implicit/implicit	allows	−	−	+	NA	channels
PGI	all implicit	implicit/implicit	obstructs	∼	+	∼	seq	loops
PathScale	all implicit	implicit/implicit	obstructs					
RapidMind	all implicit	implicit/implicit	obstructs	∼	+	+	seq	programs(kernels)
Pattern			obstructs	−−	++	?	NA	patterns
OpenCL	explicit, implicit	explicit, implicit	supports	+	++	∼	seq/NA	kernels

Table 7.2: *A comparison of multi-core programming models. F/G stand for Fragmented/Global, Y/N stand for Yes/No, NA stands for Not Applicable.*

(2) they automate DMA transfers, transparently, and with data-locality optimizations, and (3) they simplify the required PPE-SPE and SPE-SPE communication. Low-level optimizations are mostly left to the compiler. Generic multi-core programming models address productivity in its simplest form, focusing on reusing legacy code and exposing the hardware as little as possible. Parallel design is of little interest, and low-level optimizations are waved-off to the compiler. In the case of GPUs, the programming models are very close to the architecture. Most optimizations and design decisions are completely left to the user, with very little help from the model itself.

7.4 Summary and discussion

In the multi-core era, application parallelism has become mandatory. The software community is (suddenly) faced with a large scale problem: virtually every application will have to run on a parallel machine, rather sooner than later. And with multi-core architectures emerging at high-pace, addressing applications and machines as one pair simply does not scale.

One effective way to address this problem is to focus on *application-centric programming*: first design a parallel solution for the problem, and then implement it on one or multiple platforms.

However, this is easier said than done. We have analyzed 19 programming models for multi-cores. From our analysis, the major lacking feature is support for proper parallel application design. Even for portable models, where multiple platforms share the same front-end, the focus is on simplifying the user experience (usually, by starting from a sequential C/C++ solution and isolating parallel regions) and not on supporting high-level parallel design. We argue that the programmability gap cannot be solved by using these types of models, because they do no address platform programmability. Instead, they focus on increasing productivity per-application, by simplifying the implementation side. While this effort might pay off in the time-to-market of some application instances, it is not generic enough to address the multi-core programmability problem.

One major exception to this trend[6] are pattern-based languages, which *require* a parallel application design from the user. However, lacking a proper implementation or back-end, it is hard to evaluate this approach as a complete solution.

A second exception is the OpenCL language. We believe that the features and qualitative analyses, combined with its availability, prove OpenCL to be a programming model that is worth learning. Given the large consortium that supports the OpenCL standard [Khr10], and the large number of hardware vendors that are working on OpenCL drivers, we believe this is a promising alternative as an unified multi-core programming for the near future. Overall, in its current form, OpenCL provides a very good solution for prototyping portable applications: the solutions written in OpenCL can be run on all three families of hardware, with no theoretical limitations on performance. Furthermore, these applications are fairly future-proof, as new generations of platforms will most likely support OpenCL very early in their life-cycle.

For the near-future, we have four recommendations. For the OpenCL committee, to analyze how a new generation of the virtual platform model can include more flexibility in the communication and memory modules. For hardware vendors, to support OpenCL by developing drivers for their platforms, rather than offering yet another hardware-specific programming model. For (third-party) programming model designers, to support OpenCL as a compilation target for their models. For

[6] We are only discussing multi-core specific programming models; of course, an entire collection of "classical" parallel programming models combine design and implementation, but only very few of them are also usable for multi-cores, and typically only for the GPMCs

programmers, to make use of OpenCL for prototyping and/or development - its quick adoption will increase the pace of its development.

Finally, we note that despite its qualities in terms of portability and performance, the limitations of OpenCL - especially the very low-level parallel design support and a clear focus on highly data parallel platforms - leave room for improvement. We believe that a step-by-step strategy, supported by a comprehensive framework, is the only way to systematically approach multi-core application development, from design to execution. We discuss our solution in the following chapter (Chapter 8).

Chapter 8

A Generic Framework for Multi-Core Applications

In this chapter we present MAP, a framework for designing and implementing parallel applications for multi-cores.

With our case-studies, we have shown that multi-core processors *can* lead to impressive performance gains for various types of workloads. However, the time and effort spent in reaching these results is not encouraging for multi-core productivity. We believe the little design experience, as well as the lack of a strict implementation discipline drastically affected our productivity.

Further, we have shown that relying on programming models to fill in these gaps has only been partially successful: the models in use today are hardly able to cope with the challenges posed by applications in both the design and implementation phases. The main reason for this failure is the way these models have been created. Platform-specific models (the majority of these models) have been created to alleviate *platform-specific* programmability issues (e.g., ALF); therefore they lack support for parallel application design. Generic programming models have been created to re-use available sequential codes (e.g., RapidMind, PGI Accelerator) - again, a solution that leaves no space for parallel design. Finally, abstract programming models are either very limited in the class of applications they can tackle (e.g., Sequoia), or too generic to be practically built (e.g., pattern-based languages). Consequently, all these models have only little impact on platform programmability. And they are scarcely used.

We believe that programming tools that aim to have a higher impact on productivity and portability, should focus on application-centric development, guiding the user to go through all phases of application development - from design to implementation and optimization - with maximum efficiency.

In this context, this chapter presents a strategy for application-centric multi-core programming (in Section 8.1), arguing that such a top-down approach encourages application-level programmability. Further, we present MAP (in Section 8.2), our prototype of a generic framework for designing, implementing, and tuning multi-core applications. MAP has been designed to fully support the proposed strategy. To show how MAP can be applied for a real application, we also discuss a design case-study (see Section 8.3), using the MARVEL application (discussed in Chapter 4). Based on both requirements and empirical findings, we analyze the tool support for MAP (see Section 8.4), and discuss how such a framework can be implemented.

8.1 A Strategy for Multi-core Application Development

In this section, we synthesize a strategy for effective implementation of parallel applications for multi-cores. We strongly believe that following such a systematic, step-by-step strategy is highly beneficial for most applications, resulting in increased productivity and portability. While we describe an application-centric approach, we also compare it against the more commonly used hardware-centric solutions, showing how the latter is, in fact, a specific case of the former.

8.1.1 An application-centric approach

Our application-centric strategy has six stages: (1) analysis, (2) design, (3) prototyping, (4) implementation, (5) tuning, and (6) roll-back. In the remainder of this section, we discuss each stage in detail, explaining what it does, which are its inputs and outputs, and what are the challenges it poses to the user.

Application analysis

This first stage analyzes the application, extracts the problem, and generates a set of solutions; all these steps take place in the application domain, typically not directly related to computer science. Analysis starts from an application specification, and obtains (1) a clear view of the application requirements, and (2) a set of algorithmic solutions to solve the problem.

Application analysis investigates three application characteristics: *functionality*, *input and output data sets*, and *performance requirements*. Note here an important issue: even if source code is available, we investigate the functionality, the data characteristics, and the performance bounds of the application in *the application domain*, unbiased by constraints from the implementation or the hardware platform.

Functionality analysis investigates what is the application doing and how. Consequently, the *functionality specification* should reflect the application structure and computation requirements. Also, the first functional and data dependencies are already identified here, hinting towards potential parallelization options.

Data requirements analysis focuses on finding out what are the main data structures, types, and sizes for both the input and the output data, what is the distribution of the data (is it sparse or dense, one- or multi-dimensional), and what are the requirements for temporary data (in terms of memory-footprint). We are not (yet) interested in a detailed analysis, with highly accurate values for these parameters, but rather in a coarse set of estimates and ratios to guide memory allocation and re-use potential.

Performance requirements are usually provided by the nature of the application or by the user as bounds on the execution time or platform utilization. For instance, "real-time movie-frame analysis" poses a time limit on the processing of each movie frame. Performance bounds are essential for any problem as they set a pragmatic goal for the application and provide a clear stop condition for the iterative application implementation and optimization processes.

Once the application analysis is done, several potential solutions emerge. In most cases, these "algorithms" are defined in the problem domain, but they might as well be designed by the programmers themselves. Ideally, the outcome of this stage is a set of solutions - the most promising alternatives in terms of meeting the target performance goals. However, in practice, due to development time constraints, it is very common to only use one solution, and tune it until it fits inside the performance boundaries.

Hardware modeling

We use hardware models to enable performance prediction and estimates along the design and prototype process. Although platform-agnostic, our models, prototypes, and designs do require hardware support for performance comparison and ranking.

Note that the multi-core families are not hardware compatible; therefore, per-family hardware models are needed for performance estimates. Depending on the desired accuracy level, the collection

of models can increase even further, taking into account more details within families of platforms or generations. Alternatively, to make portable predictions, one can use a generic model, built either using a standard "virtual" platform (e.g., the OpenCL platform model) or extracting only the common hardware features of all families (leading to a very high-level, PRAM-like model). We believe that family-level modeling provides sufficient accuracy at the design level.

Application Design

In this stage, solutions from the problem domain are translated into a set of parallel algorithms in the computer-science domain. Further, a compliance filter is added to eliminate those parallel solutions that are obviously not suitable for the concurrency requirements or for the application performance constraints. Filtering is based on performance evaluation, which in turn is based on a high-level prediction tool. The outcome of this stage is a set of parallel algorithms, preferably platform-agnostic, that could potentially be implemented as solutions.

Application Prototyping

The parallel algorithms are implemented into functional prototypes, for functionality and coarse performance verification. For prototyping, there are two orthogonal approaches: (1) the breadth-first approach, in which we briefly prototype multiple solutions, ranking them by their performance, and choosing the best for further implementation, and (2) the depth-first approach, in which a prototype is built only for one solution, which is further implemented and optimized before testing other solutions. In practice, the depth-first approach embeds prototyping into the implementation of the solution with the best predicted performance.

Note that depending on the target level of platform-agnostic behavior, prototypes need not be implementations, but can be more detailed models, for which detailed performance prediction is possible. As this approach is typically quicker than implementation, it is particularly useful for a breadth-first analysis of the parallel solutions.

A performance checkpoint is inserted at the end of this stage to investigate how far off the best prototype (i.e., the solution that will be implemented first) is from the required performance. If the gaps are too large, one should consider trying multiple prototypes before moving on to application implementation.

Application Implementation

The prototype is transformed into a complete implementation. For a platform-agnostic implementation, one should use a generic programming tool (in most cases, that means expanding the prototype). If a target platform has already been chosen, using a platform-specific programming model typically leads to better performance and productivity, at the expense of portability.

If a generic programming model is used, we recommend exploring and implementing generic optimizations - such as computation optimization(s) and data-dependent optimizations in this stage. As these optimizations preserve the application portability, implementing them here enables fair per-platform performance studies in later stages.

Once an implementation is available, its performance is evaluated by benchmarking and profiling. The metrics to use are: execution time, computation throughput, and memory throughput. The measurements have to be done with representative data, i.e., data that matches the properties identified in the application specification phase.

These performance measurements show how large the gap between the current implementation and the required performance is. In most cases, a performance boost is still needed, and it can only be achieved by applying lower-level optimizations, which are platform-specific. This is the point where the trade-off between performance and portability, specific to generic models, becomes visible.

If the current solution satisfies the performance requirements on (any of) the selected target platform(s), the development can and (in most cases) should stop. However, it is rarely the case, and going one step further to look into lower-level optimizations is usually mandatory. Once on this path, however, the portability of the implementation diminishes significantly - not only in terms of code and language, but also in terms of parallelization, structure, and granularity.

Iterative Optimizations and Tuning

This final stage deals exclusively with low-level, platform-specific optimizations (i.e., the target hardware platform is known). The iterations continue as long as the following conditions are satisfied: there are still optimizations available, the time budget allows optimizations, and the performance goals are not yet reached. When the process stops, we either have a performance-compliant implementation, or we need to roll-back.

Roll-back

If the required performance is not met after the optimizations have been exhausted, the process can be restarted from any of the checkpoints along the path. We can re-evaluate our choices, and:

- Replace the platform: one can also investigate another target platform (see the platform-specific optimizations step), having the advantage that there is no need to start the parallelization process all over again.
- Change the parallel algorithm: one can get back at the pool of potential parallel algorithms to choose another path.
- Relax the performance requirements: it might be the case that the current performance requirements are too strict. Relaxing these requirements is equivalent to a "best-effort" approach in the previous step, where the expiring of the development time or the optimization completion can deliver the best possible performance for the application running on each potential target; the best performance of all platforms can be taken, if needed.
- Constrain the data: if the performance penalties are due to over-relaxed assumptions on the data set (i.e., data is considered random and dense, although it might not be) or due to outlier data cases, a more constraint data set might allow better data-dependent optimizations, which in turn might increase the overall application performance.

8.1.2 The hardware-centric approach

A hardware-centric approach to multi-core application implementation is, in fact, a specific case of an application-centric solution. The differences are as follows:

- Application parallelization can be platform-specific. This has the advantage that performance evaluation and prototyping are simplified.

- Platform-specific performance evaluation is simpler and more accurate; therefore, using a DFS approach in parallelizing the most promising parallel solution from the initial set will probably lead to a good solution quicker than a BFS approach. In other words, in most hardware-centric approaches, prototyping might be skipped for a shorter development process.
- Implementation can and should be made using a platform-specific programming model. However, if the platform is supported by generic programming models, we still recommend investigating if such a portable solution suffices in performance.
- The roll-back offers fewer intermediate points for backtracking.

Overall, we note that the main differences between hardware-centric and application-centric approaches appear in the design phases of the application, and propagate to the implementation. As hybrid approaches are possible, we strongly recommend starting with a portable, platform-agnostic design, and preserving the platform independence as long as the time, resources, and performance requirements allow it.

8.2 MAP: A Framework for Platform-Centric Application development

To support the use of the systematic strategy presented in Section 8.1, we propose MAP[1], a generic framework for application-centric parallel programming for multi-cores. MAP is presented in Figure 8.1.

In the remainder of this section, we detail the stages of the MAP framework, focusing on the actions required for each one of them.

8.2.1 User input

Application Specification We define *application specification* (shortly, an application) as a triplet consisting of: a problem to solve (i.e., functionality), the I/O data sets, and a set of performance requirements.

Functionality specification Obtaining a correct and complete functionality specification requires, in most cases, an iterative dialogue with the experts from the problem domain. Expressing these findings in a form suitable for future implementation work is non-trivial and there is no standard form for representing application functionality and requirements.

Inspired by the theoretical work of Blelloch et. al [Ble96b], we use "loose" application dependency graphs[2] as a suitable abstraction for the functional requirements of the application.

I/O Data specification Data characteristics have become quite significant for application design due to the complex and fragmented memory models of the multi-core architectures. Application I/O data specification has to include a list of the most important input and output data structures the application uses, together with information on their type, size(s), dimensionality, and any available

[1] **MAP** stands for "Multi-core Application Performance", the three major elements of our framework

[2] Application Dependency Graphs are oriented graphs where nodes represent computation units, and edges represent functional dependencies.

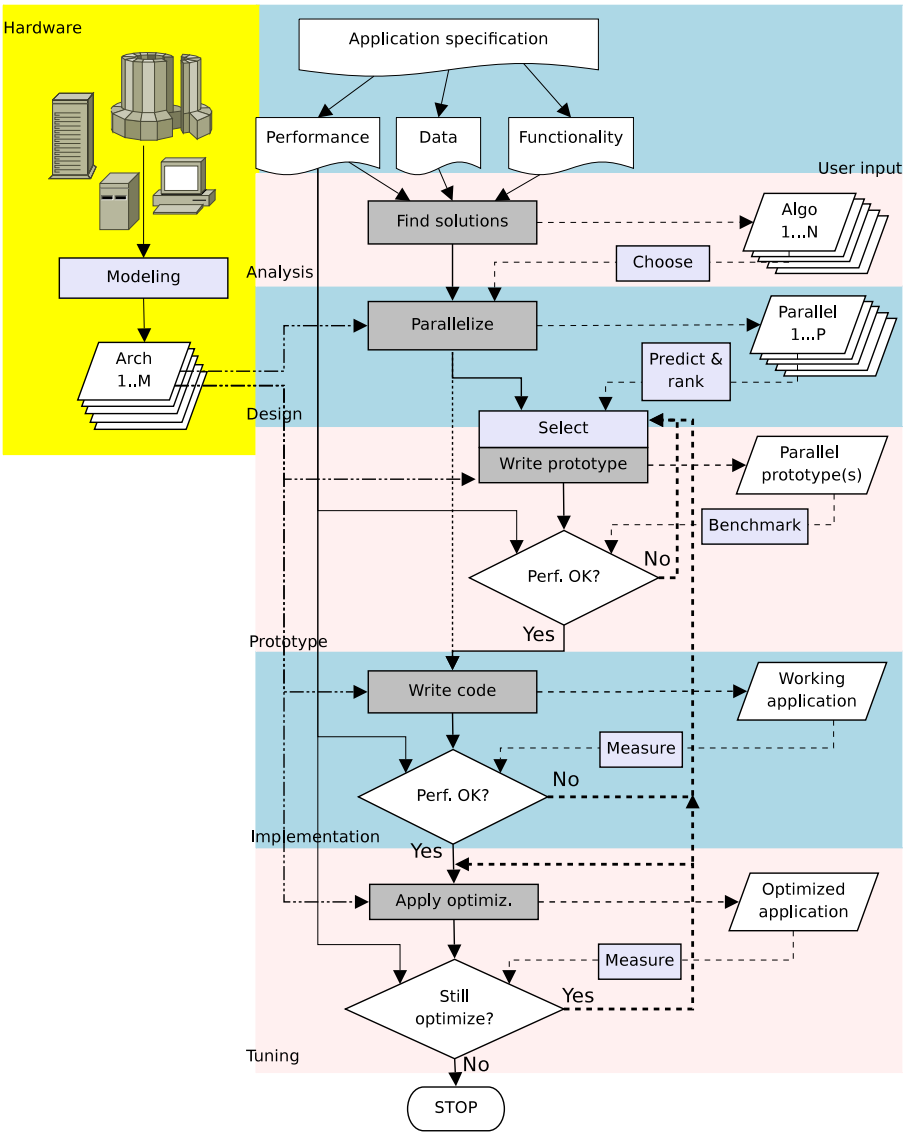


Figure 8.1: *The MAP framework flowchart. Application specification (performance requirements, data requirements, and functionality) are user-specified inputs. Dark-gray components are the framework building blocks. Light-gray components are support-tools.*

distribution information. This information is used for memory-footprint estimates and parallelization granularity assessments. Furthermore, any ordering information and any (mathematical) correlations between these data structures should be noted for potential use in data-dependent optimizations.

Performance requirements Setting application performance requirements is essential in both setting a pragmatic goal, and a stop condition in the iterative process of code implementation, tuning, and optimization. Performance requirements are typically specified as upper bounds on the execution time or power consumption, and imposed by the nature of the application. However, metrics like

efficiency or throughput (GFLOPS) are becoming popular as well.

Note that performance requirements can also be "detailed" as the application evolves. Thus, for the parallel models of the application, the performance requirements are less accurate than those imposed on the implementation itself. However, this adaptation has to be controlled entirely by the user, and we foresee no generic way to automate this process.

8.2.2 Analysis

Solutions and Algorithms We call *solution* an algorithm for solving a given problem. Belonging to the problem domain, solutions are platform-agnostic. Therefore, they are specified either using high-level pseudo-code or dependency graphs. We recommend using the same abstraction as the problem specification (dependency graphs, in this case), because these solutions are often just extended problem specifications.

Find solutions Based on the application specification, several solutions should be already available or specifically designed to solve the problem. As we are still in the problem domain, an automation or generalization of the process is hardly possible, excepting classical types of algorithms (such as sorting or searching). Thus, in most cases, solution search is a process to be performed entirely by the user.

Choose solution(s) While discovering as many solutions as possible allows for a larger collection to extract from, we limit our selection to a set of promising algorithms. This selection is, for now, entirely based on empirical observations and expertise. Current developments in application characterization and application-architecture fit aim to enable a systematic approach to this selection: based on several application characteristics, visible in the solution specification, one can choose a platform and a parallelization strategy for the application; consequently, the most promising solutions for parallelization will be easier to identify.

8.2.3 Design

Parallel algorithms A *parallel algorithm* is generically defined as an algorithm that uses parallelism to solve a certain problem. In the context of the MAP framework, a parallel algorithm is a parallel version of a solution. Note that a single solution can "generate" multiple parallel algorithms.

At this level, parallel algorithms are specified as platform-agnostic models. Besides allowing a compact form that focuses on parallelism, keeping specifications as high-level models enables the comparisons between parallel solutions and, eventually, quick performance estimates. One example of such modeling language is PAMELA [van03a], suitable for nested-parallel applications and DAGs. Furthermore, PAMELA allows for symbolic performance prediction, useful for the performance-based ranking of the parallel solutions. An example of using the PAMELA language can be seen in Chapter B.

Parallelization We define *parallelization* as the operation of translating a problem-domain solution into a parallel algorithm (in the computer science domain). Overall, we claim that parallelization is the most difficult part of the application development process and it still is the responsibility of the user. However, depending on the application and on the choice of platform, it can be assisted by semi-automated tools.

For example, one can use a programming model able to extract (some degree of) parallelism from an available sequential implementation. Alternatively, applications can be parallelized using a pattern-based approach, where several patterns of computation are identified and parallelized accordingly. Another option is to use application characterization, which is based on evaluating a set of metrics to characterize the application behavior, and further recommends a parallelization strategy based on similarity. In all these scenarios, the resulting parallel solutions are platform-agnostic.

In the case of platform-centric design, we assume that a platform or set of platforms has already been selected. Therefore, additional information and specific tools can be used for parallelization. In this case, we obtain a platform-specific parallel solution, with limited portability.

Predict and rank We aim to rank the collection of parallel algorithms in terms of potential performance. To do so, we have to rely on some form of performance prediction. By *performance prediction* we mean a way to estimate the application performance (execution time, in our case) when the code or the hardware are not fully available to actually measure the execution time.

For such a predictor to be effective, all solutions have to be expressed using a common modeling language, that allows performance estimates and, therefore, allows the weaker solutions to be detected and eliminated. Similarly, all platforms need to be modeled using the same strategy, granularity, and accuracy.

Such a performance predictor can be built using model-based symbolic analysis techniques, similar to those used by PAMELA[van03b]. For example, PAM-SoC [Var06c] is a predictor for system-on-chip platforms, which uses the PAMELA models of both the application and the platform to determine a rough performance behavior of the pair. Although the absolute performance numbers are only coarsely accurate, they suffice for ranking purposes. As the prediction is very fast, it allows multiple (application,platform) pairs to be quickly ranked by their relative performance. More detailed descriptions of PAMELA and PAM-SoC are given in Appendix B.

8.2.4 Prototype

Parallel prototype(s) A parallel prototype is an implementation of a parallel algorithm, without any optimizations. Prototypes can be used further as reference implementations. Implementing multiple prototypes is useful for checking solution correctness and efficiency, and eventually refining the ranking provided by the prediction. For hardware-centric application development, prototyping is typically skipped or "embedded" in the implementation itself.

Select Typically, the parallel algorithm with the best predicted execution time is chosen for further prototyping and implementation. In case multiple algorithms share the top spot, there are three options: (1) choose arbitrarily or based on a brief similarity analysis, (2) refine application models and update the performance prediction, and (3) prototype multiple solutions.

Implement prototype The prototype implementation should follow closely the parallel algorithm design, while the platform and the programming model are chosen such that the prototyping is quick: the hardware should be available for direct performance benchmarking and the programming model should require little user intervention to transform an algorithm into an implementation.

If prototypes are to be kept platform-agnostic, a generic programming model can be used as a prototyping tool. If sequential code is available, OpenMP can be useful for quickly getting a working data-parallel solution, and it is ported on most platform families. If there is no code available, OpenCL

is probably a better option, given it specifically targets multi-cores. For large and complex applications, it is probably best to use more refined models as prototypes (instead of real code), and choose the most promising one based on higher accuracy predictions (instead of measurements). The choice of a programming model and platform should follow the modeling.

Benchmark Once the prototype is available, it is benchmarked using either real data sets, or data that fits as much as possible in the original I/O specification.

Performance checkpoint and roll-back The achieved performance can be checked against the imposed requirements. If performance falls inside the limits, one can further proceed to implementation. If the current benchmark is way off the set boundaries, the platform choice, parallelization, and solution selection should be reconsidered, in this order. Note, however, that this early performance check-point enables early drop-out, without wasting too much time on application implementation.

8.2.5 Implementation

Working application The working application is a complete implementation of a solution, able to run on the final target platform(s). It is usually a direct "translation" of the prototype, eventually in a new programming model and for a new platform.

Write code Application implementation should make use, as much as possible, of high level programming models. Using generic models, like OpenCL, preserves the platform independence even further. A choice for platform specific models limits the hardware flexibility, but might increase overall productivity - for example, the use of CellSS for the Cell/B.E. is more productive than the use of OpenCL (given that Cell/B.E. has already been chosen as a target).

Measure Once the working application is available, it is executed and its performance is measured. Again, the performance measurements should be performed using either real data sets, or at least data that fits as much as possible in the original specification.

In this stage, we also determine if we implement a compute-bound or a memory-bound application. One way to systematically investigate this is to build a Roofline model [Wil09c] of the application. This is a fairly easy model to build, and gives a clear graphical view on the application bounds on a specific platform; using this graphical model, we immediately see on which side of the performance spectrum we are: compute-bound or memory-bound. Note that we need one Roofline model for each potential target platform, as the same application can be compute-bound on one platform, and memory-bound on another.

Performance checkpoint and roll-back A second performance checkpoint is placed here, with the same "safeguard" role: if the performance of the implementation is far off the requirements, one has to check first the implementation itself, then the eventual overhead induced by the programming model. If both are within reasonable limits, a roll-back to the choice of platform, programming model, parallelization, and even the solution design might be needed.

8.2.6 Tuning

Optimized application The (fully) optimized application is the end target of the development process: it is a correct application, with all the required platform-level and application-level optimizations activated.

Implement optimization Low-level optimizations are specific techniques to further improve the performance of an implementation. There are multiple types of optimizations to be applied to any parallel implementation; however, their specific implementation can be both application and platform specific. The impact various optimizations have on the overall application performance depends on the application class.

Compute-bound applications are typically optimized by reducing the number of operations. In most cases, this means re-using some results instead of recalculating, or reordering instructions in a more compiler-friendly way. Specifically, application-wise modifications include pre-calculating and/or re-using results by storing and loading them from memory, which is in fact a way to trade calculations for the memory traffic. Alternatively, most platforms offer different types of low-level optimizations that speed-up the available operations by fusing them (e.g. multiply-add) and/or group them with SIMD-like techniques.

Memory-bound applications can be optimized by reducing or speeding-up the memory traffic. Application-wise, we can trade memory communications for computation, if there are computation parts that are pre-calculated: instead of loading data from the memory, one can simply re-calculate it. Platform-wise, memory-optimizations are very diverse. For example, GPUs have a complex memory hierarchy, where the lower the memory is in the hierarchy, the faster and the smaller it is. Using this hierarchy properly may require quite some trickery, but it can reduce the memory access times by up to two orders of magnitude. Alternatively, one can try to increase data locality by properly reordering the data. Finally, there are platform-dependent optimization, like data coalescing and the use of very specific cached memories, usually available in GPUs.

Note that most optimizations are still manually implemented by the programmer. While some optimizations are supported by compilers, the obtained performance is, for now, significantly lower than the performance of hand-optimized code. On the other hand, it is worth mentioning that these optimizations are always fairly difficult and time-consuming; hence our proposal to postpone them till the very last stages of application development, to be applied only if really needed.

Measure The application performance is measured using real data sets. Furthermore, after each optimization, an update of the compute-/memory-bound is required, for choosing the next optimization step properly.

The stop checkpoint and roll-back options Application optimization is an iterative process. In each iteration, one single optimization is applied, followed by an update of the compute-/memory-bound and performance measurements. The iterations stop when any of these three conditions is met: (1) there are no more optimizations to be applied, (2) the development time budget is finished, or (3) the application performance is already meeting the requirements.

8.3 A MAP Example: MARVEL

To demonstrate how the MAP framework is applied to design and implement a parallel application, we re-use the MARVEL case-study (see Chapter 4 for a detailed description of the application). We show that, following the framework step-by-step, we can reach a functional and performance-compliant parallel implementation.

8.3.1 User input and the analysis stage

The raw application specification is:

MARVEL is a content analysis application. Its inputs are a collection of pictures and a collection of models; the output is a classification score per image. Each image is analyzed for extracting different features. Image features are used for concept detection. The application should analyze pictures at 30 fps.

After further inquiries, we extract the application specification and it components presented below.

Data I/O specification `IMG` is a collection of N color images, all with the same resolution, $H \times V$. `MODELS` is a collection of P models (one per feature); each model is a collection of feature vectors, all the same length, representing values of known concepts.

Functionality specification From the raw specification, we extract the functionality presented in Listing 8.1. An alternative representation of MARVEL's functionality, in a Directed Acyclic Graph (DAG) format, can be seen in Figure 4.1 (in Chapter 4).

Listing 8.1: *The functionality specification for MARVEL (two designs).*

```
1
2 MARVEL = {
3 [..]
4 forall images i=1..sizeof(IMG) {
5     read_and_preprocess(IMG[i])
6     forall features f=1..sizeof(FEATURES) {
7         K[i,f] = extract_feature(IMG[i],FEATURES[f])
8         V[f] = extract_vectors_from_model(MODELS[f])
9         concept[i,f]=detect_concept(K[i,f],V[f])
10    }
11    final_score[i]=compute_score(concept[i,_all_])
12 }
13
14 detect_concept(K,V) = {
15         forall v=1..sizeof(V)
16             score += weight[v] * detect_concept(K,v)
17 }
18 }
```

The functionality specification needs to be further refined by adding the specification for the `extract_feature` and `detect_concept` functions. Thus, `extract_feature` is actually a collection of four different algorithms, one for extracting each type of feature: `extractCH` for color histograms, `extractCC` for color correlograms, `extractTX` for texture wavelets, and `extractEH` for edge histograms. The `detect_concept` function is using the extracted features to identify

pre-defined concepts in the images; it works the same for all features. For detailed functionality descriptions of all these functions, see Chapter 4.

The solution presented in Listing 8.1 is the only high-level solution we have investigated for MARVEL. In the analysis phase, one can find different algorithms for the feature extraction or use different solutions for concept detection. For now, we choose a single implementation of each feature extraction and for the concept detection; consequently, the analysis phase output is the algorithm sketched above.

Performance specification From the required execution rate of 30 fps, we extract the performance requirements: the application has to process 30 images in 1 s.

We note that the performance is underspecified: we know a time requirement, but we have no restrictions on latency or throughput. For example, 30 fps can be reached by either processing 30 images in parallel, where each image takes 1 s to process, or by processing one image within 1s/30, and traversing the image collection sequentially. The first solution has high latency (1s) and high throughput (30 images), while the second has low latency, and low throughput.

8.3.2 The Design Stage

Here we transform the solution found in the previous stage in one or several parallel algorithms. We aim to extract *all* concurrency levels of the application, and to expose them as layers in our design models. Thus, we decompose the application to its minimal granularity units, following top-down approach: (1) expose task parallelism, (2) determine and expose all data parallel layers, and (3) specify synchronization and reduction points.

Two task parallel versions of MARVEL are presented in Listing 8.2.

Listing 8.2: *Exposing task-parallelism for MARVEL (two designs).*

```
1  //task paralel algorithm - T-par1:
2  par (i=1..sizeof(IMG)) {
3    read_image(IMG[i]);
4    {  // task parallelism
5      {K[CH,i]=extractCH(IMG[i]); s[CH,i] = concept_detection(MODELS[CH], K[CH,i])} ||
6      {K[CC,i]=extractCC(IMG[i]); s[CC,i] = concept_detection(MODELS[CC], K[CC,i])} ||
7      {K[TX,i]=extractTX(IMG[i]); s[TX,i] = concept_detection(MODELS[TX], K[TX,i])} ||
8      {K[EH,i]=extractEH(IMG[i]); s[EH,i] = concept_detection(MODELS[EH], K[EH,i])}
9    } ;
10   final_score[i] = compute(s[CH,i],s[CC,i],s[TX,i],s[EX,i]);
11 }
12
13 //task paralel algorithm - T-par2:
14 par (i=1..sizeof(IMG)) {
15   read_image(IMG[i]);
16   {  // task parallelism
17     K[CH,i]=extractCH(IMG[i]) ||
18     K[CC,i]=extractCC(IMG[i]) ||
19     K[TX,i]=extractTX(IMG[i]) ||
20     K[EH,i]=extractEH(IMG[i])
21   };
22   syncronize(_all_)
23   { // task parallelism
24     par (f=CH,CC,TX,EH) {
```

```
25        s[i,f] = concept_detection(MODELS[f], K[f,i])
26      };
27      seq (f=CH,CC,TX,EH) {
28      final_score[i] = compute(final_score[i], s[i,f])
29    }
30 }
```

To add data parallelism to the task parallel version of MARVEL, all kernels need to be able to run on slices or blocks of the image, and the reduction function (i.e., the computation that builds the overall result from the pieces) needs to be simple enough to be quickly executed.

Thus, we need to parallelize each feature extraction and the concept detection, focusing on data partitioning and simple results reduction. For all feature extractions we exploit one data-parallel dimension: we slice the images and run the extraction on multiple slices in parallel. For the concept detection, we expose two layers of data parallelism: (1) we compare each image feature vector against different concepts (i.e., vectors from each model) in parallel, and (2) we compute the dot-product of each vector pair multiplying multiple blocks in parallel.

The data parallel versions of `T-par1` and `T-par2` are built in the same way; therefore, we use `T-par2` as an example, and only present its data-parallel version in Listing 8.3.

Listing 8.3: *Exposing both data and task parallelism for MARVEL.*

```
1 //paralel algorithm MARVEL-DT-par2:
2 MARVEL-DT-par2 = {
3   par (i=1..sizeof(IMG))
4         analyze_image(IMG[i]);
5 }
6 analyze_image(IMG) {
7         read_image(IMG);
8         K=extract(IMG,FEATURES);
9         s=detect(K,MODELS);
10        seq (f=CH,CC,TX,EH)
11            final_score = compute(final_score, s[f])
12        return final_score
13 }
14 extract(IMG[i],FEATURES) = {
15 // task parallelism
16    par (f=CH,CC,TX,EH) {
17 // data parallelism
18      par (k = 1 .. sizeof(SLICES(IMG[i], CH)))
19              sK[CH,i,k]=extract[f](SLICE(k,IMG[i]))
20    };
21 // result reconstruction, with different operations per feature
22    par (f=CH,CC,TX,EH)
23      K[f,i]=reduce(sk[f,i,_all_], op[f])
24    return K[f,i]
25 }
26
27 detect(K[i],MODELS) = {
28 // task parallelism
29        par (f=CH,CC,TX,EH)
30            s[i,f] = concept_detection(MODELS[f], K[f,i])
31        return s[i,f]
```

```
32 }
33
34 concept_detection(MODEL, K) {
35     par (v = 1 .. sizeof(MODEL)) {
36           M = MODEL[v];
37           par (k = 1 .. sizeof(SLICES(M)))
38             ss[k,v] = compute(SLICE[k,M], SLICE[k,K])
39           s[v] = reduce(ss[_all_,v])
40     }
41     concept = reduce(s[_all_])
42     return concept
43 }
```

To evaluate the execution time of any of these models, we use the following rules: the execution time of a parallel operator (either `par` or `||`) is the maximum of the parallel runs, while the execution time of a sequential operator (either `seq` or `;`) is a sum of the sequential runs. For example, the computation of the performance expression of `DT-par2` is presented in Equation 8.1.

$$
\begin{aligned}
P_{DT-par2} &= \max_{i=1,\#IMG} T_{analyze\_image}[i] \qquad (8.1)\\
T_{analyze\_image[i]} &= T_{read\_image}[i] + T_{extract}[i] + T_{detect}[i] + \\
&+ \sum_{f=CH,CC,TX,EH} T_{final\_score[f]}[i] \\
T_{extract}[i] &= \max_{f=CH,CC,TX,EH} T_{dp-extract[f]}[i] + \\
&+ \sum_{f=CH,CC,TX,EH} T_{reduce[f]} \\
T_{dp-extract[f]}[i] &= \max_{s=1,\#SLICES} T_{extract[f]}[SLICE[i,s]] \\
&\approx \frac{T_{extract[f]}[i]}{\#SLICES} \\
T_{detect}[i, MODELS] &= \max_{f=CH,CC,TX,EH} T_{detect[f]}[i, MODELS] \\
T_{detect[f]}[i, MODELS] &= \max_{m=1,\#MODELS} T_{m-detect[f]}[i,m] \\
T_{m-detect[f]}[i,m] &= \max_{s=1,\#BLOCKS} T_{dp-m-detect[f]}[BLOCK[i,s],m] + \\
&+ \sum_{f=CH,CC,TX,EH} T_{reduce[f,m]} \\
T_{dp-m-detect[f]}[BLOCK[i,s],m] &= \max_{s=1,\#BLOCKS} T_{extract[f]}[BLOCK[i,s],m] \\
&\approx \frac{T_{detect[f]}[i,m]}{\#BLOCKS}
\end{aligned}
$$

Note that `DT-par*` expresses the *maximum* degree of concurrency at the application level, without considering any mapping and subsequent scheduling decisions that occur as soon as the parallel

application runs on a real machine. In other words, it assumes a model with infinite compute and memory resources.

We could potentially compare all these parallel designs at the expression level, and remove the ones that are obviously slower than others. For example, `T-par2` is slower than `T-par1`, as the execution stalls until all feature extraction kernels have finished, and only then continues with the concept detections. On the other hand, mapping and scheduling decisions can cancel these differences. For example, on a platform with only 4 cores, the overall result of both `T-par1` and `T-par2` is the same. However, mapping and scheduling *do not change* expression-level ranking: a design A slower than another design B will remain slower, or at most equal in performance with design B on any platform.

Overall, pruning the solution space search is not trivial using this abstract, application-only model. To really compare the options and rank them according to more realistic execution times, we need some knowledge about (1) the execution times of all the kernels, and (2) platform resources and mapping. In turn, this information is only available when a choice for a target platform has been made, which is (the earliest) in the prototyping phase.

As mentioned in the previous section, we need to make a choice here between a depth-first and a breadth-first approach to implementing a parallel MARVEL. For an application like MARVEL, using more than one parallel design in the prototyping phase is not very difficult, so we choose for a breadth-first approach. However, for larger applications, the reduction of the solution space has to be a lot more drastic here, and has to continue in the prototype phase.

8.3.3 The Prototype Stage

Due to the breadth-first approach we took in the design stage, we divide prototyping in two phases: a model-based phase and a coding phase; in the case of a depth-first search, the model-based phase would be skipped.

The model-based phase

We map the parallel designs of MARVEL, such as `DT-par1` and `DT-par2`, on the potential hardware platforms. For the sake of this exercise, we use a generic GPMC, the Cell/B.E., and a GPU as target platforms. Our goal is to achieve a more accurate performance prediction, and use it to limit the number of solutions to be implemented (ideally, no more than two per platform).

First, Listings 8.4, 8.5, and 8.6 present a high-level view of the PAMELA machine models for the three platform families we discuss.

We use the following conventions and notations:

- `core` is a processing resource; `memory`, `local_memory`, `shared_cache`, `cache` are classes of memory resources; depending on the resource class, certain usability restrictions may apply.
- `resource(m)` is a resource from the class `resource` with multiplicity `m` (i.e., the platform has `m` identical resources of this particular type);
- `use(R,t)` acquires a resource `R` and uses it exclusively for time `T`; `use(R[_all_],t)` acquires all resources `R` for the time `t`; `use({R,Q},t)` uses *both* resources `R` and `Q` for time `t`;
- `t_op` is the notation for the time taken by the operation `op`. Note that `op` can be any function.
- `threads(N)` are the application threads;

- `schedule` encodes the way the platform distributes threads on cores; a schedule is a tuple of four components for each type of core: it specifies what is the logical to physical mapping (threads on cores, in all three cases), with allowed multiplicity (i.e., how many logical units "fit" on a physical one), with the placement policy (i.e., FCFS = first come, first served) and the preemption time (preempt=0 is equivalent with no preemption).
- `map(threads:function)` assigns each thread the function to run `function`; the `_all_`, `*`, `n` qualifiers specify that the function runs on all, any, or any `n` threads, respectively; once a function is mapped on a thread, *all* its `par` sections become `seq` sections, and eventual data parallelism is collapsed into complete sequential loops.

Listing 8.4: *A high-level machine model for a GPMC.*

```
1 //a machine model for a GPMC with no hardware multi-threading
2 L1     = cache(P)
3 L2     = cache(P)
4 CORE   = core(P)
5 MEM    = memory(1) // main memory, shared
6 L3     = shared_cache(1) // L3 cache, shared
7
8 compute(op) = use(CORE, t_op) //use any core, exclusively
9 reduce(op)  = use(CORE(_all_), t_op) //use all cores
10
11 //an application is a collection of threads
12 T      = thread(N);  //from the application
13 schedule(T:CORE, 1:1, FCFS, preempt=t)
```

Listing 8.5: *A high-level machine model for the Cell/B.E..*

```
1 //a machine model for the Cell/B.E.
2 SPE      = core(P); // from the HW
3 LS       = local_memory(P)
4
5 PPE      = core(2);
6 MEM      = memory(1);
7 PPE_L2   = shared_cache(1);
8
9 SPE_compute(op) = use(SPE, t_op);
10 SPE_reduce(op)  = use({SPE[_all_],PPE}, t_reduce);
11 PPE_compute(op) = use(PPE, t_op);
12
13 //an application is a collection of two types of threads
14 T_work = thread(N);  //worker application threads
15 T_main = thread(M);  //main app threads
16 schedule({T_work:SPE, m:1, static, preempt=0},
17          {T_main:PPE, 1:1, FCFS, preempt=t});
```

Listing 8.6: *A high-level machine model for a GPU.*

```
1 //a machine model for the GPUs
2 HOST_CORES   = core(HP);
3 DEVICE_CORES = core(DP);
```

```
4
5 HOST_MEM    = memory(1);
6 DEVICE_MEM  = memory(1);
7 SH_MEM      = local_memory(HP);
8 CONSTANT    = memory(1);
9 [...] // other types of memory
10
11 DEV_compute(op)   = use(DEVICE_CORES(_all_), t_op);
12 DEV_reduce(op)    = use({HOST_MEM, DEVICE_MEM}, t_copy); HOST_compute(op)
13 HOST_compute(op)  = use(HOST_CORES, t_op)
14 HOST_reduce(op)   = use(HOST_CORES(_all_), t_reduce)
15
16 //an application is a collection of threads of two types of threads
17 T_work = thread(N); //from application
18 T_main = thread(M); // main app thread
19 schedule({T_work:DEVICE_CORES, _all_:_all_, HW, preempt=0},
20          {T_main:HOST_CORES, 1:1, FCFS, preempt=t});
```

Note that the machine models presented above are not complete - we omit on purpose the models of memory transfers (i.e., read/write operations from various memories) and communication for each platform, because they are not used in our high-level evaluation. The complete models are presented in Appendix B.

For each of these platforms, there are several potential mappings of a parallel design. A few examples, for `DT-par2`, in decreasing order of their granularity, are:

1. `map(threads(_all_):image_analysis)`.
   Each thread will execute the full image analysis, but different threads analyze different images. In terms of load balancing, this is an ideal solution, and it works best for GPMC platforms. The performance prediction for this mapping is given in Equation 8.2, where P is the number of cores the GPMC has, and $T_{analyze\_image}$ is the time taken by the sequential application to execute for one image.

$$T_{map1} = \sum_{i=1..\#IMG/P} \max_{1..P} T_{analyze\_image} \tag{8.2}$$

$$\tag{8.3}$$

   For the Cell/B.E., a memory footprint analysis - i.e., comparing the required kernel memory with the available memory - shows that the kernels do not fit in the SPE memory, so this mapping is not possible. For the GPU, the task-parallel approach of this mapping is not suitable.

2. `map(threads(*):{extract(IMG[i],FEATURE[f]);detect(IMG[i],FEATURE[f])}`.
   Each thread executes one extraction and the corresponding detection for the entire image; the `IMG` collection is processed sequentially, one image at a time. The performance prediction for this mapping is given in Equation 8.4, where $T_{extract}[f]$ and $T_{detect}[f]$ are the times taken by the extraction and detection of feature f, respectively. These times are measured for the isolated, sequential execution of each kernel on the chosen platform, for a "random" test image. As expected, performance varies with P, the number of available cores and the number of features to be extracted (we omit the parameterized expression as it is hardly readable).

$$T_{map1} = \sum_{i=1..\#IMG} T_{analyze\_image}[i] \tag{8.4}$$

$$P = 2 \Rightarrow T_{analyze\_image}[i] = \max_{f=CH,TX} T_{extract[f]}[i] + \max_{f=CH,TX} T_{detect[f]}[i] + \max_{f=CC,EH} T_{extract[f]}[i] + \max_{f=CC,EH} T_{detect[f]}[i]$$

$$P = 4 \Rightarrow T_{analyze\_image}[i] = \max_{f=CH,CC,TX,EH} T_{extract[f]}[i] + \max_{f=CH,CC,TX,EH} T_{detect[f]}[i]$$

$$P = 8 \Rightarrow T_{analyze\_image}[i] = \max \max_{f=CH,CC,TX,EH} T_{extract[f]}[i], \max_{f=CH,CC,TX,EH} T_{detect[f]}[i-1]$$

Suitable for both the GPMC and the Cell/B.E., this mapping is equivalent with a task-parallel mapping. However, a memory footprint test is still required to assess if the image size is not too large to allow one image to be processed entirely by one SPE. The lack of data parallelism makes also this mapping unsuitable for the GPU.

3.
```
seq(f=(CH, CC, TX, EH) {
  map(threads(_all_):extract(SLICES(IMG[i],k),FEATURE[f]);
  reduce(threads(_all_));
  map(threads(_all_):detect(BLOCKS(K[i,f],MODELS[f]);
}
```

Each thread executes one extraction for one image slice. All threads execute the same extraction, on the same image. As soon as the processing is finished, the results are reconstructed, and a new data-parallel execution is launched, for the concept detection. The performance prediction for this mapping is given in Equation 8.5, where $T_{slice\_extract}[f]$ and $T_{block\_detect}[f]$ are the times taken by the extraction and detection of feature f for a slice of the image and a block of the featrue vector, respectively. These times are either measured for the data parallel execution of each kernel on the chosen platform, for a "random" test image, or approximated from the values measured per kernel, as $T_{slice\_extract[f]}[i] = fracT_{extract[f]}[i]\#SLICES$ and$T_{block_detect[f]}[i] = fracT_{detect[f]}[i]\#BLOCKS$.

$$T_{map3} = \sum_{i=1..\#IMG} \Big(\sum_{f=CH,CC,TX,EH} T_{slice\_extract[f]}[i] + T_{reduce(\_all\_)} + \sum_{i=1..\#IMG} \Big(\sum_{f=CH,CC,TX,EH} T_{block\_detect[f]}[i] + \tag{8.5}$$

This mapping is equivalent to the pure data-parallel approach we have discussed in Chapter 4. While we have showed that this is not suitable for a platform like the Cell/B.E., it is a suitable mapping for a GPU, which performs best when all threads run the same computation. Note that, for the same platform, the reduce operation is quite expensive, so any solution that removes that synchronization point should be evaluated as well.

Many other mappings exist, but these examples are sufficient to show how application models can use the mapping information.

Still, to generate and evaluate expressions like those in Equations 8.2, 8.4, and 8.5, we need (1) information about the execution times of the various kernels, parameterized with the number of cores and/or image size, (2) clear and deterministic scheduling policies (i.e., what happens if computing resources are less than running threads), (3) memory footprint information per kernel, parameterized, and (4) data distribution and memory affinity information. To gather any of these, benchmarking of code prototypes is the most convenient way. This solution implies coding and, therefore, platform-dependent results. While detailed machine modeling might seem a better option, the complexity of these multi-core architectures, especially in terms of memory hierarchies and cores-memories interaction, is too high to allow enough accuracy with reasonable prediction times. Furthermore, complex platform models require complex application models, which, for large applications, are simply a lot more difficult to develop by hand than regular source code. Therefore, lowering the modeling granularity below the one discussed in Listing 8.3 is not effective anymore.

To summarize, the MARVEL model-based prototyping has proved that once the parallel application model is written such that concurrency is exposed at all granularity levels, mapping becomes a trial-and-error fitting exercise. We believe all mappings can be generated automatically, as permutations of the loops in the code, limited by whatever kernels fit on the avalable compute cores. To evaluate the mappings, we require micro-benchmarking on the finest grain kernels, and rules for composing them. Our experience shows that these rules are simpler for platforms with no caches (like the Cell/B.E.) and more complex for platforms with multiple cache levels. Further, the finer the granularity, the more difficult the composition performance is to predict (i.e., less accurate). Therefore, we recommend (1) to choose for mappings which match the degree of concurrency exposed by the application with the degree of parallelism exposed by the platform, and (2) to evaluate/implement prototype models in the decreasing order of their granularity, for each platform. Finally, note that inter-platform comparisons of prototype models (aiming to find the absolute best performance) requires machine models and application models (the kernel implementation, micro-benchmarking, and mapping) to be similarly accurate. Again, our experience shows this is a difficult task to achieve. Therefore, unless the comparison of the symbolic expressions already indicates the better platform, numeric comparisons (especially comparable numbers - within 20-30%) should be used with caution.

The coding phase

Once the choice for the models to be prototype is made, it is time for coding. If portability to other platforms is not an issue, platform-centric programming models are easier to use and lead to good performance. If multiple platforms are of interest, OpenMP (for homogeneous, low-scale platforms) and OpenCL (for heterogeneous, hierarchical platforms) can be used for prototyping.

Once the coding is done, a small set of running prototypes is available. From this moment on, implementations and optimizations are platform specific. These two phases are not covered by this example - details on how they are performed can be seen in our case-studies (Chapters 3, 4, and 5).

8.3.4 The Roll-back

Parallel MARVEL's implementation can roll-back to choosing a different `par*` design, a different mapping, or a different target platform. At every step back, the entire process starting at the roll-back checkpoint has to be repeated, including all decisions, using new parameters. Ideally, this roll-back to

intermediate development points makes it easier to test multiple solutions and recover quicker from "dead-ends".

8.4 Current status

We believe that MAP framework covers the strategy in its completeness, but it still lacks the necessary tool support. We present an overview of the required MAP tools and the available ones in Table 8.1. For each stage, we summarize the required actions and the tools recommended/required to perform them, we evaluate the automation potential, and we give examples of such tools, when available.

Table 8.1: *An overview of the MAP framework required and available support tools.*

MAP Stage	Action(s)	Tool(s)	Automated?	Examples
User Input	Specify functionality	Language	NO	-
	Specify/analyze data	Analyzer	PARTIAL	Matlab, R
	Specify performance	-	NO	-
Hardware	Model architecture	Modeling language	NO	PAMELA
		HW-to-model compilers	YES	-
		Micro-benchmarking	YES	Code suites
Analysis	Find solutions	Patterns	PARTIAL	OPL
	Specify solutions	Modeling language	NO	DAGs, pseudo-code
	Select solution	Symbolic analyzer	PARTIAL	High-level metrics, Roofline
Design	Parallelize	Similarity analyzer	PARTIAL	Patterns, metrics
		Modeling language	NO	PAMELA
	Predict/rank/select	Predictor	YES	Roofline, PAMELA (the predictor)
Prototype	Model solutions	Mapping generators	YES	-
	Predict/rank	Performance predictor	YES	PAM-SoC
	Code	Programming models	PARTIAL	OpenMP, OpenCL
	Benchmark	Benchmarking suites	PARTIAL	ParSEC
	Check performance	Timing	YES	profilers
Implementation	Coding	Programming models	YES	programming tools
	Check performance	Timing	YES	profilers
Optimizations	Apply optimizations	Tuners/Wizards	YES	Auto-tuning
	Check performance	Timing	YES	profilers
Roll-back	Back-track	Selector	PARTIAL	-

Note that most framework stages can be covered by tools, with moderate to minor user intervention (most of these tools can be automated). Still, available tool support is very scarce: tools inherited from large-scale parallel machines are not yet able to cope with multi-core complexity and multiple granularity scales, while new tools are very platform-centric, and therefore hardly suitable for generic parallel design and evaluation. Implementation and optimization are probably the best supported stages, with multiple choices of programming tools and models, as well as optimization guidelines and wizards. However, these tools do not cover early application design and evaluation. Benchmarking and micro-benchmarking of multi-cores have recently become fields of active interest, so there are not enough choices in either synthetic or real benchmarking codes that expose enough concurrency to make performance analysis and prediction challenging. Finally, application specification and modeling is currently (indirectly) studied by researchers working on patterns and application metrics; hopefully, they will be able to determine ways to use application similarity to increase the efficiency of the application analysis stage, one of the most cumbersome and error-prone steps in the development flow.

8.5 Summary and discussion

As more and more applications need to be run on multi-core platforms, effective parallelization becomes mandatory. The experimental, hardware-centric approach the community has taken so far - mapping and super-optimizing each application on each single platform, just to find out what is the best solution - does not scale.

As an alternative, in this chapter we have described an application-centric approach to parallel programming for multi-core processors. In a nutshell, this approach focuses on the application parallelization solutions and alternatives, and matches those on suitable platforms early in the design process. Further, being based on performance feedback, this solution provides a natural stop-condition for the otherwise endless optimize-and-test loop seen for many case-studies. Furthermore, although applications implemented this way might not achieve the best performance they could ever get in theory, the increased productivity and portability are more important gains. And these optimizations are still allowed in the final stages, if they are required.

Our application-centric development strategy is supported by our generic framework, MAP, that enables the user to design, develop, and tune a parallel application on multi-cores. Our framework closely matches each step of the strategy, as we analyze both manual and potential (semi-)automated solutions to perform the operations in each stage. To show how MAP works for a given application, we revisit the MARVEL case-study (see Chapter 4) and discuss in detail how each component on the framework can and should be used. Finally, generalizing from this example, we have shown that the tool support for MAP has a lot of gaps between required and the available tools.

Despite our observation that most of the operations are still to be performed by the user (i.e., we severely lack tool support for an application-centric development of parallel workloads for multi-cores), the MAP framework sets the right discipline for approaching multi-core programming in an effective way. If work in progress on topics like application characterization, architecture modeling and simulation, and performance prediction is successful, MAP can be 75% automated (the remaining 25% are in the high-level application specification and the performance measurements).

Chapter 9

Conclusions and Future Work

In this chapter, we present our conclusions and sketch future research directions.

Less than six years ago, processor designers declared the future of computing to be parallel. Since then, multi-core processors have divided the research community in two camps: those who advertise them as essential for high performance computing (HPC) applications [Car07; Wil06] and those who question their value due to programmability concerns [Aga07; Ass08].

Despite the controversy, one thing is clear: until new technologies will emerge to provide solutions to dig through (or avoid) the memory, parallelism, and energy walls [McK04; Asa09b], multi-core processors are here to stay. With that, application parallelism has become mandatory. And the software community is (suddenly) faced with a large-scale problem: virtually every application will have to be parallel, rather sooner than later.

While the software community came up with various solutions for specific cases, a systematic, generalized approach has not yet appeared, and we see no convergence on the horizon. But with multi-core complexity steadily increasing (see Chapter 2), addressing these cases one at the time will simply be too slow.

Along this thesis (Chapters 3,4,5), we have shown that the problem is not easy: getting close to the theoretical peak performance of multi-cores requires applications to exploit multiple layers of parallelism, investigate memory locality, perform data-dependent optimizations, and, in many cases, get to know the inside details of complex architectures to be able to tweak applications to the right performance level. Even worse, some (traditional) parallel algorithms are not naturally suitable for these processors. In these cases, additional effort needs to be put in algorithm analysis and parallel design, to enable decent performance.

Our experience shows that one effective way to address the multi-core programming challenges is to find the right level of abstraction to reason about applications, supported by a clear parallel programming discipline to implement it. However, programming models available today lack either one or the other: hardware-centric approaches lack high-level application design features, while application-centric strategies lack a stronger connection to the hardware (Chapter 7). To enable these two components, we have proposed MAP, a multi-level framework that tackles the entire multi-core development process, from analysis to implementation and optimization (Chapter 8).

9.1 Conclusions

Our research on application parallel application development for multi-core processors has lead to the following conclusions.

1. Multi-core processors are taking over the HPC field due to their high performance. Current architectures are separated in three main classes: heterogeneous master-slave architectures, gen-

eral purpose multi-cores, and graphics processing units. The major differences arise in (1) the number and complexity of cores, (2) the core type(s), and (3) the memory hierarchies.

The multi-core architectures of the near future will be harder to characterize. The differences in core complexity will attenuate, and it is likely that most architectures dedicated to HPC will be heterogeneous: one (or few) big, complex, core(s) for control threads, and a multitude of accelerators, potentially of different types. As a result, the parallelism layers will increase and diversify even further. We note that current programming models are definitely not prepared for this level of complexity.

2. Traditional HPC applications, even if already available in parallel versions for massively parallel machines, are not an already solved category for the multi-core processors. These applications still require careful parallelization and optimization and, it has to be said, they rarely preserve the elegant and concise form of scientific kernels.

   However, there is a methodical way to address the problem, and the exposure of this unavoidable multi-core complexity in a clear, unified manner, focusing on the multi-layered parallelization opportunities and challenges, and showing a possible, but difficult path to follow for extracting the expected high level of overall performance.

   Once this path is found and correctly followed for a real life application, like Sweep3D (Chapter 3), it is worth the effort: the implementation achieved an improvement factor of over 20 on the Cell/B.E. versus commodity processors. This proves that the Cell/B.E. processor, although initially designed mainly for computer games [D'A05] and animation [Min05], is a very promising architecture for scientific workloads as well. Even though the simple architectural design of its SPUs forces the user to consider a number of low-level details, it offers good opportunities to achieve high floating point rates: in our case we were able to reach an impressive 64% of peak performance in double precision (9.3 GFLOPs/second) and almost 25% in single precision (equivalent to 50 GFLOPs/second).

3. Running multi-kernel applications on Cell/B.E. should use both data and task parallelism (preferably allowing for a dynamically configurable hybrid approach), and requires optimized high-level mapping and scheduling. To enable this, SPE kernels require additional flexibility.

   For our experiments, we have used MarCell, a multimedia analysis and retrieval application which has five computational kernels, each one of them ported and optimized for the SPEs. We have shown how exploiting the SPEs sequentially leads to an overall application speed-up of more than 10, task parallelization leads to a speed-up of over 15, while the data parallelization worsens performance due to a high SPE reconfiguration rate. We have shown how certain scheduling scenarios can increase the overall application speed-up over 20, with a maximum of 24.

   Specifically, we have drawn three important conclusions. First, we have proved how task parallelism can be a good source of speed-up for independent kernels, but we have also shown how a significant imbalance in the execution times of some of these kernels can alter overall performance. Second, we have shown how data parallelism is, in most cases, inefficient due to the very high overhead of SPE reprogramming. Finally, we have shown how an application combining the two approaches - data and task parallelism - can gain from both; however, this reconfigurability potential does come at the price of programming effort, because all kernels need to be programmed to accept variable I/O data sizes and local structures, and the main thread needs to be able to run various SP combinations of these kernels.

4. Implementing data-intensive and dynamic application kernels on the three main multi-core families available today is non-trivial.

   When a data-intensive HPC application, possibly with irregular access patterns, has to be implemented, the biggest problems are platform choice and a prohibitively high programming effort to optimize for any or even a small number of platforms. Each platform is not only different in hardware architecture, but also in programming method. Although at a higher abstraction level, similarities in optimization strategies for all platforms can be found, the performance gains are different, as is the effort and techniques to get there.

   As for the platform choice, there is no clear winner, as it depends completely on the specific application, platform experience, available time and funds. We do have presented a list of guidelines to make an educated choice. For porting and optimization efforts, we have also presented guidelines, centered around memory and data optimizations, rather than optimizations on computing that have been predominant up till now.

5. Effective parallel application development for multi-cores requires (as multiple case studies show) both careful high-level parallelization and hardware-aware low-level implementation.

6. However, available multi-core programming models are separated into two disjoint classes: application-centric and hardware-centric. This split mimics, and widens, the programmability gap: the former models focus on design, and disregard many low-level optimizations, while the latter models focus on platform-specific implementation.

   The only available partial solution to this problem is OpenCL, which addresses the problem by providing a common platform model, that becomes the target platform for both application design and implementation, literally bridging the gap between the two.

7. Tool support for application development on multi-cores is minimal: debuggers, profilers, performance predictors, code analyzers are almost inexistent.

   Generic application-centric development is possible if it is supported by a disciplined strategy. This, however, can add extra overhead to programmers, unless it is supported by the right framework and tool-chain. While the framework prototype can be designed and used "by hand", for various case studies, the tool support is, for now, minimal.

9.2 Future Work Directions

As seen from our conclusions, despite the huge research efforts already put in simplifying multi-core programming, there are still significant gaps to be filled. We believe the MAP framework is a suitable solution for solving some of these problems. Therefore, our future work targets improving the framework itself, adding the required tools, and testing it on multiple applications.

Specifically, we propose the following research directions:

1. *Application specification and analysis*

   Our application specification strategy is, for now, ad-hoc, and not compared against alternatives. One possible extension of our work is into defining a more powerful solution for application specification, which should enable a more detailed analysis of the application in its earliest stage. Such a strategy should comply with two important conditions: (1) it has to make it easier for

the application domain specialists to provide more details on the computation kernels and their dependencies, and (2) it should allow an automated application analysis tool to extract/compute application characteristics such as concurrency levels, data and task parallelism potential, arithmetic intensity, and so on.

2. *Parallelization strategies*

   Another interesting research direction opened by the MAP framework is related to parallelization strategies. Note that the framework we propose is essentially a breadth-first search in the space of the potential parallel solutions for an application. However, creating this space is in itself a non-trivial research problem, as it requires generating all parallelizations for a given algorithm. For example, if the algorithm is expressed as a dependency graph, a promising approach can be to investigate various graph permutations and the rules they have to obey to still be a valid parallelization of the original algorithm. While we had no time to investigate the problem further, we note that such a tool will be of wide interest, far beyond the MAP framework context.

3. *Performance analysis and prediction*

   Following up on the idea that MAP enables a BFS search in the parallel solutions space, we emphasize that such a search has to be guided by a predictor, which has to rank the available solutions according to their performance potential. Such predictors are hardly available, and they are, in most cases, application-class or platform specific. Developing a more generic platform predictor is yet another interesting research direction, given its visibility and its challenges, as well as its novelty - while it has been proven possible for simpler parallel architectures, multi-core performance prediction is yet to be tackled generically.

4. *Platform and Application modeling*

   To enable performance prediction, both platforms and applications require modeling. While there are several strategies already available, we argue that prediction modeling requires extra care, due to the trade-off between model complexity, prediction accuracy, and prediction time. Modeling techniques borrowed from (larger) massively parallel machines are typically not fine-grained enough, and application modeling techniques from the same domain turn out to be too fine grained. We recommend another research direction in finding the right balance between the two.

5. *Application-to-platform matching*

   We have argued that application-centric design is the way to solve the portability problem, by preserving platform independence for as long as performance requirements allow it. Of course, a different approach is to find the best platform for a given application early in the design phases. To do so, both applications and platforms need to thoroughly characterized by a large set of features, which should be visible early enough in the design process (preferably not later than the parallelization). Furthermore, the two classes of features need to be "fit": the applications characteristics that match platform metrics need to be paired correctly and, based on the match in values, a fit can be calculated. This application-platform fit is another novel research area, which would re-enable hardware-centric application design, focusing on exploiting the most out of the most suitable target platform for an application.

Appendix A

Data-Intensive Kernels on Multi-Core Architectures ††

As discussed in Chapter 5, data-intensive applications are not multi-core friendly, as they defeat the number crunching purpose of these architectures. However, there are more and more scientific and consumer applications which are data intensive. Fields like multimedia and gaming, radioastronomy, medical imaging, or radar processing require larger and larger data collections to increase the accuracy and/or the resolution of their results. Therefore, we are facing yet another pair of contradictory trends: hardware processors increase their computational power, while applications become more data-intensive.

In this appendix we extend our discussion on designing and implementing parallel solutions for multi-core processors. If Chapter 5 discussed mainly the case of the Cell/B.E., in this chapter we analyze in more detail the parallelization of these applications on the other two families of multi-core processors - namely, general purpose multi-cores (GPMCs) (in Section A.1) and graphical processing units (GPUs) (in Section A.2). Further, we compare the parallelization of the same case-study application - the radio-astronomy kernels for gridding and degridding - on all three platforms, aiming to understand what are the advantages and disadvantages that each multi-core offers. Based on several types of experiments (presented in Section A.3), we are able to determine (1) a common path to parallelize data-intensive applications on all these multi-core families, as well as a set of empirical guidelines for choosing the most suitable platform for another (similar) data-intensive application (see Section A.4).

A.1 General-purpose multi-cores (GPMCs)

Multi-core CPUs have the advantage of availability of hardware and development tools, and of cost, as they are by far the most extensively used processors in server, desktop, and HPC machines. Our test platforms in this category are three systems with up to eight x86-64 cores and with SSE2 support for SIMD operations. On systems with Xeon 5320 processor(s), some cache coherency traffic passes through the front-side bus. The Core i7 has four cores, but makes them available as eight virtual cores using hyperthreading. All systems use a UMA memory model with at least 4 GB RAM. The standard programming model for this platform type is based on multi-threading.

A.1.1 Gridding on the GPMCs

Using block-wise, symmetrical visibility data distribution, we managed to parallelize the application using `pthreads` within few days. To solve the problem of concurrent writes on the grid, each thread

†† This chapter is based on previous work published in the proceedings of CF 2009 ([Ame09]).

works on a private grid. This approach is more efficient than any locking scheme, because the accesses into the grid and convolution matrices for each data value can start at any position would otherwise require a shared locking structure, wasting precious memory bandwidth. Besides, conflicts are rare, except in a number of hot spots. A final reduction step centralizes the private grids, a step that can be parallelized as well without access conflicts by partitioning.

As a next step, symmetrical data distribution was replaced for a master-worker approach. Workers are still operating on a private grid, but fetch jobs from a private queue, which is filled by the master. The master improves the worker cache benefits by increasing data locality in the **C** and/or **G** matrices. To do so, it places jobs with (semi-)overlapping matrix areas in the same queue, without leaving any queue empty.

In this case, the computation of grid and convolution matrix offsets has to be performed by the master. We limit the filling rate with a maximum queue size to avoid effectively sorting too much data, which would be unrealistic in a streaming environment. We have also improved in-core performance of the workers with SSE2 intrinsics.

A.2 Graphical Processing Units (GPUs)

For this category, we use NVIDIA Geforce and Tesla GPUs with 16 or 30 SIMD multi-processors. Each multi-processor has 8 execution units; it is typical to count these units as cores in this Single Instruction, Multiple Threads (SIMT) model, because each can reference any memory address (scatter/gather) and take any branch (although diverging threads lead to execution serialization, with serious performance penalties). Each multi-processor offers various local storage resources, such as 8k or 16k registers and 16 KB scratch-pad ("shared") memory, which are dynamically partitioned, and also a constant cache of 8 KB. Each cluster of 2 or 3 multi-processors share a single texture cache (8 KB per multi-processor). Constant memory (64 KB) and texture memory are stored in the global memory (the slowest device memory).

To program NVIDIA GPUs, we have used the API from the Compute Unified Device Architecture (CUDA) [nVi07]. CUDA is a hardware/software architecture developed by NVIDIA to perform general purpose computations on a GPU, without the need to first translate the application into the computer graphics domain. A compiler, runtime libraries and driver support are provided to develop and run CUDA applications.

CUDA extends the C/C++ programming languages with vector types and a few declaration qualifiers to specify that a variable or routine resides on the GPU. It also introduces a notation to express a kernel launch with its execution environment. A kernel is launched by specifying the number of threads in a group (block) and the number of blocks that will execute the kernel. In a routine on the GPU, a thread can compute the locations of its data by accessing built-in execution environment variables, such as its thread and block indexes. Blocks are automatically assigned to one of the multi-processors when enough resources are available and a hardware scheduler on each multi-processor determines periodically, or on thread stalls, which (partition of a) block runs next.

A.2.1 Gridding on GPUs

NVIDIA provides BLAS (CUBLAS) and FFT (CUFFT) libraries to accelerate the porting effort of an application based on BLAS or FFT kernels. Required functions of the CUBLAS library were taken as a starting point to port our application kernel. Since the source code of CUBLAS and CUFFT are available for download, we could specialize the required routines. We pushed looping over the

Table A.1: *The characteristics of the platform-specific parallelizations*

Platform	Memory Model	Core type	Programming model	In-core optimizations	Data distribution	Task Scheduling
GPMC	Shared memory Coherent caches	Homogeneous	Multi-threading	SIMD	Symmetric, blocks	OS
Cell	Distributed memory	Heterogeneous	Cell SDK	SIMD , Multi-buffering	Dynamic	Application specific
GPU	Distributed memory Shared Memory Small caches	Homogeneous	CUDA multi-threading	SIMD Memory accesses	Symmetric, blocks	Hardware

visibilities into the kernel to launch a kernel only once. The visibility data is partitioned block-wise, but to run efficiently, we need at least 100 to 200 thread blocks. With each block writing to a private grid, the size of the grid is limited to only a few MB, which is unfortunate for real astronomy usage. Apart from waiting for GPUs with more memory, this limitation is hard to solve in software. Especially on this platform, where we cannot synchronize write conflicts efficiently. Atomic memory update instructions, as available on some GPUs, don't operate on floating point data and we rejected a locking scheme for the same reason as described in the multi-core CPU sub-section. We cannot send computed grid contributions to host memory instead of to GPU memory, because this turns a data reduction into a data expansion, raising an even worse I/O bottleneck on the PCIe bus.

We also considered abandoning CUDA in favor of OpenGL/GLSL to take advantage of the graphics framebuffer blending hardware of GPUs to atomically update grid values. This approach was rejected, because the performance of blending hardware on 32 bit floating point data is too low.

To improve performance, we only generate aligned and contiguous memory requests to allow the hardware to generate coalesced memory transfers on complex values (float2), boosting memory efficiency. Other minor optimizations we tried include various texturing variants to utilize texture caches, fetching common data for all threads in a block only once and spreading it through shared memory and software prefetching.

For our GPU implementation, we performed no data-dependent optimizations, like having the host CPU fill private queues with jobs that overlap as much as possible. Although this solution may improve data locality, the benefits are limited by the small texture caches. Further, the coordination provided by the host CPU is too slow for the GPU's embedded hardware multi-processor scheduler, which only works efficiently on large amounts of jobs submitted at once. Therefore, we stick to the traditional model, where the CPU works on other (parts of the) application and interrupts occasionally to launch a kernel on a GPU.

A.2.2 Platform comparison

We summarize the platform descriptions for parallelization in Table A.1. An essential part of the performance of data-intensive applications is how each platform type deals with memory latency hiding. Multi-core CPUs hide memory latency with a cache hierarchy and prefetching. This requires temporal (and spatial) data locality. Local memory on the Cell/B.E. has to be managed by software and can hide main memory latency with prefetching (double buffering). Data for the next computation can already be fetched, while computing on the other half of local memory. On the GPU, memory latency is hidden with rapid context switching between many threads.

Table A.2: *Platform characteristics*

Platform instance	Cores	Clock (GHz)	Memory per core (kB)	Main Memory	Peak compute (GFlops)	Memory BW (GB/s)	FLOP/ Byte
Dual Intel Xeon 5320	2x4	1.86	32+32 L1 2x4096 L2	8 GB	60.2	21.3	2.8
Quad AMD Opteron 8212	4x2	2.0	64+64 L1 4x1024 L2	4 GB	64.7	12.5	5.2
Intel Core i7 920	4(HT)	2.66	32+32 L1 4x256 L2+8192 L3	6 GB	42.6	32	1.33
PS3-Cell/B.E.	1+6	3.2	256	256 MB	153.6	25.6	6.0
Cell/B.E.	1+8	3.2	256	1GB	204.8	25.6	8.0
QS20-Cell/B.E.	2+16	3.2	256	2GB	409.6	51.2	8.0
NVIDIA GeForce 8800 GTX	16	1.35	16+8+8	768 MB	346.0	86.4	4.0

A.3 Experiments and Results on multiple platforms

In this section we present the performance of the sequential and parallel versions of the application kernel at various optimization levels and with various parameters.

A.3.1 Experimental Set-up

For each test platform, we present performance graphs for convolution sizes of 16x16, 32x32, 64x64, and 128x128. We also discuss the effect of the optimizations proposed for each platform on both performance and programming effort.

All tests were run under the Linux operating system, using the GNU C compiler for all multi-core CPUs and the Cell/B.E. systems, and using CUDA 2.0 with driver version 177.67 for the GPU. We have used several platform instances, presented in Table A.2.

From the FLOP:Byte ratio values, we can already conclude that for data-intensive applications, we are going to severely under-utilize the computing capabilities of all platforms, unless we can expose significant data reuse in local memories. As expected, the arithmetic intensity variation experiments resulted in little performance difference (5-10%) for values below 2.0.

A.3.2 Experiments with the GPMCs

For the GPMCs, we have three milestone implementations.

The first parallel version (GPMC-Para-1) is the straightforward parallelization, using a symmetrical data distribution of the visibilities stream over the available cores; We implemented it in 4 days using the `pthreads` library. GPMC-Para-1, running on the 8-core machine for the largest convolution kernel (128×128) achieves 7.7 GFLOPs and a 4.4x speed-up versus the base implementation. The speed-up is clearly affected by the data-intensive application behavior.

Version GPMC-Para-2 uses a "queuing-system", where the master thread decides which set of visibilities is assigned to each thread (and, ideally, core). Further, the master assigns adjacent visibilities to cores such that data from the same grid/convolution region can be reused. GPMC-Para-2 took 3 more days to implement, reaching 18 GFLOPs and a 11.25x speed-up.

The third and final version, GPMC-Para-3, we sorted the "local queues" locally (i.e., in-core), thus increasing temporal data locality even more. GPMC-Para-3 took 2 more days to implement. This final version reached 24.8 GFLOPs, and a speed-up of 15.5x when compared with the basic implementation.

Figure A.1 presents the results GPMC-Para-3 has achieved on the different platforms we used for experiments. We note an interesting fact: we have included measurements on three platforms, none of which has more than 8 processors. However, for two of them, enabling more threads than cores seems beneficial technique, leading to visible performance gains. We believe this is the result of both cache effects and OS thread scheduling.

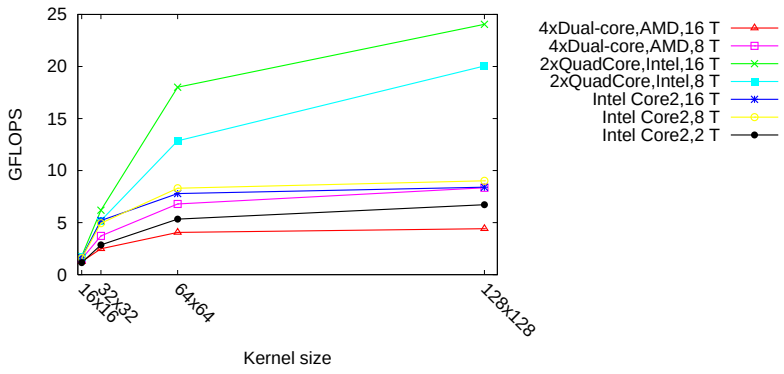


Figure A.1: *The performance (GFLOPS) of parallel gridding (GPMC-Para-3) running on various instances of GPMC platforms, using different numbers of threads.*

A.3.3 Experiments with the Cell/B.E.

Compared with the GPMC and GPU, the Cell/B.E. was a much more difficult target platform for the gridding kernel, mainly due to the large pool of options for both parallelization and optimization. More details on the chosen solutions (and the way we found them) are presented in Chapter 5; we only present here a very brief overview.

We have tried several optimizations, each one leading to more or less surprising results. For example, we have applied the low-level SIMD optimizations and obtained about 30% overall improvement over the non-SIMD code, which is a measure of how low the arithmetic intensity of the application is. Further, double buffering has added some additional 20% to the original code. Our reference Cell/B.E. implementation (the "Cell-Base") already includes the core-level optimizations.

To increase performance further, Cell-Para-1 replaced the original symmetrical data distribution with a dynamic, round-robin one, based on individual SPE queues. The PPE acts as a master and fills the queues with equal chunks of data. Cell-Para-1 (update from the Cell-Base in 10 days) reached 6 GFLOPs and it a 3x speed-up over the Cell-Base implementation.

Following the same idea, in Cell-Para-2, the PPE acts as a smarter dispatcher, and distributes the computation such that the regions used by the SPEs from either the convolution matrix or the grid overlap. Data reuse increased significantly, which significant impact on performance (the amount of DMA transfers decreased). Cell-Para-2 reached 14 GFLOPs and a 7x speed-up over the Cell-Base implementation. However, it took us 25 days to reach these performance levels.

Finally, in Cell-Para-3, each SPE sorts its local copy of the data queue, such that the locality is increased even more. With only 3 more days of work, Cell-Para-3 reached 57 GFLOPs with a speed-up of 23.5x over the reference Cell implementation.

Note that we have applied more aggressive tuning on the Cell/B.E. by using data-dependent optimizations. However, despite the excellent performance outcome (a further improvement by a

factor of 2 in execution time), these optimizations minimize the number of performed operations by using an application-specific feature of the computation. We do not report here these final results[1] because of two reasons: (1) the same optimizations could be envisioned for all platforms, and could lead to similar gains, and (2) because they alter the generic data-intensive behavior we focus on with the other steps.

Figure A.2 presents the Cell-Para-3 results for various kernel sizes when using the Cell/B.E. Note that the application scales with the number of utilized cores.

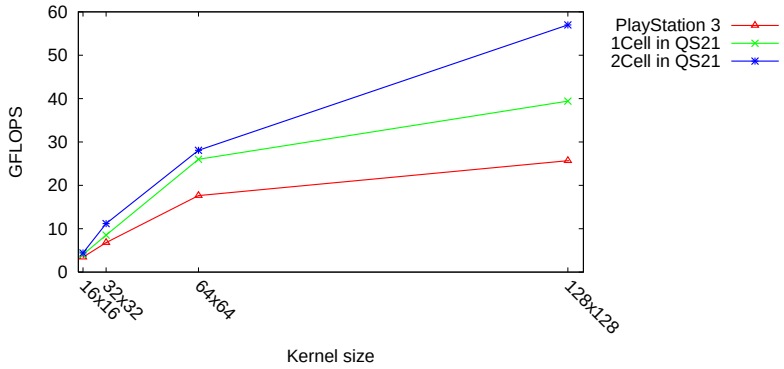


Figure A.2: *The performance (GFLOPS) of gridding running on three Cell/B.E. platforms.*

A.3.4 Experiments with the GPU

For GPUs, the first optimized version, GPU-Para-1 uses the modified CUBLAS routines; additionally, we move the visibilities loop on the GPU. We note that the the effective memory bandwidth peaks at 13.8 GB/s (as measured with the bandwidth test program from the NVIDIA CUDA SDK) out of the theoretical bandwidth of global memory, given at 86.4 GB/s. GPU-Para-1 reaches 4.9 GFLOPs (severily limited by low platform utilization, also visible in the low memory bandwidth utilization). Still, the speed-up over the naive implementation is over 200x. This implementation required 10 days.

After using coalesced memory accesses for version GPU-Para-2, the memory bandwidth reaches 94% of the theoretical peak. According to the NVIDIA CUDA Programming Manual [nVi07], bandwidth of non-coalesced transfers is about four times as low as coalesced transfers on 64 bits requests, which confirms our experiences quite well. Although it would be possible to generate larger transfers using CUDA-specific SIMD data types, performance would only marginally improve, at the expense of increased code complexity. Still, built in 8 days, GPU-Para-2 reaches 22.3 GFLOPs, (with memory bandwidth peaking at 62.2GB/s) and a speed-up of more than 350x. These large speed-ups only demonstrate that the GPU-Base version was performing very poorly, achieving less than 0.1% from the platform peak.

Our final version, GPU-Para-3, combines a few minor optimizations that together deliver under 10% extra performance. An example is the use of the texture caches. The optimizations have a larger impact when tested against non-coalesced memory transfers. However, this would be the wrong optimization order for data-intensive applications. We didn't include here the effect of software prefetching

[1]The final performance results for gridding on the Cell/B.E. are included in Chapter 5.

(i.e., pre-loading the convolution matrix and the grid values for the next processing iteration). The performance gain is only 3%. GPU-Para-3 required 12 additional days (mostly spend on optimizations that only gave marginal improvements, if any) to reach 22.3 GFLOPs, (with memory bandwidth peaking at 66.5GB/s).

Figure A.3 presents the best performance results we have obtained on GPU architectures for various kernel sizes.

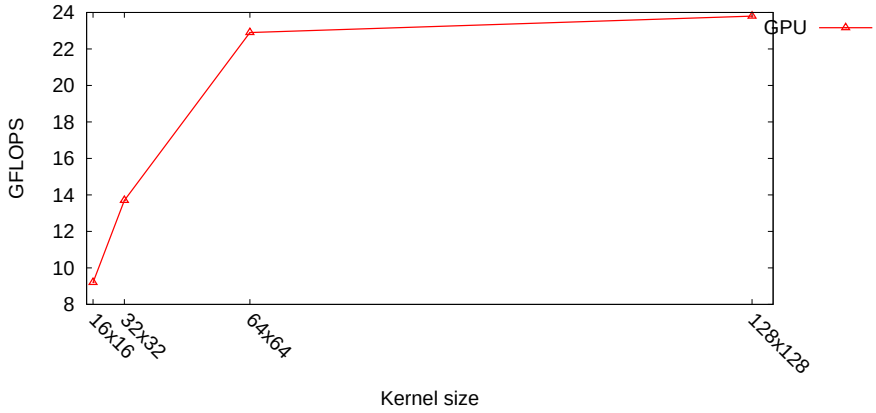


Figure A.3: *The performance (GFLOPS) of gridding when running on the NVIDIA GeForce.*

A.3.5 A Brief Comparison

After implementing data-dependent optimizations, the Cell/B.E. is a lot faster on all kernel sizes. If these optimizations don't apply well to the input data, GPUs score best performance-wise, due to superior memory bandwidth. Although extensive optimizations can also be applied to multi-core CPUs with great relative effect, its performance is still below that of the other platforms. In data-intensive applications, its strength is not the performance-per-core. Note that further experiments

on the latest instances of the three platform types, which provide (1) higher memory throughput on multi-core CPUs (Core i7), (2) higher 64 bit floating point performance on Cell/B.E. (PowerXCell 8i), (3) 64 bit floating point support on GPUs (Geforce GTX 280) are in progress. Further, due to the low computational utilization, early results show little performance decrease when operating on double precision numbers, but detailed results with more diverse arithmetic intensity ratios will follow for the final version the paper.

A.4 Discussion

Although a data-intensive application can be sped up to excellent performance when spending a lot of effort, available memory bandwidth and the ability to efficiently exploit local memory are the main factors in the attained performance-level for a reasonable effort approach. Next, arithmetic intensity and memory access patterns are additional limiting factors for the overall application performance.

Based on our experiments with gridding and degridding on different families of multi-core processors, we have drawn a set of guidelines to make an educated platform choice for implementing data-intensive applications:

- For data-intensive applications, choose a platform with high memory bandwidth and with local memory size large enough to fit more than twice the size of a job.
- Overlapping memory transfers with computation (prefetching) is easier and more efficient to implement on architectures with software-managed local memory, such as Cell/B.E, and a GPU. Applications that have larger job footprints, need the large cache memories of multi-core CPUs, although we expect that higher level caches will be increasingly more distant to access in the next generations.
- GPUs are especially suitable for tasks with a lot of data parallelism. A feature that can be useful for some applications is that caches can be programmed to work for 2D or 3D spatial locality. Synchronization across thread blocks (which should be very limited) is usually performed by letting the kernel finish and launching a new kernel. The overhead for kernel launches is in the order of tens of microseconds, depending on the number of blocks and the size of parameters.
- With the hardware platform market in major turbulence and developments following each other rapidly, the advantage of general-purpose flexibility, availability and ease of programming of GPMCs should not be underestimated.
- With the ongoing trend that computation rate grows faster than memory and I/O bandwidth, more and more applications will become data-intensive. With unfortunately designed applications, all platforms may be a bad choice, unless different algorithms are developed to solve the same problem more efficiently.

Below we give a list of guidelines to assist porting and optimization efforts for data-intensive applications, which is also useful for algorithm design.

- Optimize first for maximum effective memory bandwidth, instead of maximum compute rate. Data (structures) must often be re-outlined (for example, tiled) to make memory accesses contiguous and aligned. Such transformations can force major code overhauls, which is often unavoidable for data-intensive applications. It is the only way to ensure that hardware can apply request optimizations, such as hardware-initiated prefetching, request reordering and coalescing, which has a major effect on performance.
- Only by effectively using local memories, data-intensive applications can perform truly well.
- Irregular access patterns must be resolved as much as possible, usually at a higher level in the hardware hierarchy. In our case, both the multi-core CPU and Cell B.E. profited from job queuing and smart filling.
- If the application kernels contain more than a couple of computations, overlap of memory transfers with computation (prefetching) must be implemented, as in-core optimizations on top of compiler optimizations is often more work than is justified by the pay-off.

Finally, we present an efficiency overview for each platform type for data-intensive applications, shown in Table A.3. Based on all our findings, we conclude that there is not yet a "best-platform" available in the multi-core field and, furthermore, it is still hard to choose such a winner even when for more restrictive application classes.

Table A.3: *Efficiency overview of the experimental results.*

Platform	Performance	Scalability	Cost	Design and Implementation	Efficiency
GPMC	∼	+	++	++	
Cell/B.E.	++	∼	−	−	+
GPU	+	∼	−	∼	∼

A.5 Related Work

In the past two years, all available multi-core processors have been tested in the context of application porting and optimizations. While GPMCs have been mainly targeted by user-level applications, there are still a couple of interesting case-studies that link them to high-performance computing [Con05; Bod07]. The results obtained so far show that previous expertise in SMP machines is very useful, and the new results seem to be coherent. On the other hand, new developments in the homogeneous GPMCs - and mostly the sheer size expected for these platforms (80 cores!) will generate new problems with regard to memory contention and task-parallelism granularity. Therefore, a systematic way to program these platforms at a higher level of abstraction is mandatory for dealing with their next generation [Moo08]. Therefore, most of the software and hardware vendors already provide HPC solutions for multi-core machines - see, for example, HP's multi-core toolkit, Microsoft's HPC Server, or Intel's Thread Building Blocks model. Still, most of these models are very new and not mature enough for fully fledged comparisons. As a result, most programmers still rely on basic multi-threading for HPC applications implementation, making our study an useful information source.

On the Cell/B.E. side, applications like RAxML [Bla07] and Sweep3D [Pet07], have also shown the challenges and results of efficient parallelization of HPC applications on the Cell/B.E. Although we used some of their techniques to optimize the SPE code performance, the higher-level parallelization is usually application specific. Therefore, such ad-hoc solutions can only be used as inspiration for the class of data-intensive applications. Typical ports on the Cell/B.E., like MarCell [Liu07], or real-time ray tracing [Ben06] are very computation intensive, so they do not exhibit the unpredictable data access patterns and the low number of compute operations per data byte we have seen in this application. Irregular memory access applications like list-ranking have been studied in [Bad07a]. Although based on similar tasks decompositions, the scheme proposed in our work is simpler, being more suitable for data-intensive applications.

Another option is to use one of the many programming models available for the Cell/B.E., but most of them are also focused on computation-intensive applications, and do now address the memory hierarchy properly. A notable exception is Sequoia [Fat06], which focuses on the memory system performance, but it is hard to use for irregular parallel applications. Further, although RapidMind [McC07], a model based on data parallelism, provides back-ends for all three platform types, there are no performance evaluations and guidelines for exposing the required (low-level) data parallelism from a data-intensive application. Therefore, we consider a comparison of the results generated by these abstract models with our hand-tuned application a good direction of future work, which would help prepare these programming models to tackle data-intensive applications more efficiently. Note that both these models are fairly portable for all three platforms studied here.

In the case of GPU programming, the NVIDIA CUDA model is widely used, replacing the somewhat more cumbersome "traditional" graphics units programming. As a result, there are many studies of HPC applications running on GPUs [Sto08; Way07; Krg05]. Most of these studies conclude, as well

as we do, that a lot of effort is required to put the application in the right form for being properly exploited with CUDA. These transformations are application specific. Furthermore, our study is the first to expose the additional difficulties that irregular data-intensive applications pose on GPUs. We also note the recent release of the OpenCL 1.0 [com08] specifications, a standard for general purpose parallel programming on heterogeneous systems, with support from all major CPU, GPU, embedded system and gaming vendors.

An interesting overview that covers some of the memory-related problems we have encountered when optimizing a data-intensive application is to be found in [Dre07]. This extensive study treats properties and trade-offs of caches, main memory, NUMA architectures, access patterns and what programmers can do to optimize for them.

Several similar application-oriented comparative studies have been developed to investigate multiple multi-cores [Dat08; Wil08], but these studies are focused on computation intensive applications. Finally, we note the availability of the Roofline model [Wil09a] as a tool to investigate application bounds on multi-core platforms. While the model is very elegant, and useful for many "classical" HPC applications, its applicability for such "nasty" memory and computation patterns as the gridding kernel exposes is still under investigation.

A.6 Summary

After the good results we have obtained with data-intensive applications running on the Cell/B.E. processor, we extended the case-study to cover the other two important multi-core families: general purpose multi-cores (GPMCs) and graphical processing units (GPUs). We have designed and implemented different versions for each platform, and different subversions for major optimization steps. Our results show that porting is possible, with good performance results, but also with high implementation efforts. Furthermore, we have presented several optimization guidelines for data-intensive applications. Finally, we showed that the most significant gains (for data-intensive applications) come from memory and data optimizations, and not from computation optimizations, which typically receive much more attention.

As for choosing a plaform suitable for all data-intensive applications, we don't have a winner. In fact, we show that the choice is not universal, but rather application *and* data specific. We include a list of guidelines to make an educated choice, but no guarantees nor solutions for validating the choice without exhaustive trials (all platforms, all parallelization options). This is, in fact, an important future work direction.

Appendix B

PAMELA for Multi-Cores[‡‡]

The critical part of the multi-core programmability gap is in fact the performance gap: too many applications achieve performance levels far from the specified peak performance of the hardware platforms. While in many cases, the complexity of the hardware requires different design and programming tricks, there are still applications that are just not suitable for some of these platforms. In this context, performance estimation and/or prediction are extremely useful tools for limiting application solution space. They have to be (1) high-level enough to be applied very soon in the design chain, (2) specific enough to allow solution ranking, (3) flexible enough to allow various accuracy levels, and (4) fast enough to allow multiple tests to be performed for every step in the design flow.

Research on performance prediction is not new. Static performance predictors, targeted at (scientific) parallel applications [van03b; Adv93; Adv98] have been very effective for large parallel machines. Based on application and platform modeling, followed by a (partially) symbolic simulation of the actual execution, these methods can only be as accurate as the models are. Traditional parallel systems - such as clusters and supercomputers - are fairly easy to model at the architectural level due to the small number of parallelism layers and the very large difference in performance between them. For multi-cores, the increased parallelism, the complex memory systems, and the relatively small difference between the compute cores and the memory performance make hardware modeling a lot more difficult. Application models follow the same trend.

In this appendix, we discuss how PAMELA [van93], of these model-based performance prediction tools can and/or should be used for multi-core processors. We present the original PAMELA in Section B.1, explaining the modeling and symbolic prediction philosophy.

Further, we present a successful attempt to extend PAMELA for MPSoCs (embedded systems multi-cores): the PAM-SoC predictor (Section B.2). We discuss how the application and machine modeling had to be adapted for the specifics of MPSoCs. We also show a few validation experimental results and two interesting usage scenarios.

However, our further attempts to make PAMELA suitable for multi-cores were less successful. Section B.3 explains briefly what we have tried and where we failed. The lessons we learned are used to sketch research directions for future attempts.

B.1 The PAMELA Methodology

PAMELA (PerformAnce ModEling LAnguage) [van96] is a performance simulation formalism that facilitates symbolic cost modeling, featuring a modeling language, a compiler, and a performance analysis technique. The PAMELA model of a Series-Parallel (SP) program[GE03b] is a set of explicit,

[‡‡]This chapter is partially based on previous work published in the proceedings of Euro-Par 2006 ([Var06c]). The chapter includes a summary of the results included in the above mentioned paper, but also adds several different examples and considerations for multiple architectures.

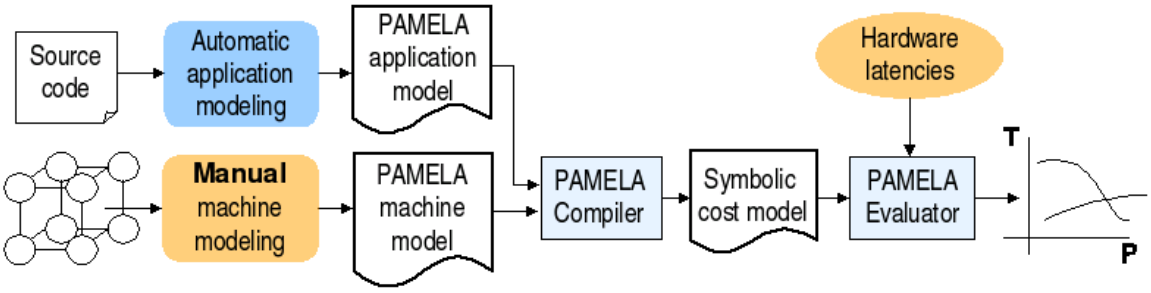


Figure B.1: *The* PAMELA *symbolic cost estimation*

algebraic performance expressions in terms of program parameters (e.g., problem size), and machine parameters (e.g., number of processors). These expressions are automatically *compiled* into a *symbolic cost model*, that can be further reduced and compiled into a *time-domain cost model* and, finally, evaluated into a *time estimate*. Note that PAMELA models trade prediction accuracy for the lowest possible solution complexity. Fig. B.1 presents the PAMELA methodology.

Modeling.

The PAMELA modeling language is a process-oriented language designed to capture concurrency and timing behavior of parallel systems. Data computations from the original source code are modeled into the *application model* in terms of their resource requirements and workload. The available resources and their usage policies are specified by the *machine model*.

Any PAMELA model is written as a set of process equations, composed from `use` and `delay` basic processes, using sequential, parallel, and conditional composition operators. The construct `use(Resource,`t`)` stands for exclusive acquisition of `Resource` for t units of (virtual) time. The construct `delay(`t`)` stalls program execution for t units of (virtual) time. A *machine model* is expressed in terms of available resources and an abstract instruction set (AIS) for using these resources. The *application model* of the parallel program is implemented using an (automated) translator from the source instruction set to the machine AIS. The example below illustrates the modeling of a block-wise parallel addition computation $y = \sum_{i=1}^{N} x_i$ on a machine with `P` processors and shared memory `mem`:

```
// application model:                    // machine model
   par (p=1,P) {                            load=use(mem,t_access)
       seq (i=1,N/P) { load ; add } ;       add=delay(t_add)
       store }
```

Symbolic Compilation and Evaluation.

A PAMELA model is translated into a *time-domain performance model* by substituting every process equation by a numeric equation that models the execution time associated with the original process. The result is a new PAMELA model that only comprises numeric equations, as the original `process` and `resource` equations are no longer present. The PAMELA compiler can further reduce and evaluate this model for different numerical values of the parameters, computing the lower bound of the application execution time. The analytic approach underlying the translation, together with the algebraic reduction engine that drastically optimizes the evaluation time, are detailed in [Gau00].

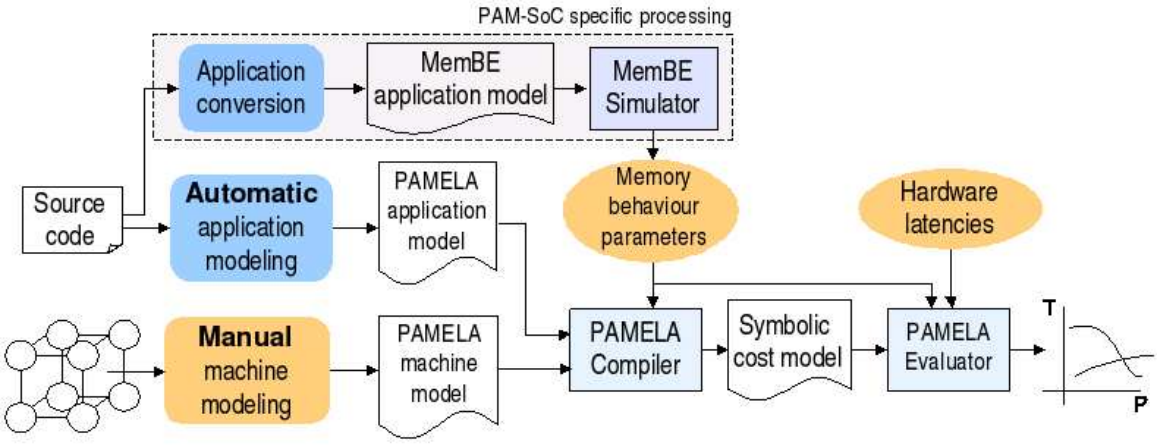


Figure B.2: *The PAM-SoC toolchain*

B.2 PAM-SoC: PAMELA for MPSoCs

Using PAMELA for MPSoC performance predictions is difficult because of the architecture and application modeling efforts required. Details that can be safely ignored for GPMC models, as they do not have a major influence on the overall performance, may have a significant influence on MPSoC behavior. As a consequence, for correct modeling of MPSoC applications and architectures, we have extended PAMELA with new techniques and additional memory behavior tools. The resulting PAM-SoC tool-chain is presented in Fig. B.2. In this section we will further detail its specific components.

B.2.1 MPSoC Machine Modeling

The successful modeling of a machine architecture starts with accurate detection of its important *contention points*, i.e., system resources that may limit performance when used concurrently. Such resources can be modeled at various degrees of detail, i.e., *granularities*, by modeling more or less from their internal sub-resources. The model granularity is an essential parameter in the speed-to-accuracy balance of the prediction: a finer model leads to a more accurate prediction (due to better specification of its contention points), but it is evaluated slower (due to its increased complexity). Thus, a model *granularity boundary* should be established for any architecture, so that the prediction is still fast and sufficiently accurate. This boundary is usually set empirically and/or based on a set of validation experiments.

Previous GPMCs experiments with PAMELA typically used coarse models, based on three types of system resources: the processing units, the communication channels and the memories. For MPSoC platforms, we have established a new, extended set of resources to be included in the machine model, as seen in Fig. B.3. The new granularity boundaries (the leaf-level in the resource tree in Fig. B.3) preserve a good speed-to-accuracy balance, as proved by various experiments we did [Var06b], while allowing drastic simplification of the MPSoC machine modeling procedure. Some additional comments with respect to the machine modeling are the following:

- When on-chip programmable processors have subunits able to work in parallel, they should be modeled separately, especially when analyzing applications that specifically stress them.

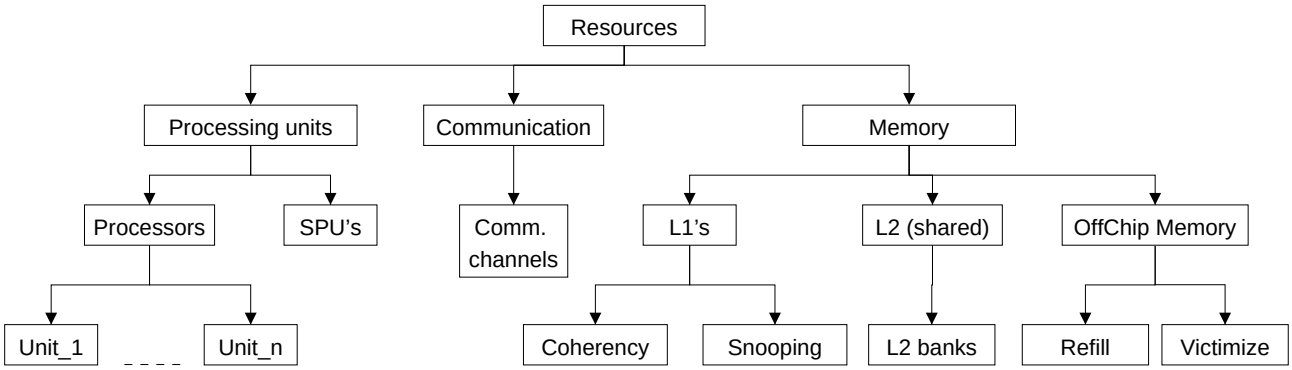


Figure B.3: *The extended set of resources to be included in a PAM-SoC machine model*

- The communication channels require no further detailing for shared-bus architectures. For more complex communication systems, involving networks-on-chips or switch boxes, several channels may be acting in parallel. In this case, they have to be detected and modeled separately.
- The memory system is usually based on individual L1's, an L2 shared cache (maybe banked) and off-chip memory (eventually accessed by dedicated L2 Refill and L2 Victimize engines). If hardware snooping coherency is enforced, two more dedicated modules become of importance: the Snooping and the Coherency engines. Any of these components present in the architecture must be also included in the model.

After identifying model resources, the AIS has to be specified as a set of rules for using these resources, requiring (1) in-depth knowledge on the functionality of the architecture, for detecting the resource accesses an instruction performs, and (2) resource latencies. As an example, Table B.1 presents a snippet from a (possible) AIS, considering an architecture with: several identical programmable processors (`Procs(p)`), each one having parallel arithmetic (`ALU(p)`) and multiplication (`MUL(p)`) units and its own `L1(p)` cache; several Specialized Functional Units (`SFU(s)`); a shared `L2` cache, banked, with dedicated `Refill` and `Victimize` engines; virtually infinite off-chip memory (`mem`).

Table B.1: *Snippet from an AIS for a generic MPSoC*

Operation	**Model**
p: ADD	use(ALU(p), t_{ADD})
p: MUL	use(MUL(p), t_{MUL})
s: EXEC	use(SFU(s), t_{SFU})
p: accessL1(addr)	use(L1(p), $t_{L1}^{hit} * h_{L1}^{ratio} + t_{L1}^{miss} * (1 - h_{L1}^{ratio})$)
p: accessL2(addr)	use(L2(bank(addr)), $t_{L2}^{hit} * h_{L2}^{ratio} + t_{miss} * (1 - h_{L2}^{ratio})$)
p: refillL2(addr)	use(Refill, t_{Mem}^{RD})
p: victimizeL2(addr)	use(Victimize, $victimization^{ratio} * t_{Mem}^{WR}$)

p: READ(addr)	accessL1(addr);
	if (missL1) { accessL2(addr);
	if (missL2) {
	if (victimize) victimizeL2(addr);
	refillL2(addr)}}

The cache hit/miss behavior cannot be evaluated using the cache (directory) state, because PAMELA, being algebraic, is a state-less formalism. Thus, we compute a probabilistic average cache latency, depending on the cache hit ratio, h^{ratio}, and on the hit/miss latencies, t^{hit} and t^{miss}. Also the `if` branches in the `READ(addr)` model are addressed in a probabilistic manner. For example, `if(missL1)` is replaced by a quantification with $(1 - h_{L1}^{ratio})$, which is the probability that this condition is true. All these probabilistic models are based on *memory behavior parameters* which are both application- and architecture-dependent. PAM-SoC uses an additional tool for computing these parameters, which is presented in Section B.2.3.

B.2.2 Application modeling

Translating an application implemented in a high-level programming language to its PAMELA application model (as well as writing a PAMELA model from scratch) implies two distinct phases: (1) modeling the application as a series-parallel graph of processes, and (2) modeling each of the processes in terms of PAMELA machine instructions. However, modeling an existing application to its PAMELA model is a translation from one instruction set to another, and it can be automated if both instruction sets are fully specified as exemplified in [van03b].

B.2.3 The memory statistics

For computing its prediction, PAM-SoC uses two types of numerical parameters: (1) the hardware latencies (measured under no-contention conditions), and (2) the memory statistics. While the former have been also required by GPMC models, the latter become of importance mainly for modeling MPSoC platforms.

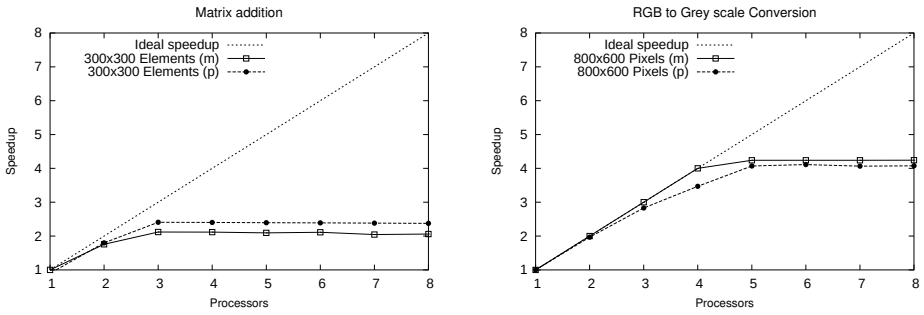
The hardware latencies are fixed values for a given architecture and can be either obtained from the hardware specification itself (i.e., theoretical latencies) or by means of micro-benchmarking (i.e., measured latencies). We have based our experiments on the theoretical latencies.

The memory statistics are both machine- and application-dependent, and they have to be computed/evaluated on a per-application basis. For this, we have built MemBE, a custom light-weight *Memory system Behavior Emulator* able to obtain memory statistics like cache hit ratios, snooping success ratios, or vitimization ratios, with good speed and satisfactory accuracy. MemBE is built as a multi-threaded application that permits the (re)configuration of a custom memory hierarchy using the memory components supported by PAM-SoC. MemBE emulates the memory system of the target architecture and executes a memory-skeleton version of the analyzed application[1]. The memory skeleton is stripped of any data-processing, which allows MemBE to run faster and to focus exclusively on monitoring the application data-path.

B.2.4 Validation experiments

The validation process of PAM-SoC aims to prove its abilities to correctly predict application behavior on a given MPSoC platform. For these experiments, we have modeled the Wasabi platform, one tile of the CAKE architecture from Philips [Str01; Bor05]. A Wasabi chip is a shared-memory MPSoC, having 1-8 programmable processors, several SFUs, and various interfaces. The tile memory hierarchy has three levels: (1) private L1's for each processor, (2) one shared on-chip L2 cache, available to

[1]Currently, the application simplification from the source code to the memory-skeleton is done by hand. In principle, we believe that PAMELA and MemBE can both start from a common, autmatically-generated application model.



(a) (b)

Figure B.4: *Predicted and measured speedup for (a) matrix addition and (b) RGB-to-Grey conversion*

all processors, and (3) one off-chip memory module. Hardware consistency between all L1's and L2 is enforced. For software support, Wasabi runs eCos[2], a modular open-source Real-Time Operating System (RTOS), which has embedded support for multithreading. Programming is done in C/C++, using eCos synchronization system calls and the default eCos thread scheduler.

The simulation experiments have been run on Wasabi's configurable cycle-accurate simulator, provided by Philips. For our experiments, we have chosen a fixed memory configuration (L1's are 256KB, L2 is 2MB, and the off-chip memory is 256MB) and we have used up to 8 identical Trimedia processors[3].

For validation, we have implemented a set of six simple benchmark applications, each of them being a possible component of a more complex, real-life MPSoC application. These applications are: (1) **element-wise matrix addition** - memory intensive, (2) **matrix multiplication** - computation intensive, (3) **RGB-to-YIQ conversion** - a color-space transformation filter, from the EEMBC Consumer suite[4], (4) **RGB-to-Grey conversion** - another color-space transformation, (5) **high-pass Grey filter** - an image convolution filter, from the EEMBC Digital Entertainment suite[5], and (6) **filter chain** - a chain of three filters (YIQ-to-RGB, RGB-to-Grey, high-pass Grey) successively applied on the same input data. All benchmark applications have been implemented for shared-memory (to comply with the Wasabi memory system), using the SP-programming model and exploiting data-parallelism only (no task parallelism, which is a natural choice for the case of these one-task applications).

The results of PAM-SoC prediction and Wasabi simulation for matrix addition and RGB-to-Grey transformation are presented together in Fig. B.4. We have only included these two graphs because these are the applications that clearly exhibit memory contention and therefore show the prediction abilities of PAM-SoC. The complete set of graphs (for all applications) and a complete record of the experimental results are presented in [Var06b].

For all the applications, the behavior trend is correctly predicted by PAM-SoC. The average error between simulation and prediction is within 19%, while the maximum is less than 25% These deviations are due to (1) the differences between the theoretical Wasabi latencies and the ones implemented in the simulator (50-70%), (2) the averaging of the memory behavior data, and (3) the PAMELA accuracy-for-speed trade.

[2] http://ecos.sourceware.org/
[3] TriMedia is a family of Philips VLIW processors optimized for multimedia processing
[4] http://www.eembc.org/benchmark/consumer.asp
[5] http://www.eembc.org/benchmark/digital_entertainment.asp

Table B.2: *Simulation vs. prediction times [s]*

Application	Data size	T_{sim}	T_{MemBE}	T_{Pam}	T_{PAMSoC}	Speed-up
MADD	3x1024x1024 words	94	2	1	3	31.3
MMUL	3x512x512 words	8366	310	2	312	26.8
RGB-to-YIQ	6x1120x840 bytes	90	7	4	11	8.1
RGB-to-Grey	4x1120x840 bytes	62	3	1	4	15.5
Grey-Filter	2x1120x840 bytes	113	6	4	10	11.3
Filter chain	8x1120x840 bytes	347	20	12	32	10.8

While Fig. B.4 demonstrates how PAM-SoC is accurate in terms of application behavior, Table B.2 emphasizes the important speed-up of PAM-SoC prediction time ($T_{PAMSoC} = T_{MemBE} + T_{Pam}$) compared to the cycle-accurate simulation time, T_{sim}, for the considered benchmark applications and the largest data sets we have measured. While a further increase of the data set size leads to a significant increase for T_{sim} (tens of minutes), it has a minor impact on T_{Pam} (seconds) and leads to a moderate increase of T_{MemBE} (tens of seconds up to minutes). Because MemBE is at its first version, there is still much room for improvement, by porting it on a parallel machine and/or by including more aggressive optimizations. In the future, alternative cache simulators or analytical methods (when/if available) may even replace MemBE for computing memory statistics.

B.2.5 Design space exploration

PAM-SoC can be successfully used for early design space exploration, both for *architecture tuning*, where architectural choices effects can be evaluated for a given application, and for *application tuning* where application implementation choices effects can be evaluated for a given architecture.

Architecture tuning.

For *architecture tuning*, we have considered a hypothetical system with up to eight identical programmable processors, each processor having its own L1 cache, a fast shared on-chip L2 cache and practically unlimited external memory (i.e., off-chip). We have modeled three variants of this hypothetical system, named **M1**, **M2**, and **M3**, and we have estimated the performance of a benchmark application (matrix addition, implemented using 1-dim block distribution) on each of them. In this experiment we have chosen to tune the memory configuration of the architecture for these different models, but different processing and/or communication configurations can be evaluated in a similar manner.

M1 is a basic model presented in Fig. B.5(a), being a good starting point for modeling any MPSoC architecture. Its key abstraction is the correct approximation of the off-chip memory latencies - for both `READ` and `WRITE` operations.

M2 is an improved version of **M1**, presented in Fig. B.5(b). It has multiple interleaved banks for the L2 cache (providing concurrent access to addresses that do not belong to the same bank) and buffered memory access to the external memory. Due to these changes, we expect the execution of an application to speed-up on **M2** compared to **M1**. The **M2** model can be adapted to suite any MPSoC architecture with shared banked on-chip memory, if the number of banks, the sizes of the on-chip buffers, and the banking interleaving scheme are adapted for the target machine.

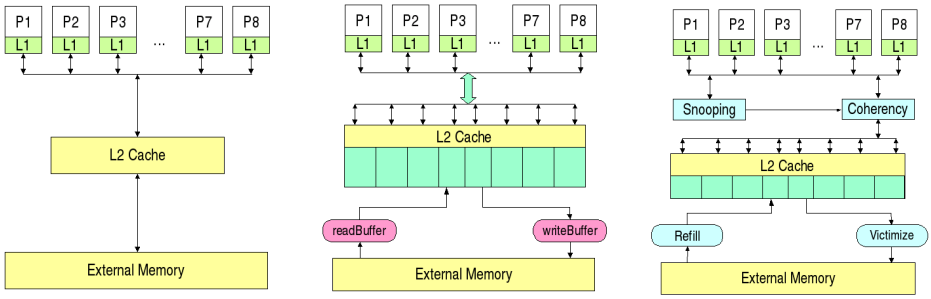


Figure B.5: *The hypothetical MPSoCs: (a) **M1**, (b) **M2**, (c) **M3***

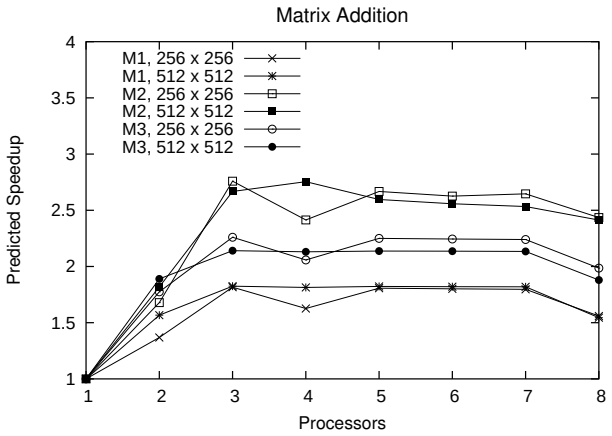


Figure B.6: *Architecture tuning results (predicted)*

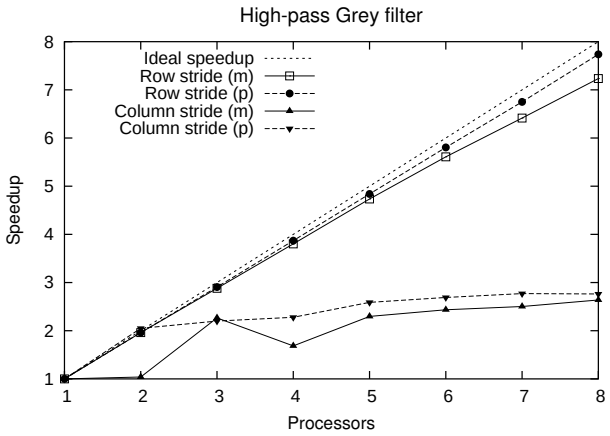


Figure B.7: *Application tuning results (predicted vs. measured)*

M3, presented in Fig. B.5(c), has hardware-enforced *cache coherency*, based on a snooping protocol. The snooping mechanism may increase performance for successful snoopings, when an eventual L1-to-L1 transfer is replacing a slower L2-to-L1 fetch. On the other hand, the L1-to-L2 coherency writes may slow down the application execution. Furthermore, due to the L2-to-Memory transfers performed by the `Victimize` (for `WRITE`) and `Refill` (for `READ`) engines, two new contention points are added. Overall, because matrix addition has almost no successfull snoopings, the application execution on **M3** is slowed down compared to its run on **M2**. **M3** covers the most complex variants of a three-level memory hierarchy to be found in shared memory MPSoCs.

Fig. B.6 shows that PAM-SoC is able to correctly (intuitively) predict the behavior of the given application on these three models. Unfortunately, we could not run validation experiments for the same data sets because these would require cycle-accurate simulators for M1, M2, and M3[6], which are not available. However, previous PAMELA results [van03b] provide ample evidence that PAM-SoC will not predict a wrong relative order in terms of performance: it either correctly identifies the best architecture for the given application, or it cannot distinguish one single best architecture.

[6]Wasabi is a variant of M3, but its simulator is implemented as a combination of M2 and M3

Application tuning

The aim of *application tuning* experiments is to try evaluate several possible implementations of the same application and choose the best one. To prove the use of PAM-SoC for application tuning, we have used the model of the Wasabi platform and we have implemented the high-pass Grey filter mentioned in Section B.2.4 using row and column stride distribution. For this example, based on the PAM-SoC predictions (which are validated by the simulation results), we can decide that row-stride distribution is the best for the Wasabi architecture. Similarly, the experiments like, for example, matrix addition (see Fig. B.4(a)), can detect the maximum number of processors to be used for the computation. Fig. B.7 shows how PAM-SoC correctly detects the application implementation effects on the given architecture. The simulation results that validate the predictions are also included in the graph.

B.3 PAMELA for Multi-Cores

Being able to prove PAM-SoC useful for MPSoC platforms, we further investigated what changes are needed - if any - for enabling PAMELA-like performance prediction for multi-core platforms used for general purpose, as well as for high performance computing. In the following sections we show how the machine models for the three common types of multi-core processors can be built, and discuss their feasability for quick and fairly accurate performance prediction.

B.3.1 Examples of Multi-Core Machine Models

Modeling for PAM-SoC is already more complex than modeling for the original PAMELA due to the increased complexity of the MPSoC platforms. Multi-core processors make this problem even worse: the number and complexity of the cores, the multiple memory layers, and the interconnections require fine-grain modeling if all performance factors are to be properly captured. As examples, we discuss here three PAMELA machine models for the three major multi-core families: the Cell/B.E., GPMCs, and GPUs.

A (generic) GPMC machine model

Listing B.1 presents a GPMC machine model.

Listing B.1: *A GPMC machine model.*

```
1 //a machine model for a GPMC with no hardware multi-threading
2 L1     = cache(P)
3 L2     = cache(P)
4 CORE   = core(P)
5
6 MEM    = memory(1) // main memory, shared
7 L3     = shared_cache(1) // L3 cache, shared
8
9 compute(op) = use(CORE, t_op) //use any core, exclusively
10 reduce(op)  = use(CORE(_all_), t_op) //use all cores
11
12 mem_read(addr, size) = cache_read(L1, addr, size);
13                        cache_read(L2, addr, size);
```

```
14                          cache_read(L3, addr, size);
15                          use(MEM, t_mem_read(size))
16 mem_write(addr, size)= cache_write(L1, addr, size);
17                          cache_write(L2, addr, size);
18                          cache_write(L3, addr, size);
19                          use(MEM, t_mem_write(size))
20
21 cache_read(L, addr, size) = hit(addr)*t_RD_hit(L,size)+(1-hit(addr))*t_RD_miss(L,size)
22 cache_write(L, addr, size) = hit(addr)*t_WR_hit(L,size)+(1-hit(addr))*t_WR_miss(L,size);
23                             update_mem_layers_below()
```

In the proposed model, `hit(addr)` is a boolean operation that returns `1` if the address is available in the cache, and `0` otherwise; `t_op` is the time taken by an operation to be computed on the CORE. Resources under contention are, most likely, the `CORE`s, and the `L3` and `MEM` memories.

The Cell/B.E. machine model

Listing B.2 presents the Cell/B.E. machine model.

Listing B.2: *A Cell/B.E. machine model.*

```
1 SPE      = core(P)
2 LS       = local_memory(P)
3 mailbox = comm(P,N) //each SPE has N mailboxes
4 DMA_ch   = comm(P,M) // each SPE has M DMA channels
5
6 PPE      = core(2);
7 MEM      = memory(1);
8 PPE_L2   = shared_cache(1);
9
10 SPE_compute(op) = use(SPE, t_op);
11 SPE_reduce(op)  = sync_SPE_mail_to_PPE(_all_,request);
12                   async_SPE_DMA_write(_all_,data);
13                   async_SPE_mail_from_PPE(_all_,ack);
14                   PPE_compute(op);
15 PPE_compute(op) = use(PPE, t_op);
16
17 async_SPE_mail_to_PPE(i,data) = use({SPE(i),mailbox(i)}, t_send);
18 sync_SPE_mail_to_PPE(i,data)  = use({SPE(i),mailbox(i),PPE}, (t_send+t_ack));
19
20 async_SPE_mail_from_PPE(i,data) = use({SPE(i),mailbox(i)}, t_recv);
21 sync_SPE_mail_from_PPE(i,data)  = use({SPE(i),mailbox(i),PPE}, (t_recv+t_ack));
22
23 SPE_read(data)  = use({SPE,LS}, t_read(sizeof(data)))
24 SPE_write(data) = use({SPE,LS}, t_write(sizeof(data)))
25
26 async_SPE_DMA_read(i,data)  = use({LS(i),MEM}, t_DMA_read(sizeof(data)))
27 sync_SPE_DMA_read(i,data)  = use({SPE(i),LS(i),MEM}, t_DMA_read(sizeof(data)))
28
29 async_SPE_DMA_write(i,data) = use({LS(i),MEM}, t_DMA_write(sizeof(data)))
30 sync_SPE_DMA_write(i,data) = use({SPE(i),LS(i),MEM}, t_DMA_write(sizeof(data)))
```

Note that the model accounts for the three types of SPE communication: local memory RD/WR operations, mailbox operations (for messaging), and DMA operations (for bulk data read/writes). Also note that, because the Cell/B.E. is a distributed memory archtiecture, the reduce operation is more complex than for the GPMC archtiecture (which uses shared memory). For example, to make a data reduction from all the SPEs, we propose a two-way request/ack protocol, and a separate DMA data transfer. Of course, this is not the only possible solution - one can choose to try different approaches here, extending the model to cover all alternatives.

A GPU machine model

Listing B.3 presents a GPU machine model. Note that we focus here on NVIDIA's GPUs; ATI GPUs require slight modifications in modeling both the memory system and the multi-processors/threads layers.

Listing B.3: *A GPU machine model.*

```
1 HOST_CORES    = core(1)
2 DEVICE_MP     = core(M)   // M microprocessors
3 DEVICE_PE     = core(M,P) // the processing elements, P per microprocessor
4
5 HOST_MEM    = memory(1)
6 DEVICE_MEM = memory(1)
7 REGISTERS  = local_memory(M,R) // R registers per microprocessor
8 SH_MEM      = local_memory(M)
9 CT_MEM      = memory(1)
10 CT_CACHE    = cache(DP)
11 TEXT_MEM    = memory(1) // texture memory
12
13 DEV_compute(op)   = use(DEVICE_PE(_all_), t_op);
14 DEV_reduce(op)    = use({HOST_MEM, DEVICE_MEM}, t_copy); HOST_compute(op)
15 HOST_compute(op) = use(HOST_CORES, t_op)
16
17 HOST_RD(data) = use(HOST_MEM, t_RD(sizeof(data)))
18 HOST_WR(data) = use(HOST_MEM, t_WR(sizeof(data)))
19
20 REG_RD(data)   = use(REG, t_access_REG)
21 REG_WR(data)   = use(REG, t_access_REG)
22
23 DEV_SH_MEM_RD(data) = use(SH_MEM, t_SHMEM_RD)
24 DEV_SH_MEM_WR(data) = use(SH_MEM, t_SHMEM_WR)
25
26 cache_read(L, addr, size) = hit(addr)*t_RD_hit(L,size)+(1-hit(addr))*t_RD_miss(L,size);
27 cache_write(L, addr, size) = hit(addr)*t_RD_hit(L,size)+(1-hit(addr))*t_RD_miss(L,size);
28                             update_mem_layers_below()
29
30 DEV_CT_MEM_RD(data) = cache_read(CT_CACHE, addr, sizeof(data));
31                       use({MEM, CT_CACHE}, t_ct_read(sizeof(data)))
32 DEV_CT_MEM_WR(data) = error()
33
34 DEV_TX_MEM_RD(data) = cache_read(TX_CACHE, addr, sizeof(data));
35                       use({MEM, TX_CACHE}, t_tx_read(sizeof(data)))
36 DEV_TX_MEM_RD(data) = cache_write(TX_CACHE, addr, sizeof(data));
```

```
37                    use({MEM, TX_CACHE}, t_tx_write(sizeof(data)))
```

We assume, for this model, that the host is a sequential machine, with predictable behavior. Thus, we can ignore its memory hierarchy - we consider that the latency of RD/WR operations only depends on the size of the transfer. If more details are needed for the host itself (i.e., if the target architecture features a GPMC and a GPU, for example), a combination of the GPMC and the GPU models can be built. Further, note that we model the GPU as able to only perform in SIMT mode - all threads execute the same instruction.

B.3.2 Discussion

Despite being able to design generic machine models for each of the three architectures, we identify three major problems in applying these models for application performance prediction: computation granularity, memory behavior, and the mapping/scheduling conflicts.

Computation granularity

To denote computation, all three platforms consider a generic operation `compute`, which acquires a computing resource and uses it exclusively for `t_op`, while executing the operation `op`. As computing resources are modeled at the core-level (i.e., coarse granularity, higher than PAM-SoC, which may hide inner-core parallelism), operations should also be considered at the core-level. Thus, `op` is not a simple arithmetic operation, but a sequence of instructions that must have a deterministic execution time (i.e., its execution time can be benchmarked on the core itself). In turn, these complex operations are application specific, requiring some coding for benchmarking. But the coarser the granularity of these operations is, the higher the coding effort is, and the simpler the machine/application models become. Unfortunately, the optimal granularity is both application and platform dependent, and there are no other rules but empirical similarity to determine it (i.e., similar applications should be modeled at the same granularity).

For GPUs, we make one additional comment: this approach is not trivial to apply, because benchmarking is hardly deterministic unless performed at the kernel level. This is due to the hardware scheduler and the unpredictable behavior of data-dependent branches, which lead to thread serialization. Therefore, for codes with branches, benchmarking becomes more complicated, as a probabilistic model of the input data is required to determine the time spent in branching.

The alternative is to capture all core-level operations. This approach leads to a very detailed model. The complexity of such a model not only requires a lot of microbenchmarking, but reflects immediately in the application model, which has to expose *the same level of granularity* for the predictor to models to merge correctly. Modeling an application at such fine granularity becomes equivalent with modeling its assembly code - an operation that takes a lot more time than coding itself. Therefore, we rule out this option.

Memory behavior

For GPMCs predicting the memory behavior is hindered by caches. The same discussion as for PAM-SoC applies here: as PAMELA has no state, maintaining a cache history is not possible (and, even if it would be, searching it would be fairly slow for PAMELA's standards). Thus, evaluating the `hit(addr)` as a boolean function is not really possible in practice.

We can apply the same trick as for PAM-SoC and use statistical modeling to assume an average hit probability. In this case, `hit(addr)` is the probably of hitting in the cache, and it's the same for all addresses. Next, the time taken for the operation itself depends on the hit/miss and the size of data to be read/written. The simplest solution to solve this dependency is to microbenchmark the platform for cache operations with different sizes (the sizes are limited, as RD/WR operations are always performed per cache lines).

For the Cell/B.E., the SPEs have no caches, but we need to separate local and remote memory transfers. However, given the predictable nature of the architecture, benchmarking this communication leads to deterministic results, which can be further used as constants in a high-level model such as the one discussed here.

For GPUs, the different types of memories and caches, as well as the rules and restrictions that they impose on mapping and scheduling (i.e, the scheduler can take different decisions depending on the *number of registers* that a kernel is using) make memory behavior prediction very difficult. Again, we have to chose between a very complex, fine-grain model, and coding. In our experience, coding is faster in most cases.

Mapping/scheduling conflicts

A Pamela application model expresses all available theoretical parallelism, and the engine maps it on the machine resources. Thus, Pamela determines the mapping information from the application model, using the resources and their scheduling policies as given by the machine model.

For multi-core applications, however, mapping and scheduling are much more embedded in the application design and the architecture itself, respectively. This is, again, due to modeling granularity. Choosing a fine-grain model allows Pamela to use the traditional mapping rules, but leads to very complex models, which are too difficult to build. Opting for a coarse-grain model requires some mapping to be done (implicitly, via coding) to allow benchmarking. In this latter case, Pamela's mapping engine becomes obsolete, and a manual mapping has to be coded in the application model.

Furthermore, the traditional scheduling policies, like First Come First Served (FCFS), can no longer be applied by the Pamela prediction engine to core and/or memory resources, as these are managed by other schedulers: the OS in the case of GPMCs, the user in the case of the Cell/B.E., and the hardware scheduler in the case of GPUs. This further limits the usability of the predictor, as the contribution of contention can no be computed without inner knowledge in these schedulers. Despite quite some efforts with various application, we have not been able yet to find a generic way to include this new mapping and scheduling constraints into a fully Pamela compliant chain.

B.4 Summary

Performance prediction is a very useful tool for parallel application development - be it for large, traditional parallel machines, embedded systems multi-cores (MPSoCs), or HPC multi-cores. For the first two classes of platforms, we are able to present working model-based solutions - both using the Pamela methodology. For HPC multi-core processors, however, the use of Pamela is hindered by the complexity of the hardware, the multiple parallelism layers and their combinations, and the fairly small performance differences between various compute and memory units. While we are able to show how both machine and application models can be built empirically, this approach proves to be difficult to generalize and use for real-life applications.

The major issues that drastically limit the use of model-based predictors are: variable computation granularity, memory behavior, and the mapping/scheduling conflicts.

First, modeling the computation granularity for multi-core applications becomes a combination of coding, benchmarking, and abstract modeling, impossible to generalize. Furthermore, the more coding is performed, the simpler (and more accurate) the model becomes. In fact, for many applications for GPMCs and the Cell/B.E., these models become simple enough to be manually evaluated. Still, for GPUs, this modeling approach can become highly inaccurate due to the non-determinism in scheduling and branching. Second, multi-cores include very different memory hierarchies and conflict resolution policies, impossible to generalize in a single, deterministic engine. Finally, mapping and scheduling decisions are now split between the platform, the user/OS, and the application design, prohibiting a clear separation between a generic machine model and an application-only model, to be merged and evaluated by the prediction engine.

We conclude that PAMELA-based generic prediction methodology is not yet feasible - at least with the variety of workloads and platforms we encounter today. While we are able to gather valuable performance information using the model-based approaches, the predictor engine itself requires multiple, platform-specific additions to allow for a semi-automated solution. We leave these modifications for future work.

List of Acronyms

SMP	Symmetrical Multiprocessor
DMA	Direct Memory Access
GPU	Graphical Processing Unit
HPC	High Performance Computing
MPSoC	Multi-processor System-on-Chip
CMP	Chip Multi-Processors
Cell/B.E.	Cell Broadband Engine
RISC	Reduced Instruction Set Computer
DMA	Direct Memory Access
CLP	Core-Level Parallelism
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
SPMD	Single Process Multiple Data
MPMD	Multiple Process Multiple Data
ILP	Instruction Level Parallelism
GPP	General Purpose Processor
MPP	Massively Parallel Processor
GPGPU	General Processing on GPUs
GPMC	General purpose Multi-core
QPI	Intel®Quick Path Interconnect

HT3	AMD®Hyper Transport 3
GB/s	GigaBytes per second
GFLOPS	Giga FLoating Operations per Second
FB-DIMM	Fully Buffered DIMM
TBB	Intel®Threading Building Blocks
IPP	Intel®Integrated Performance Primitives
MKL	Intel®Math Kernel Library
Ct	Intel®C for throughput
ACML	AMD®Core Math Library
SIU	System Interface Unit
TPC	Thread Processing Cluster
SM	Streaming Multiprocessor
TPA	Thread Processor Arrays
CUDA	Compute Unified Device Architecture
SP	Streaming Processor
SIMT	Single Instruction Multiple Threads
ALU	Arithmetical Logical Unit
TMU	Texture Mapping Unit
SFU	Special Function Unit
SMT	Simultaneous Multithreading
PGAS	Partitioned Global Address Space
DAG	Directed Acyclic Graph
FCFS	First Come First Served
SPU	Synergistic Processing Unit
SPE	Synergistic Processing Element
PPU	Power Processing Unit
PPE	Power Processing Element
EIB	Element Interconnection Bus

SCC	Single-Chip Cloud Computer
PS3	PlayStation 3
LS	Local Storage
MFC	Memory Flow Controller
MPB	Message Passsing Buffer
DDR	Double Data Rate
DIMM	Dual In-line Memory Module
PCIe	PCI Express (Peripheral Component Interconnect Express)
DRAM	Dynamic Random Access Memory
GDDR5	Graphical Double Dynamic RAM 5
MIC	Many Integrated Cores
NoC	Network on Chip
ASCI	Accelerated Strategic Computing Initiative
QMF	Quadrature Mirror Filter
LL	Low-Low
LH	Low-High
HL	High-Low
HH	High-High
SVM	Support Vector Machine

Bibliography

[Note] *For the convenience of the reader, each bibliographic entry was appended with a list of back-references to the page(s) where the entry is referenced.*

[Adv93] V. S. Adve. *Analyzing the behavior and performance of parallel programs*. Ph.D. thesis, Dept. of Computer Sciences, University of Wisconsin-Madison, December 1993. 177

[Adv98] V. S. Adve and M. K. Vernon. A deterministic model for parallel program performance evaluation. Tech. Rep. TR98-333, Dept. of Computer Sciences, University of Wisconsin-Madison, 3, 1998. 177

[Adv08] Advanced Micro Devices Corporation (AMD). *AMD Stream Computing User Guide*, august 2008. Revision 1.1. 133

[Aga07] A. Agarwal and M. Levy. Going multicore presents challenges and opportunities. `http://www.embedded.com/columns/technicalinsights/198701652?_requestid=592311`, April 2007. Embedded Systems Design, embedded.com. 163

[Aka93] A. N. Akansu, R. A. Haddad, and H. Caglar. The binomial QMF-wavelet transform for multiresolution signal decomposition. *IEEE Transactions on Signal Processing*, vol. 41(1):pp. 13–22, 1993. 55

[Aka00] A. Akansu and P. Haddad. *Multiresolution Signal Decomposition Transforms, Subbands, and Wavelets*. Elsevier Academic Press, October 2000. 55

[A.L09] A.L.Varbanescu, X. Matorell, R. M. Badia, and H. Sips. Models for the Cell/B.E. an overview with examples. In *CPC '09*. 2009. 8, 115

[Ale77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. 134

[AMD08] AMD. The Framewave project. , 2008. 13

[AMD09] AMD. *ATI Stream Computing Technical Overview*, 2009. 20, 132

[Ame09] A. S. van Amesfoort, A. L. Varbanescu, H. J. Sips, and R. van Nieuwpoort. Evaluating multi-core platforms for hpc data-intensive kernels. In *CF '09*, pp. 207–216. ACM, May 2009. 95, 123, 167

[Ami03] A. Amir, W. Hsu, G. Iyengar, C.-Y. Lin, M. Naphade, A. Natsev, C. Neti, H. J. Nock, J. R. Smith, B. L. Tseng, Y. Wu, and D. Zhang. IBM Research TRECVID-2003 system. In *NIST Text Retrieval (TREC)*. Gaithersburg, MD, November 2003. 52, 108

[Asa09a] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commununication of ACM*, vol. 52(10):pp. 56–67, 2009. 64

[Asa09b] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, vol. 52(10):pp. 56–67, 2009. 163

[Ass08] M. Association. Multicore association creates working group to ease the challenges of software programming for multicore platforms. http://www.multicore-association.org/press/080522.htm, May 2008. 163

[Bad07a] D. Bader, V. Agarwal, K. Madduri, and S. Kang. High performance combinatorial algorithm design on the Cell Broadband Engine Processor. *Parallel Computing*, vol. 33(10–11):pp. 720–740, 2007. 94, 175

[Bad07b] D. A. Bader and V. Agarwal. FFTC: Fastest fourier transform for the ibm cell broadband engine. In *HiPC*, pp. 172–184. 2007. 64

[Bak95] R. S. Baker, C. Asano, and D. Shirley. Implementation of the first-order form of the 3-D discrete ordinates equations on a T3D. Tech. rep., Los Alamos National Laboratory, Los Alamos, New Mexico, 1995. 36

[Bal98] H. E. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, vol. 6:pp. 74–84, July 1998. 64

[Bar10] M. Baron. The single-chip cloud computer. In *Microprocessor*, vol. 1. Microprocessor online, April 2010. 24, 25, 33

[Bec87] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. In *Workshop on Specification and Design*. 1987. 134

[Bel06] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A programming model for the Cell BE architecture. In *SC '06*. IEEE Computer Society Press, November 2006. 66, 113, 126

[Ben06] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the Cell processor. In *IEEE Symposium of Interactive Ray Tracing*, pp. 15–23. IEEE Computer Society Press, September 2006. 24, 64, 66, 93, 113, 175

[Bla06a] N. Blachford. Cell architecture explained version 2 part 3: Programming the Cell. http://www.blachford.info/computer/Cell/Cell3 v2.html, 2006. 107

[Bla06b] N. Blachford. Programming the Cell processor-part 2: Programming models. http://www.blachford.info/computer/articles/CellProgramming2.html, 2006. 106

[Bla07] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. S. Nikolopoulos. RAxML-CELL: Parallel phylogenetic tree construction on the Cell Broadband Engine. In *IPDPS '07*. IEEE Press, March 2007. 51, 64, 66, 93, 175

[Bla08] F. Blagojevic, X. Feng, K. Cameron, and D. S. Nikolopoulos. Modeling multi-grain parallelism on heterogeneous multicore processors: A case study of the Cell BE. In *HiPEAC 2008*. January 2008. 66

[Ble96a] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, vol. 39(3):pp. 85–97, 1996. 130

[Ble96b] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, vol. 39(3):pp. 85–97, 1996. 145

[Bod07] A. Bode. High performance computing in the multi-core area. *Parallel and Distributed Computing, International Symposium on*, vol. 0:p. 4, 2007. 175

[Bor05] D. Borodin. *Optimisation of Multimedia Applications for the Philips Wasabi Multiprocessor System*. Master's thesis, TU Delft, 2005. 10, 181

[Bro06] D. A. Brokenshire. Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance. http://www-128.ibm.com/developerworks/power/library/pa-celltips1/, 2006. 100

[Buc04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM Transactions on Graphics, Proceedings of SIGGRAPH 2004*, pp. 777–786. ACM Press, Los Angeles, California, August 2004. 133

[But07a] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the playstation 3. Tech. Rep. UT-CS-07-595, ICL, University of Tennessee Knoxville, May 2007. 126

[But07b] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. SCOP3: A rough guide to scientific computing on the PlayStation 3. Tech. Rep. UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville, April 2007. 75

[Car99] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Tech. Rep. CCS-TR-99-157, LBNL and UC Berkeley, 1999. 15

[Car06] J. Carter, L. Oliker, and J. Shalf. Performance evaluation of scientific applications on modern parallel vector systems. In *VECPAR*, p. to appear. 2006. 47

[Car07] J. Carter, Y. H. He, J. Shalf, H. Shan, E. Strohmaier, and H. Wasserman. The performance effect of multi-core on scientific applications. Tech. Rep. LBNL-62662, National Energy Research Scientific Computing Center, LBNL, 2007. http://repositories.cdlib.org/lbnl/LBNL-62662/. 163

[Cha00] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. Zpl: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, vol. 26(3):pp. 197–211, 2000. 130

[Cha07] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int J High Perform Comput Appl*, vol. 21(3):pp. 291–312, 2007. 121

[Coa05] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, D. Chavarria-Miranda, F. Cantonnet, T. El-Ghazawi, A. Mohanti, and Y. Yao. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *PPoPP 2005*. June 2005. 15, 48

[Coa06] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with sweep3d implementations in co-array fortran. *The Journal of Supercomputing*, vol. 36(2):pp. 101–121, May 2006. 48

[com08] O. committee. OpenCL 1.0 standard. http://www.khronos.org/opencl/, December 2008. 176

[Con05] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput Archit News*, vol. 33(4):pp. 80–91, 2005. 175

[Cop95] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995. 134

[Cor92] T. J. Cornwell and R. A. Perley. Radio-interferometric imaging of very large fields - The problem of non-coplanar arrays. *Astronomy and Astrophysics*, vol. 261:pp. 353–364, July 1992. 71

[Cor04a] T. Cornwell, K. Golap, and S. Bhatnagar. W-projection: A new algorithm for wide field imaging with radio synthesis arrays. In *Astronomical Data Analysis Software and Systems XIV*, vol. 347, pp. 86–95. ASP Press, October 2004. 71

[Cor04b] T. J. Cornwell. SKA and EVLA computing costs for wide field imaging. *Experimental Astronomy*, vol. 17:pp. 329–343, Jun 2004. 69, 70

[D'A05] B. D'Amora. Online Game Prototype (white paper). http://www.research.ibm.com/cell/whitepapers/cellonlinegame.pdf, May 2005. 24, 113, 164

[Dat08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*. 2008. 176

[Dre07] U. Drepper. What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf, November 2007. 176

[eCo] eCos Home page. http://ecos.sourceware.org/. 11

[Eic05] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the cell processor. In *PACT 2005*, pp. 161–172. IEEE Computer Society, 2005. 38, 124

[Fat06] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *SC*. ACM Press, November 2006. 66, 127, 175

[Fis96] B. Fisher, S. Perkins, A. Walker, and E. Wolfart. Hypermedia image processing reference. `http://www.cee.hw.ac.uk/hipr/html/hipr_top.html`, 1996. 53

[Gau00] H. Gautama and A. van Gemund. Static performance prediction of data-dependent programs. In *Proc. WOSP'00*, pp. 216–226. ACM, Sep 2000. 178

[GE03a] A. Gonzalez-Escribano. *"Synchronization Architecture in Parallel Programming Models",*. Ph.D. thesis, Dept. Informtica, University of Valladolid, 2003. 64

[GE03b] A. Gonzalez-Escribano. *Synchronization Architecture in Parallel Programming Models*. Ph.D. thesis, Dpto. Informatica, University of Valladolid, 2003. 177

[Ged07] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: high performance sorting on the Cell processor. In *VLDB '07*, pp. 1286–1297. VLDB Endowment, 2007. 64

[Geu94] U. Geuder, M. Hardtner, B. Worner, and R. Zink. Scalable execution control of grid-based scientific applications on parallel systems. In *Scalable High-Performance Computing Conference*, pp. 788–795. May 1994. 47

[Ghu07] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, M. So, B. Ghuloum Rajagopalan, Y. Chen, and C. B. Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture. *Intel Technology Journal*, vol. 11(4):pp. 333–348, 2007. 130

[Gsc06] M. Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF*, pp. 1–8. ACM Press, 2006. 65

[Guo04] M. Guo. Automatic parallelization and optimization for irregular scientific applications. *ipdps*, vol. 14:p. 228a, 2004. 47

[Hil08] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, vol. 41:pp. 33–38, 2008. 33, 64

[Hof05] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA'05*, pp. 258–262. IEEE Computer Society, February 2005. 35, 51, 69

[Hoi00a] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications. *International Journal on High Performance Computing Applications*, vol. 14(4):pp. 330–346, 2000. 37

[Hoi00b] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP-2000)*, pp. 21–24. Toronto, Canada, August 2000. 35, 37

[How10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC '10*, pp. 108–109. 7-11 2010. 24

[Hua97a] J. Huang, S. R. Kumar, and M. Mitra. Combining supervised learning with color correlograms for content-based image retrieval. In *MULTIMEDIA 97*, pp. 325–334. ACM, New York, NY, USA, 1997. 54

[Hua97b] J. Huang, S. R. Kumar, M. Mitra, W.-J. Zhu, and R. Zabih. Image indexing using color correlograms. In *CVPR 97*. IEEE Computer Society, Washington, DC, USA, 1997. 53, 110

[Hua98] J. Huang, S. R. Kumar, M. Mitra, and W.-J. Zhu. Spatial color indexing and applications. In *ICCV 98*. IEEE Computer Society, Washington, DC, USA, 1998. 54

[IBM] IBM. *Cell Broadband Engine Programming Handbook*. www-128.ibm.com/developerworks/power/cell/downloads_doc.html. 39

[IBM06] IBM. *Cell Broadband Engine Programming Tutorial*, 2.0 edn., December 2006. 125, 126

[IBM09] IBM. Introducing the next generation of power systems with POWER7. http://www-03.ibm.com/systems/power/news/announcement/20100209_systems.html, 2009. 9

[Ier01] C. S. Ierotheou, S. P. Johnson, P. F. Leggett, M. Cross, E. W. Evans, H. Jin, M. A. Frumkin, and J. Yan. The Automatic Parallelization of Scientific Application codes using a Computer Aided Parallelization Toolkit. *Scientific Programming*, vol. 9(2-3):pp. 163–173, 2001. 39

[Int07] Intel. Measuring application performance on multi-core hardware. Tech. rep., September 2007. 51

[Int09] Intel. Intel Hyper-Threading Technology (Intel HT Technology). http://www.intel.com/technology/ platform-technology/hyper-threading/index.htm, 2009. 2, 11

[Int10a] Intel. SCC platform overview. http://communities.intel.com/docs/DOC-5173, 2010. 24

[Int10b] Intel. SCC programmer's guide. http://communities.intel.com/docs/DOC-5354, 2010. 26

[JaJ92] J. JaJa. *Introduction to Parallel Algorithms*. University of Maryland, 1992. 80

[Kah05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, vol. 49(4/5), 2005. 9, 35, 51, 69

[Kal95a] L. V. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM parallel programming language and system: Part I – description of language features. Tech. Rep. 95-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995. 15

[Kal95b] L. V. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM parallel programming language and system: Part II – the runtime system. Tech. Rep. 95-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995. 15

[Kal96] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pp. 175–213. MIT Press, 1996. 15

[Kan06] D. Kanter. Niagara ii: The Hydra returns. http://www.realworldtech.com/page.cfm?ArticleID=RWT090406012516, April 2006. 14

[Kan08] D. Kanter. Inside Nehalem: Intelś future processor and system. http://www.realworldtech.com/ page.cfm?ArticleID=RWT040208182719, April 2008. 12, 13

[Kan09] D. Kanter. Inside Fermi: NVIDIAś HPC push. http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932, September 2009. 18, 19

[Ker06] D. Kerbyson and A. Hoisie. Analysis of Wavefront Algorithms on Large-scale Two-level Heterogeneous Processing Systems. In *Workshop on Unique Chips and Systems (UCAS2)*. Austin, TX, March 2006. 39

[Keu10] K. Keutzer and T. Mattson. A design pattern language for engineering (parallel) software. Tech. rep., UC Berkeley, 2010. 135

[Khr10] Khronos Group. OpenCL overview. http://www.khronos.org/developers/library/overview/opencl_overview.pdf, June 2010. 139

[Kis06a] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, vol. 26(3):pp. 10–23, 2006. 22, 99

[Kis06b] M. Kistler, M. Perrone, and F. Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, vol. 25(3), May/June 2006. 41

[Krg05] J. Krger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH*. 2005. 175

[Kun06] D. Kunzman. *Charm++ on the Cell Processor*. Master's thesis, Dept. of Computer Science, University of Illinois, 2006. Http://charm.cs.uiuc.edu/papers/KunzmanMSThesis06.shtml. 127

[Kun09] D. M. Kunzman and L. V. Kalé. Towards a framework for abstracting accelerators in parallel applications: experience with cell. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12. ACM, New York, NY, USA, 2009. 127

[Liu07] L.-K. Liu, Q. Liu, A. P. Natsev, K. A. Ross, J. R. Smith, and A. L. Varbanescu. Digital media indexing on the Cell processor. In *IEEE International Conference on Multimedia and Expo*, pp. 1866–1869. July 2007. 7, 8, 51, 53, 66, 93, 123, 175

[Man98] B. S. Manjunath, J. Ohm, V. V. Vasudevan, and A. Yamada. Color and texture descriptors. *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11:pp. 703–715, 1998. 55

[Mat04] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *A Pattern Language for Parallel Programming*. Software Patterns. Addison Wesley, 2004. 135

[Mat10] T. G. Mattson, R. F. V. Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *SC '10*, p. TBD. November 2010. 26

[MC09] J. Mellor-Crummey, L. Adhianto, G. Jin, and W. N. S. III. A new vision for coarray fortran. In *Partitioned Global Address Space Programming Models '09*. October 2009. 15

[McC07] M. McCool. Signal processing and general-purpose computing on GPUs. *IEEE Signal Processing Magazine*, pp. 109–114, May 2007. 94, 175

[McC08] M. McCool. Developing for GPUs, Cell, and multi-core CPUs using a unified programming model. `http://www.linux-mag.com/id/6374`, July 2008. 134

[McK04] S. A. McKee. Reflections on the memory wall. In *CF 2004*, p. 162. 2004. 163

[Min05] B. Minor, G. Fossum, and V. To. Terrain rendering engine (white paper). `http://www.research.ibm.com/cell/whitepapers/TRE.pdf`, May 2005. 65, 113, 164

[Mol09] A. Molnos. *Task Centric Memory Management for an On-Chip Multiprocessor*. Ph.D. thesis, Delft University of Technology, January 2009. 11

[Moo08] S. K. Moore. Multicore is bad news for supercomputers. *IEEE Spectrum*, November 2008. 175

[Mun10] A. Munshi. The OpenCL specification. Tech. rep., KHRONOS Group, June 2010. 135

[Nap02] M. R. Naphade, C.-Y. Lin, and J. R. Smith. Video texture indexing using spatio-temporal wavelets. In *ICIP (2)*, pp. 437–440. 2002. 53, 55, 110

[Nap03] M. R. Naphade and J. R. Smith. A hybrid framework for detecting the semantics of concepts and context. In *CIVR*, pp. 196–205. 2003. 53

[Nat04] A. Natsev, M. Naphade, and J. R. Smith. Semantic space processing of multimedia content. In *ACM SIGKDD 2004*. Seattle, WA, August 2004. 52, 109

[Nat05] A. Natsev, M. Naphade, and J. Tesic. Learning the semantics of multimedia queries and concepts from a small number of examples. In *ACM Multimedia 2005*. Singapore, November 2005. 52

[Nat07] A. Natsev, J. Tesic, L. Xie, R. Yan, and J. R. Smith. IBM multimedia search and retrieval system. In *CIVR*, pp. 645–645. ACM, 2007. 52

[Nij07] M. Nijhuis, H. Bos, and H. E. Bal. A component-based coordination language for efficient reconfigurable streaming applications. In *ICPP*, pp. 60–72. 2007. 128

[Nij09] M. Nijhuis, H. Bos, H. E. Bal, and C. Augonnet. Mapping and synchronizing streaming applications on Cell processors. In *HiPEAC*, pp. 216–230. 2009. 128

[nVi07] nVidia. *CUDA-Compute Unified Device Architecture Programming Guide*, 2007. 94, 168, 172

[NVI10] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi. http://www.nvidia.com/content/PDF/ fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010. 9

[O'B07] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on the Cell. In *International Workshop on OpenMP*, pp. 65–76. Springer, June 2007. 94

[Oha06a] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming th Cell Broadband Engine processor. *IBM Systems Journal*, vol. 45(1):pp. 85–102, January 2006. 94

[Oha06b] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, vol. 45(1), March 2006. 112

[Oli04] L. Oliker, R. Biswas, J. Borrill, A. Canning, J. Carter, M. J. Djomehri, H. Shan, and D. Skinner. A performance evaluation of the cray x1 for scientific applications. In *VECPAR*, pp. 51–65. 2004. 47

[Ora08] Oracle/Sun Microsystems. UltraSPARC T2 and T2 Plus processors. http://www.sun.com/ processors/ UltraSPARC-T2/index.xml, 2008. 14

[Pet07] F. Petrini, J. Fernàndez, M. Kistler, G. Fossum, A. L. Varbanescu, and M. Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, March 2007. 7, 35, 51, 64, 65, 66, 69, 93, 107, 113, 123, 175

[Phi04] Philips Semiconductors. *Programmable Media Processor: TriMedia TM-1300*, 2004. http://www.datasheetcatalog.org/datasheet/philips/PTM1300.pdf. 10

[Por10] The Portland Group. *PGI Fortran & C Accelerator Programming Model white paper*, version 1.2 edn., March 2010. http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.2.pdf. 133

[Pra07] M. Prange, W. Bailey, H. Djikpesse, B. Couet, A. Mamonov, and V. Druskin. Optimal gridding: A fast proxy for large reservoir simulations. In *SPE/EAGE International Conference on Reservoir Characterization*, pp. 172–184. 2007. 92

[Ren97] R. Rengelink, Y. Tang, A. d. Bruyn, G. Miley, M. Bremer, H. Rttgering, and M. Bremer. The Westerbork Northern Sky Survey (WENSS), a 570 square degree mini-survey around the North ecliptic pole. *Astronomy and Astrophysics Supplement Series*, vol. 124:pp. 259 – 280, 1997. 71

[Ros98] D. Rosenfeld. An optimal and efficient new gridding algorithm using singular value decomposition. *Magnetic Resonance in Medicine*, vol. 40(1):pp. 14–23, 1998. 92

[R.T07] R.T.Schilizzi, P.Alexander, J.M.Cordes, P.E.Dewdney, R.D.Ekers, A.J.Faulkner, B.M.Gaensler, P.J.Hall, J.L.Jonas, and K. I. Kellermann. Preliminary specifications for the square kilometre array. Tech. Rep. v2.4, www.skatelescope.org, November 2007. 69, 70

[Sch84] F. Schwab. Optimal gridding of visibility data in radio interferometry. In *Indirect Imaging*, pp. 333–340. Cambridge University Press, 1984. 72

[Sch95] H. Schomberg and J. Timmer. The gridding method for image reconstruction by Fourier transformation. *IEEE Transactions on Medical Imaging*, vol. 14(3):pp. 596–607, Sep 1995. 92

[Sch04] K. van der Schaaf, C. Broekema, G. van Diepen, and E. van Meijeren. the lofar central processing facility architecture. *Experimental Astronomy, special issue on SKA*, vol. 17:pp. 43–58, 2004. 69, 70

[Sei08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans Graph*, vol. 27(3):pp. 1–15, 2008. 20, 21

[Smi96] J. R. Smith and S.-F. Chang. Tools and techniques for color image retrieval. In I. K. Sethi and R. C. Jain, editors, *SPIE '96*, vol. 2670, pp. 426–437. March 1996. 53, 54, 109

[Smi05] M. C. Smith, J. S. Vetter, and X. Liang. Accelerating scientific applications with the src-6 reconfigurable computer: Methodologies and analysis. *ipdps*, vol. 04:p. 157b, 2005. 47

[Sou07] L. de Souza, J. D. Bunton, D. Campbell-Wilson, R. J. Cappallo, and B. Kincaid. A radio astronomy correlator optimized for the Xilinx Virtex-4 SX FPGA. In *International Conference on Field Programmable Logic and Applications*. 2007. 94

[ST00] J. Shawe-Taylor and N. Cristianini. *Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000. 52

[Sto08] S. S. Stone, J. P. Haldar, S. C. Tsao, W. m. W. Hwu, B. P. Sutton, and Z. P. Liang. Accelerating advanced mri reconstructions on gpus. *J Parallel Distrib Comput*, vol. 68(10):pp. 1307–1318, 2008. 175

[Str01] P. Stravers and J. Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *Proc. VLSI-TSA'01*. April 2001. 10, 181

[Sun08] Sun Microsystems. Sun Studio. http://developers.sun.com/sunstudio/features/index.jsp, 2008. 15

[Tho01] A. Thompson, J. Moran, and G. Swenson. *Interferometry and Synthesis in Radio Astronomy*. Wiley-Interscience, 2001. 70

[van93] A. van Gemund. Performance prediction of parallel processing systems: The PAMELA methodology. In *International Conference on Supercomputing*, pp. 318–327. 1993. 8, 75, 177

[van96] A. van Gemund. *Performance Modeling of Parallel Systems*. Ph.D. thesis, Delft University of Technology, April 1996. 177

[van03a] A. van Gemund. Symbolic performance modeling of data parallel programs. In *Proc. CPC'03*, pp. 299–310. January 2003. 147

[van03b] A. van Gemund. Symbolic performance modeling of parallel systems. *IEEE TPDS*, vol. 14(2):pp. 154–165, February 2003. 148, 177, 181, 184

[Var06a] A. Varbanescu, M. Nijhuis, A. Gonzalez-Escribano, H. Sips, H. Bos, and H. Bal. SP@CE - an SP-based programming model for consumer electronics streaming applications. In *LCPC 2006*, LNCS 4382. Springer, nov 2006. 127

[Var06b] A. L. Varbanescu. PAM-SoC experiments and results. Tech. Rep. PDS-2006-001, Delft University of Technology, 2006. 179, 182

[Var06c] A. L. Varbanescu, H. J. Sips, and A. v. Gemund. Pam-soc: A toolchain for predicting mpsoc performance. In *Euro-Par*, pp. 111–123. 2006. 148, 177

[Var07] A. L. Varbanescu, H. J. Sips, K. A. Ross, Q. Liu, L.-K. Liu, A. Natsev, and J. R. Smith. An effective strategy for porting c++ applications on cell. In *ICPP '07*, p. 59. IEEE Computer Society, 2007. 51, 66, 97, 123

[Var08a] A. L. Varbanescu, A. van Amesfoort, T. Cornwell, B. G. Elmegreen, R. van Nieuwpoort, G. van Diepen, and H. Sips. The performance of gridding/degridding on the Cell/B.E. Tech. rep., Delft University of Technology, January 2008. 84, 85

[Var08b] A. L. Varbanescu, A. van Amesfoort, T. Cornwell, B. G. Elmegreen, R. van Nieuwpoort, G. van Diepen, and H. Sips. Radioastronomy image synthesis on the Cell/B.E. Tech. rep., Delft University of Technology, August 2008. 8, 69, 84

[Var09a] A. L. Varbanescu, A. S. van Amesfoort, T. Cornwell, G. van Diepen, R. van Nieuwpoort, B. G. Elmegreen, and H. J. Sips. Building high-resolution sky images using the Cell/B.E. *Scientific Programming*, vol. 17(1-2):pp. 113–134, 2009. 8, 69, 123

[Var09b] A. L. Varbanescu, H. J. Sips, K. A. Ross, Q. Liu, A. Natsev, J. R. Smith, and L.-K. Liu. Evaluating application mapping scenarios on the Cell/B.E. *Concurrency and Computation: Practice and Experience*, vol. 21(1):pp. 85–100, 2009. 7, 51

[Wan00] Y. Wang, Z. Liu, and J.-C. Huang. Multimedia content analysis using both audio and visual clues. *IEEE Signal Processing Magazine*, vol. 17(6):pp. 12–36, November 2000. 52

[Way07] R. Wayth, K. Dale, L. Greenhill, D. Mitchell, S. Ord, and H. Pfister. Real-time calibration and imaging for the MWA (poster). In *AstroGPU*. November 2007. 94, 175

[Wik08] Wikipedia. IBM RoadRunner. `http://en.wikipedia.org/wiki/IBM_Roadrunner`, 2008. 24

[Wil06] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientic computing. In *ACM International Conference on Computing Frontiers*, pp. 9–20. ACM Press, 2006. 24, 35, 64, 163

[Wil08] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. A. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *IPDPS*, pp. 1–14. 2008. 176

[Wil09a] S. Williams, A. Waterman, and D. Patterson. "roofline: An insightful visual performance model for floating-point programs and multicore architectures". *Communications of the ACM (CACM)*, p. to appear, April 2009. 176

[Wil09b] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, vol. 52(4):pp. 65–76, 2009. 65

[Wil09c] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, vol. 52(4):pp. 65–76, 2009. 149

[Wol10] M. Wolfe. Compilers and more: Knights Ferry versus Fermi. `http://www.hpcwire.com/features/Compilers-and-More-Knights-Ferry-v-Fermi-100051864.html`, august 2010. 21, 22, 33

[Zhe10] Y. Zheng, F. Blagojevic, D. Bonachea, P. H. Hargrove, S. Hofmeyr, C. Iancu, S.-J. Min, and K. Yelick. Getting multicore performance with UPC. In *SIAM Conference on Parallel Processing for Scientific Computing*. February 2010. 15

Summary

On the Effective Parallel Programming of Multi-core Processors

Until the early 2000s, better performance was simply achieved by waiting for a new generation of processors, which, having higher operational frequency, would run the same applications faster. When the technological limitations of this solution (e.g., power consumption and heat dissipation) became visible, multi-core processors emerged as an alternative approach: instead of a single, large, very complex, power hungry core per processor, multiple "simplified" cores were tightly coupled in a single chip and, by running concurrently, they yielded better overall performance than the equivalent single-core.

Multi-cores are now designed and developed by all the important players on the processors market. In only five years, this competition has generated a broad range of multi-core architectures. Major differences between them include the types and numbers of cores, the memory hierarchy, the communication infrastructure, and the parallelism models.

But as multi-core processors have become state-of-the-art in computing systems, applications have to make use of massive parallelism and concurrency to achieve the performance these chips can offer. A failure to do so would make multi-cores highly inefficient and, ultimately, application performance would stagnate or even decrease.

Despite the very large collection of (traditional) parallel programming models, languages, and compilers, the software community is still lagging behind: the applications, the operating systems, and the programming tools/environments are not yet coping with the increased parallelism requirements posed by this new generation of hardware. Furthermore, the newly developed, multi-core specific programming models lack generality, infer unacceptable performance penalties, and pose severe usability limitations.

In fact, multi-core programmers base application design and implementation on previous experience, hacking, patching, and improvisation. Therefore, we claim that *the state-of-the-art in programming multi-cores is no more than a "needle-in-the-haystack" search, being driven by programmer experience and ad-hoc empirical proves. This approach hinders performance, productivity, and portability. In this thesis, we investigate whether a structured approach to parallel programming for multi-cores is possible and useful, as well as what conditions, actions, and tools are necessary to transform such an approach into a complete framework able to improve the design and development of parallel applications for multi-core processors.*

Our work focuses on defining a step-by-step strategy for effective multi-core application development. This strategy is supported by the **MAP** framework, which guides the user through the phases and decisions required in the application design and implementation. Specifically, MAP takes as input an *application specification* and a *set of performance targets*, and allows the programmer to use a step-by-step approach to analyze, design, prototype, implement, and optimize a multi-core application. The framework enables a user-controlled mix of application-driven and hardware-driven design and development, by enabling and disabling various platform-specific and application-specific

shortcuts. A performance feedback loop, also part of the framework, provides an effective roll-back mechanism, based on performance checkpoints, which allow the user to return to a previous design stage and take a different path without sacrificing the entire design flow.

This thesis is split into four logical parts: we discuss multi-core processors in detail, we present a few representative application case-studies, we evaluate the current status of multi-core programmability, and we discuss the MAP framework.

An investigation of state-of-the-art multi-cores is presented in Chapter 2, where we show the diversity of the multi-core processors landscape when it comes to cores, memory systems, and communication infrastructures. However, all these platforms do share three main features: (1) they offer very high (theoretical) performance (hundreds to thousands of GFLOPs), (2) they have hierarchical parallelism (between two and five distinct layers), and (3) the only way for applications to get close to the theoretical peak performance is to expose enough concurrency to match each parallelism layer of the hardware platform.

Next, we focus on three different case-studies (Chapters 3, 4, 5), aiming to determine the major challenges that arise in application development, from design to implementation and optimization; we also give a sample of the practical challenges of porting legacy code onto multi-cores (Chapter 6).

Our first case-study (Chapter 3) is a classic high-performance computing (HPC) application (Sweep3D) to be ported onto a heterogeneous multi-core processor (the Cell/B.E.). This case-study emphasizes the transition from the traditional model, where parallelism was typically expressed in a single dimension (i.e., local versus remote communication, or scalar versus vector code), to the complex, multi-dimensional parallelization space of multi-core processors, where multiple levels of parallelism *must be* exploited in order to gain the expected performance.

Our Sweep3D implementation proves that exploiting all the available parallelism layers of a multi-core does provide high performance for real applications (we achieved over 20x speed-up over the original implementation). Furthermore, we have exposed the limiting factor for the application performance, which is not the computational capabilities of the Cell/B.E., but its memory bandwidth. Finally, the Sweep3D case study proved that programming complex multi-cores like the Cell/B.E. is significantly simplified if done systematically, by matching the multiple layers of hardware parallelism with corresponding layers of application concurrency.

The second case-study (Chapter 4) focuses on a multi-kernel application. We claim that for such applications, low-level parallelization and core-level optimizations must be augmented by application-level optimizations, which are likely to provide a significant performance boost. We analyze a concrete case in MarCell, the Cell/B.E. version of MARVEL, a multimedia content analysis application with five computational kernels. For MarCell, we show that by combining aggressive core-level optimizations, we gain an overall speed-up above 10x, when only using the SPEs sequentially. On top of these optimizations, we have added application-level parallelization, using task parallelization (by firing different tasks in parallel - an MPMD model) and data parallelization (by using multiple SPEs to compute a single kernel faster - an SPMD model). Results show that task parallelism performs better, leading to a 15x improvement over the original code. Finally, we have tested a hybrid solution, which enables both data and task parallelism, and were able to increase the overall application speed-up to over 20x.

Our third and final case-study (Chapter 5) focuses on data-intensive applications. These applications are not naturally suitable for multi-core processors: their focus on data rather than computation does not make good use of the computing capacity of these architectures and, in turn, leads to low per-

formance. Using gridding and degridding (two data-intensive kernels often found in signal-processing pipelines) we show several generic algorithmic changes and optimizations that can compensate the gap between the required bandwidth and the communication capabilities of the platform. Furthermore, besides the application-level parallelization and the low-level optimizations (already seen in the previous case-studies), our case-study exposed the performance-boosting effect of data-dependent optimizations. Effectively, we show that application behavior has to be studied taking into account the structure of the I/O data, which in turn can be exploited to increase performance.

Our case-studies exposed a lot of challenges we had to solve manually. In Chapter 7, we investigate if any of the modern programming tools, mostly designed to support multi-cores, offers the right mix of features to help the user solve such challenges effectively. Our survey, including twenty programming models and tools, exposes two disjoint classes of models - application-centric and hardware-centric. This split widens the programmability gap, as none of the available programming models can provide *all* the needed support for the complete design and development of multi-core applications.

Our solution for bridging this programmability gap is a framework that enables a complete analysis-to-implementation strategy for parallel programming of multi-core applications. Thus, Chapter 8 explains our step-by-step strategy and describes the MAP framework prototype. To determine the applicability of our solution, we investigate the effort of building a potential support tool-chain. We find that despite the completeness of the solution, the state-of-the-art tool support is not yet sufficient. Therefore, we show what are the missing links and give guidelines for building them.

Overall, we conclude (in Chapter 9) that effective parallel programming of multi-core processors requires a systematic approach to application specification (including the analysis of the performance requirements and I/O data-sets), design, implementation, optimization, and performance feedback. However, we are not there yet: high-level general solutions for each of these problems are hardly supported by the available tools. Therefore, our future research directions focus on support for application specification and analysis, performance analysis and prediction, automated parallelization strategies, and, ultimately, platform-to-application matching.

Samenvatting

Effectief Parallel Programmeren van Multi-core Processoren

Tot grofweg 2002 werden betere prestaties eenvoudig bereikt door te wachten op een nieuwe generatie processoren, die met een hogere clock frequentie dezelfde taken sneller zouden uitvoeren. Toen de limieten van deze technologie (zoals het hoge stroomverbruik en warmteafgifte) zichtbaar werden, zijn multi-core processoren naar voren gekomen als een alternatieve oplossing. In plaats van een enkelvoudige, grote, zeer complexe, en energieslurpende core per chip, werden meerdere vereenvoudigde cores in een enkele chip geplaatst en door het gebruik van parallelisme zijn nu betere algemene prestaties mogelijk dan in een gelijkwaardige single-core oplossing.

Multi-cores worden nu ontworpen en ontwikkeld door alle belangrijke spelers op de processor markt. In slechts vijf jaar heeft deze competitie geleid tot veel verschillende multi-core architecturen. De grote verschillen binnen deze architecturen zijn de soort en het aantal cores, de geheugen hiërarchie, de communicatie-infrastructuur, en de modellen van parallelleverwerking.

Aangezien de multi-core processoren de state-of-the-art technologie in computersystemen zijn geworden, moeten applicaties derhalve parallellisme gebruiken om de prestaties die deze chips kunnen bieden ook echt te bereiken. Als dit niet gebeurt maakt dit multi-cores zeer inefficiënt en zullen uiteindelijk de prestaties van applicaties stagneren of zelfs teruglopen.

Ondanks de zeer grote collectie van (traditionele) parallelle programmeer modellen, talen en compilers, heeft de software gemeenschap nog steeds een grote achterstand: de applicaties, besturingssystemen en programmeer tools kunnen nog niet goed omgaan met eisen ter aanzien van parallellisme van deze nieuwe generatie hardware componenten. Nog erger, de nieuwe multi-core specifieke programmeer modellen zijn niet algemeen toepasbaar en leiden tot verminderde prestaties en/of ernstige gebruikbaarheidsbeperkingen.

In werkelijkheid, baseren multi-core programmeurs nog steeds hun applicatie ontwerp en uitvoering op hun eerdere ervaringen, hacking, patching en improvisatie. Daarom stellen wij dat de state-of-the-art in de multi-core programmeren niet meer is dan het zoeken van een “naald-in-de-hooiberg”, gedreven door programmeur ervaring en ad-hoc empirische keuzes. Deze aanpak belemmert optimale prestaties, productiviteit en toepasbaarheid van allerlei applicaties, en maakt in feite de programmeerbaarheidskloof groter. In dit proefschrift onderzoeken we of er een gestructureerde aanpak van parallel programmeren voor multi-cores mogelijk en zinvol is, alsmede welke voorwaarden, acties en tools nodig zijn om deze aanpak te transformeren in een compleet raamwerk dat het ontwerp en de ontwikkeling van parallelle applicaties voor multi-core processors kan verbeteren.

Ons werk richt zich op het bouwen van een stap-voor-stap strategie voor een effectieve multi-

core applicatie ontwikkeling. Deze strategie wordt ondersteund door **MAP**, een volledig raamwerk dat de gebruiker leidt door de verschillende fasen en beslissingen die zijn vereist voor het ontwerp en de ontwikkeling van de applicatie. MAP heeft als input een applicatie specificatie en een set van prestatie-doelstellingen en begeleidt de programmeur om stap-voor-stap een multi-core applicatie te analyseren, ontwerpen, implementeren en optimaliseren. MAP maakt een gebruikersgecontroleerde mix van applicatie-gedreven en hardware-gedreven ontwerp en ontwikkeling mogelijk door het in- en uit-schakelen van verschillende platform-specifieke en applicatie-specifieke snelkoppelingen. Een prestatie terugkoppeling, ook deel van het raamwerk, biedt een effectief roll-back mechanisme, gebaseerd op prestatie controlepunten, die de gebruiker toestaat om terug te keren naar een eerdere ontwerpfase en een ander pad te nemen zonder dat de gehele ontwerp-flow wordt vernietigd.

De inhoud van dit proefschrift bestaat uit vier delen: we behandelen in detail de architecturen van multi-core processoren, analyseren drie representatieve applicaties middels case-studies, evalueren de huidige status van multi-core programmeerbaarheid, en bespreken het MAP raamwerk.

Een onderzoek naar state-of-the-art multi-core architecturen wordt gepresenteerd in Hoofdstuk 2, waar we de diversiteit in het multi-core processor landschap laten zien. Deze diversiteit betreft vooral de cores zelf, de geheugen-systemen en de communicatie-infrastructure. Deze platformen hebben drie belangrijke dingen gemeen: (1) ze bieden zeer hoge prestaties (honderden tot duizenden GFLOPs), (2) ze gebruiken een hiërarchie van parallellisme (twee tot vijf verschillende lagen), en (3) de enige manier voor applicaties om dicht bij de theoretische piekprestaties te komen is om in de software op elke hardware laag voldoende concurrency te laten zien.

Vervolgens, richten we onze focus op drie verschillende case-studies (Hoofdstukken 3, 4, 5), gericht op het bepalen van de grote uitdagingen die zich voordoen in de ontwikkeling van applicaties, van ontwerp tot implementatie en optimalisatie; we geven ook een voorbeeld van de praktische uitdagingen bij het porten van legacy-code naar multi-core systemen (Hoofdstuk 6).

Onze eerste case-studie (Hoofdstuk 3) is een een klassieke high-performance computing (HPC) applicatie (Sweep3D) overzetten naar een heterogene multi-core processor (de Cell/B.E.). Deze case-studie benadrukt de overgang van het traditionele model, waar parallellisme meestal wordt uitgedrukt in een enkele dimensie (zoals lokaal versus communicatie op afstand, of scalaire versus vector code), naar de complexe multi-dimensionale parallellisatie van multi-core processoren, waarbij meerdere lagen van parallellisme moeten worden benut om de verwachte prestaties te krijgen.

Onze Sweep3D implementatie bewijst dat het benutten van alle beschikbare lagen van parallellisme van een multi-core leidt tot hoge prestaties (we hebben een factor 20 versnelling bereiken ten opzichte van de originele implementatie). Verder hebben we gevonden dat de beperkende factor voor de prestatie van deze applicatie niet de berekeningsmogelijkheden van de Cell/B.E. zijn, maar de geheugen bandbreedte. Ten slotte heeft de Sweep3D case studie bewezen dat het programmeren van complexe multi-cores, zoals de Cell/B.E. aanzienlijk vereenvoudigd wordt indien dit systematisch wordt gedaan door het afstemmen van de verschillende lagen van hardware parallellisme met overeenkomstige lagen van de applicatie concurrency. Bij een correcte toepassing strekt deze correspondentie zich uit tot de prestaties: hoe meer concurrency een applicatie kan gebruiken, hoe meer zijn prestaties het maximum van het platform benaderen.

De tweede case-studie (Hoofdstuk 4) richt zich op een multi-kernel applicatie. We tonen aan dat voor dergelijke applicaties, low-level parallellisatie en core-level optimalisaties veelal moeten worden aangevuld met applicatie-niveau optimalisaties, die voor een significante prestatieverbetering zorgen. We analyseren een concrete multi-core applicatie genoemd MarCell, welke de de Cell/B.E. versie van MARVEL is. MarCell is een multimedia-content analyse applicatie met vijf computationele kernels.

Voor MarCell laten we zien dat het combineren van agressieve core-level optimalisaties leidt tot een versnelling van meer dan 10x door alleen al de cores sequentieel te gebruiken. Naast deze optimalisatie, hebben we applicatie-niveau parallellisatie toegevoegd, met behulp van taak parallellisatie (door verschillende taken parallel te starten - een MPMD model) en gegevens parallellisatie (door meerdere cores een enkele kernel sneller te laten berekenen - een SPMD model). Resultaten laten zien dat taak parallellisme beter presteert met een 15x verbetering ten opzichte van de oorspronkelijke code. Tot slot hebben we een hybride oplossing getest, die zowel gegevens als taak parallellisme gebruikt, en bereikten een algemene applicatie versnelling van meer dan 20x.

Onze derde en laatste case-studie (Hoofdstuk 5) richt zich op data-intensieve applicaties. Deze applicaties zijn niet van nature geschikt voor multi-core processoren: hun focus op de data in plaats van de berekening maakt weinig gebruik van de rekeningscapaciteit van deze architecturen en leidt vervolgens tot onvoldoende prestaties. Met behulp van gridding en degridding (twee data-intensieve kernels vaak te vinden in het digital signal processing applicaties) tonen we aan dat een aantal generieke algoritmische wijzigingen en optimalisaties de kloof kunnen compenseren tussen de vereiste bandbreedte en de communicatie mogelijkheden van het platform. We tonen ook aan dat naast de applicatie-niveau parallellisatie en de low-level optimalisaties (al gezien in de vorige case-studies) kunnen data-afhankelijke optimalisaties de prestaties nog verder kunnen verhogen. We kunnen dan ook concluderen dat bij een applicatie rekening moet worden gehoudt met de structuur van de I/O-data, die op haar beurt gebruikt kan worden om data-afhankelijke optimalisaties uit te voeren en uiteindelijk de prestatie te verbeteren.

Onze case-studies brachten diverse uitdagingen aan het licht die we handmatig moesten op te lossen. In Hoofdstuk 7 onderzoeken we of een van de moderne programmeer tools, meestal bedoeld om multi-cores programmeren te ondersteunen, de juiste mix van functies biedt waarmee de gebruiker effectief dergelijke uitdagingen kan oplossen. Onze overzicht van twintig programmeer modellen en tools laat twee disjuncte klassen van modellen zien - de applicatie-gerichte en de hardware-gerichte modellen. Deze splitsing vergroot de programmeerbaarheidskloof omdat geen enkel programmeer model *alle* benodigde ondersteuning kan bieden voor het ontwerp en de ontwikkeling van multi-core applicaties.

Onze oplossing voor het overbruggen van deze programmeerbaarheidskloof is een raamwerk dat ondersteuning biedt voor een stap-voor-stap analyse-tot-implementatie strategie voor het parallel programmeren van multi-core applicaties. Hoofdstuk 7 beschrijft deze stap-voor-stap strategie en het MAP raamwerk prototype. Voor het bepalen van de toepasbaarheid van onze oplossing, hebben we de moeite van het bouwen van een mogelijke support toolchain onderzocht. Onze bevindingen zijn dat ondanks de volledigheid van de oplossing, de state-of-the-art tool ondersteuning nog lang niet afdoende is. We identificieren laten we zien wat de ontbrekende schakels zijn en geven we richtlijnen om ze te kunnen bouwen.

Over het algemeen kunnen we concluderen (Hoofdstuk 9) dat het effectief parallel programmeren van multi-core processoren een systematische aanpak van de applicatie specificatie (met inbegrip van de analyse van de prestatie-eisen en I/O-data-sets), ontwerp, implementatie, optimalisatie, en prestatie feedback vereist. Echter, we zijn er nog niet: high-level algemene oplossingen voor elk van deze problemen worden nog nauwelijks ondersteund door de beschikbare tools. Het toekomstige onderzoek moet derhalve richt op het ondersteunen van applicatie specificatie en analyse, prestatie analyse en voorspelling, geautomatiseerde parallellisatie strategieën en uiteindelijk, het vinden van het juiste platform voor elke applicatie.

About the Author

Ana Lucia Varbanescu was born on the 2nd of February 1978 in Bucharest, Romania. In 1996, Ana Lucia graduated the special Informatics class of the National College "Sf. Sava" in Bucharest, Romania, and started her studies at the POLITEHNICA University of Bucharest (UPB), in the Computer Science and Engineering faculty. In 2001, she received her BSc./Eng. degree in Computer Science and Engineering from UPB. In 2002, she received her MSc degree in Advanced Computing Architectures from the same university.

In the summer of 2001, Ana Lucia was a Socrates exchange-student in the Computer Engineering department of the Faculty of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology, where she worked together with Anca Molnos, under the supervision of Prof. Sorin Cotofana. In the spring of 2002, she was a visiting researcher (with a UPB grant) in the same group, where she worked together with Prof. Sorin Cotofana towards her MSc degree.

After finishing her studies, Ana Lucia worked for a couple of years as a teaching assistant at UPB for topics like Computer Architecture and Microcontroller/Microprocessor Programming. In September 2004, she became a PhD student with the Faculty of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology (TU Delft), as a member of the Parallel and Distributed Systems group, under the supervision of Prof. Henk Sips.

In 2006 and 2007 Ana Lucia was a summer intern at IBM TJ Watson (USA), working under the supervision of Dr. Michael Perrone in the Cell Solutions Department. In the sprig of 2007, she received a HPC-Europa travel grant and was a visiting researcher at the Barcelona Supercomputing Center (Spain), working under the supervision of Prof. Rosa Badia and Prof. Xavier Matorell. In 2009, she was a summer intern at NVIDIA (USA), working in the CUDA Solutions group, under the supervision of Ian Buck and Stephen Jones. During her PhD studies, Ana Lucia received the SuperComputing Broader Engagement Grant (November 2008 and 2009), she was a Google Anita Borg Finalist (June 2008), and she received a HPC-Europa Young Researcher Mobility Scholarship (in April 2007).

Between 2004 and 2010, Ana Lucia was also involved in various teaching activities at TUDelft and VU, including the supervision of the Parallel Algorithms for Parallel Computers practicum (TUDelft) and Parallel Programming practicum (VU), as well as the supervision of several MSc students.

Ana Lucia's research interests are in the areas of high-performance computing, novel multi-core architectures and their programmability, programming models, and performance analysis and prediction.

In 2009, Ana Lucia received a NWO IsFast grant, for the ART-Cube project, which focuses on the use of multi-cores for astronomy imaging applications. Since 2009, she is a part-time PostDoc in the Parallel and Distributed Systems group at TUDelft, working with Prof. Henk Sips, and in the Computer Systems group at the Vrije Universiteit Amsterdam, working with Prof. Henri Bal.

Ana Lucia loves basketball in all its forms (playing, coaching, watching) and cycling (mostly the outdoor biking part). She enjoys traveling, small cities, and supermarkets. She likes good food - both cooking and eating. She has mild addictions to dark chocolate, coffee, and cheese (the latter mainly in combination with red wine). She collects and (sometimes) repairs bikes. She likes good cartoons, and she rarely skips any news on xkcd.com and phdcomics.com.

Selected publications*

International (refereed) journals

1. A.L.Varbanescu, A.S. van Amesfoort, R. van Niewpoort, H.J. Sips: **Building Sky Images on the Cell/B.E.** *In Scientific Programming, special issue on HPC on the Cell/BE (2008)*

2. A.L. Varbanescu, L.-K. Liu, K.A. Ross, Q. Liu, A. Natsev, J.R. Smith, H.J. Sips: **Evaluating Application Mapping Scenarios on the Cell/B.E.** *Concurrency and Computation: Practice and Experience (2009)*

International (refereed) conferences

1. J.Keller, A.L.Varbanescu: **Performance Impact of Task Mapping on the Cell/B.E. Multicore Processor.** *In A4MMC, 2010 (co-located with ISCA 2010).*

2. A.S.van Amesfoort, A.L. Varbanescu, R. van Niewpoort, H.J. Sips: **Evaluating multi-core platforms for HPC data-intensive kernels.** *In Computing Frontiers 2009.*

3. A.L. Varbanescu, A.S. van Amesfoort, T. Cornwell, B. Elmegreen, A. Mattingly, Rob van Niewpoort, G. van Diepen, H.J. Sips: **Radioastronomy Imaging on the Cell/B.E..** *In EuroPar 2008.*

4. A.L. Varbanescu, L.-K. Liu, K.A. Ross, Q. Liu, A. Natsev, J.R. Smith, H.J. Sips: **An Effective Strategy for Porting C++ Applications on Cell/B.E..** *In ICPP 2007.*

5. L-K. Liu, Q. Li, A. Natsev, K.A. Ross, J.R. Smith, A.L. Varbanescu: **Digital Media Indexing on the Cell Processor.** *In ICME 2007.*

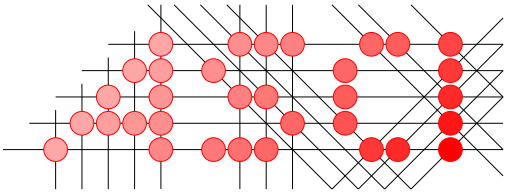
6. F. Petrini, G. Fossum, J. Fernandez, A.L. Varbanescu, M. Kistler, M. Perrone **Multicore Surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine.** *In IPDPS 2007.*

7. A.L. Varbanescu, H.J. Sips, A.J.C. van Gemund: **PAM-SoC: a Toolchain for Predicting MPSoC Performance.** *In Europar 2006.*

8. A.L. Varbanescu, M. Nijhuis, A. Gonzalez-Escribano, H.J. Sips, H. Bos, H.E. Bal **SPCE: An SP-based Programming Model for Consumer Electronics Streaming Applications.** *In LCPC 2006.*

*A complete list of Ana Lucia Varbanescu's publications can be found at http://www.st.ewi.tudelft.nl/~varbanescu/.

Colophon



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number **223**.

This manuscript was typeset by the author with the LaTeX 2_ε The text was edited using Kile and TeXnic Center. The illustrations and graphs were created with gnuplot, bargraph, Corel Draw, Dia, and various pre- and post-processing scripts.

The LaTeX 2_ε page formatting is ensured by a modified version of the "It Took Me Years To Write" template by Leo Breebaart[7].

The cover image (designed by Ana Lucia Varbanescu and produced by Alexandru Iosup) is entitled "Curiosity or skepticism?", and it can be considered a metaphorical portait of the author, who found herself quite often in the same dilemma - at least research-wise.

The black cat is designed by Allan Cheque Chaudar (aka Miaaudote, see http://www.toonpool.com/artists/Miaaudote_1029). While a connection between the legendary multiple lives of a cat and the multiple-cores of our target architectures could be made, the true reason behind the main character on the cover is that the author likes black cats. A lot.

Finally, the "word cloud" represents a graphical histogram of the conclusions of this thesis, as generated by Wordle (http://www.wordle.net/). The complete image, as well as the similar Wordle for the introduction of this thesis can be found on the page dedicated to this thesis at:
http://www.pds.ewi.tudelft.nl/~varbanescu/PhDThesis.html

[7]Leo Breebaart, "It Took Me Years To Write" template for Delft University PhD Thesis, 2003. [Online] Available: http://www.kronto.org/thesis/ (checked Nov 2010).