VU Research Portal

Programming Many-Cores on Multiple Levels of Abstraction

Hijma, H.P.

2015

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
Hijma, H. P. (2015). *Programming Many-Cores on Multiple Levels of Abstraction*. [PhD-Thesis - Research and graduation internal, Vrije Universiteit Amsterdam].

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

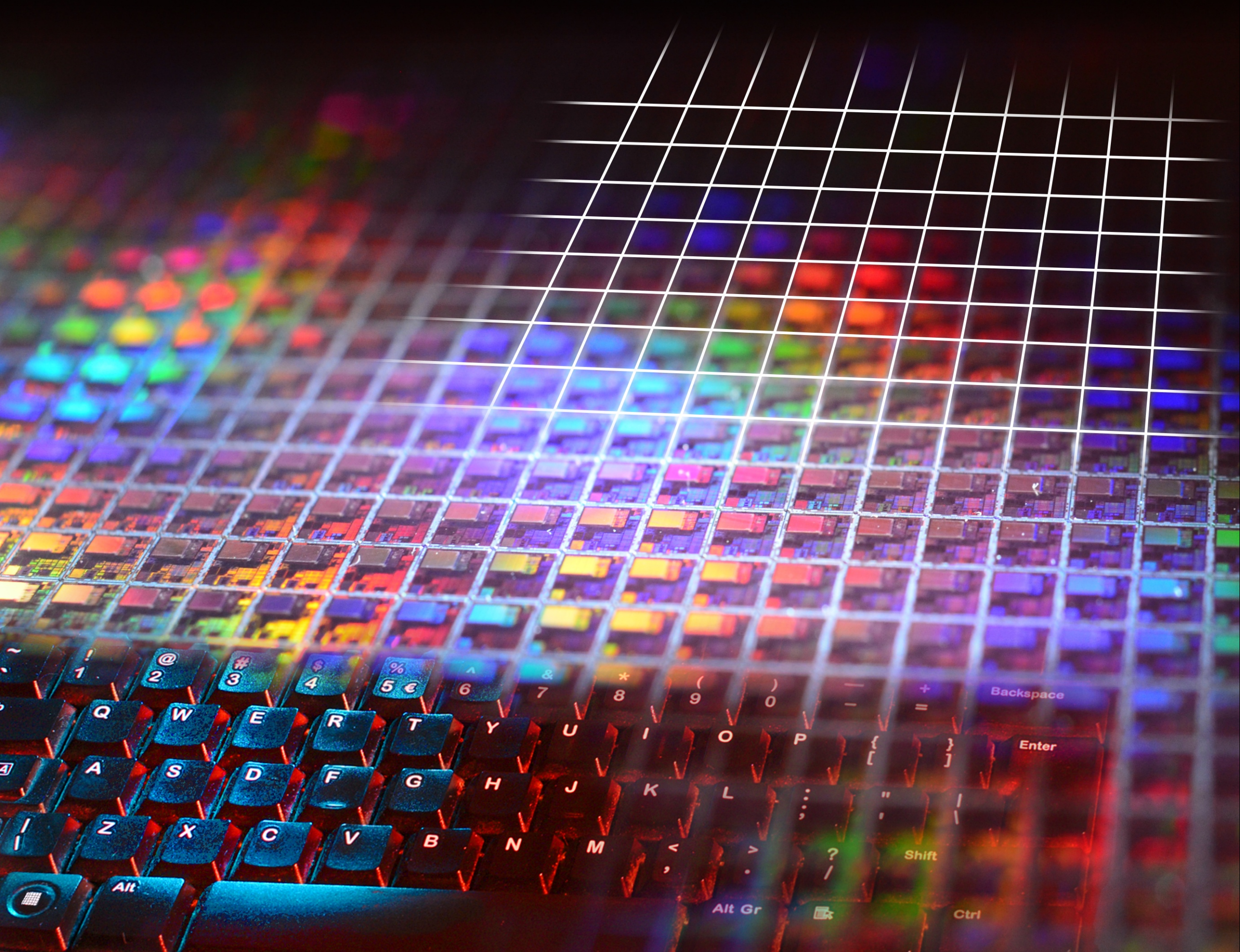
Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 02. May. 2023

Programming Many-Cores on Multiple Levels of Abstraction

Pieter Hijma



Programming Many-Cores on Multiple Levels of Abstraction

Pieter Hijma

VRIJE UNIVERSITEIT

Programming Many-Cores on Multiple Levels of Abstraction

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op dinsdag 9 juni 2015 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Hein Pieter Hijma

geboren te Dokkum

promotor:	prof.dr.ir. H.E. Bal
copromotor:	dr. R.V. van Nieuwpoort

members of the thesis committee:	dr. Rosa M. Badia
	prof.dr.ir Henk J. Sips
	prof.dr. Paul Klint
	dr. Clemens Grelck
	dr.-Ing.habil. Thilo Kielmann

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 329.

This thesis was realized with support from the Wijbenga-leen.

Cover image by Pieter Hijma, adapted from the photograph "wafer 3" from Santi: `https://www.flickr.com/photos/s4nt1/15137081219`. The photograph was licensed with the Creative Commons Attribution-ShareAlike 2.0 license: `https://creativecommons.org/licenses/by-sa/2.0`. The cover image is licensed with the same license, but version 4.0: `http://creativecommons.org/licenses/by-sa/4.0`.

Copyright © 2015 by Pieter Hijma

Acknowledgements

Although it has not been easy at times, I have learned so much these last years. I am very grateful for the faith and continuous support from my promotor Henri Bal and co-promotor Rob van Nieuwpoort. Henri and Rob, you have given me much freedom and from the beginning we have tried to balance this freedom with minimizing the risk by avoiding to implement a large compiler. We have failed in doing so, but I am now glad that we took those risks and I am happy with the end-result: this thesis.

I have been very lucky to receive much help from many people. Firstly, Ceriel Jacobs helped in virtually all my papers. In all cases I would start to implement my ideas, would realize after a while that it would take too much time for one person, after which Ceriel would step in and learn my completely undocumented code. This must not have been easy, especially with the Many-Core Levels (MCL) system as it had a very large code-base. However, Ceriel, for me it was invaluable to be able to discuss the low-level details with you.

Secondly, I want to thank the Software Analysis and Transformation (SWAT) team at the CWI. They invited me to come work at the CWI with their system Rascal in which I implemented MCL. Their direct and active support allowed me to learn Rascal fast. Rascal has been invaluable for prototyping my language designs and it helped me to work very quickly, allowing me to almost keep up with “a compiler pass a day”. I want to thank Paul Klint, for his enthusiasm about me using Rascal; Jurgen Vinju, for working incredibly hard to fix bugs in Rascal; Tijs van der Storm, from whom I learned many things about programming languages; and Atze van der Ploeg. Atze, thanks for the countless discussions we have enjoyed. I learned much about my own ideas by trying to convince you that you were wrong, especially in reaction to your standard phrase: “Oh, that’s trivial!”. Let us not comment on how many times you actually were wrong, as it would start another discussion.

Furthermore, I want to thank my committee for finding time to take part in

my defense and to consider whether I deserve the degree of Ph.D.: Rosa Badia, Henk Sips, Paul Klint, Clemens Grelck, and Thilo Kielmann. Thank you for your kind words and your constructive feedback that allowed me to improve the thesis further.

I want to thank all my colleagues at the VU: My paranymphs Ben van Werkhoven and Stefan Vijzelaar: Ben, I am glad that after sharing the same office for so long (and after you left), we have finally started working together with a nice paper as outcome. Stefan, thank you for the nice collaboration in the Concurrency and Multithreading class. I hope the students learned many things, but in any case we learned more and more about concurrency each iteration. I want to thank all my office mates during the years for the enjoyable time: Albana Gaba, Suhail Yousaf, Christian Rossow, Rena Bakhshi, Ben, and Alessio Sclocco.

I want to thank Andy Tanenbaum for appreciating the proposal I wrote for the Research Proposal Writing class and notifying Henri Bal about it. This led to my Master project which transitioned into my Ph.D.; Kees van Reeuwijk for supervising me initially in the sync generator project; Kees Verstoep for keeping the DAS-4 always up, especially in paper deadline periods; and Wan Fokkink for working together on the Concurrency and Multithreading class and giving me the opportunity to do a guest lecture.

Alessandro Margara, I much enjoyed our discussions during the many bike rides to the swimming pool, even though I knew you would beat me in swimming afterwards. Thanks also for your feedback on the MCL papers from which I learned much; Roelof Kemp, thank you for the coffee moments. I especially remember the coffee breaks in the summer of 2012 in which you looked back on your Ph.D. time. This has influenced the way I worked; Timo van Kessel, I regret that we did not manage to work together on a paper but I wish you all the best; Remco Vermeulen, I admire the fact that you took even more risk than I did with the incredibly difficult subject that you chose.

Many thanks to everybody of Henri's group: Ismail El-Helw, Jacopo Urbani, Kaveh Razavi, Alex Uta, Ana Oprescu, Stefania Costache, Rutger Hofman, Daniela Remenska, and old colleages Nick Palmer, Frank Seinstra, Jason Maassen, Niels Drost, and Maarten van Meersbergen. Many thanks as well to all others in the Computer Systems group. Thank you Caroline Waij, for arranging many things so swiftly, from barbecues to printer paper.

The Netherlands eScience Center needs special mention as they allowed Rob van Nieuwpoort to continue supervising me. Thanks to Cees de Laat and Ana Varbanescu from the UvA for allowing me to continue in academia; Raphael Poss and Merijn Verstraaten for their interest in my work and discussing new angles for MCL I did not think about. I want to thank Zeno Gerardts, Pelle Barens, Ewald Snel, and Arjan Mieremet from the NFI for their interest in MCL and Cashmere and providing an interesting forensics application.

I want to thank my parents for always supporting but never pushing me, even when I took drastic decisions such as starting with a Computer Science study. I am very grateful for all the chances you offered me. My brother Durk, thanks for not taking such drastic decisions as starting a Computer Science study. Let us agree on me doing the programming, while you remain doing what you do so well.

Dear Dorine, thank you for all the support and thank you for accepting the too many times that I had to work: weekends, evenings, and nights. It has been amazing for me that we could talk about my computer science topics and that you would not only listen, but also understand and ask questions. It is not for nothing that you came up with the title “Cashmere” for my system. I am very grateful I have you in my life.

Finally, all my friends that showed support: Sandra, Mandana, Haiko, Thomas, Renée, Atze, Herman, Melanie, Remco, Steven, Martijn, Despo, Abhi, Gerard, Femke, Martin, Hannah, Maarten, Koen, and Alessandro, thank you for your understanding in case I was stressed or busy, probably both. I hope we can catch-up the lost time of the last few years after the defense.

Contents

Acknowledgements **ix**

Contents **xiii**

1 Introduction **1**

1.1 Background 4
1.2 Scope 6
1.3 Problem Statement and Research Questions 6
1.4 Outline of this thesis 7

2 Generating Synchronization Statements **11**

2.1 Introduction 12
2.2 The Satin programming model 13
2.3 Problem description 16
2.4 Implementation 19
2.4.1 Basic algorithm 19
2.4.2 Analysis phase 20
2.5 Evaluation 26
2.5.1 Evaluation per application 29
2.6 Discussion 31
2.6.1 Precision of the Alias-Analysis 32
2.6.2 Improving Sync Generation with Programmer Support . . 34
2.6.3 Cilk 35
2.6.4 Futures 36
2.7 Programming model design considerations 36
2.8 Related Work 37
2.9 Conclusion 38

3 Stepwise-refinement for performance 41
3.1 Introduction . . . 42
3.2 Related work . . . 44
3.2.1 Programming Many-cores . . . 44
3.2.2 Other related Work . . . 47
3.3 Stepwise-refinement for Performance . . . 48
3.3.1 Philosophy . . . 49
3.3.2 Methodology . . . 50
3.4 Design of MCL . . . 51
3.4.1 Overview . . . 52
3.4.2 Hardware Description Language HDL . . . 54
3.4.3 Programming Language MCPL . . . 57
3.4.4 Compiler . . . 59
3.5 Example: Matrix Multiplication . . . 61
3.5.1 GTX480 . . . 65
3.5.2 Xeon Phi . . . 72
3.5.3 Summary . . . 75
3.6 Implementation . . . 76
3.6.1 Translation between Abstraction Levels . . . 76
3.6.2 Operation Statistics . . . 79
3.6.3 Data Reuse Analysis . . . 80
3.6.4 Cache Analysis . . . 82
3.6.5 Performance Feedback Functions . . . 83
3.7 Evaluation . . . 84
3.8 Discussion . . . 91
3.9 Conclusion . . . 92

4 Cashmere: Heterogeneous many-core computing 95
4.1 Introduction . . . 96
4.2 Cashmere Programming Model . . . 98
4.2.1 Satin . . . 98
4.2.2 MCL . . . 100
4.2.3 Cashmere programming model . . . 100
4.3 Implementation . . . 103
4.3.1 MCL . . . 103
4.3.2 Cashmere . . . 104
4.4 Methodology . . . 106
4.5 Evaluation . . . 109
4.5.1 Kernel performance . . . 109

4.5.2 Scalability 110
4.5.3 Heterogeneity 111
4.6 Related Work 117
4.7 Conclusion 120

5 Conclusions 123
5.1 Summary 123
5.2 Future Directions 126
5.2.1 Conclusions 128

A Many-Core Levels Language Descriptions 129
A.1 Hardware Description Language HDL 129
A.2 Programming Language 134

B Translating to a lower level of abstraction 139
B.1 The top-level functions 140
B.2 Finding equivalent ParGroups 141
B.3 Translating memory spaces 142
B.4 Translating ForEach statements 144

References 151

List of Publications 163

Samenvatting 165

Chapter 1

Introduction

In the last decades we have witnessed an incredible increase in CPU performance. As a result, we were in the fortunate position of obtaining higher performance by running virtually the same software on newer generations of hardware. Fig. 1.1 illustrates Moore's Law, the exponential increase of the number of transistors over time that made this performance increase possible.

However, Fig. 1.1 also shows the clock-frequency over time. It shows that clock-frequency increased exponentially until the mid 2000s after which it leveled, while Moore's Law remained as before. Thus, every 18 months the number of transistors that we could fit on a die increased, but we were not able to increase the clock-speed due to power dissipation excesses. This meant a major change in the design of processors: the solution was to fit multiple processors or cores on one die running on a lower clock-frequency and therefore reducing the power dissipation. We call these processors *multi-core* processors (starting from "Core 2" in Fig. 1.1).

This was a fundamental change from a software viewpoint because from that moment processors exposed *parallelism* to the programmer. In fact, the traditional single-core machines were already highly parallel machines with much logic dedicated to optimize *sequential* instruction streams. However, to gain the same performance increase associated with the increase of the number of transistors for *multi-core* processors, the programmer has to offer the processor multiple parallel instruction streams. In other words, traditional sequential software will not run faster automatically on multi-core processors, but the software has to be made parallel to benefit from the increased performance of the multi-core processors.

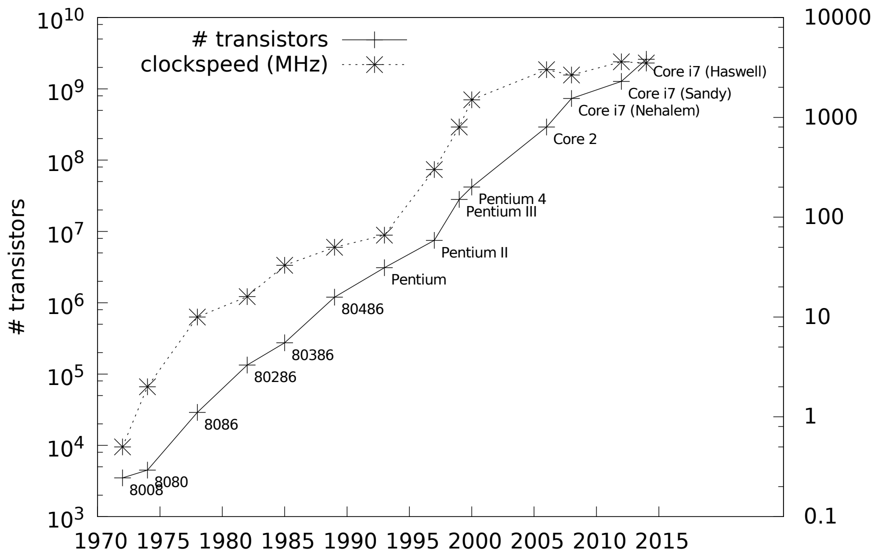


Figure 1.1: Development of processors over time (source Intel).

So far, we have seen two types of processors: traditional *single-core* processors, with much logic dedicated to optimize sequential programs and *multi-core* processors with multiple cores of which each core has still much logic dedicated to optimize sequential programs. It exposes parallelism to the programmer, but in a moderate way (the number of cores). This thesis, however, studies how to program *many-core* processors. Instead of dedicating logic to optimize sequential programs as single-core and multi-core processors do, many-core processors use most logic to expose a high degree of parallelism to the programmer in the form of many simple cores. Its programs need to be highly parallel, but in return the programmer is able to achieve much higher performance than with multi-core processors because a large degree of the logic of a chip is used for computing instead of analyzing sequential instruction streams.

Examples of the many-core processors that we use in this thesis are the Graphics Processing Unit (GPU) and the Xeon Phi from Intel. GPUs have become more general over the years and are now often used to accelerate large scientific codes or other compute intensive applications. The Xeon Phi is also typically used in this way. Both processors have in common that they expose large amounts and multiple levels of parallelism. Typically, they provide tens of cores that need hundreds of parallel tasks to keep the device busy. Each core then exposes another level of parallelism in the form of small, simple data-parallel cores or vector processor units.

The design of these processors is important for the energy consumption. Because most of the logic of the chip is dedicated to computation and not optimizing instruction streams, many-cores deliver much higher performance per Watt [1, 2]. This is not only important for mobile devices, but also for supercomputers that start to use unacceptable amounts of energy. From 1972 to 2007, the performance of supercomputers increased 10,000 fold, but the performance per Watt ratio only increased 300 fold. Nowadays, supercomputers draw power equivalent to small cities [3]. It is clear that many-cores provide an opportunity in this context and many supercomputers are already fitted with many-core processors today. It is highly likely that massively parallel hardware in the form of many-cores will play an important role in computing in the future.

However, many-core processors are difficult to program for multiple reasons. The only goal of many-core processors is obtaining high performance. They expose a complicated interface to the programmer with many levels of parallelism and a complicated memory hierarchy, all meant to reach high performance. They do not have a standardized instruction set as single-core and multi-core processors have, there are many different kinds of many-core processors, and the hardware evolves quickly. Because of the many levels of parallelism that closely interact with the memory subsystem, the performance of a program can improve drastically if it accounts for the limitations of the processor and makes proper use of the available hardware. Therefore, optimizing a program for many-cores is often beneficial but also very difficult.

This thesis proposes solutions for this programming problem. It takes a fundamental approach: It considers the hardware limitations that we face, for example the memory wall (computation speed increases faster than memory access speed) and the energy wall (clock-frequency cannot increase any further) as programming problems. Thus, we consider the *single-core era* to be a fortunate situation in which the increase of clock-frequency meant automatic performance gains without changing the software, but it was never truly sustainable; *multi-core hardware* shows a transition from offering a sequential hardware interface to a parallel hardware interface with still much logic to optimize sequential programs. This transition will ultimately lead to hardware that will not make any compromises in its interface to programmers to overcome the hardware limitations that we face. This thesis considers *many-core processors* as a first manifestation of this kind of hardware. To summarize what this thesis is about: Eventually we will reach the limits of hardware which will result in a complicated interface to the programmer, making this hardware increasingly more difficult to program. The main question that this thesis tries to answer is how to effectively program this hardware while still achieving high performance.

The next section describes some background to clarify why we believe that the shift from single-core and multi-core processors to many-core processors should change the traditional view on the role of programmers and compilers. In Sec. 1.2 we explain the scope of the thesis and Sec. 1.3 defines the problem and presents the research questions. Section 1.4 presents the outline of the thesis.

1.1 Background

Computer hardware exposes a set of instructions that one can execute on this hardware. Programming means to find a sequence of instructions such that a problem is solved. Problems can be solved in many ways and the particular way a problem is solved is called an algorithm. Given an algorithm for solving a problem, programming is finding the sequence of instructions that implements this algorithm.

Usually, the exposed instructions are very low level and cumbersome to use. As a result, a fundamental step in computer science is to find a sequence of instructions that occurs often and find a simple expression that can be translated into this sequences of instructions. For example, adding two numbers in memory locations `a` and `b` and loading the result in memory location `c` could be encoded into the following instructions:

```
mov(r1, a)
mov(r2, b)
add(r1, r2)
mov(c, r1)
```

The first two instructions move the value in memory locations `a` and `b` into registers `r1` and `r2` as an add instruction cannot operate directly on memory but only on special purpose registers. The add instructions computes the addition and puts the result in register `r1` that we move back into memory location `c` in the final instruction. Because this is cumbersome we prefer an expression such as

```
c = a + b
```

which computes the same. We call this a higher-level abstraction because it abstracts away the movement from and to registers. The task of translating a higher-level abstraction to the lower-level representation is performed by a compiler. In a way, the compiler exposes a new interface to the hardware that matches better with how we think.

Another task of the compiler is optimizing code. For example, it may be natural for us to write an expression as

```
c = (a + b) * (a + b)
```

The compiler may be able to perform common sub-expression elimination and transforming this code into

```
t = a + b
c = t * t
```

removing one addition. This seems like a good idea, but it is important to note that by doing this, the compiler makes certain assumptions about the hardware. In this case, the compiler assumes that using a temporary memory location `t` is beneficial for the overall performance. However, dependent on the hardware, this may not be the case. For example in many-core hardware, it may be faster to recompute the value than reusing a register.

This scheme, consisting of

- provide the programmer high-level abstractions,
- automatically translate to lower-level representations and optimize the instructions where possible, and
- run the instructions while extracting parallelism from the instruction-stream on the fly in hardware

has been very successful in the single-core era because 1) the interface of hardware has not changed much over the years, and 2) obtaining higher performance could be achieved by buying newer hardware.

Due to the increasing complexity of the hardware interface that is exposed to programmers, we reconsider the responsibilities in achieving high performance in this thesis. We envision that programmers will be responsible for achieving performance and expressing parallelism. We change the role of compilers as being a black box translator and optimizer to a more advisory role where programmers have much control. Because performance becomes a software problem and not a hardware problem we also reconsider high-level abstractions in relation to control over the hardware.

1.2 Scope

In this thesis we study the problem of programming many-cores. Programming in itself is already very challenging. Humans are experts in communicating through natural language where a context is always implied to resolve the many ambiguities that natural language contains. However, to communicate with (read: programming) a machine we need formal languages that leave no room for ambiguities. For humans, it is very challenging to adapt to these formal languages and remove all ambiguities, especially when programs become larger and more complex [4].

It is also important to note that software is not only written for machines, but also for humans. Typically, software grows slowly and remains in use much longer than anticipated. This means that software has to be maintained. Already in 1968 it was clear that the cost of software engineers writing and maintaining codes exceeded the cost of hardware [5].

Sequential programming is challenging, but parallel programming is even more demanding. Instead of expressing the steps for one sequential program, programmers have to decide on the granularity of parallelism, how the parallel tasks interact and how shared data can be protected. Lee explains the many problems that can arise in parallel programming [6].

Within the parallel programming field, this thesis focuses on programming many-cores. As explained above, many-cores expose all kinds of architectural intricacies to achieve high performance and the performance differences can be quite large when a program is well-balanced in how the hardware is used.

In general, many-core processors will be used as components in larger systems such as supercomputers or clusters, so we also study how to program *clusters* of many-cores. More specifically, as many-core hardware shows much variety and evolves fast, we study clusters of *heterogeneous* many-core processors. We expect that in the near future heterogeneity of many-cores in clusters can play an important roll in improving power consumption and performance.

1.3 Problem Statement and Research Questions

The limitations in hardware (memory wall, energy wall) will have an effect on the design of hardware that will eventually have repercussions on how to program these chips. Many-core hardware is the first manifestation of hardware that provides little compromise in the hardware design and the interface that is exposed to programmers. The main research question of this thesis is:

- How can we support programmers in their responsibility to achieve high performance from many-core hardware?

There are several sub-questions that follow from this question that we try to answer in the following chapters:

1. What are important design considerations for parallel programming models and their compiler analyses?
2. How to balance control over hardware with raising the level of abstraction?
3. How can we manage the many different types of many-core hardware that exist?
4. Can we provide programmers a structured approach with which a programming system can assist them to achieve high performance?
5. How to achieve good scalability when programming *clusters* of many-cores?
6. How to program *heterogeneous* many-core clusters?

1.4 Outline of this thesis

The major contribution of this thesis is a methodology for programming many-core devices that we call "Stepwise-refinement for performance". We present two programming systems for many-core devices: Many-Core Levels (MCL) and Cashmere. MCL focuses on writing computational kernels for many-core devices on multiple levels of abstractions and as a system, it supports the *stepwise-refinement for performance* methodology. We define a computational kernel, or just kernel, as a compute-intensive set of functions within an application that as a whole can be run on a many-core device. Cashmere combines MCL with Satin, an existing divide-and-conquer programming model [7] to bring the many-core compute power to clusters of heterogeneous many-core devices.

Chapter 2 This chapter does not yet focus on many-core devices but presents an analysis of Satin programs. This chapter gives an answer to research question 1 and has important conclusions that influenced the design of MCL and Cashmere. Most importantly, it led to the insight that it may be preferable to have compilers in a more advisory role while programmers remain in control.

This is a major theme throughout the thesis. This chapter is based on the following publications:

> Automatically Inserting Synchronization Statements in Divide-and-Conquer Programs. Pieter Hijma, Rob V. van Nieuwpoort, Ceriel J.H. Jacobs, and Henri E. Bal. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1233–1241, May 2011.
>
> Generating synchronization statements in divide-and-conquer programs. Pieter Hijma, Rob V. van Nieuwpoort, Ceriel J.H. Jacobs, and Henri E. Bal. *Parallel Computing*, 38(1-2):75 – 89, January–February 2012.

Chapter 3 This chapter answers research question 2, 3, and 4. It presents the Many-Core Levels programming system that supports the *stepwise-refinement for performance* methodology with which programmers can optimize computational kernels guided by the compiler. The novelty in MCL is that it supports multiple levels of abstraction, giving programmers a trade-off in portability and program maintainability against control to reach high performance. This chapter is adapted from the following publications:

> Programming Many-Cores on Multiple Levels of Abstraction. Pieter Hijma, Rob V. van Nieuwpoort, and Henri E. Bal. In *Proceedings of the 5th USENIX Conference on Hot Topics in Parallelism (Poster presentation)*, HotPar '13, pages 1–7, Berkeley, CA, USA, 2013. USENIX Association.
>
> Stepwise-refinement for performance: a methodology for many-core programming. Pieter Hijma, Rob V. van Nieuwpoort, Ceriel J.H. Jacobs, and Henri E. Bal. *Concurrency and Computation: Practice and Experience*, 2015. `http://dx.doi.org/10.1002/cpe.3416`.

Chapter 4 In this chapter we introduce Cashmere, a programming system that combines MCL with Satin's divide-and-conquer programming model to program heterogeneous clusters of many-core devices. The major contribution of this chapter is a programming system that achieves good scalability even with many-cores that vary widely in architectures and performance. This chapters answers research questions 5 and 6 and is based on the following publication:

Cashmere: Heterogeneous Many-Core Computing. Pieter Hijma, Ceriel J.H. Jacobs, Rob V. van Nieuwpoort, and Henri E. Bal. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015), 25-29 May 2015, Hyderabad, India.*, 2015.

Chapter 5 This last chapter presents a summary of the thesis, our conclusions, and future directions. We conclude that MCL and Cashmere provide promising solutions to support programmers in their responsibility to extract performance from many-core devices, whether or not arranged in cluster computers. MCL provides a trade-off in control over hardware and the level of abstraction, and it supports a methodology that gives programmers insight in the compiler and the performance of their application in relation to the hardware. Cashmere provides a solution for heterogeneous many-core clusters with automatic load-balancing and detailed performance feedback in the form of Gantt-charts of the execution.

Chapter 2

Generating synchronization statements in divide-and-conquer programs

Divide-and-conquer is a well-known and important programming model that supports efficient execution of parallel applications on multi-cores, clusters, and grids. In divide-and-conquer systems such as Satin or Cilk, recursive calls are automatically transformed into jobs that execute asynchronously. Since the calls are non-blocking, consecutive calls are the source of parallelism. However, programmers have to manually enforce synchronization with `sync` statements that indicate where the system has to wait for the result of the asynchronous jobs.

In this chapter, we investigate the feasibility of automatically inserting `sync` statements to relieve programmers of the burden of thinking about synchronization. We investigate whether correctness can be guaranteed and to what extent the amount of parallelism is reduced when synchronization statements are inserted automatically. We discuss the code analysis algorithms that are needed in detail.

To evaluate our approach, we have extended the Satin divide-and-conquer system, which targets efficient execution on grids, with a sync generator. Our experiments show that, with our analysis, we can automatically generate synchronization statements in virtually all real-life cases: in 31 out of 35 real-world applications the `sync` statements are placed optimally. The automatic placement is correct in all cases, and in one case the sync generator corrected synchronization errors in an application (FFT).

Finally, this chapter presents important design considerations that influenced the design of MCL and Cashmere: most importantly, we conclude that programmers have application knowledge that compilers lack and that compilers can overlook optimizations because they have to be conservative to guarantee correctness.

2.1 Introduction

Writing parallel programs is difficult in general. Writing parallel programs that execute efficiently on multiple clusters or clouds is even more demanding. Satin [7] makes cross-grid computing accessible to programmers who are not parallel programming experts. It allows programmers to write parallel programs without much effort by offering a sequential divide-and-conquer programming model. Typical applications for Satin are large scientific computations or other compute-intensive problems.

In Satin, programmers annotate recursive methods to indicate that calls to these methods are spawnable, which means that they can be executed asynchronously. Consecutive spawnable method calls create parallelism in the program. These method calls are transformed into jobs that are executed efficiently on grids or clouds using the Ibis platform [8].

However, since spawnable method calls are non-blocking, programmers also have to annotate where in the program the system has to block until the results of the jobs are available. Programmers indicate this by carefully inserting `sync()` statements. Placing sync statements too soon results in less parallelism than possible and placing them too late gives incorrect results. Our goal is to make grid computing even more accessible to programmers who are not parallel programming experts by making sync insertion automatic.

Having automatic sync insertion, by means of a sync generator, means that programmers no longer have to think about synchronization. The following questions arise: can syncs always be automatically inserted in such a way that the resulting program is correct; how much parallelism can be obtained; and what analysis is needed to accomplish this.

We found that the implementation of the sync generator program is able to insert sync statements in such a way that the resulting program is always correct, but alias and control-flow analysis are needed to accomplish this. More extensive analysis will not lead to optimal placement in all cases: programmers sometimes *deliberately* use unsynchronized variables for performance reasons. An automatic generator cannot determine this.

In practice, our sync generator achieves excellent results. We tested the sync generator on 35 pre-existing real-world Satin applications. In 31 of them, the sync generator found the optimal locations for the sync statements. In all but one of the remaining cases, the sync generator gave a warning that the placement was likely suboptimal. In one case, it even corrected an originally incorrect application (FFT).

Our contributions are the following:

- We make implementing parallel divide-and-conquer applications even more effortless than before. Programmers only have to indicate parallel methods but not the synchronization points.
- We offer a good understanding of the problems involving automatically inserting synchronization statements.
- We provide a working implementation in the form of a compilation pass for the Satin compiler that generates synchronization statements automatically.
- We answer research question 1: What are important design considerations for parallel programming models and their compiler analyses?

The following section discusses the Satin programming model and explains some basic concepts. Section 2.3 defines the problem in detail and Sec. 2.4 discusses the implementation of the sync generator. We evaluate the sync generator in Sec. 2.5 using 35 real-world Satin applications while Sec. 2.6 discuss the results and presents important design considerations that affected the design of MCL and Cashmere. Section 2.8 describes related work after which the chapter concludes.

2.2 The Satin programming model

This section briefly introduces the Satin programming model. Satin [7] is a divide-and-conquer framework similar to Cilk [9]. The main differences are that

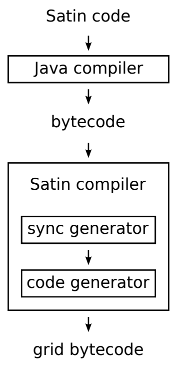


Figure 2.1: Compiler procedure with automatic sync generation.

programs are written in Java instead of Cilk, a C derivative, and that programs are deployed on a grid and not on shared memory systems.

The Satin compiler rewrites Satin programs in such a way that they can run in parallel on grids and clouds using Ibis [8]. Ibis is an environment that provides communication primitives to compute nodes in a grid. Rewriting Satin programs is performed with help of the bytecode rewriting library BCEL [10].

In order to create parallelism in a Satin program, programmers have to make some annotations. They have to indicate which methods are spawnable to ensure that the Satin compiler will rewrite those methods to versions that spawn jobs on the grid. The second annotation programmers have to make is a special `sync()` statement which is a barrier local to the method. The system will block at the sync statement until all results of the spawnable calls in the current method have become available. A node that blocks in a sync may execute jobs resulting from spawnable calls in the meantime.

The Java type and class systems provide all means to annotate Satin programs. Programmers create spawnable classes by extending the class `ibis.satin.SatinObject` that provides methods such as `sync()`. Programmers can also create an interface that extends `ibis.satin.Spawnable`. The method calls of the methods that are declared in this interface will be rewritten by the Satin compiler and executed in parallel.

The procedure from writing to deploying a Satin program is as follows: programmers write a sequential recursive program in Java and annotate the (possibly recursive) calls that have to be spawnable. They also place sync statements

```java
interface SpawnProg extends ibis.satin.Spawnable {
  int spawningMethod(Data data);
}

int spawningMethod(Data data) {
  if (stopCondition) return 0;

  int result1 = spawningMethod(data.part1);
  int result2 = spawningMethod(data.part2);

  sync();
  return result1 + result2;
}
```

Figure 2.2: A basic Satin program.

in places where the program has to wait for the results of the spawnable calls. The program is compiled to normal, sequential Java bytecode. This serves as input for the Satin compiler that rewrites the bytecode in such a way that Satin jobs are spawned. The application is now ready to be deployed on the grid.

The procedure in combination with the sync generator differs slightly and is depicted in Fig. 2.1. Again, programmers write a sequential recursive program but leave out the sync statements. They compile the program to ordinary bytecode. The Satin compiler takes the Java bytecode as input and generates sync statements in an extra compiler pass. It then rewrites the bytecode so that it spawns Satin jobs. It is still an option to insert sync statements manually. The sync generator will ignore methods that already contain sync statements.

Throughout the rest of the document the following terms will be used: A *spawning class* is a class that contains spawning methods. A *spawning method* is a method that contains *spawnable calls*. A *spawnable call* is an invocation of a method with a *spawnable method signature*. This is the signature of the method annotated by the programmer to be spawnable. A *sync statement* is a local barrier synchronization primitive which makes sure that all spawnable calls have returned their values.

Parameters of spawnable method signatures need to be serializable. Serializable classes provide a means to create a deep copy of the object including all objects that are referenced within the object and even cycles between these references. This allows applications to store objects on disk or transfer objects over the network. This last feature is used by the Satin runtime when a spawnable call is to be executed on a different node in the grid. Satin serializes every object in the argument list of the spawnable call to transfer the objects over the network.

Figure 2.2 depicts a typical Satin program. On line 2, `spawningMethod()` is marked to be a spawnable method, as it is defined in an interface that extends `ibis.satin.Spawnable`. The method on line 5 is a *spawning method*, because it contains two *spawnable calls* on line 8 and 9 (two recursive calls). These calls are non-blocking, and as a result the two calls run in parallel. At the statement in line 8, the Satin runtime may serialize the argument `data.part1` depending on whether the data will be sent over the network or stays on the current node. It then creates a job, spawns the job, and returns to do the same for the spawnable call on line 9. The system will block at the `sync()` statement on line 11 until both parallel calls have finished and returned their results into `result1` and `result2`. The method can now safely return the sum of `result1` and `result2`.

Besides returning values using the `return` statement, a spawning method can also return using exceptions. Figure 2.3 illustrates this. The non-blocking spawning method is called on line 10 in the `try` block, and the program continues immediately beyond the `catch` clause. It executes the second `try` block, calling the second spawnable call on line 17 in parallel. The program continues beyond the `catch` clause and blocks on the `sync()`. When the first spawnable call has finished and the result has been thrown, a new thread stores the results in local variables in the catch block on line 12 and returns from the method. When this has happened for all spawnable calls, the main thread that blocked on the sync can continue and throw the result of this method.

This exception mechanism allows applications to perform work immediately after a spawning call has finished and is often used for speculative parallelism. An example is a parallel search. As soon as a spawnable call locates the item that is searched for and throws an exception, the new thread in the catch block can abort the other spawnable calls in order to prevent that more items are searched than necessary.

2.3 Problem description

The sync generator's objective is to determine a suitable location for sync statements. Results of spawnable calls are generally stored temporarily to allow parallelism. For instance, on lines 8 and 9 in Fig. 2.2, the two spawnable calls store the results in `result1` and `result2`. Sync statements should precede the part of the code where these results are used again. Therefore, because `result1` and `result2` are used again on line 12, there is a sync statement on line 11.

```java
interface SpawnProg extends ibis.satin.Spawnable {
  void spawningMethod() throws Result;
}

void spawningMethod() throws Result {
  if (stopCondition) throw new Result();

  int result1, result2;
  try {
    spawningMethod();
  }
  catch (Result r) {
    result1 = r.result;
    return;
  }
  try {
    spawningMethod();
  }
  catch (Result r) {
    result2 = r.result;
    return;
  }

  sync();

  Result finalResult = new Result();
  finalResult.result = result1 + result2;
  throw finalResult;
}
```

Figure 2.3: A basic Satin program throwing exceptions.

Placing the sync statements too soon may reduce parallelism. For example, placing sync statements between lines 8 and 9 and after line 9 in Fig. 2.2 results in sequential execution of the two spawnable calls. Placing sync statements too late, after `result1` and `result2` have been used again, can lead to incorrect results.

The implementation of the sync generator focuses first of all on correctness. The main question is: is it possible to create a sync generator compilation pass that inserts sync statements in such a way that the resulting program is guaranteed to be correct? Correctness in this sense means that the placement of sync statements is such that the parallel version delivers the same results as the sequential version.

The second issue is then: is it possible to find a location for the sync statements in such a way that parallelism is created? If this is true, then the question is how optimal the placement is. Other questions are whether this can be achieved in a way that does not require complicated analysis or results in many spurious sync statements.

The sync generator operates under the assumption that it has as input all classes that are needed to determine which classes are spawning classes. In addition, there are some assumptions about spawnable calls. Figures 2.2, 2.3, and 2.4 show parts of typical Satin programs using return values, exceptions, and loops respectively.

A spawnable call can return in three ways. The spawnable method signature can be of type `void`. It then returns with an empty `return` statement. The second way is by returning a value using the `return` keyword. This happens in Fig. 2.2 and Fig. 2.4. Finally, it can return using exceptions as shown in Fig. 2.3.

In Satin, it is not possible to return values via parameters. Spawnable calls have either call-by-reference or call-by-value semantics, depending on whether a spawnable call is executed on the current node or on a different node in the grid respectively. In the latter case, the Satin runtime serializes the arguments of the spawnable call to send them over the network. It is undefined which mechanism is used, and therefore, programmers cannot assume call-by-reference or call-by-value semantics.

For example in Fig. 2.2, programmers cannot rely on storing information in the `data` parameter at line 5 in order to make this information available to the method that called this method. The calling method may be on a different machine in the grid and in that case will not receive the information. So, to communicate information to the calling method, programmers need to either return values or throw exceptions.

Return values from spawnable calls are likely to be stored in local variables to allow parallelism, but this may not be the case. Also, these local variables may not be loaded again. In this case and the case that the spawnable call is of type `void`, the sync generator needs to place a sync as last instruction of the spawning method.

To conclude the boundaries for the sync generator, it is not an issue when multiple syncs are placed behind each other. The system will notice during the sync that there are no spawnable calls running and will just continue without any problem. It is also no problem to have sync statements when no spawnable call will be called. The overhead of these sync statements is negligible.

```
int spawningMethod() {
  if (stopCondition) return 0;

  int[] results = new int[NR_SPAWNS];
  int finalResult = 0;

  for (int i = 0; i < results.length; i++) {
    results[i] = spawningMethod();
  }

  sync();
  for (int i = 0; i < results.length; i++) {
    finalResult += results[i];
  }

  return finalResult;
}
```

Figure 2.4: A basic Satin program using loops.

2.4 Implementation

The sync generator analyzes Java bytecode with help of the BCEL bytecode rewriting library [10]. It implements each of the terms *spawning class*, *spawning method*, *spawnable call*, and *spawnable method signature* introduced in Sec. 2.2 as classes that hold all necessary information to do the analysis. The implementation also provides extensive debugging capabilities, a library to analyze control-flow graphs, and utilities to inspect the bytecode of classes and control-flow graphs. The implementation is written in Java and is about 4400 lines of code including comments and whitespace.

The first subsection describes the basic algorithm of the sync generator compilation pass. The second subsection discusses several analysis strategies and gives insight into which problems need to be solved to automatically generate sync statements.

2.4.1 Basic algorithm

The basic algorithm of the sync generator is composed of three phases: the recording phase, the analysis phase, and the generator phase. During the recording phase, the sync generator reads in all class files. Next, it finds all spawnable method signatures from the interfaces and it then tries to create spawning classes for every class file. This succeeds if the class contains spawning methods.

For a spawning method, the recording phase records all spawnable calls, and for every spawnable call, it keeps track of:

- the invoke instruction of the spawnable call
- the load instruction of the object reference on which the spawnable call is invoked
- the indices of the local variables in which the results are stored

A spawnable call tracks multiple local variables in case an exception is thrown. The `catch` block may store results in multiple local variables and subsequent loads of these local variables require a sync statement.

When the result is stored into an array or a field of an object, the analysis uses the local variable index of the object reference. This means that when a spawnable call stores into a field of `this`, the analysis tracks the local variable index of the `this` reference. In case the result is stored into a static variable, the analysis will default to no parallelism.

The analysis phase (discussed in detail in the following subsection) takes as input a spawning method and proposes one or more places to insert a sync statement. The generator phase will then insert a sync statement at those places.

2.4.2 Analysis phase

On the basis of four strategies that we employ in the analysis phase, we discuss the problems involved in automatically generating sync statements. The analysis phase takes as input a spawning method and the spawnable calls with the information provided by the recording phase. The analysis phase returns the instructions in front of which sync statements need to be inserted.

Fallback strategy

The fallback strategy is used when all other analysis fails. It proposes sync statements immediately behind the spawnable calls. This provides us guaranteed correctness as the resulting program is equivalent to the sequential program. However, this also means that there is no parallelism.

```
result1 = spawnableMethod();
result2 = spawnableMethod();

if (someCondition) {
  sync();
  return result1;
} else {
  return result2;
}
```

Figure 2.5: Jumping over a sync() statement when `someCondition` evaluates to `false`.

On-first-load strategy

This strategy increases parallelism by postponing the sync statement until the first load of a variable in which one of the spawnable calls stores. In Fig. 2.2 that would be exactly where the sync is now, on line 11, because `result1` is loaded first.

Unfortunately, this places multiple syncs in case of loops. In Fig. 2.4 this would mean that the sync statement is placed inside the loop between lines 12 and 13. It is preferable to put the sync in front of the for-loop, but it is not a problem since multiple syncs are allowed.

The strategy fails in the example in Fig. 2.5. When `someCondition` evaluates to `false`, there would be no sync and the result would be incorrect.

This can easily be solved by inserting sync statements in front of all loads of local variables in which spawn results are stored. However, there is a larger problem. The correctness is based on the assumption that loads always occur in the instruction range behind the spawnable call. In many cases this will be true, but there may be situations with a backward jump after a spawnable call. The analysis has to be control-flow aware to place sync statements in these situations. However, within basic blocks, where no control-flow occurs, the on-first-load strategy suffices.

Control-flow-aware strategy

The problems discussed above are solved with this strategy. It succeeds in placing the sync in front of both loads in Fig. 2.5 and, for example, just in front of a loop that accesses the result of a spawn (Fig. 2.4).

With help of control-flow information provided by the BCEL library [10], a graph of *basic blocks* is constructed. A *basic block* is a sequence of instructions with only one entry and one exit point. So, within the basic block there is no branch instruction other than the last instruction and no instruction is targeted by any branch instruction except the first. The basic block analysis maintains a successor relation between the basic blocks and determines whether a basic block is an *ending basic block* in the graph. This is the case if the last instruction is a `return` or a `throw` instruction.

A *path* is a sequence of basic blocks, one a successor of the other. An *ending path* is a path where the last basic block is an ending basic block. For every spawnable call in the method the following analysis takes place: find all *ending paths* from the spawnable call on. This includes all possible loops.

Then, for every ending path, the implementation of this strategy tries to construct a *store-to-load path* for every local variable index in which the spawnable call stores. Note that there can be multiple local variable indices when spawnable calls return using exceptions. A *store-to-load path* is a path from the basic block of the spawnable call to a basic block with a load of one of the local variables in which the spawnable call stores.

In this stage, all spawnable calls of the spawnable method have been analyzed, and every spawnable call is associated with one or more store-to-load paths. The implementation retrieves all these paths and removes duplicates. For all these paths, it tries to find a basic block from the end that is not in a loop. The result is now one or more basic blocks in which a sync needs to be placed.

For each of these basic blocks, the on-first-load strategy finds the first load of a variable in which one of the spawnable calls stores, and this instruction is proposed as an instruction in front of which a sync statement should be inserted.

In the case that a spawnable method signature is of type `void`, results of a spawnable call are not read, or exceptions are not handled (there is no catch block), it is not possible to create a store-to-load path. In this case sync statements will be placed at the ends of all ending paths. This means that on every exit from the method, a sync will be placed.

The control-flow-aware strategy contains two additional optimizations:

Unnecessary sync statements It is possible that a spawning method contains multiple exclusive store-to-load paths, but where the sync statement is placed in such a way that it syncs the other paths as well. In this case it is not necessary to insert later sync statements. An example is Fig. 2.6. There is a path *without* the first read on line 7 and then *with* the second read on line 12,

```
 1  for (int i = 0; i < MAX_SPAWNS; i++) {
 2    result[i] = spawnableMethod();
 3  }
 4
 5  sync(); // placed by sync generator
 6  for (int i = 0; i < MAX_SPAWNS; i++) {
 7    readFirstTime(result[i]);
 8  }
 9
10  // sync left out by sync generator
11  for (int i = 0; i < MAX_SPAWNS; i++) {
12    readSecondTime(result[i]);
13  }
```

Figure 2.6: Multiple exclusive store-to-load paths.

which is exclusive from the path *with* the first read and *without* the second read. Without this optimization, the sync generator would also place a sync in front of the third for-loop on line 10. However, because there is already a sync in front of the second for-loop, the second sync is left out.

Array stores and object putfield If a spawnable call stores into an object, the load in a store-to-load path is based on the load of the object reference. In many cases, the load is used to read from the object, which needs a sync statement. However, there are situations in which the load of an object reference is used to store into the object, for example a store into an array or to store something in a field of an object. This does not need a sync statement.

Figure 2.7 shows a store into an array on line 8. The object reference for `result` needs to be loaded. Because of this, without optimization, the sync generator would insert a sync in front of the load. However, this optimization recognizes that the load of the object references is used for a store into an array or object field and places it beyond the load of the object reference on line 10.

Alias-aware strategy

The previous strategy places sync statements optimally in many cases, but the analysis is incorrect when *aliases* to object references are loaded instead of the *original* object references. Figure 2.8 illustrates this. On line 8 and 9, two result objects are created, and on line 10 and 11 new references are pointing to the same objects. The results are stored using references `r1` and `r2` (lines 13

```
for (int i = 1; i < MAX_SPAWNS; i++) {
  result[i] = spawnableMethod();
}

// the reference to object result is loaded,
// but sync is not placed here, because it
// is a store into an array
result[0] = someValue;

sync(); // by sync generator
for (int i = 0; i < MAX_SPAWNS; i++) {
  read(result[i]);
}
```

Figure 2.7: Ignoring array stores.

and 14), but the results are loaded using the aliases `a1` and `a2` (line 16). The `sync()` on line 17 is incorrectly placed.

Alias analysis is typically imprecise [11], but aliasing within a spawning method is not so common in the Satin programming model. Therefore, we take a pragmatic approach and try to detect situations where aliasing can occur.

Aliases introduced *after* the spawnable call has executed can only be introduced by loading the original object reference. The implementation detects this as a load and will insert a sync in front of the load. Therefore, aliases that are introduced after the spawnable call has executed do not need special treatment for providing correctness.

Aliases that are introduced *before* the spawnable call has executed do pose a problem. Spawnable calls that store into parameters of the spawning method form a special case. Aliases to these parameters could have been created before the spawning method was called, and since the implementation has no knowledge of these aliases, it reverts back to the fallback strategy and issues a warning. For aliases that are created within the method, as shown in Fig. 2.8 on lines 10 and 11, are handled by the alias-detection algorithm.

The alias-detection algorithm proceeds as follows: if a spawnable call stores into an object, it is necessary to check for aliases. If the object is a parameter of the spawning method, the analysis stops, gives a warning, and reverts back to the fallback strategy. Otherwise, the analysis uses the control-flow graph to find all predecessors of the basic block that contains the spawnable call. For every predecessor, the analysis determines whether the object is loaded or stored. If the object is loaded, it may be introducing an alias, unless the load instruction

```
 1  class Result {
 2    int v;
 3  }
 4
 5  int spawningMethod() {
 6    if (stopCondition) return 0;
 7
 8    Result r1 = new Result();
 9    Result r2 = new Result();
10    Result a1 = r1;
11    Result a2 = r2;
12
13    r1.v = spawnableMethod();
14    r2.v = spawnableMethod();
15
16    int sum = a1.v + a2.v;
17    sync(); // incorrect, should be on line 15.
18    return sum;
19  }
```

Figure 2.8: Incorrect sync placement due to aliasing within a spawning method.

is used in the following cases: an array load or store, retrieving the length of an array, or putting data into or retrieving data from a field of the object. If the object reference is stored into, it may be introducing aliases unless it is used in the following situations: a *non-escaping constructor*, a cast, creating a one- or multi-dimensional array, or setting the reference to `null`.

A *non-escaping constructor* is a constructor of an object in which the `this` reference does not escape. To verify this, the analysis checks every instruction of the constructor. If the `this` reference is used, it may only be used for putting something into a field or calling the constructor of the super-class. These super-class constructors are verified recursively as well. The constructor of `Object` is non-escaping.

As a result of this algorithm, many situations that are common in Satin programs and that do not introduce aliases are detected, such as array creation, object creation, and putting data into a field. As soon as the object references are used for different reasons, and the analysis cannot guarantee that aliases are not introduced, the sync generator reverts back to the fallback strategy and gives a warning that aliases may have been created and that the sync placement is likely to be sub-optimal (but still correct).

For example, in the Satin programming model, the results of a spawnable call are often stored into an array, for instance similar to Fig. 2.7. As the array often needs to be created first, this means that the object reference is stored into before the spawnable calls are executed. The alias-detection algorithm recognizes that this does not introduce aliases.

It is also common that results are stored in a result object, similar to `r1` and `r2` on lines 8 and 9 of Fig. 2.8. The class is defined on line 1. The alias-detection algorithm recognizes the creation of a new object and performs the escape analysis on the default constructor. Also in this case, it concludes that the statements on line 8 and 9 do not introduce aliases.

However, on line 10 and 11, the object references `r1` and `r2` are loaded but do not comply to the cases outlined above. The alias-detection algorithm concludes that aliases may have been introduced here, reverts back to the fallback strategy, and warns for non-optimal sync placement.

To summarize: the alias-aware strategy cannot optimally place sync statements when a spawnable call stores into object references, except for the cases described above. However, it can detect possible aliasing and warn for non-optimal sync placement. This gives the programmer the opportunity to either restructure the code to reduce possible aliasing, or insert sync statements manually.

2.5 Evaluation

The sync generator is evaluated using 35 pre-existing real-world Satin applications, amongst others a SAT solver [12], N–body simulation [7], Grammar-based text analysis [12], Grammar induction [7], Gene sequence alignment [7], FFT, and Game-tree search [13].

To evaluate the performance of the sync generator, the applications are stripped from sync statements. The applications are recompiled with the Satin compiler that runs the sync generator pass. The resulting bytecode is carefully examined and compared to the original bytecode. The placement of sync statements is compared in relation to the control-flow and variables on which the correctness and amount of parallelism depends.

The placement is said to be optimal, if it is as late as possible in the control-flow and allows all spawnable calls to run in parallel. The placement by the sync generator may be later in the control-flow than the placement of the programmer (but still correct). The sync generator may also place sync statements in such a way that sync is called multiple times. This has a negligible performance effect.

Table 2.1 shows the results. The first column shows the name of the ap-

Table 2.1: Automatic sync statement generation in real-world Satin applications.

application	optimal	alias warning	notes
Adaptive integration	yes	-	programmer placed unnecessary syncs
Awari: game tree search with the mtdf algorithm [13]			
Reference version with transposition tables	yes	-	
Pre-allocated transposition tables	yes	-	
Transposition tables as structure of arrays	yes	-	
Replicated transposition tables with Java RMI	yes	-	
Replicated transposition tables with sockets	yes	-	
No speculative parallelism	yes	-	
Shared objects version	yes	-	
Binomial coefficients	yes	-	programmer placed unnecessary syncs
Checkers			
Reference version	yes	-	
Negamax search with alpha-beta pruning	**no**	**no**	based on false dependency
Fast Fourier Transform (FFT)	yes	-	originally incorrect, corrected by sync generator
Fifteen puzzle, iterative deepening A* algorithm	yes	-	programmer placed unnecessary syncs
Gene sequence alignment [7]	yes	-	
Grammar induction [7]	yes	-	
Grammar-based text analysis [12]	yes	-	
Knapsack	yes	-	

Continued on next page

application	optimal	alias warning	notes
Matrix multiplication			
Standard version	**no**	**yes**	writing into parameter
shared objects version	**no**	**yes**	writing into parameter
N-body simulation (Barnes-Hut) [7]	yes	-	
N-Queens problem			
Reference version	yes	-	
Using exceptions	yes	-	
Using speculative parallelism and aborts	yes	-	
Using speculative parallelism, aborts and thresholding	yes	-	
Non-speculative version, counting total number of solutions	yes	-	
2nd prize winner in the Grids@work 2005 contest [14]	**no**	**yes**	writing into object reference, writing into parameter
Othello: game tree search with the mtdf algorithm [13]	yes	-	
Prime factorization	yes	-	
Raytracer	yes	-	
SAT solver [12]	yes	-	
Text indexing	yes	-	
Traveling Salesman Problem			
Reference version	yes	-	
Shared objects version	yes	-	
Young-brothers-wait version	yes	-	
VLSI cell router (LocusRoute) [15, 16]	yes	-	

plication. Column two shows whether the generated syncs are inserted at an optimal place. In the case that sync statements are not placed optimally, the third column indicates whether the programmer receives a warning due to aliasing. Column four shows some additional notes.

The table does not show whether the applications are correct, because this is true for all the applications. Correct means that the applications with automatically inserted sync statements give the same result as the original applications.

The overall result is that the sync generator is able to compute an optimal sync placement for 31 out of the 35 applications. In the application FFT, the sync generator inserted correct sync statements that were missing in the original incorrect application. We also tested the sync generator on 12 additional test applications, such as a Fibonacci application and a hello world application. Because the placement is optimal in all cases and these applications are not real-world applications, we do not discuss these applications.

There are some applications that print the timing measurements incorrectly with automatically inserted syncs. Programmers often measure execution time behind the sync statement, but the sync generator postpones the sync beyond this measurement. We do not regard this as a problem as programmers will place the timing statements differently when relying on the sync generator.

2.5.1 Evaluation per application

To get more insight in the results, we discuss the applications separately. The applications that have optimal sync placement and no further problems are omitted.

Adaptive integration, Binomial coefficients, Fifteen puzzle In these applications the programmer placed syncs where the sync generator leaves them out. The sync generator is correct and these syncs can be safely left out. This shows that unnecessary sync insertion is no problem.

Checkers (negamax version) This application is the only application that does not have optimal placement and has no warning. Figure 2.9 shows the cause. The method `srch()` has a variable `beta_cutoff` that has value 0 initially. Depending on this value, the control-flow breaks out of the for-loop (line 19). The programmer knows that this variable is not important for the result, only for performance. The sync generator must regard the variable `beta_cutoff` as a variable in which some value is stored inside the catch block (line 13). This

```
 1 void srch() throws Result {
 2   int beta_cutoff = 0;
 3
 4   for (x = 0; x < count; x++) {
 5     // sequential work
 6     try {
 7       spawn_srch();
 8     }
 9     catch (Result res) {
10       killer[p.ply] = move_list[res.choice_ix];
11       // other stuff
12       if (-res.score >= p.beta) {
13         beta_cutoff = 1;
14         abort();
15       }
16       return;
17     }
18     // sync generator's sync
19     if (beta_cutoff != 0) break;
20   }
21   // programmer's sync
22 }
```

Figure 2.9: Skeleton of method `srch()` in Checkers (negamax version).

means that before this variable is loaded, there should be a sync statement. Unfortunately, this leads to sync placement right behind the try and catch blocks of the spawnable call `spawn_srch()` within the for-loop on line 18, resulting in sequential execution of this application. To compare, the programmer placed the sync on line 21, outside of the for-loop which does result in parallelism.

FFT The sync generator inserts more statements than were inserted by the programmer. Surprisingly, the original application was incorrect. The sync generator corrected it. This error was most probably introduced after a refactoring of the code.

Matrix multiplication (both versions), N-Queens (contest version) These three applications are affected by possible aliasing. The two matrix multiplication applications write into a parameter of the spawning method. Therefore, only the first four of eight calls of the application are executed in parallel. The N-Queens (contest version) application stores into an object that is used by

```java
int spawningMethod() {
  Result result1 = createResult();
  Result result2 = createResult();

  result1.x = spawningMethod();
  result1.doSomethingHarmless();

  result2.x = spawningMethod();

  //sync(); preferred place
  return result1.x + result2.x;
}
```

Figure 2.10: Situation not optimally analyzed.

a different spawn and that is also a parameter. Because the sync generator cannot assure that there are no aliases, syncs are placed conservatively taking away much of the parallelism. However, the sync generator warns for non-optimal sync placement in all three cases. This allows the programmer to restructure the code or insert sync statements manually. Methods that already contain sync statements are ignored by the sync generator.

2.6 Discussion

The topic of this chapter is to investigate whether sync generation can be made automatic in such a way that the resulting program is correct, and to what extent the sync placement is optimal. Another question is with what kinds of analysis we can solve this problem.

The evaluation shows that all applications remain correct after the sync generator inserted sync statements automatically. In many cases (31 out of 35 real-world applications), the automatic sync generation resulted in optimal sync placement. Three out of 35 applications suffered from sub-optimal sync placement because of aliasing, but the sync generator was able to warn for this. Analyzing one version of the Checkers application, the sync generator was unable to warn for sub-optimal sync placement, despite the fact that the sync placement resulted in sequential code.

To obtain these results and to deal with many cases that are common in Satin programs, it is not enough to only have an analysis that is control-flow aware.

```
 1  Matrix mult(Matrix a, Matrix b, Matrix c) {
 2    // other code
 3
 4    Matrix f_00 = mult(a._00, b._00, c._00);
 5    Matrix f_01 = mult(a._00, b._01, c._01);
 6    Matrix f_10 = mult(a._10, b._00, c._10);
 7    Matrix f_11 = mult(a._10, b._01, c._11);
 8
 9    c._00 = mult(a._01, b._10, f_00);
10    c._01 = mult(a._01, b._11, f_01);
11    c._10 = mult(a._11, b._10, f_10);
12    c._11 = mult(a._11, b._11, f_11);
13
14    return c;
15  }
```

```
WARNING: The result of the spawn at line 9
  is stored in an object given as parameter
  to the spawning method. This case is not
  handled by the sync generator. The
  resulting sync placement is likely not
  optimal.
WARNING: The result of the spawn at line 10
  is stored in an ... etc.
```

Figure 2.11: Matrix multiplication that stores into a parameter and the warnings given by the sync generator.

Alias analysis is also necessary, but it is sufficient to restrict the analysis to alias-detection with extensions that are relatively straightforward to implement, such as analysis of array creation and an escape analysis on constructors.

The following subsection will discuss how the precision of the alias analysis affects the overall precision. Section 2.6.2 shows what programmers can do when they receive aliasing warnings from the sync generator. We discuss the applicability of our analysis to Cilk programs in Sec. 2.6.3 and finally, we discuss making Satin more general to support futures.

2.6.1 Precision of the Alias-Analysis

Although Java restricts aliasing more than languages such as C, it is still relatively straightforward to write code that contains many potential aliases. There-

```
Matrix mult(Matrix a, Matrix b, Matrix c) {
  // other code

  Matrix f_00 = mult(a._00, b._00, c._00);
  Matrix f_01 = mult(a._00, b._01, c._01);
  Matrix f_10 = mult(a._10, b._00, c._10);
  Matrix f_11 = mult(a._10, b._01, c._11);

  Matrix l_00 = mult(a._01, b._10, f_00);
  Matrix l_01 = mult(a._01, b._11, f_01);
  Matrix l_10 = mult(a._11, b._10, f_10);
  Matrix l_11 = mult(a._11, b._11, f_11);

  c._00 = l_00;
  c._01 = l_01;
  c._10 = l_10;
  c._11 = l_11;

  return c;
}
```

Figure 2.12: Matrix multiplication that is restructured to minimize aliasing based on the warning given in Fig. 2.11.

fore, we expected that aliasing would give problems in some applications. This chapter gives an indication to what extent this is a problem.

More precise aliasing analysis will make the complete analysis more precise. Figure 2.10 illustrates this. Due to possible aliasing and loading of `result1` on line 6, the sync generator will not place the sync on line 10 where we want it. We could extend the analysis to verify that no aliases are created in `createResult()` on lines 2 and 3. We could also analyze `doSomethingHarmless()` on line 6 to verify that it does not load variable `x`. However, this would make the analysis much more complicated, as it has to extend the analysis to different methods that may be part of other classes.

Extending the alias analysis to make it more precise, however, will not solve the problem in the Checkers application. No matter how precise the alias-detection is, the problem in Fig. 2.9 can never be solved. The `beta_cutoff` variable is a variable that is written inside the catch block and is inserted deliberately by the programmer for performance reasons. However, the analysis will never be able to determine whether this write is relevant for a correct result or not, which led to sequential code in the Checkers application. This case is a good example of compilers lacking application-specific knowledge that programmers have. The conclusion is that there may always be applications that do not have optimal sync placement, even if the alias-detection would be very precise.

Spawnable calls that store into objects that are parameters of the spawning method also form a problem. Figure 2.11 shows a sketch of the problem of Matrix multiplication. On lines 9, 10, 11, and 12, the spawnable calls store into `c`, which is a parameter of the spawning method (line 1). For the subsequent spawnable calls, the sync generator cannot verify that `a` or `b` is not an alias of `c` (aliases between objects in argument lists are preserved during serialization). Therefore, the sync generator needs to place a sync statement behind each spawnable call from line 9 on.

This problem occurs in both of the Matrix multiplication versions and in the N-Queens problem. The sync rewriter would be able to place sync statements optimally in these cases as well, if the Satin programming model would be somewhat more restrictive. If the Satin programming model would disallow aliases between parameters of spawnable calls, which is not the case, these applications would also have optimal sync placement.

2.6.2 Improving Sync Generation with Programmer Support

Although aliasing restricted the parallelism in three cases, the alias-detection algorithm was able to give very precise warnings about the problems in the code. Programmers then have the choice to restructure the code to eliminate the aliasing problems or to insert sync statements manually. In the latter case, the sync generator will recognize that the spawning method already contains sync statements and will move on with the analysis.

Figure 2.11 shows the warnings that are given by the sync generator in the Matrix multiplication programs. The warnings indicate that on lines 9, 10, 11, and 12 the spawnable calls store into an object that is a parameter (`c` in this case).

Programmers can restructure the code with help of these warnings. Figure 2.12 shows a possible solution. Instead of storing the results of spawnable calls immediately in a parameter, programmers can decide to store the results in local variables. Using this simple technique, the sync generator is able to place sync statements optimally.

```
1  cilk void bucket_sort(float *array, int left, int right) {
2
3    if (right - left >= BASE) {
4      int mid = (left + right) / 2;
5      spawn bucket_sort(array, left, mid);
6      spawn bucket_sort(array, mid + 1, right);
7    }
8    ...
```

Figure 2.13: Part of a Cilk bucket sort test application in which parameters are used to return data.

2.6.3 Cilk

In principle, the analysis we describe in this chapter can be used to analyze Cilk [9] programs as well. However, due to the use of C as a base language, the implementation needs to take into account unrestricted aliasing (aliasing to variables of basic types, casting) and global variables which are more common on shared memory machines. In addition, our analysis operates on bytecode and does not need the source code. A similar binary rewriting approach would be difficult with Cilk code.

Comparing Satin applications with Cilk applications and imposing a similar analysis on the Cilk applications reveals that the Satin programming model has an important assumption that helps the analysis. Figure 2.13 shows a piece of a bucket sorting test application from the Cilk distribution. Lines 5 and 6 show two spawnable calls. The `bucket_sort` function is of type `void` which means that the result is returned via the parameter `array` (line 1).

A similar approach is not possible in Satin, because the array may be serialized to be sent to other nodes in the grid. If we would apply the analysis proposed in this chapter on this Cilk application, the analysis would not be able to determine whether the result that is returned in the `array` parameter in line 5 is needed in the spawnable call in line 6. It would therefore place a sync behind both spawnable calls, which would result in sequential execution. Therefore, applying the same analysis on Cilk programs would be much more difficult, because it would need a very detailed view on aliasing relationships between variables. Due to the unrestricted aliasing in Cilk which would make such a detailed view even harder, the analysis would probably result in less impressive results than with Satin programs.

2.6.4 Futures

The sync statement in Satin is a local barrier that waits for *all* spawnable calls to be completed. However, for instance the Matrix multiplication application could benefit from syncing on a specific spawn. To make the first part of Fig. 2.12 correct, there needs to be a sync statement on line 8 that waits for all four spawnable calls on the preceding lines. However, it might be the case that the first spawnable call on line 4 finishes first, which would mean that the spawnable call on line 9 could be started because it depends on `f_00`. If Satin would provide a means to sync specifically on the spawnable call on line 4, it might give a slightly more efficient execution. This mechanism would then resemble futures [17]. This might make Satin more general and perhaps more powerful, but the recursive tasks that are now used provide the basis for creating a task hierarchy that takes into account data locality. The hierarchical nature of grids allows an efficient mapping of the hierarchical Satin programs to grids [18]. Satin might lose this ability if we would make the programming model more general with futures.

Although we consider extending Satin with sync statements for specific spawnable calls future work, we have our reservations because in our experience, it is precisely the restrictions on the Satin programming model that makes the model successful. This is underlined by the efficient execution of Satin applications on grids [18] but also by our sync generation analysis that is in part possible because of the restrictions of the Satin programming model. Moreover, if we would restrict the Satin programming model even more by disallowing aliases in parameter lists, we would be able to insert synchronization statements optimally in 34 out of 35 real-world applications.

Overall, the main contribution of this chapter is that it provides a thorough understanding of the problems that involve generating synchronization statements automatically. Moreover, we provide a real implementation in the form of a compiler pass for the Satin compiler that makes creating parallel divide-and-conquer programs for grids even more effortless than before.

2.7 Programming model design considerations

There are several issues that stand out in this chapter and should be considered when designing a parallel programming model.

Aliasing Firstly, aliasing has severe ramifications for analysis of programs. This chapter needs aliasing analysis and from the comparison with Cilk in Sec. 2.6.3 it is clear that unrestricted aliasing has impact on the feasibility of the analysis.

Restrictions Secondly, restrictions on the language have a positive effect on the feasibility of the analysis. Our analysis is helped by the fact that Java restricts aliasing more than Cilk and also by the fact that Satin is more restricted than Java in returning values through parameters. Additionally, if Satin would have restrictions on aliasing between parameters in spawnable calls, the analysis in this chapter would perform better. Finally, the analysis is also possible because Satin has all input classes available. This would allow us to do even more precise alias analysis.

Adapting to compilers Section 2.6.2 touches an interesting issue. It explains a strategy to adapt the program such that a compiler understands the program better. This sounds like a good idea, but it also shows a slippery slope. The question becomes what the platform is that the programmer is programming for. Are programmers programming hardware or are they programming a compiler? Do the programmers have to know the inner workings of the compiler to adapt their program to the compiler? This is an important design consideration for programming models.

Compiler/programmer interaction Finally, Sec. 2.6.1 shows that compilers need to be conservative to guarantee correctness. The compiler lacks a small piece of information, namely that the variable `beta_cutoff` is not important for correctness. However, programmers typically have this piece of information, this application knowledge. On the other hand, the evaluation showed that programmers are not infallible and make mistakes. Our conclusion is that parallel applications may benefit from a compiler and programmer that interact more closely together and use each other strengths.

2.8 Related Work

Satin [7] is closely related to Cilk by Blumofe et al. [9]. Both systems provide a divide-and-conquer programming model with asynchronous spawnable calls and a local barrier synchronization primitive called *sync*. Cilk differs from Satin in the base language. Where Cilk extends C and C++ with keywords such as `spawn` and `sync`, Satin uses the Java type and class systems to make similar annotations. As a result, a Satin program is still a valid Java program, whereas a Cilk program is not a valid C or C++ program. Another difference is that Cilk targets multi-core architectures, whereas Satin targets execution on clusters and grids using the Ibis system [8] that provides communication primitives to nodes on a grid.

We are unaware of any previous work on trying to make sync statements implicit in a divide-and-conquer model. Implicit sync statements show resemblance with implicit futures, introduced by Baker and Hewitt [17]. A future evaluates an expression in a separate thread or process. The original process can perform other work and blocks when it tries to use the expression of the future until the result is computed. Flow Java [19] is a variant of Java that implements futures through the use of single assignment variables. The difference between implicit syncs and implicit futures is that a future can block on a specific spawn, whereas a sync blocks for all spawns in the method.

The sync generator uses several well known techniques to decide where to place sync statements. We are interested in which variables a spawnable call stores and where these variables are loaded again. The analysis can be incorrect due to aliasing of variables before a spawnable call is called. We perform simple alias-detection, but the analysis could be made more precise through more advanced alias analysis, for example with help of the Spark framework [20]. In the context of Java, a type-based alias analysis would be a good choice [21], opposed to analysis that can handle unlimited pointer access [22]. In addition, we perform a simple escape analysis that can also be made more precise, for example by implementing work of Whaley and Rinard [23] or Blanchet [24].

2.9 Conclusion

Satin provides a divide-and-conquer programming model to execute applications efficiently on clouds and grids. Programmers annotate recursive calls to be spawnable so that these calls are executed in parallel on the grid. Sync statements indicate where the system has to block to wait for the result. In this chapter we discussed a sync generator that automatically generates these sync statements.

We conclude that it is possible to automatically insert sync statements in Satin code in such a way that every resulting program is correct. Correctness in this sense is that the resulting program that is run on a grid will deliver the same result as the sequential version. The fact that we provide correctness is well illustrated by a previously incorrect application (FFT) that was corrected by the sync generator.

In many cases the sync generator will be able to place sync statements optimally. An optimal place in this sense is that as many independent asynchronous spawnable calls as possible are called within a spawning method. To perform the analysis, the sync generator needs to be control-flow and alias aware.

We evaluated the sync generator using 35 pre-existing real-world applications and an additional 12 test applications. The sync generator finds an optimal place in all tests and in 31 out of 35 real-world applications. For three applications, the sync generator does not place syncs optimally due to possible aliasing. However, these applications still have some parallelism and the sync generator also warns for non-optimal placement. One application is completely sequentialized by the sync generator due to a false dependency, deliberately introduced by the programmer, that does not affect the result of the computation but only the performance. However, no analyzer can dismiss this false dependency and place sync statements in such a way that more parallelism is enabled.

The current analysis could be made more precise by extending the analysis to methods that are called from the spawning method. However, this analysis also cannot disregard false dependencies as above.

Although this chapter is not about many-core hardware, it still led to valuable considerations for the design of MCL and Cashmere and as such gives answers to the sub-question:

1. What are important design considerations for parallel programming models and their compiler analyses?

Aliasing is an important issue and restrictions in the language may be beneficial to limit these kinds of problems. This chapter also shows that compilers can become the platform: programmers adapt their programs so that the compiler can do proper analysis, whereas for many-cores it may be preferable to target actual hardware instead of the compiler. Finally, this chapter shows that compilers have to be conservative and therefore miss out on optimizations, while programmers having better application knowledge can make better decisions.

Chapter 3

Stepwise-refinement for performance: a methodology for many-core programming

Many-core hardware is targeted specifically at obtaining high performance, but reaching high performance is often challenging because hardware-specific details have to be taken into account. Although there are many programming systems that try to alleviate many-core programming, some providing a high-level language, others providing a low-level language for control, none of these systems have a clear and systematic methodology as foundation.

Based on the design considerations we found in Chapter 2, we propose *stepwise-refinement for performance*: a novel, clear, and structured methodology for obtaining high performance on many-cores. We present a system that supports this methodology and offers multiple levels of abstraction to provide programmers a trade-off between high-level and low-level programming. The system contains a compiler that gives programmers detailed performance feedback. We evaluate our methodology with several widely varying compute kernels on two different many-core architectures: a GPU and the Xeon Phi. We show that our methodology gives insight in the performance and that in almost all cases we gain a substantial performance improvement using our methodology.

3.1 Introduction

The high performance that many-cores offer makes them a compelling target for the growing performance needs in industry and science. However, obtaining high performance – the main purpose of many-cores – is challenging as hardware-specific details need to be taken into account, such as multiple levels of parallelism, explicit fast memories, and memory access patterns.

Languages such as CUDA and OpenCL offer much control over hardware-specific details. However, managing all these hardware-specific details to obtain high-performance is a complex task. Manually optimized programs are usually difficult to read and do often not portray well why they obtain high performance.

A common approach to simplify many-core programming is to raise the level of abstraction. However, whereas a high abstraction-level simplifies programming in *general purpose computing* where performance is less critical, in *many-core computing* it often results in hiding the very details that are important for performance.

In our view, this tension between control over hardware-specific details and raising the level of abstraction is not supported by a proper methodology in current work. In this chapter we present a methodology that we call *stepwise-refinement for performance*. This methodology presents programmers a structured approach in which they can start on a high-level and can then, if they desire, move on to lower levels of abstraction that expose more hardware details. We can illustrate this with a quote from Alan J. Perlis: “A programming language is low-level when its programs require attention to the irrelevant.” In our experience, the difficult issue in many-core programming is deciding which details are relevant for performance. Our methodology aims to assist programmers with this decision.

Inspired by the observations made in Chapter 2, we made the interaction between the compiler and programmer to obtain high performance a crucial aspect of the methodology. The compiler is augmented with hardware knowledge with varying levels of detail resulting in multiple levels of abstraction. Using this hardware knowledge, the compiler assists the programmer by providing detailed *performance feedback*. The goal of our methodology is to structure the optimization process, give programmers a *trade-off* between high-level and low-level programming, and give programmers *understanding* of the performance they obtain.

We evaluate our methodology with Many-Core Levels (MCL), a system that supports multiple abstraction-levels for multiple many-core architectures. Our system has knowledge of many-core hardware by means of hardware descriptions encoded in our Hardware Description Language (HDL). Compute kernels are expressed in Many-Core Programming Language (MCPL) that makes the mapping between algorithm and hardware description explicit. Our compiler can generate code for each level of abstraction and combines the knowledge of the program and the hardware to provide detailed *performance feedback* to the programmer who can then *manually* transform the program.

This approach was inspired by the conclusions from Chapter 2 and combines the strengths of both the compiler and the programmer. On the one hand, programmers gain insight in the compiler, remain in control, and can use their application knowledge to transform the program. On the other hand, the compiler does not have to be as conservative in providing feedback as an automatically optimizing compiler has to be.

To show that our approach is effective, we implemented several widely varying and well-known compute kernels in MCL for two very different many-core architectures: an NVIDIA GPU and Intel's Xeon Phi. In almost all cases our approach results in substantial performance improvements over our highest-level code. We show that programmers in cooperation with our compiler have enough control to obtain high performance and gain an understanding of the performance during the stepwise-refinement process that leads the programmer through multiple abstraction-levels.

To summarize, this chapter presents the following contributions:

- A clear methodology for optimizing many-core programs,
- our system MCL that implements this methodology, released as open source [25],
- an evaluation of the methodology with several widely different many-core programs on two different many-core architectures,
- several implementation techniques that exploit the strong relation between hardware description and program, and
- answers to research questions 2, 3, and 4 of this thesis.

Section 3.2 elaborates how various many-core programming approaches relate to MCL. In Sec. 3.3 we introduce our methodology *stepwise-refinement for performance*. Section 3.4 gives an overview of Many-Core Levels and how our system implements our methodology. In Sec. 3.5 we give a detailed example of how the process of *stepwise-refinement for performance* takes place. Section 3.6 discusses several of the implementation techniques of our system. Section 3.7

Table 3.1: Comparison of several programming approaches.

approach	control	portability	understanding	high-level
high-level programming	-	+	-	+
separation of concerns	-	+	-	+
tuning cycle approach	+	-	+	-
MCL	+	(choice)	+	(choice)

evaluates our techniques for various well-known compute kernels. We conclude the chapter with a discussion and conclusion.

3.2 Related work

The challenges in many-core programming are widely recognized and there are many approaches that try to alleviate it. This following section discusses the current status of programming many-cores and identifies issues (summarized in Table 3.1) that we try to address in our work. We distinguish three programming approaches: high-level programming, separation of concerns, and a tuning cycle approach. Section 3.2.2 discusses systems that influenced MCL.

3.2.1 Programming Many-cores

High-level programming The fact that low-level many-core programming is difficult is a well-recognized problem and several systems raise the level of abstraction to make many-core programming more accessible. Often, these systems center around a main abstraction, such as streaming [26, 27], Bulk-Synchronous Parallelism [28, 29], Nested Data-Parallelism [30, 31, 32, 33], divide-and-conquer [34], or powerful arrays [35, 36, 37, 38, 39, 40, 41].

Automatic optimization and algorithmic skeletons are two means to obtain performance from high-level programs. Automatic optimization relies on the premise that high performance can be obtained in all cases. This is difficult to realize in practice, because compilers have to work in the general case, have no application knowledge, and have to be conservative (e.g. assume aliasing). In general-purpose computing, since the focus is not on high-performance, sub-optimal performance may not be a problem, but it is an issue in many-core programming because obtaining high performance is the main goal.

There are two examples of automatic optimizers that optimize naively written GPU kernels [42, 43]. Although this is valuable work, a drawback of automatic optimizers is that the optimization knowledge is contained in the compiler and cannot be reused by programmers for applications that are less amenable to automatic optimization.

A second means to obtain performance from a high-level programs is to provide a programming model in which programmers express their algorithms in terms of algorithmic skeletons [44, 45, 46]. The skeletons are often manually implemented and optimized. These skeleton-based programming models rely on programmer insight to select the right parallel patterns for the application. Additionally, performance often relies on the composition of these algorithmic skeletons, which is a challenging transformation [47, 48], but has been applied to GPU programming [49].

Although raising the level of abstraction often provides a cleaner programming model and helps to achieve (non-performance) portability, it also means that hardware-specific details are hidden that may be necessary for obtaining high performance. This makes understanding why a particular program has a particular performance more difficult and in case an application performs less than expected, programmers have fewer means to adapt the code to gain more performance.

Separation of concerns Another approach is based on separation of concerns. Delite [50] and the work by Cartey et al. [51] provide frameworks for building DSLs (Domain-Specific Languages) on top of a performance library to separate the concerns between domain-experts and performance-experts. This is an interesting approach that can be tailored to the needs of domain-experts. However, building DSLs is a difficult task, especially when two different goals need to be addressed, namely making them expressive for the domain-experts and making the generated code amenable to optimizations. Besides, a DSL provides the domain-experts, the ones requiring performance, no understanding of and no control over the performance. Additionally, the performance depends on good communication between the performance-experts and the domain-experts.

Tuning cycle approach The tuning cycle approach is an iterative process that usually consists of the following steps: evaluate the performance of an application, analyze the gathered results, and refactor the code to increase the performance. This approach can be applied to directive based systems such as OpenACC [52, 53], becoming an industry standard, and others [54, 55, 56, 57].

In each iteration more detailed directives can be inserted to attempt to increase the performance. However, this approach usually suits low-level languages such as CUDA [58] or OpenCL [59] better because these languages offer programmers more control in restructuring the program to increase the performance.

The first step in this process, evaluating the performance of the application, can be done in several ways. The performance can be measured using profilers or it can be estimated using performance models. Lopez-Novoa et al. conducted an excellent survey on performance modeling for many-cores [60].

The second step, analyzing the results, can be very challenging. Profilers usually give feedback in the form of statistics about the code, which makes it difficult to understand how to act on the feedback. Some tools, however, help programmers to interpret the result. PerfExpert [61], for instance, does support many-core architectures but only gives feedback about which code-parts of an application could run well on a Xeon Phi or GPU [62, 63]. It is not capable of optimizing those kernels taking into account non-standard architectural details, such as warp execution or scratchpad memories. PerfExpert operates by interpreting the results from measurements, matching it against rules about the architecture to find a bottleneck in the program and recommends a list of optimizations, complete with code patterns that may solve the bottleneck.

In the final step, the code is refactored to increase the performance. Often this is done manually by the programmer, but PerfExpert can in some cases also apply the optimizations automatically [64].

This approach is performed on the lowest level of abstraction, which makes the applied optimizations not portable to other architectures. This is a major disadvantage considering the rapid evolution of many-core hardware. In addition, optimizations in low-level languages make the code difficult to understand.

MCL Table 3.1 presents an overview of the discussed programming approaches in relation to our work. In comparison to *high-level approaches* and *separation of concerns*, MCL also provides a high-level of abstraction, but offers programmers control by allowing lower levels of abstraction, although this diminishes the portability. MCL is specifically targeted at providing programmers understanding of the performance they obtain and MCL offers a choice in the level of abstraction to work on.

In relation to the *tuning cycle approach*, tuning is often performed only to the lowest level of abstraction, where programmers have much control. This makes the optimizations not portable to other many-core architectures. In contrast, MCL allows to write optimizations on each level of abstraction, making the optimization available to all lower levels of abstraction in the sub-tree of the hardware-description hierarchy (Sec. 3.5 gives an example of this).

Additionally, low-level languages do often not provide enough language features to express optimizations clearly in relation to the underlying hardware. For example, CUDA and OpenCL have *explicit* constructs for low-level hardware features, such as accessing fast on-chip memory, but other important hardware features remain *implicit*, such as parallel banks in accessing memory, or GPU threads that execute in warps, a pipelined manner to overcome memory latency.

Since not all hardware features are explicitly part of these languages but are important for performance, in our experience, optimizing programs often leads to code that does not explain well why certain optimization decisions were made. Examples are data structures with a non-standard layout and loops that iterate in a particular manner for improved memory access. This makes implementing suggestions from analysis tools difficult because programmers cannot relate the code to the underlying hardware. MCL provides more means to express the optimizations in relation to the hardware, because hardware descriptions in MCL are tightly coupled to the programming language.

Furthermore, because optimizations are applied on multiple levels of abstraction, the optimizations are also traceable. It is clear which optimizations have been applied for which hardware and it is also clear on which feedback the optimizations are based.

MCL mainly relies on static information encoded in the program and hardware descriptions, which allows a fast development cycle for experimentation. In contrast, profilers need several runs to record data or runs with instrumented code for tools such as PerfExpert, which results in a longer development cycle. However, since run-time information is more accurate than static information, we consider this type of analysis as very valuable and in alignment with MCL. However, in MCL run-time analysis is typically performed at the lowest-level of abstraction where static information is no longer precise enough and where the longer development cycle can be traded-off with performance.

In summary (as can be seen from Table 3.1), MCL is a different and novel choice in the design space of programming approaches for accelerators, favoring ‘control’ and ‘understanding’ by the programmer, and allowing the programmer the freedom to work on different levels of abstraction and portability.

3.2.2 Other related Work

There are several approaches from which our work draws inspiration. NVIDIA’s Thrust library [65] provides a C++ Standard Template Library-like interface with a vector data-structure. Thrust is tightly integrated with CUDA which makes it possible to replace performance critical Thrust code with specialized CUDA functions. This programming model advocates a methodology, albeit a simple one: prototype with Thrust, rewrite the hot-spots by falling back to CUDA and optimize these kernels.

Sequoia introduces tasks that recursively call each other as main programming abstraction [66]. It also provides user-defined descriptions of memory hierarchies that define how different task variants are mapped to the memory hierarchy. In comparison to Sequoia, our system generalizes the memory hierarchy descriptions to hardware descriptions, does not depend on task abstractions, provides a more direct mapping between algorithm and hardware than Sequoia does, and offers multiple levels of abstraction.

MCL's hardware description language is inspired by Aspen, a performance modeling language [67]. Aspen describes high-level properties of compute kernels with hardware of cluster computers. In MCL we focus on describing many-core hardware instead of cluster computers and we do this on a lower level. Our approach also differs in that we do not describe properties of compute kernels, but instead explicitly define the mapping between the algorithm and hardware constructs.

In general our hardware descriptions could be seen as models of the hardware that have certain performance characteristics. In this sense our work relates to performance modeling. In the context of distributed systems, Ammar [68] worked on hierarchical performance models with multiple abstraction-levels. At the highest level they describe the performance characteristics and requirements of a distributed system at the application level. At lower levels, the performance characteristics of interacting software components are modeled with increasing precision, resulting in an accurate performance model. While Ammar's hierarchy models performance characteristics of software components, our hierarchy entails multiple abstractions for hardware.

3.3 Stepwise-refinement for Performance

From the previous section we can distill several requirements for many-core programming. First, there is a need for low-level programming languages that provide programmers control over the hardware. Solutions differ in how explicit this control is in the language. Second, architectures of many-cores change rapidly which makes portability an important issue. A third important aspect is understanding performance. Tools such as profilers and PerfExpert try to fill this gap. Furthermore, there is also a need for high-level programming that abstracts away hardware-details. However, this comes at the cost of loss of control. Finally, none of the approaches has a good solution for handling the tension between control over hardware and raising the level of abstraction, and shows support for the optimization process over more than one abstraction level. This is an important point because optimizing is a laborious process which often leads to code that is difficult to read and maintain. Often, it is also unclear why exactly these codes perform as well as they do.

In this section we propose the *stepwise-refinement for performance* methodology, a structured approach that combines all of these requirements to help the programmer to obtain high performance. For our methodology, we assume an existing application with computational hot-spots or kernels that are to be ported to a many-core device. Such a computational kernel is the starting point of our methodology.

3.3.1 Philosophy

Our *stepwise-refinement for performance* methodology addresses the tension between low-level programming that offers control and raising the level of abstraction. It gives programmers a trade-off between high-level programming which is good for portability and code maintainability, and low-level programming that gives programmers a clear, explicit, and well-defined interface to hardware features that are necessary for performance. To summarize, *stepwise-refinement for performance* provides programmers control and understanding on levels of abstraction they can choose themselves.

Inspired by the observations in Chapter 2, our approach relies on cooperation between the compiler and programmer, unlike relying on black-box compilers that automatically optimize. The compiler is extended with knowledge about many-core architectures and combines this with the knowledge about the program to give detailed performance feedback. The programmer acts on the feedback and rewrites the program to obtain higher performance. This approach combines the strengths of both the compiler and programmer. The compiler does not have to be as conservative in providing feedback as it would normally be in applying transformations, whereas the programmer has application knowledge that the compiler lacks and gains an understanding of the performance of the program.

In each stage during the iterative process that our methodology advocates, a part of a program is rewritten to incorporate the feedback from the compiler. The different versions of the program that arise as a result of this process in

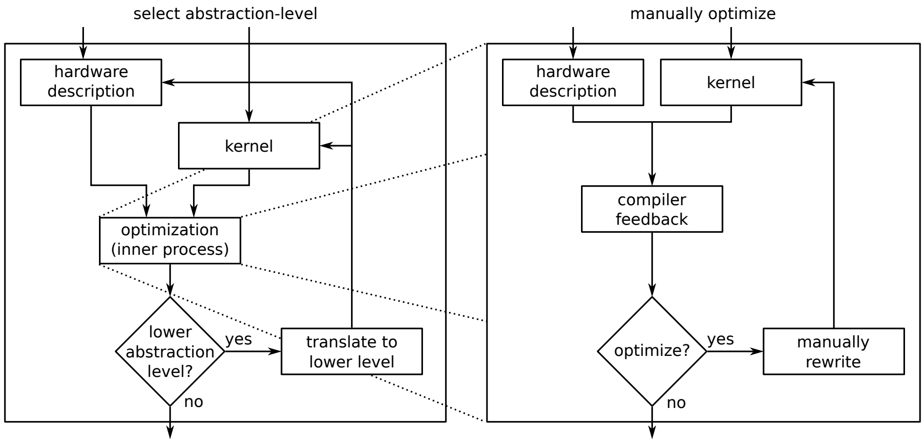


Figure 3.1: A diagram of the iterative processes. At the left is the outer process, at the right the inner process.

effect capture the reasoning and knowledge about the optimizations.

3.3.2 Methodology

The *stepwise-refinement for performance* methodology contains two intertwined iterative processes where in each stage a program is rewritten to incorporate feedback from the compiler. The outer process centers around moving from a high abstraction-level to a lower abstraction-level. Hardware descriptions defined with different levels of detail effectively create multiple levels of abstraction.

Figure 3.1 shows that the outer process at the left has as input a hardware description and a computational kernel. With these inputs, the iterative inner process is started (discussed below). After this process has finished, the programmer has to decide whether to move to a lower level of abstraction to introduce more hardware details, or to stop, for example to remain portable among architectures.

If a programmer decides to move to a lower level of abstraction, the kernel has to be translated to a version that adheres to the rules of the lower-level hardware description. This process may be automated, but does not necessarily result in a kernel that has higher performance. It merely results in a kernel that represents the more detailed hardware features defined in the lower-level hardware description. The outer process restarts with this new kernel and its lower-level hardware description that serve as input for the inner process. If a programmer decides to not move to a lower level of abstraction, the outer process stops. The resulting kernel can now be further processed to be incorporated in an application.

The outer process gives programmers a trade-off in the level of abstraction for the inner process that focuses on optimizing the kernel. For example, there could be a hardware description that defines a generic GPU independent of vendor. The hardware description at the next level of abstraction could introduce hardware features that are vendor-specific. At this point, the programmer may decide to stop at this level to remain portable between GPU vendors.

The inner process at the right in Fig. 3.1 has as input a hardware description and a kernel and focuses on optimizing the kernel at the current level of abstraction. Applying the lessons learned from Chapter 2, instead of automatically optimizing the code, our approach gives the compiler a different role, namely providing the programmers with detailed performance feedback based on analysis of the kernel and the hardware description.

The role of the programmer is to decide whether the feedback is useful and can lead to better performance. If this is the case, the programmer manually rewrites the kernel to incorporate the feedback from the compiler. This rewritten kernel can then be reevaluated by the compiler to generate additional feedback. If the programmer decides that no further optimizations are necessary at this specific level of abstraction, the inner process stops after which the outer process continues giving the programmer an option to move to lower levels of abstraction.

3.4 Design of MCL

The previous section discussed our methodology without any implementation details. This section introduces Many-Core Levels (MCL) and discusses our choices in implementing the *stepwise-refinement for performance* methodology. MCL has been released as open source [25].

There are several consequences for programming systems that support the *stepwise-refinement for performance* methodology. First of all, compilers usually have a view of how hardware behaves, but to support this methodology, the compiler has to be able to interpret hardware descriptions with varying levels of detail to support the multiple abstraction-levels. Second, the compiler has to combine the knowledge from these hardware descriptions with the knowledge about the program to generate feedback for the programmer.

There are also consequences for the programming language. It has to show how the program is mapped to the hardware, so that the compiler can reason about the mapping. Moreover, as we have seen in Chapter 2, the programming language may have to be more restricted to allow the compiler to give feedback. Finally, for the feedback to be useful, it has to be closely related to the code that the programmer wrote and not to already optimized machine code, as is traditionally done.

In MCL, we consider a program as a mapping of an algorithm to hardware. Since hardware-specific details are so important for performance, we make the mapping explicit in the programming model. Therefore, MCL introduces hardware descriptions in the programming model. The hardware descriptions are user-defined and can describe hardware with different levels of detail.

Usually, a programming language exposes a machine model to the programmer through the abstractions it defines. For example, OpenCL exposes the abstractions of local memory and global memory to the programmer even though those memories may be represented by the same physical memory (in case of CPUs) or separate physical memories (usually the case in GPUs). So, the programming language defines abstractions that may or may not have their physical counterparts in hardware. In MCL we take a different approach: it allows us to describe the physical entities on a many-core device to create programming abstractions from them, such that programmers can map their algorithm to the physical device.

We can define two roles for our system: Programmers are the *users* of the system writing code in the programming language. The other role *describes hardware* in the hardware description language, ideally fulfilled by hardware vendors. However, the hardware descriptions can be adjusted by programmers to define new abstraction-levels, for example if programmers want a hardware description that focuses mainly on the memory hierarchy when working on data-intensive applications. MCL provides a library of predefined hardware descriptions. Before describing the hardware description language and the programming language, we first provide an overview of the interaction between the two languages.

3.4.1 Overview

MCL is solely targeted at writing computational kernels for many-core hardware. These kernels are often just a small part of a larger application. Since many-core

```
 1  perfect void matmul(int n, int m, int p,
 2      main float[n,m] c,
 3      main float[n,p] a, main float[p,m] b) {
 4
 5    foreach (int i in n threads) {
 6      foreach (int j in m threads) {
 7        float sum = 0.0;
 8        for (int k = 0; k < p; k++) {
 9          sum += a[i,k] * b[k,j];
10        }
11        c[i,j] += sum;
12  } } }
```

Figure 3.2: A matrix multiplication program for hardware description *perfect*.

hardware is highly parallel with complicated memory hierarchies, programmers need *programming abstractions* to control these hardware features. In Many-Core Programming Language (MCPL) parallelism is expressed in terms of *units of parallelism* such as threads or vectors and it is possible to target specific memories by declaring variables to reside in specific *memory spaces*.

Figure 3.2 shows a matrix multiplication program written in MCPL. The `foreach`-loops on line 5 and 6 express parallelism in terms of parallelism units `threads`. The variables `a`, `b`, and `c` are declared to reside in memory space `main` (line 2 and 3). The units of parallelism and the memory-spaces are defined in the hardware description.

For each compute kernel, to select a level of abstraction, programmers indicate which specific hardware description they target. The function in Fig. 3.2 targets hardware description *perfect* (line 1). The targeted hardware description governs which memory spaces and units of parallelism are available to the programmer for a specific compute kernel. MCPL is explained in more detail in Sec. 3.4.3 and we refer to App. A.2 for a complete description of the syntax.

A hardware description written in HDL has two distinct sections. The first section is called the *parallelism hierarchy* and defines the programming abstractions that are available to the programmer in terms of units of parallelism and memory spaces. Lines 1 to 9 in Fig. 3.3 show that a memory space `main` is defined (line 2) and that a `par_group threads` is defined (line 4) that can be used in `foreach`-statements.

The second part of the hardware descriptions defines the physical device. The main components are *execution units*, *memories*, and *interconnects* that connect the execution units and memories. The memories are associated with

```
 1  parallelism hierarchy {
 2    memory_space main {
 3    }
 4    par_group threads {
 5      nr_units = unlimited;
 6      par_unit thread {
 7        memory_space reg {
 8          default;
 9  } } } }
10
11  device perfect {
12    mem;
13    ic;
14    cores;
15  }
16  memory mem {
17    space(main);
18    capacity = unlimited B;
19  }
20  execution_group cores {
21    nr_units = unlimited;
22    execution_unit core {
23      slots(thread, 1);
24  } }
25  interconnect ic {
26    connects(mem, cores.core[*]);
27    latency = 1 cycle;
28    bandwidth = unlimited bit/s;
29  }
30  ...
```

Figure 3.3: Part of the hardware description *perfect*.

the memory spaces and the execution units with the units of parallelism. The next section explains HDL in greater detail, while we refer to App. A.1 for a more thorough specification.

3.4.2 Hardware Description Language HDL

The main purpose of HDL is to describe hardware features, but only those that are important for performance. Hardware descriptions may vary in the level of detail to define the abstraction-levels. Another goal is to formalize the knowledge that is often informally described in programming guides to make it accessible to both the programmer and the compiler, and to relate the hardware features with the program.

As shown in Fig. 3.4, hardware descriptions are organized in a hierarchy. The root is hardware description *perfect* that describes idealized many-core hardware and provides the highest level of abstraction for programmers. This description is provided in MCL and cannot be modified. Each lower level describes hardware in more detail and extends a parent resulting in the hierarchy.

Figure 3.3 shows a part of hardware description *perfect*. We first present an overview of what the hardware description expresses and then discuss the details. Hardware description *perfect* provides the programmer a flat parallelism hierarchy with only two memory spaces to consider. The memory `mem` is responsible for `memory_space main` (line 17) and has unlimited capacity (line 18) device `perfect` has an unlimited amount of `cores` (line 21-24) and each core

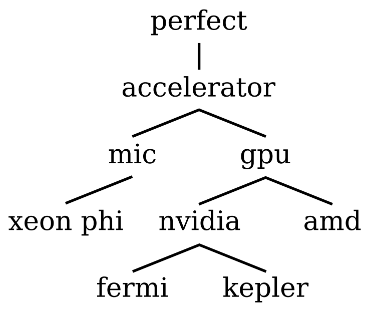


Figure 3.4: An example of a possible hierarchy of hardware descriptions. All hardware descriptions except *perfect* are user-defined.

can run 1 `thread` (line 23). The memory is connected with all cores (line 26) and has 1 cycle latency (line 27) and unlimited bandwidth (line 28).

In HDL, the main syntactic construct to express hardware properties is a block with syntax:

```
Block          =  BlockKeyword Identifier "{" Statement* "}"
Statement      =  PropertyStatement
               |  Block
BlockKeyword   =  "parallelism"  |  "memory_space"
               |  "par_unit"  |  ...
```

Hardware description *perfect* shows several of those blocks, for example on lines 1, 11, 16, 20, and 25. The semantics of a block are primarily determined by the specific BlockKeyword, such as `parallelism`, `memory_space`, or `interconnect`. Blocks contain a list of Statements that define some properties of the Block. A Statement can be a PropertyStatement or a nested Block. HDL defines precisely what kind of properties and nestings of Blocks are allowed. For example a `parallelism` block must have a `memory_space` block. A more complete description of the syntax and semantics is given in App. A.1.

A valid hardware description with the name *name* has at least two blocks, a `parallelism` block (line 1 in Fig. 3.3) and a `device` block with *name* as Identifier (line 11). The `parallelism` block defines a hierarchy of programming abstractions and the `device` block describes the physical device.

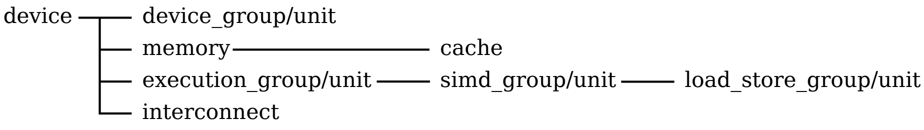


Figure 3.5: The device hierarchy.

Programming abstractions In a `parallelism` block the hardware description enforces which and how many units of parallelism programmers can use and to which memory spaces each unit of parallelism has access. This is achieved by nesting `memory_space`, `par_group`, and `par_units` in the `parallelism` block. The scope of the memory spaces defines which units of parallelism can make use of the memory space. For example, if a `memory_space` is defined within a `par_unit`, then this memory space is local to the `par_unit`.

Line 4 in Fig. 3.3 shows a `par_group threads`. Blocks ending with `_group` are special blocks that can group matching `_unit` blocks. The `_group` blocks must always contain a `nr_units` or `max_nr_units` property that expresses the allowed number of units.

Lines 1-9 show that the parallelism hierarchy `hierarchy` defines (on line 2) a `memory_space main` and a `par_group` block `threads` (line 4) with an unlimited number of parallelism units `par_unit thread` (lines 4-6). This means that the programmer can use a memory space `main` in a program and can express parallelism with threads. The scoping of the `memory_space` blocks indicates that a `thread` has private access to `reg`, but that all `par_units` have access to `memory_space main`. The `default` property will be explained in Sec. 3.4.3.

Physical device Describing the physical device is achieved with `device` blocks that can be organized in a subclass hierarchy similar to a class hierarchy in Object-Oriented Programming. The hierarchy is shown in Fig. 3.5. The root is a generic `device` block. A `device_group` and `device_unit` are generic `devices` to make it possible to group `devices`.

A `memory` is a more specialized `device` of which we can describe, among other properties, its capacity. The `memory` blocks need to have a `space` property with an identifier as argument. The identifier has to refer to a `memory_space` in the parallelism hierarchy and indicates for which memory space the memory is responsible. For instance, in Fig. 3.3, the `memory mem` is responsible for `memory_space main` (line 17). A `cache` is a specialized `memory`. It inherits all the properties of memory, but it is also necessary to specify the cache-line size.

Execution units can be described by the various `execution_group/unit` blocks. Execution units execute instructions that are defined in the `instruction` block (not shown in Fig. 3.3). The block `execution_unit` describes a generic execution unit. A `simd_group` describes an execution unit that executes its instructions in lock-step: each `simd_unit` executes the same instruction in parallel. A `load_store_group` is a `simd_group` that only executes the two instructions “load” and “store”. This execution unit is responsible for moving data between the memories and each hardware description must have a `load_store_unit` (not shown in Fig. 3.3).

Execution groups and units must have a property `slots`. This property takes as argument an identifier and an integer expression. The identifier has to refer to a `par_unit` and the integer expression defines how many “slots” or “contexts” are available for an execution unit or group, which indicates how much parallelism is available on an execution unit. For example, if an execution group has 4 execution units and 8 slots for threads, this would mean that 2 threads have to share the time on each execution unit and have to be scheduled after each other. Scheduling cannot be further expressed in HDL.

An `interconnect` block describes a device that connects the various `device` blocks with each other. It is possible to specify the latency, the bandwidth or the width of the bus (in bits). The `connects` property specifies how execution units and memories are connected to each other. It takes two identifiers as arguments with possibly a [*] suffix. The suffix [*] has to be used for a device in a `_unit` in a `_group` and means that the connection leads to all units.

Figure 3.3 shows that the `device perfect` on line 11 holds a `memory mem`, an `interconnect ic` and an `execution_group cores`. A statement with solely an identifier means that this specific statement can be replaced with the block it refers to. This mechanism also supports inheriting blocks from other hardware descriptions, but in this case the identifiers have to be fully qualified. For example, a hardware description may reuse the `par_group threads` from hardware description *perfect*. It then has to refer to this block with the qualified identifier `perfect.hierarchy.threads`.

3.4.3 Programming Language MCPL

To provide a familiar interface to many-core programmers, MCPL is imperative and C-like. In contrast to C, MCPL has constructs to explicitly map the program to hardware using the programming abstractions defined in the `parallelism` block in a hardware description. We describe the syntax in more detail in App. A.2.

Figure 3.2 shows matrix multiplication written for hardware description *perfect*. On line 1, the function `matmul()` is declared to adhere to the rules of hardware description *perfect*. Thus, it is possible to specify a hardware description per function. The hardware description determines which memory spaces are available and how parallelism can be expressed. MCPL has a `foreach` construct that expresses parallelism in terms of units of parallelism. In Fig. 3.2 on lines 5 and 6, the units of parallelism are `par_group threads` (defined on line 4 in Fig. 3.3). Barrier statements (not shown in Fig. 3.3) and declarations are associated with memory spaces. Figure 3.2 shows that variables `c`, `a`, and `b` are declared to reside in memory space `main` (lines 2 and 3).

In an MCPL program, the compiler can automatically infer which variables are constant or written. In an MCPL module, the whole call hierarchy has to be fully specified. Library functions are annotated with signatures that specify which of the parameters are written and/or read.

Variables that have an array type or that are written need to have a memory space modifier unless the hardware description has the property `default` (line 8 in Fig. 3.3 for `reg`). Thus, `sum` (line 7 in Fig. 3.2) is implicitly declared to reside in memory space `reg`. Variables are passed by reference unless types are primitive and constant. Guided by our conclusions in Chapter 2 MCPL does not allow aliasing and global variables.

MCPL provides multi-dimensional arrays with array-size specifiers. This helps the compiler to reason about different dimensions and ensures that the compiler can express feedback in terms of array-sizes. Variables that are used in array-size specifications have to be constant. Figure 3.2 shows on lines 2 and 3 variables `c`, `a`, and `b` being declared as two-dimensional arrays.

MCPL supports views and tiles in multiple dimensions on arrays. For example, an array `int[36]d` can be declared as `int[2,2][3,3]d2` with 2×2 tiles of 3×3 elements. This helps programmers to specify data-layout. More details are provided in App. A.2.

A `foreach`-statement declares an indexing variable, contains a dimension expression and an identifier that indicates which units of parallelism the `foreach` targets. Line 5 in Fig. 3.2 states that there will be n units of parallelism of `par_group threads` each with a unique index i where $0 \leq i < n$. The dimension expressions in `foreach`-statements are not allowed to depend on the index of an outer `foreach`.

The `parallelism` block in the hardware description governs how `foreach` loops can be nested and where memory spaces can be declared. The nesting has to follow the nesting in the hardware description with an exception for `foreach`

```
par_group blocks {
  par_unit block {
    memory_space main {}
    par_group threads {
      par_unit thread {
        memory_space reg {}
} } } }
```

```
foreach (int ib in nb blocks) {
  foreach (int jb in mb blocks) {
    main int[n] a;
    foreach (int it in nt threads) {
      foreach (int jt in mt threads) {
        reg int b;
} } } }
```

Figure 3.6: Nesting of memory spaces and units of parallelism in relation to nesting of `foreach` statements and declarations.

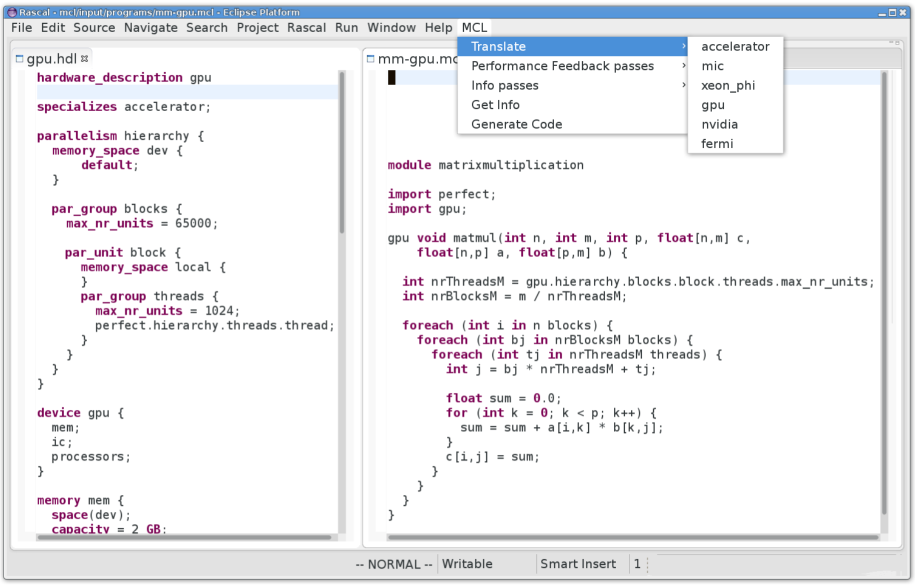
loops belonging to the same `par_group`, which are allowed to be nested (as shown on lines 5 and 6 in Fig. 3.2). Executing statements are only allowed in the innermost `foreach` loop. For a declaration of a variable, its nesting must match the nesting of its `memory_space` in the hardware description.

Figure 3.6 explains the nesting rules in more detail. It shows at the left a possible nesting of `par_groups` in a `parallelism` block inside a hardware description. The right side shows a possible nesting of `foreach` statements. The nesting follows the nesting in the hardware description and the figure shows that `foreach` statements of the same `par_group` can be nested. At the right side there is a declaration `a` with memory space `main`. The declaration is inside a `foreach`-loop with `par_group blocks`, which means that the declaration is inside a `block`. The left side shows that there is indeed a memory space `main` defined inside a block `par_unit`.

Units of parallelism can communicate with each other through memory that is declared in a scope outside of the units of parallelism. For example, in Fig. 3.6, threads cannot communicate with each other using memory space `reg`, but they can communicate with each other using memory space `main`. To ensure that a thread can read what another thread wrote, it is possible to insert a `barrier(main)` statement, with a parameter for the memory space.

3.4.4 Compiler

To support the *stepwise-refinement for performance* methodology, the compiler is not only responsible for generating code, but also for generating performance feedback for the programmer. MCL has been implemented in the metaprogramming language Rascal [69] which allows us to create an Eclipse Plugin with feedback annotations and syntax highlighting for both HDL and MCPL. Figure 3.7 shows a screenshot of the Eclipse Plugin with a menu for activating


Rascal - mcl/input/programs/mm-gpu.mcl - Eclipse Platform
File Edit Source Navigate Search Project Rascal Run Window Help MCL
gpu.hdl
mm-gpu.mc
Translate
Performance Feedback passes
Info passes
Get Info
Generate Code
accelerator
mic
xeon_phi
gpu
nvidia
fermi
-- NORMAL --
Writable
Smart Insert
1

```
hardware_description gpu

specializes accelerator;

parallelism hierarchy {
  memory_space dev {
      default;
  }

  par_group blocks {
    max_nr_units = 65000;

    par_unit block {
      memory_space local {
      }
      par_group threads {
        max_nr_units = 1024;
        perfect.hierarchy.threads.thread;
      }
    }
  }
}

device gpu {
  mem;
  ic;
  processors;
}

memory mem {
  space(dev);
  capacity = 2 GB;
```

```
module matrixmultiplication

import perfect;
import gpu;

gpu void matmul(int n, int m, int p, float[n,m] c,
    float[n,p] a, float[p,m] b) {

  int nrThreadsM = gpu.hierarchy.blocks.block.threads.max_nr_units;
  int nrBlocksM = m / nrThreadsM;

  foreach (int i in n blocks) {
    foreach (int bj in nrBlocksM blocks) {
      foreach (int tj in nrThreadsM threads) {
        int j = bj * nrThreadsM + tj;

        float sum = 0.0;
        for (int k = 0; k < p; k++) {
          sum = sum + a[i,k] * b[k,j];
        }
        c[i,j] = sum;
      }
    }
  }
}
```

Figure 3.7: Screenshot of the MCL Eclipse Plugin.

several feedback passes in the compiler. The programmer receives messages from the compiler associated directly with the source code, for example messages for a specific variable, `foreach`-loop, or function.

Another task of the compiler is to automatically translate between the abstraction-levels. Programs written for hardware x can automatically be translated to hardware description y if y is a child of x in the hierarchy in Fig. 3.4. The edges in the hierarchy define the translation possibilities, so it is for example possible to translate from *perfect* to *gpu*.

Finally, the compiler can generate OpenCL and C++ code from the leaf nodes of Fig. 3.4. For example, the compiler takes as input a function written for hardware description *fermi*. It reads in the hardware description *fermi* but also a configuration file that tells the compiler how the programming abstractions defined in hardware description *fermi* can be mapped to OpenCL and C++ constructs. Our system generates OpenCL code for the computational kernels, header files, and C++ code for setting up the execution on a many-core device, including the necessary data-transfers. This means that the generated code can easily be included in many applications.

As the compiler automatically translates between abstraction-levels and can generate code from each leaf node, it can generate code for each abstraction level. For example, following the hierarchy of hardware descriptions in Fig. 3.4, we can generate code for *fermi* from a program written in *gpu*. First, the compiler translates the program to *nvidia*, then it translates to *fermi*, and from there the compiler generates OpenCL and C++ code.

This section presented an overview of the MCL programming system. Section 3.5 will give an example of the *stepwise-refinement for performance* methodology using our system. Section 3.6 will discuss several interesting implementation details.

3.5 Example: Matrix Multiplication

In this section we give a detailed overview of how the process of the *stepwise-refinement for performance* methodology took place for matrix multiplication. We refined the kernel in a stepwise manner to obtain high performance for two different many-core architectures: a GTX480 GPU and a 60-core Intel Xeon Phi.

Tables 3.2 and 3.3 give an overview of the feedback that the we received on each level, what refinements we applied, and the performance of each kernel. Below we will discuss the details. From level *accelerator* we will split in Sec. 3.5.1 for the GTX480 and Sec. 3.5.2 for the Xeon Phi. How the compiler translates and gives feedback is explained in Sec. 3.6. In Sec. 3.7 we will evaluate our approach with more applications.

perfect The goal of hardware description *perfect* is to define a high level of abstraction by making the many-core hardware as simple as possible. Usually, many-cores expose multiple layers of parallelism and several memories, but to create a high level of abstraction, hardware description *perfect* has only one layer of parallelism and two memory spaces (Fig. 3.3).

Figure 3.2 shows matrix multiplication written for this hardware. Since programmers write for idealized hardware, they do not have to pay attention to hardware-specific details. At this level of abstraction, it is no problem to create $n \times m$ threads or read the same data multiple times, because the hardware description defines that we have an unlimited number of execution cores and an access time of 1 cycle to data in main memory.

In the inner process, the compiler is not triggered to give optimization feedback on this level of abstraction, because the properties of the device do not give

Table 3.2: Feedback from compiler for the GTX480 on different abstraction-levels for matrix multiplication. The versions with (v*) are modified by the programmer according to the feedback from the compiler.

Version	Change compared to previous version	Feedback	Performance (GFLOPS)
NVIDIA GTX480 GPU			
perfect	(initial version)	#computational instructions: $2nmp$, #data accesses: $2nmp + nm$	89.0
accelerator	(auto-translate)	PCIe transfers: $4np + 4mp + 4nm$ bytes	
gpu	(auto-translate)	expression `a[i,k]` loaded for each thread `t`: local memory can be leveraged	
gpu (v1)	Use local memory for p elements	consider computing multiple elements per threads	100
gpu (v2)	Compute multiple elements per thread	expression `b[k,j]` loaded for each block and each iteration in for-loop (n times)	91.6
gpu (v3)	pull `b[k,j]` out of for-loop	expression `b[k,j]` loaded for each block `bl` (n/x times)	205
nvidia	(auto-translate)	Using 1/8 blocks per SMP. Reduce the amount of shared memory used by storing/loading shared memory in phases	
nvidia (v1)	multiple store-load phases	Using 4/8 blocks per SMP.	489
fermi	(auto-translate)	Optimal memory access. `b[k,j]` in loop *bi* does not benefit from cache, best case: 256 cache-line fetches	
fermi (v1)	compute more elements for `b[k,j]`	`b[k,j]` in loop *bi* does not benefit from cache, best case: 128 cache-line fetches	564

Table 3.3: Feedback from compiler for the Xeon Phi on different abstraction-levels for matrix multiplication. The versions with (v*) are modified by the programmer according to the feedback from the compiler.

Version	Change compared to previous version	Feedback	Performance (GFLOPS)
Intel Xeon Phi (5110P)			
perfect	(same version as in Table 3.2)	(same feedback as in Table 3.2)	39.9
accelerator	(same version as in Table 3.2)	(same feedback as in Table 3.2)	
mic	(auto-translate)	Consider computing multiple elements per threads	
mic (v1)	Compute multiple elements per thread	Data reuse: `a[i,k]` accessed in loop with index `ej` but does not depend on it.	35.7
mic (v2)	Pull `a[i,k]` out of for-loop		47.6
xeon_phi	(auto-translate)	`b[k,j]` in loop *k* does not benefit from cache, *p* cache-line fetches	
xeon_phi (v1)	Pull `b[k,j]` out of for-loop	`c[i,j]` in loop in loop *ei* does not benefit from cache, 32 cache-line fetches	87.8
xeon_phi (v2)	`c` in temporary variable	Try to adjust size `bTemp` with size `4*p` bytes, to cache (32 kB total, cache_line 64 B)	115
xeon_phi (v3)	Decrease `bTemp`		488

rise to this. For example, there is no faster memory. However, we can choose to investigate the algorithmic properties of the program. For instance, we can request statistics on the number of operations as shown in Table 3.2.

Matrix multiplication written for *perfect* is a portable program. We first automatically translate this program down the hierarchy in Fig. 3.4 via *accelerator*, *gpu*, and *nvidia* to *fermi* to generate code. Running this on an NVIDIA GTX480 GPU delivers 89.0 GFLOPS. Automatically translating the program to *xeon_phi* (via *accelerator* and *mic*) and generating code from there results in a performance of 39.9 GFLOPS. Since the inner process does not give feedback that we can use to optimize, we move to a lower level of abstraction in the outer process. We let the compiler automatically translate the program to hardware description *accelerator* below *perfect* in Fig. 3.4.

accelerator In the library of predefined hardware descriptions, an accelerator is defined to be a many-core device that is connected to a host computer by means of a PCI-express bus, a common setup for many-core hardware. The *accelerator* hardware description specializes hardware description *perfect*. It inherits most features from *perfect*, but a special device `host` is added that is responsible for setting up the computation for the many-core machine. We define an interconnect `pcie` in a similar way as the interconnect `ic` in Fig. 3.3 on line 25, but now it connects the memory `accelerator.mem` and the device `host` and has different values for the latency and bandwidth.

Since the parallelism hierarchies of *perfect* and *accelerator* are the same, the translation pass of the compiler only has to change the keyword `perfect` on line 1 to `accelerator` to let the program adhere to the rules of hardware description *accelerator*.

At this level of abstraction, we receive feedback that the program has $4nm$ bytes of data transfers from device to host, and a data transfer of $4np$ bytes plus a data transfer of $4pm$ from host to device. The analysis to provide this feedback infers that variables `a` and `b` are read and that `c` is written inside the `foreach` loops. The compiler can give this feedback because the sizes of `a`, `b`, and `c` are known in terms of parameters `n`, `m`, and `p`.

At this level of abstraction and with this feedback, we have no opportunity to optimize the existing program. Therefore, we move to lower level hardware descriptions giving up portability between GPU and Xeon Phi. The next subsection will discuss the process for the GTX480, while Sec. 3.5.2 discusses the process for the Xeon Phi.

```
 1  parallelism hierarchy {
 2    memory_space dev {
 3    }
 4    par_group blocks {
 5      max_nr_units = 65535;
 6      par_unit block {
 7        memory_space local {
 8        }
 9        par_group threads {
10          max_nr_units = 1024;
11          perfect.hierarchy.threads.thread;
12  } } } }
13
14  memory on_chip {
15    space(local);
16    capacity = 16 kB;
17  }
18  execution_group processors {
19    nr_units = 16;
20    execution_unit processor {
21      slots(block, 1);
22      on_chip;
23      execution_group alus {
24        nr_units = 32;
25        execution_unit alu {
26          slots(thread, 1);
27  } } } }
```

Figure 3.8: Part of the hardware description *gpu*.

3.5.1 GTX480

gpu At this level of abstraction, the many-core device becomes more concrete. Representative values obtained from programming guides are filled in to give the compiler an indication of the amount of parallelism and the amount of memory to define a generic GPU. For example, we could define that a *gpu* has a device memory of several GBs, 16 processors, and each processor has 32 small ALUs capable of performing floating point operations and a fast on-chip memory.

Conceptually, a GPU exposes two layers of parallelism to the programmer. The 32 ALUs can run threads in parallel and the 16 processors can run blocks of threads in parallel. Figure 3.8 shows how this is reflected in the parallelism hierarchy. Lines 4 to 12 express that there are two layers of parallelism. Programmers can define up to 65535 `blocks` to run in parallel. Outside the block

```
1  int nrThreadsM = gpu.hierarchy.blocks.block.threads.max_nr_units;
2  int nrBlocksM = m / nrThreadsM;
3
4  foreach(int bi in n blocks) {
5    foreach (int bj in nrBlocksM blocks) {
6      foreach (int tj in nrThreadM threads) {
7        int i = bi;
8        int j = bj * nrThreadsM + tj;
9
10       float sum = 0.0;
11       for (int k = 0; k < p; k++) {
12         sum += a[i, k] * b[k, j];
13       }
14       c[i, j] = sum;
15 } } }
```

Figure 3.9: Transformation of the loops on level *gpu*

scope there is a memory space `dev` that is shared by blocks. Inside the scope of a block there is a memory space `local`. It is not possible to synchronize between blocks using this memory because it is not declared in a scope that surrounds blocks. However, threads can synchronize using this memory because the group of parallelism units threads is defined in the same scope. Per `block`, programmers can use up to 1024 threads (line 10). On line 11, this hardware description inherits the unit of parallelism `thread` from hardware description *perfect*.

To automatically translate matrix multiplication to *gpu*, the compiler has to split the `foreach` loops for parallelism units `threads` to multiple `foreach` loops for parallelism units `blocks` and `threads` (loop-tiling). The translation process is explained in Sec. 3.6, the result is listed in Fig. 3.9 (assuming that `m` can be divided by `nrThreadsM` for simplicity). The code from line 10 on remains unchanged compared to the code in Fig. 3.2.

Arrived in the inner process (Fig. 3.1), the compiler combines the knowledge from the hardware description and the program to give the following three very specific feedback messages: (1) “expression `a[i,k]` loaded for `nrThreadsM` threads `tj`: local memory can be leveraged.”, (2) “expression `b[k,j]` is accessed for `n` blocks `bi`.”, and (3) “It may be beneficial to consider computing more than one element of `c` per thread.”

The compiler issues the first two messages after analyzing data reuse. Using

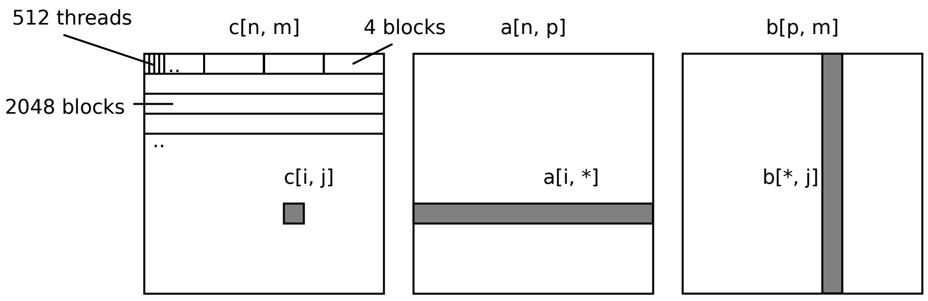


Figure 3.10: Schematic view from block division in a 2048 × 2048 matrix multiplication.

data-flow analysis, it discovers that expression `a[i,k]` does not depend on `tj` and `bj`. Figure 3.10 shows a schematic view of how the blocks and threads defined in Fig. 3.9 use the input and output data. Each of the 512 threads (identified by `tj`) computes a single output element in a row in array `c`. To compute one element, a thread `tj` (and a block `bj`) needs access to a complete row in the `a` matrix. The compiler discovers this by concluding that expression `a[i,k]` does not depend on variable `bj` or `tj`. Similarly, the compiler notices that expression `b[k,j]` does not depend on `bi`. Therefore, the compiler issues the messages that `a[i,k]` will be loaded for each thread `tj`, each block `bj`, and that `b[k,j]` will be loaded for each block `bi`.

Additionally, the compiler combines this information with the information in the hardware description. In Fig. 3.8 on lines 7 to 12, `par_group threads` shares a memory space `local`, which means that each thread `tj` can access this memory. The compiler investigates whether this memory space resides in faster memory than the memory of array `a` (memory space `dev`). Hardware description *gpu* shows that there is a memory `on_chip` that holds memory space `local` (lines 14 to 17 in Fig. 3.8). The compiler can infer that this memory is faster than the memory that holds memory space `dev` (not shown). Since `local` memory is shared between threads `tj` and is faster than `dev`, the compiler can suggest to use memory space `local` for each thread `tj` for expression `a[i,k]`.

The feedback that we receive in the inner process is now sufficient to act upon. We decide to first listen to the most specific feedback message, which is feedback message (1). We implement the suggested change in version *gpu (v1)* by loading p elements in `local` memory.

gpu (v1) In this version we want to load p elements of `a[i,*]` in `local` memory. To accomplish this, we add a declaration with memory space `local`, a `for`-loop that loads p elements in cooperation with the threads, and a `barrier(local)` statement to make sure that each thread can observe all the updates from the other threads. This is a standard technique in GPU programming.

The resulting version delivers 100 GFLOPS. The feedback we acted on disappears, but the other two messages remain. We decide to take another iteration in the inner process. Since it is not clear at this stage how to resolve the issue in feedback message (2), we decide to act on feedback message (3) by computing multiple elements per thread.

gpu (v2) We use this version to illustrate that although feedback from the compiler may not immediately lead to better performance, the feedback is not bad per se and can ultimately lead to better performance if we listen to the compiler carefully. In contrast to our structured approach to optimizing, in ad-hoc optimization, programmers might disregard this opportunity for higher performance because initially it leads to lower performance.

We implement computing multiple elements in a column per thread, but we do this naively by splitting the `foreach`-loop `bi` into two. We decide to change the loop having n iterations into a `foreach`-loop of n/x iterations, where x is a value that we chose. We add a sequential `for`-loop of x iterations in the body of the `foreach`-statements. (This technique is commonly referred to as vertical thread-block merge in GPU programming.)

Indeed feedback message (3) disappears. However, feedback message (2) is repeated in a slightly different form and gives us a deeper understanding in the performance issue. The compiler repeats that expression `b[k,j]` does not depend on each block `bi` (but now for n/x blocks), and that it is also independent of the `for`-loop which we have just introduced having x iterations. This means that effectively, since `b[k,j]` is independent of both loops, it is still loaded n times and that we have only added control flow, compared to the previous version. This explains the worse performance.

Fortunately, this steers us in the right direction: since, `b[k,j]` is independent of the `for`-loop, we now understand that we have to try to pull expression `b[k,j]` outside of this `for`-loop to reuse the value for multiple iterations. We implement this in version *gpu (v3)*.

gpu (v3) It is more challenging to incorporate the feedback in this version. However, if programmers fail to act on the feedback, at least they are informed

```
local float[x][p] l_a;
...
// load x * p elements in l_a
...
for (int k = 0; k < p; k++) {
  float bkj = b[k, j];
  for (int ei = 0; ei < x; ei++) {
    sums[ei] += l_a[ei][k] * bkj;
} }
```

Figure 3.11: Structure of the loops in *gpu (v3)*.

of the performance issue.

The insight for this version is that we want to move the `for`-loop with x iterations that we have just introduced, inside the loop on line 11 in Fig. 3.9 so we can reuse `b[k,j]` for each of the x iterations. Unfortunately, this is not immediately possible, since loading the elements in `local` memory introduced in *gpu(v1)* also relies on this loop. The solution is to use more `local` memory and load these elements in a second, separate `for-loop` of x iterations. Figure 3.11 shows the structure of the loop after rewriting.

This version delivers 205 GFLOPS. For programmers that want to remain portable between AMD and NVIDIA GPUs and are satisfied with the obtained performance, this is a good moment to stop optimizing since the hardware descriptions below *gpu* contain vendor-specific information. Therefore, this optimization applies to both AMD and NVIDIA GPUs. This clearly shows that MCL offers programmers a trade-off in high-level programming, portability, and performance. We decide to stop optimizing in the inner process and move to level *nvidia* in the outer process.

nvidia The parallelism hierarchy on level *nvidia* is almost the same as on level *gpu*, but memory space `dev` is renamed to `global` and memory space `local` is renamed to `shared` to conform to NVIDIA terminology. As a result of more specific details in the hardware description, the compiler triggers an analysis that requests example values for `m`, `n`, and `p` from the programmer. It then tells us that memory `on_chip` is a scarce resource and that we should carefully consider the usage of memory space `shared`, since we use this in our program. The compiler is even more specific. It tells us that we execute only 1 out of 8 blocks per SMP in parallel which depends on the number of threads, the number

```
 1  execution_group smps {
 2    nr_units = 16;
 3    execution_unit smp {
 4      slots(block, 8);
 5      slots(thread, 1024);
 6      performance_feedback("shMem");
 7      on_chip;
 8      regs;
 9      gpu.processors.processor.alus;
10  } } } }
```

Figure 3.12: Part of the hardware description *nvidia*.

of registers, and the amount of shared memory we use. Finally, it proposes a strategy to manage shared memory, namely by reducing the amount of shared memory by using multiple store-load phases.

To be able to give this feedback, the compiler combines many pieces of information from the hardware description. Figure 3.12 shows a part of hardware description *nvidia*. It defines that an SMP has 8 slots for `blocks` (line 4). It also defines that an execution-unit `smp` contains a memory `on_chip` (line 7) that holds memory space `shared` (not shown). Because of what is defined in the `parallelism_hierarchy` block, the compiler also knows that `blocks` cannot communicate through memory space `shared` (similar to the situation in *gpu* where blocks cannot communicate through `local` in Fig. 3.8 on line 7). Therefore, the compiler knows that memory `on_chip` has to be divided among 8 blocks, which makes memory `on_chip` a scarce resource.

On an NVIDIA GPU, the maximum number of blocks that can run on an SMP (between 1 and 8) is important but governed by complex rules. Therefore, MCL provides an API in which these complex rules can be encoded. Line 6 in Fig. 3.12 points the compiler to a performance feedback function "shMem" that expresses these complex rules. Part of the feedback function is shown in Fig. 3.13. The compiler exposes an API in Java through a data-structure `Kernel` that we can ask for the size of memory spaces (line 2), and the number of slots (line 7).

Because of the feedback, we understand that we have to minimize the amount of allocated `shared` memory to run more than one block on an SMP. In version *nvidia (v1)* we implement the suggestion from the compiler to load the elements in multiple load-store phases.

```
String shMem(Kernel k) {
  int usedShared = k.context.sizeMemSpace("shared");
  if (usedShared == 0) {
    return "";
  }
  else {
    int nrSlotsBlock = smps.smp.nrSlots("block");
    ...
} }
```

Figure 3.13: Part of user-defined performance function used in hardware description *nvidia*.

nvidia (v1) It may not be immediately clear how to implement this optimization. In the previous versions we loaded p elements cooperatively with threads in `shared` memory (or `local` in hardware description *gpu*) which happens in y iterations. To create multiple store-load phases, we make this loop the outer loop moving the loading of elements and the computation inside. The insight for this optimization is that we can merge the loop on line 5 in Fig. 3.11 with the loop of y iterations. The resulting loop structure is shown in Fig. 3.14. As a result of this optimization we use y times less `shared` memory.

We now receive the feedback that we use 4 out of 8 blocks which gives us a performance of 489 GFLOPS. We translate this version to level *fermi*.

fermi Hardware description *fermi* introduces a new layer `warp` in the parallelism hierarchy, as well as `SIMD`-units and caches with cache-line sizes. The difference between a normal `execution_group` and a `simd_group` is that the `simd_units` are defined to execute in lock-step, so each unit performs the same instruction in parallel. In the *fermi* hardware description, it is defined that a `warp` of 32 threads is executed on a SIMD-unit, which means that each thread executes in lock-step with the others.

Using a performance feedback function similar to the one of Fig. 3.13, the compiler can give feedback on the complex memory-access rules of the *fermi* architecture. It tells us that the memory access patterns of `a`, `b`, and `c` are optimal, and that there are no bank conflicts in the access of shared memory.

The new information about SIMD-units and how warps execute in lock-step in the hardware description is used in the cache analysis. If a thread in a warp can access an element in memory that is cached, then the other threads in a warp also have access to the cache-line and do not need a new cache-line fetch.

```
shared float[x][nrThreadsM] l_a;
...
for (int l = 0; l < y; l++) {
  for (int ei = 0; ei < x; ei++) {
    l_a[ei][tj] = a2[bi,l][ei,tj];
  }
  barrier(shared);

  for (int k2 = 0; k2 < p/y; k2++) {
    int k = l * p/y + k2;
    float bkj = b[k,j];

    for (int ei = 0; ei < x; ei++) {
      sums[ei] += l_a[ei][k2] * bkj;
  } }
  barrier(shared);
}
```

Figure 3.14: Structure of the loops in *nvidia (v1)*.

The cache feedback gives us a worst case scenario and a best case scenario in terms of the number of cache line fetches. This makes clear that in a specific loop, access `b[k,j]` needs at least 256 cache-line fetches.

We decide to try to improve the cache behavior of the memory access of `b[k,j]` in version *fermi (v1)*.

fermi (v1) In this version we tweak the parameters x and y such that we compute twice the amount of elements for each fetch of `b[k,j]`. This gives us a performance of 564 GFLOPS.

At this point the compiler gives us no more feedback we can act on. We are also on the lowest level of abstraction, which ends the *stepwise-refinement for performance* process for the GTX480.

3.5.2 Xeon Phi

mic Intel's Many Integrated Core (MIC) architecture contains several tens of in-order x86 cores with powerful vector units and several hardware threads connected through a ring network. The MIC exposes two layers of parallelism: vector instructions and threads. In the library of hardware descriptions, the *mic* hardware description represents this as shown in Fig. 3.15. Besides registers (line 10), there is only one memory space `dev` (line 2). It is possible to define

```
 1  parallelism hierarchy {
 2    memory_space dev {
 3    }
 4    par_group threads {
 5      max_nr_units = unlimited;
 6      par_unit thread {
 7        par_group vectors {
 8          nr_units = 16;
 9          par_unit vector {
10            memory_space reg {
11  } } } } } }
12
13  interconnect ring {
14    connects(mem, cores.core[*]);
15    latency = 20 cycles;
16  }
17  execution_group cores {
18    nr_units = 60;
19    execution_unit core {
20      slots(thread, 4);
21      regs;
22      vector_group;
23  } }
```

Figure 3.15: Part of the hardware description *mic*.

an unlimited amount of threads (line 5) and the vector unit is exposed through a `par_group` of 16 vectors (lines 7 to 11). The *mic* has a ring interconnect that connects the memory to all cores (line 13). A representative value for the number of cores is 60 (line 18). Each `core` has slots for 4 hardware threads (line 20), registers (line 21) and a `simd_unit` called `vector_group` that has slots for the `vector` `par_units` defined on lines 9 to 11).

After automatically translating the *accelerator* version to *mic*, the compiler tells us about data-reuse in variables `a` and `b`, similar to the feedback on level `gpu`. The compiler does not propose a faster memory because the *mic* hardware description does not expose faster memories. However, since the number of execution units is limited, it does propose to compute multiple elements per vector.

We implement this in a new iteration of the inner process. We decide to compute multiple elements in both dimensions in version *mic (v1)* as it is not clear what dimension we should compute multiple elements in.

mic (v1) In this version we naively introduce a `for`-loop for each dimension: a `for`-loop with index `ei` and a `for`-loop with index `ej`. For the same reasons as in version *gpu (v2)*, we get lower performance (35.7 instead of 39.9 GFLOPS), but again this leads to feedback that can help us to gain more performance.

The compiler tells us that `a[i,k]` is accessed in a loop that we have introduced with index `ej`, but that it does not depend on it. Similarly, it tells us that `b[k,j]` does not depend on the loop we have introduced with index `ei`.

Because of the way the loops are organized, it is difficult to do something about both feedback messages at the same time. Therefore, we decide to choose one of the two variables and pull `a[i,k]` out of the `for`-loop with index `ej` and store it in a temporary array to increase data-reuse. This optimization is comparable to the one we did in version *gpu (v1)*.

mic (v2) Having implemented the above in this version, we still get feedback for data-reuse for `b`, but similar to the previous version, it is not clear how to act on the feedback. We measure the performance at 47.6 GFLOPS and decide to automatically translate this version to *xeon_phi*.

xeon_phi The *xeon_phi* hardware description introduces many more details compared to *mic*. The most important difference is the introduction of level 1 and 2 caches. The hardware description language is expressive enough to give a realistic representation of the Xeon Phi. The ring interconnect connects the memory to 60 L2 caches of 512 kB. Each cache is associated with an execution unit `core` that holds 32 kB of L1 cache, which is in accordance with the real hardware. Both caches have cache lines of 64 bytes.

At the *xeon_phi* level, we receive feedback that we did not receive for GPUs, namely: "This is a cache-oriented architecture. Make sure that each access benefits from the cache." The compiler gives this feedback if it learns from the hardware description that there are no other opportunities for fast memory access than caches, which is the case on a Xeon Phi. For every variable declared in `dev` memory space we get cache-behavior feedback (explained in Sec. 3.6). We learn that variable `a` does benefit from the cache, but that `b` and `c` do not benefit from the cache. The compiler tells us that `b` has the worst behavior, namely at least `p` cache fetches.

We now realize that on level `mic` we pulled the wrong array out of the `for`-loop, since array `a` has better cache behavior than `b`. We decide to undo pulling out `a` and pull out `b` instead of `a`, making sure that we compute as much as possible for each cache-line fetch for `b`. This leads to version *xeon_phi (v1)*.

xeon_phi (v1) This change almost doubles the performance to 87.8 GFLOPS. The compiler gives us two feedback messages: (1) "`c` may benefit from the cache: best case $2m$ cache-line fetches.", (2) "try to adjust the size of declaration `bTemp` with $4p$ bytes in relation to the cache with capacities 32kB and 512kB and cache-line size 64B.", where `bTemp` is the temporary variable we have just introduced.

Feedback message (1) tells us that we have bad cache behavior for variable `c` because m is typically large. A solution would be to store this in a temporary array. Investigating feedback message (2), we come to the conclusion that if we split up `bTemp`, then we have to store partial results for `c` for which we need temporary storage as well. Therefore, we decide to act on feedback message (1) first, which leads to the next version.

xeon_phi (v2) Measuring the performance for this version where we introduced temporary storage for variable `c` gives 115 GFLOPS. We decide now to act on feedback message (2) to reduce the size of `bTemp` to better fit the caches. This means that we need to compute partial results in the temporary variable of `c` we have just introduced. We implement this in version *xeon_phi (v3)*.

xeon_phi (v3) This has a dramatic effect on performance. Making the array small, for example a value of 16 elements for the `bTemp` array leads to a performance of 488 GFLOPS. The compiler gives us no longer feedback we can act on, so we stop the process.

3.5.3 Summary

In the process above, most optimizations are straightforward to do. However, several optimizations may require considerable thought: From *gpu (v2)* to *gpu (v3)*, the compiler complains that expression `b[k,j]` is loaded many times. To solve it, we allocate more `local` memory which may be counter-intuitive for some programmers. When more hardware details become available, we reduce this size in *nvidia (v1)*, which may also be a challenging optimization. The challenge here is to reuse the iteration space of one loop (line 3 in Fig. 3.14) for another (line 9). Finally, in *xeon_phi (v1)* it may not be easy to discover how to decrease the size of the `bTemp` array.

The benefit of our approach compared to a trial-and-error process is that our compiler assists and steers the programmer. The compiler helps the programmer to focus on aspects of the many-core hardware that the compiler deems appropriate at a certain stage of the program. Moving to lower levels of abstraction, the compiler gains more and more knowledge about the hardware. This is in accordance with the increasing insight that programmers gain while creating the different versions. Another benefit is that the hardware descriptions form a well-defined set of hardware features for different architectures which makes the trade-off between portability against performance more clear.

3.6 Implementation

MCL's compiler is mostly written in Rascal [69] and partly in Java. It consists of four main parts: the *data structures* that define the grammars, the Abstract Syntax Trees (ASTs), symbol table and call graph; several *passes* where a pass defines the dependencies between passes; *micro-passes* which are small passes that do the actual work; and the *plugin* that defines the interaction with the Eclipse plugin introduced in Sec. 3.4.4.

Rascal automatically generates parsers for the grammars that instantiate detailed ASTs. An MCL *pass* defines on which other passes it depends, optional passes on which it depends, and its *entrypoint*. An entrypoint is a function that takes as input *passdata* and returns passdata. Passdata contains the data for a compilation unit such as the ASTs and the callgraph. Executing a pass is implemented as building a graph of dependent passes, computing the topological order of this graph which returns a list of passes, and applying a fold operation on this list that ultimately calls each entrypoint of the pass.

A pass often makes use of several *micro-passes*. For example, semantic analysis performs the micro-passes parsing, building up a ASTs, def-use analysis, typechecking, and verifying the mapping to hardware.

Besides passes such as generating code or pretty printing, there are passes that transform the AST, such as translating between abstraction-levels, and there are passes that output feedback for programmers, such as analyzing memory behavior. These feedback passes generate messages that can be shown in the *plugin*.

The compiler entails about 35000 lines of code including whitespace and comments. It uses many standard techniques such as data-flow analysis, and simplification of expressions to provide readable feedback. For the rest of this section, we focus primarily on the techniques that use both the information from the hardware description and the program for providing high-quality feedback. The analysis techniques work directly on the AST to create a strong relation with the actual code that the programmer wrote.

3.6.1 Translation between Abstraction Levels

The input for the translation pass is an MCPL module with functions and a string *target* indicating the target hardware description to translate to. Since each hardware description defines its parent in the hierarchy of hardware descriptions (Fig. 3.4), the compiler can find the path from *target* to the root *perfect*. Any function in the module with a hardware description that is in this path is iteratively translated to *target*.

Given hardware descriptions *hwdFrom* and *hwdTo* where *hwdTo* is a child of *hwdFrom* in the hierarchy, a function with hardware description *hwdFrom* is translated in three phases: First, the compiler finds equivalence between parallelism units and memory spaces. With this information it then translates all memory spaces of declarations and barrier statements to equivalent memory spaces of hardware description *hwdTo*. Finally, it translates the `foreach` statements, splitting them up if required. The following paragraphs give a high-level overview of the translation pass. The algorithms are provided in full in App B.

Restrictions for hardware descriptions To make the translation possible, there are several restrictions that apply to the parallelism hierarchy of a child hardware description. The basic principle is that it is only allowed to add `memory_spaces` and `par_groups` or `par_units`.

Since hardware description *hwdTo* may have more layers of units of parallelism than *hwdFrom*, `par_groups` may have to be split up. The `par_groups` are only allowed to split up using a more specific number of parallelism units. For example, the `par_group threads` in hardware description *perfect* with unlimited units of parallelism can be split up in 65535 `blocks` and 1024 `threads` in hardware description *gpu*. In *nvidia* the 1024 `threads` can be split up in 32 `warps` of 32 `threads`.

The translation of memory spaces is also governed by rules that are necessary to satisfy the synchronization conditions for barrier statements. To ensure that the translation pass can always find equivalent memory spaces, the rules for a `par_group` *pgFrom* that splits up in two or more `par_groups` *pgsTo* are as follows: If *pgFrom* contains memory spaces, then the outermost `par_group` in *pgsTo* must contain equivalent memory spaces. If the `par_unit` in `par_group` *pgFrom* contains memory spaces, then these memory spaces have to be represented in the innermost `par_unit` in *pgsTo*. These rules guarantee that the translation phase never has to insert new `barrier`-statements and that existing `barrier`-statements are translated into `barrier`-statements that use equivalent memory spaces.

Finding equivalence in units of parallelism Since the compiler may need to split `foreach` statements, the output of this phase is a list of mappings from a `par_group` to lists of `par_groups`. First, for hardware descriptions *hwdFrom* and *hwdTo*, the compiler finds the *executing* `par_unit`, which is the innermost `par_unit` defined in the parallelism hierarchy. It retrieves the `par_groups` of both and compares the number of units of parallelism to decide whether to split up layers in the parallelism hierarchy or not.

For example, in hardware descriptions *perfect* (Fig. 3.3) and *gpu* (Fig. 3.8), the executing `par_units` are `thread` and `thread`. The surrounding `par_groups` are `threads` and `threads`. However, the number of units of parallelism of `threads` in `perfect` is larger (`unlimited` vs. 1024), which means that the compiler also needs to incorporate the surrounding `par_group` of `threads` in *gpu*, which is `blocks`. The result is that `par_group threads` in *perfect* is mapped to both `blocks` and `threads` in *gpu*. More details are provided in App. B.2.

Translating memory spaces Each declaration and barrier statement in the function that has a memory space *msFrom* is translated into a declaration or barrier statement with an equivalent memory space *msTo*. To find an equivalent memory space, the compiler finds the `par_group` or `par_unit` *pgFrom* where *msFrom* is defined. Using the output of the previous phase, the compiler finds the equivalent `par_groups` *pgsTo* which will be the starting point for searching equivalent memory spaces. A memory space can be defined within a `par_group` or within a `par_unit`. If *msFrom* was defined in a `par_unit`, the compiler will look in the innermost `par_unit` to find an equivalent memory space. If the memory space was found in a `par_group`, it will look in the outermost `par_group`. The pseudo-code is provided in App. B.3.

Translating `foreach` statements We illustrate this pass using the translation from *accelerator* to *gpu*, where the code for *accelerator* is identical to the code in Fig. 3.2, except for the keyword `perfect` on line 1. The resulting code is shown in Fig. 3.9.

As explained above, a `par_group` can be mapped to multiple `par_groups`. The `foreach` statements can also define multiple dimensions as occurs on lines 5 and 6 in Fig. 3.2. The compiler generates three kinds of statements: dimension statements that indicate the new dimensions for the `foreach` statements, the new `foreach` statements themselves, and indexing statements that define how the old indices are computed from the new indices in the `foreach` statements.

First, the compiler determines the dimension of the inner `foreach` taking a standard value from the hardware description for `par_group` threads (line 1 and 6 in Fig. 3.9, for clarity we assume that `m` is larger than `nrThreadsM`). Then it generates the `foreach` remembering the old index variable in combination with the new dimension expression. This leaves the compiler with `n` threads in one dimension and `m/nrThreadsM` in the other. Since the compiler already used the maximum number of threads, it moves on to blocks and generates two `foreach` statements with `n` blocks for one dimension and `m/nrThreadsM` in the other. As all dimensions are now clear (`nrThreadsM`, `nrBlocksM`, and `n`), the compiler generates the indexing statements on lines 7 and 8. The algorithms are provided in App. B.4.

3.6.2 Operation Statistics

MCL can give users feedback on the number of data accesses, computational operations, and overhead in the form of indexing and control-flow operations, and the arithmetic intensity, the ratio between the number of data-access and computational instructions.

The compiler accomplishes this by determining the *output-defining blocks*, which are control-flow graph blocks on which the output that a function generates depend. In MCL, the compiler statically knows which variables are written, since expressions can only be written in an assignment or in a function call and the call-graph is always clear to the compiler.

To determine the *output-defining blocks* the compiler finds the declarations in the parameter list that are written and the declarations of the variables that are used in return expressions. It then uses standard data-flow techniques to find all blocks on which the output depends. We define the *control-flow blocks* as the set of blocks that perform control-flow and all of their dependent blocks that are not in the output-defining blocks. Finally, we define the *indexing blocks* as the blocks that define the indexing expressions, but are neither in the *output-defining blocks* or the *control-flow blocks.*

For all these blocks, the compiler finds the number of iterations they are executed. This analysis may be imprecise because it may not be clear statically how many times a loop is executed, and because blocks may be inside `if` statements that limit the number of executions. In these cases, the compiler tries to determine the number of iterations and issues a warning that the result is an approximation. However, often the information is still useful as an upper-bound. This is a good example of where our compiler analysis is allowed to be imprecise. If the compiler had to base a transformation on this information, the compiler would have to be conservative. However, only providing feedback opens new possibilities for the compiler to be useful.

For all of these three sets, the compiler gathers what kinds of operations are performed: arithmetic operations, data-accesses with the memory-space, and which `par_unit` performs the operation. The call-graph is analyzed from the leaves up, such that it is clear how many operations call-expressions per-

```
 1 foreach (int i in h threads) {
 2   foreach(int j in w threads) {
 3     float sum = 0.0;
 4     for (int y = 0; y < fh; y++) {
 5       for (int x = 0; x < fw; x++) {
 6         sum += f[y, x] * a[i + y, j + x];
 7       }
 8     }
 9     b[i, j] = sum / (fh * fw);
10   }
11 }
```

Figure 3.16: A convolution program for hardware description *perfect*.

form. Finally, the compiler summarizes all this data taking into account the approximation warnings if they occurred, and presents it to the user.

3.6.3 Data Reuse Analysis

To analyze possible data-reuse we limit ourselves to the variables in the *output-defining blocks* that perform indexing and are read. The compiler analyzes these variables in two phases. The first phase performs a simple data-reuse analysis, but if that analysis fails, it performs a more advanced analysis.

The simple form finds all dependencies of variable v including the dependencies of the indexing variables. If v does not depend on a loop, this is reported as data-reuse. For example, in Fig. 3.2, the data-reuse analysis reports that variable `a[i,k]` on line 9 does not depend on the loop variable `j` (line 6) and therefore this data is reused `m` times. Similarly, it reports that variable `b[k,j]` does not depend on loop variable `i` (line 5) and that the data-reuse is `n` times.

If an indexing expression depends on all loops, the compiler starts a more advanced data-reuse analysis that tries to report the data-reuse ratio. A data-reuse ratio less than or equal to one means no data-reuse, whereas a value higher than one means there may be data-reuse. Fig. 3.16 shows an example of such a case. Variable `a` on line 6 has an indexing expression that depends on all four loops.

The advanced data-reuse analysis can analyze per dimension or for all dimensions. It flattens the indexing expression (for all or one dimension) and it retrieves all loops surrounding a variable v taking into account where v was last defined. It then symbolically computes the data reuse ratio by dividing the number of iterations v is accessed divided by the indexing range of v.

For example, the data-reuse ratio for the first dimension of variable `a` on line 6 of Fig. 3.16 is (`h` * `fh`)/(`h` + `fh`). The compiler computes that the number of accesses to `a` is `h` * `fh`. However, since the indexing range of `a` is `h` + `fh` (`i` and `y` filled in, taking into account the offsets and steps of the loops), a maximum of `h` + `fh` different elements is accessed, leading to the above data-reuse ratio.

The compiler can report this as an expression (`h` * `fh`)/(`h` + `fh`), but users can also fill in representative values in the parameter list for `h` and `fh`, for example 2048 and 9 respectively. The compiler will then report back a sharing ratio of 8.96. This value is correct considering there is no sharing in the borders of a convolution.

Analyzing per dimension gives programmers an extensive insight in how data is reused. In the above situation, it makes clear that for a specific combination of `i` and `x` data in `a` will be loaded more than once. There are two features of MCL that make this possible: First, MCPL has true multi-dimensional arrays that make it possible to express the program in this manner. The second feature is that the compiler uses all the information expressed by the programmer to do analysis. For example, usually compilers work on an intermediate representation, for instance a representation that flattens the array accesses into one-dimension. However, this would make the data-reuse analysis less precise. In contrast, our compiler performs analysis using all dimensionality information available, which makes it possible to provide more precise feedback and to relate the feedback to the code that the programmer wrote.

Proposing faster memories An extension of the data reuse analysis tries to propose faster memories. Because the mapping from `par_units` and memory spaces to the physical hardware is well-defined in MCL, the compiler can give very specific feedback. For a variable that has data-reuse, the compiler retrieves the memory space it is declared in. As above there are two cases: the variable is independent from specific `foreach` statements, or the variable depends on all of them but there is still data-reuse.

In the first case, the compiler retrieves all memory-spaces that are available in the scope of this `foreach`. The `foreach` statement references the `par_group` in the parallelism hierarchy which also defines the relations between the memory spaces and the units of parallelism. It compares these memory spaces with the memory space of the variable to determine whether there is a memory space that is faster.

In the second case, the compiler performs the same analysis, but on each pair of loops for each dimension. The outer `foreach` loop is used to determine the set of available memory spaces for choosing a faster memory space from.

For determining which memory space is faster, the compiler inspects the memories that hold the memory spaces. For example, in Fig. 3.8 on lines 14 and 15, memory `on_chip` holds memory space `local`. To determine whether a memory a is faster than memory b there are several heuristics: First, the compiler can inspect the interconnects between the memories and the execution units and compare the bandwidth and latencies. Second, if these details are not available, it can determine the proximity to execution units: the closer, the faster. Finally, it can compare the capacities of the memories, the smaller the memory, the faster we assume the memory to be.

3.6.4 Cache Analysis

The cache analysis tries to provide information about the cache usage of variables inside loops in terms of best-case and worst-case scenarios. Given a cache defined in the hardware description, the analysis retrieves the memory spaces that the cache holds. The analysis proceeds for every declaration that has one of these memory spaces.

For each cache the compiler determines the `par_units` that have access to the cache. A cache can reside on an execution unit that has a restricted number of slots for the `par_units`. This is illustrated in Fig. 3.12 where execution unit `smp` has slots for 8 `blocks` and 1024 `threads`. This would represent an upper-bound for the cache misses for a variable if a cache were defined to reside on this execution unit.

For caches that do not reside on an execution unit but are shared among execution units, the compiler visits the interconnects in search of execution units and the `slots` property to find out which `par_units` have access to the cache. The upper-bounds are determined by interpreting the layout of the execution unit in relation to the cache. If a cache can be reached by multiple execution units, the upper-bound for cache misses for a variable is defined to be the number of parallelism units that can reach it. Additionally, the compiler determines whether a `par_group` is executed in lock-step for the `load` and `store` instruction, which is the case if a load-store unit resides in a `simd_group`.

For a variable, the compiler retrieves the surrounding loops taking into account where the variable was last defined. If the array expressions of the variable are unpredictable, this is reported to the user. Array expressions are unpredictable if an array expression contains non-loop variables that are written or originating from arrays.

Otherwise, the compiler analyzes each loop separately. For a loop it symbolically fills in the start value of the loop into the array expression and the start + the step of the loop into the array expression. It symbolically subtracts these values to determine the stride that a loop causes in the array expression.

Based on this stride and using the knowledge about the cache-line size from the hardware description, the compiler gives the programmer feedback about the best and worst case scenario in terms of cache-line loads. If the stride is zero, then the best case is 1 cache-line fetch, otherwise the number of accesses per cache-line is *cachelinesize/stride*. The worst case is the number of iterations in a loop, or the upper-bound that was determined above if it is a `foreach` loop that has a limited amount of slots for the `par_unit` under analysis.

By comparing the best-case and worst-case, this analysis provides the user feedback about whether this variable does, may, or does not benefit from the cache for this loop. Units of parallelism that execute in lock-step and load fewer cache-lines than the number of units of parallelism, are reported to benefit from the cache since multiple threads access the same cache-line.

3.6.5 Performance Feedback Functions

The complex many-core hardware often has complex rules for execution. As shown in Fig. 3.12 and 3.13, HDL provides an API in Java to encode complex rules, for example coalesced memory access or how many blocks can run on an SMP.

A performance feedback function is automatically called by the compiler when encountered in a hardware description. It takes as input a *Kernel* data-structure and returns a string with a message for the user that will be issued by the compiler. The *Kernel* data-structure contains a *Context*, a set of *Instructions*, and the variables from the hardware description. From an *Instruction* we can request whether it is a memory instruction or not, in which memory-space it loads or stores, which `par_unit` is executing the instruction, and we can obtain a String representation. The *Context* data-structure contains a special execution context that can obtain values from the function under investigation. For example, the *Context* has a method `int getSizeMemorySpace(String)`. Hardware description variables can be referenced using an expression such as `Kernel.on_chip.capacity` for the capacity of on-chip memory.

The above API is dynamically generated by the compiler since certain functions rely on both the structure of the hardware description and the function under investigation. An example is the call `kernel.context.hierarchy.threads.getSize()` that relies on the structure of the parallelism hierarchy and the value of the number of threads that is specified in the program. An interpreter interprets exactly those statements in the MCPL program that are needed to compute the size. If a value of variable v is not available, the compiler issues a warning that it needs a value for v to complete the analysis.

The interpreter interprets `for`-loops, but not `foreach`-statements. To iterate over the units of parallelism in a `foreach`-statement, the API provides a `reset()` and `increment()` statement. This limits the execution for the interpreter making the analysis more scalable. For example in analyzing coalescing, we are only interested in 32 iterations for threads in a warp, which can be expressed by issuing `kernel.context.hierarchy.blocks.block.threads.increment()` 32 times in a for-loop.

This API is powerful enough to analyze the number of blocks per SMP as illustrated in Fig. 3.13, coalesced memory access for threads and the number of bank conflicts for threads.

3.7 Evaluation

To evaluate our approach we implemented several computational kernels and recorded the received feedback and the performance increase for two very different architectures: an NVIDIA GTX480 GPU and the Intel Xeon Phi. The focus of our framework is to give feedback on individual kernels. Therefore, we selected a variety of programs (compute-bound, bandwidth-bound, regular, and irregular) with one kernel. In Tables 3.4 and 3.5 we give an overview of the results we obtained. To reduce the size of the table, we collapsed the different versions in the inner process (v1, v2, v3) into one step (v1) in some of the applications.

Our compiler generates OpenCL code for the kernels. For each target hardware description, a configuration file describes how the programming abstractions map to OpenCL code. This ensures that the OpenCL compiler generates executables based on the same set of parameters for each compared kernel, which limits a potential performance difference caused by the OpenCL compiler.

We chose a GPU with the Fermi architecture because the cache is important on this architecture which makes the analysis more challenging for our compiler. Additionally, much related work makes use of GTX480 cards which allows us to compare our results. With the choice for the Xeon Phi, we show that we can support a very different architecture than a GPU. Since the Xeon Phi is a new many-core card, we could not find a comparison for all kernels. In Table 3.6 we compare our results with other existing implementations. We discuss the details

Table 3.4: Feedback and performance from compiler on different abstraction-levels for several applications for the GTX480.

Application	Version	Program Change	Performance
		NVIDIA GTX480 GPU	
matmul	perfect		89.0 GFLOPS
	gpu (v1)	use faster memory	100 GFLOPS
	gpu (v2)	multiple elements	91.6 GFLOPS
	gpu (v3)	data reuse	205 GFLOPS
	nvidia (v1)	store-load phases	489 GFLOPS
	fermi (v1)	cache usage	564 GFLOPS
convolution	perfect		129 GFLOPS
	gpu (v1)	use faster memory multiple elements	215 GFLOPS
	nvidia (v1)	scarce memory	241 GFLOPS
	fermi (v1)	cache usage	302 GFLOPS
histogram	perfect		6.94 GB/s
	gpu (v1)	multiple elements data reuse	80.0 GB/s
	nvidia (v1)	scarce memory	80.0 GB/s
gesummv	perfect		2.58 GFLOPS
	gpu (v1)	use faster memory	2.63 GFLOPS
	nvidia (v1)	scarce memory	3.69 GFLOPS
	fermi (v1)	cache analysis	80.2 GFLOPS
blackScholes	perfect		297 GFLOPS
	gpu (v1)	multiple elements	432 GFLOPS
sparse matmul	perfect		1.01 GFLOPS
	gpu (v1)	data reuse	1.19 GFLOPS
	gpu (v2)	multiple elements	1.41 GFLOPS
	fermi (v1)	cache usage	3.88 GFLOPS

Table 3.5: Feedback and performance from compiler on different abstraction-levels for several applications for the Xeon Phi.

Application	Version	Program Change	Performance
		Intel Xeon Phi (5110P)	
matmul	perfect		39.9 GFLOPS
	mic (v1)	multiple elements	35.7 GFLOPS
	mic (v2)	data reuse	47.6 GFLOPS
	xeon_phi (v1)	cache usage	87.8 GFLOPS
	xeon_phi (v2)	cache usage	115 GFLOPS
	xeon_phi (v3)	cache usage	488 GFLOPS
convolution	perfect		66.4 GFLOPS
	mic (v1)	multiple elements	84.0 GFLOPS
	mic (v2)	data reuse	153 GFLOPS
	xeon_phi (v1)	cache usage	171 GFLOPS
histogram	perfect		0.35 GB/s
	mic (v1)	multiple elements	23.6 GB/s
		data reuse	
gesummv	perfect		0.84 GFLOPS
	mic (v1)	multiple elements	0.68 GFLOPS
	xeon_phi (v1)	cache usage	55.4 GFLOPS
blackScholes	perfect		66.6 GFLOPS
	mic (v1)	multiple elements	54.1 GFLOPS
	xeon_phi (v1)	cache usage	115 GFLOPS
sparse matmul	perfect		2.71 GFLOPS
	mic (v1)	data reuse	2.84 GFLOPS
	xeon_phi (v1)	cache usage	2.90 GFLOPS

Table 3.6: Performance comparison for the applications compared to known, fully optimized versions (* measured on a C2050, ** using a different input than in Tables 3.4 and 3.5).

Application	Naive	MCL	Other implementations
NVIDIA GTX480 GPU			
matmul	89.0 GFLOPS	564 GFLOPS	892 GFLOPS
convolution	129 GFLOPS	302 GFLOPS	380 GFLOPS
histogram	6.94 GB/s	80.0 GB/s	80 GB/s
gesummv*	1.25 GFLOPS	51.9 GFLOPS	2.98 GFLOPS
blackScholes	297 GFLOPS	432 GFLOPS	237 GFLOPS
sparse matmul	1.01 GFLOPS	3.88 GFLOPS	8.9 GFLOPS
Intel Xeon Phi (5110P)			
matmul	39.9 GFLOPS	488 GFLOPS	695 GFLOPS
convolution	66.4 GFLOPS	171 GFLOPS	n/a
histogram**	0.331 GB/s	11.9 GB/s	0.3 GB/s
gesummv	0.84 GFLOPS	55.4 GFLOPS	n/a
blackScholes	66 GFLOPS	115 GFLOPS	n/a
sparse matmul**	3.93 GFLOPS	5.28 GFLOPS	13 GFLOPS

below.

matrix multiplication Matrix multiplication is a compute-bound, regular application. The process for this application has been thoroughly explained in Sec. 3.5 and resulted in a speedup greater than 6 on a GPU (564 GFLOPS) and a speedup of over 10 on the Xeon Phi (488 GFLOPS) compared to the naive implementation when used with 2048×2048 matrices. The CUBLAS 5.5 library obtains 892 GFLOPS and the Intel MKL library 695 GFLOPS. We consider our result as very good performance as these libraries are heavily tuned, most likely written in assembly (a much lower level of abstraction) by experts who have much more knowledge of the architecture than the programming guides provide us.

convolution Convolution is a stencil operation computing an element in the output matrix based on its neighbors. It is somewhat less regular than matrix multiplication and more bandwidth-bound. We used a 4096×4096 matrix with a filter size of 9×9. For hardware description *gpu*, the compiler identifies data-reuse in the filter and in both dimensions in the input data. This prompted us to load both data-structures in local memory. On level *nvidia*, we received the feedback that we should use more blocks per SMP. However, we as programmers have application knowledge and understand that this is a delicate balance between what can be shared and how many blocks can be run in parallel. We succeeded in running 2 blocks per SMP. On level *fermi* we improved cache behavior by changing the number of threads.

For the Xeon Phi, the compiler advised us to compute multiple elements per thread on level *mic*. Based on the data-reuse analyzed by the compiler, we also decided to load the filter in a temporary variable since it is heavily shared. At level *xeon_phi* the compiler identifies that the cache behavior of the output data has the most severe effect. Allocating private memory for the output resulted in 171 GFLOPS. We also received the feedback that we have unaligned vector loads in accessing neighboring elements. This limits the performance on the Xeon Phi.

For the GPU we obtained a speedup of more than a factor 2.3 and on a Xeon Phi a factor 2.5 versus our *perfect* implementation. The implementation with the highest performance on a GTX480 obtains 580 GFLOPS but uses loop unrolling and a specialized kernel for each filter size [70]. Our compiler generates a kernel for any filter size, so a more fair comparison is with the version that does not do loop unrolling which obtains 380 GFLOPS. The Xeon Phi performance is limited by unaligned vector loads in accessing neighboring elements.

histogram Histogram is a bandwidth-bound application with irregular data-access in the output histogram. In effect, it is a reduction, which means that there should be synchronization overhead. For this application, we computed a histogram of 256 buckets from an input of 16384×16384 elements. Since there is only one integer addition per word that we read, we measure the performance in GB/s (where a kB is 1024 bytes).

The compiler indicated that there is much data-reuse in the output array, which we can interpret as synchronization overhead. On level *gpu*, we first computed multiple elements and minimized the synchronization overhead by replicating the histogram in local memory. This led to a performance of 80.0 GB/s. We would expect a performance close to 145 GB/s, but because atomic operations are expensive on shared memory, we can not obtain better results. In comparison with the CUDA histogram implementation, we have the same performance. On level *nvidia* we received the feedback that shared memory is scarce, but the improvement did not result in higher performance.

For the Xeon Phi, on level *mic* the compiler advised to compute multiple elements and that there is much data-reuse in the histogram. Computing more elements per thread and replicating histograms led to a dramatic increase in performance. The cache analysis on level *xeon_phi* did not give feedback we could act on. Our version outperforms other work [71] by a factor 40 (5.25 ms against 200 ms on an input of 2^{24}).

gesummv This application from the PolyBench benchmark [72] performs two matrix-vector multiplications. Implementing several improvements on level *gpu* and *nvidia* did not contribute to much performance gain. However, at level *fermi* it became clear that the application has bad cache behavior. After solving these issues, the performance reached 80.2 GFLOPS. The bandwidth reached 140 GB/s and the compiler reported that there is not much data-reuse in the matrices. Since the maximum bandwidth on a GTX480 is 145 GB/s and there is no data-reuse that we can leverage, we know that we are close to the maximum performance. We compared our results with the results in the paper by Grauer-Gray et al. [72]. Their auto-tuned version has a run-time of 25.4 ms and their manually written version is 1.5 times faster than their auto-tuned version on an input of 4096×4096 elements on a Fermi C2050 GPU. However, our version is an order of magnitude faster with a run-time of 1.46 ms on the same hardware. A possible explanation of this difference is that their version may have missed the cache optimization that resulted in a significant performance increase in our version (Table 3.5).

The Xeon Phi initially obtained much lower performance than the GPU. Unfortunately, the feedback that we received initially made matters worse due to the overhead of computing multiple elements. However, on level *xeon_phi* we received feedback about similar cache issues as on the GPU and we solved it in a similar way resulting in a dramatic speedup of over 65.

Black-Scholes This application analyzes stock options, in our case $4 * 10^6$ transactions. The compiler reports that this is a very compute intensive application and the initial performance of 297 GFLOPS at level *perfect* on the GPU confirms this. Following the feedback on level *gpu* led to a performance of 432 GFLOPS. The feedback on levels *nvidia* and *fermi* did not provide any opportunity to improve the program. The performance we obtain is good as the CUDA SDK version obtains 237 GFLOPS.

For the Xeon Phi we are advised to compute multiple elements on level *mic*. Unfortunately, this led to worse performance. On level *xeon_phi* it becomes clear that the application suffered from bad cache behavior for the vector instructions. Correcting this led to a version that obtains 115 GFLOPS.

sparse matrix multiplication Sparse matrix multiplication is a very irregular application. We used a 65536×131072 sparse matrix, randomly filled with a .001 density in CSR representation.

In the GPU version, the feedback on levels *gpu* and *nvidia* did not help much. On level *fermi*, the cache feedback initially indicated that the source vector access is irregular and that the cache effect of the matrix access cannot be determined due to unknown loop bounds. This prompted us to use another approach, to at least improve the cache behavior in the index and matrix access, by using multiple threads to compute output elements. In the end, however, the performance of this application is dominated by the irregular access to the source vector. The end result is about 2.3 times slower than the CuSparse 5.5 implementation, which achieves about 8.9 GFLOPS on the same input. We suspect that the library is using texture memory, which MCL does not support yet.

The Xeon Phi version initially has good performance and can be slightly improved by taking the cache feedback of level *xeon_phi* into account. We compared our results with the results from [73] using the Mip1 matrix and observe that we are a factor 2.5 slower. This is comparable to the difference with CuSparse on a GPU.

The evaluation shows that following the *stepwise-refinement for performance* methodology has several benefits: Firstly, we obtain significant speedup in almost all cases. Only in sparse matrix multiplication we hardly obtain speedup using a specific input. However, for a different input (the Mip1 matrix from [73]), our implementation does show significant speed-up (Table 3.6).

Secondly, we are guided by feedback from the compiler. This steers us in the right direction and helps us understand the performance bottlenecks. For example, with convolution for the Xeon Phi, we receive the feedback that we have unaligned vector accesses. In the gesummv application we understand that we are close to the maximum performance, since the compiler tells us that there is no data-reuse and we observe a performance close to the maximum bandwidth.

Finally, our methodology also shows the trade-offs between abstraction-levels. A high-level of abstraction increases portability (programs written for level *perfect* can run on both the GPU and the Xeon Phi), but incorporating hardware-specific details in the program can increase the performance significantly and also helps programmers to understand the performance in relation to the hardware.

3.8 Discussion

In this chapter we show a methodology with which programmers work in cooperation with the compiler to gain both performance and understanding of this performance. Programmers with application knowledge can judge whether the compiler gives reasonable feedback. For example, in the convolution application, the compiler suggests to minimize the use of shared memory to allow for more blocks per SMP, which is usually good feedback. However, we as programmers understand that in this application, the speed-up we gain relies on the amount of data we share.

Our methodology can be combined with auto-tuning. Often, to reach even higher performance, auto-tuning of parameters such as the number of threads per block is necessary. Since programmers gain an understanding of the limits of the program and hardware after receiving feedback from the compiler, they have the opportunity to choose a good search space for auto-tuners, possibly making auto-tuning much more efficient.

We chose to design a new programming language besides the hardware descriptions because this makes it easy to 1) define an explicit mapping to hardware and 2), based on the conclusions in Chapter 2, to restrict the language to enable better analysis. However, if the right restrictions on other languages such as CUDA or OpenCL are enforced, augmented with annotations, there is no reason why our techniques could not be applied to other languages. The annotations would have to provide data-structure size information, dimensionality of the data-structures, memory space information, and units of parallelism.

The different versions that the programmer creates could be automatically documented together with the received feedback for each version. For example a standard versioning system could maintain the different kernel versions together with meta-data files that store the feedback. This system then provides a way to not only automatically document the optimizations that have been applied, but also the trigger for applying a specific optimization in the form of the received feedback. In effect, it can document the reasoning of the programmer and it captures the optimization knowledge.

Another benefit of having multiple versions of compute kernels is that the system could verify the optimized versions in an automated way. The system could run example inputs on kernels on different abstraction-levels reporting inconsistent results between high-level and low-level versions.

The techniques that our compiler uses are mainly targeted at providing high-quality feedback. For future work, we would like to investigate more advanced techniques to make the feedback more powerful. For example, we would like to investigate whether polyhedral analysis [74, 43] can be combined with our language and compiler framework such that we can still provide high-quality feedback that helps the programmer to improve the program.

Another way to improve the analyses is to extend our performance feedback functions to support advanced performance models. For example, incorporating the Roofline model [75] might provide more accurate information on when to stop optimizing. Additionally, we would like to investigate whether we can incorporate the Xeon Phi's cache model by Ramos et al. [76] in a performance feedback function.

Finally, we would like to improve the hardware descriptions. For example, in the current hardware descriptions it is not determined how instructions are scheduled, although this may impact the performance. For instance, this may benefit the Xeon Phi programs because the Xeon Phi has multiple pipelines with complex rules for how instructions from multiple threads are scheduled onto those pipelines.

3.9 Conclusion

In this chapter we presented the *stepwise-refinement for performance* methodology, a novel approach to provide programmers control and understanding on levels of abstraction they can choose themselves. As such, this chapter proposes solutions for the following sub-questions:

2. How to balance control over hardware with raising the level of abstraction?

3. How can we manage the many different types of many-core hardware that exist?

4. Can we provide programmers a structured approach with which a programming system can assist them to achieve high performance?

In this chapter, we proposed a methodology centers around hardware descriptions that specify many-core hardware with different levels of detail, resulting in multiple levels of abstraction. This provides programmers a trade-off between high-level programming which is good for portability and code maintainability, and low-level programming with a well-defined interface to the hardware.

The *stepwise-refinement for performance* methodology supports a structured approach in which programmers are gradually exposed to more hardware details in an iterative process. During this process, the compiler gives programmers detailed performance feedback about the programs to increase their understanding of the performance they obtain. This approach leverages the strengths of both the programmer and compiler: the programmer has application knowledge that the compiler lacks, and the compiler does not have to be as conservative in providing feedback as it has to be for automatic transformations. Using our methodology, the programmer and compiler work together to produce high-performance code.

To show the process of optimizing many-core programs with this methodology, we presented Many-Core Levels, our programming system that supports our methodology. MCL contains a hardware description language and a programming language that are tightly integrated and help the compiler to produce detailed, hardware-specific performance feedback with a strong relation to the code that programmers wrote.

Since there are many types of many-core hardware, programmers need to manage the optimizations for the various devices. By organizing the hardware descriptions in a hierarchy, programmers can make an informed trade-off for the optimization process in relation to the devices that they are targeting.

We gave a thorough example of the optimization process to explain how programmer and compiler work together to produce high-performance code and how programmers can make a trade-off in targeting multiple devices or continuing to optimize for specific devices. We explained how MCL automatically translates between abstraction-levels, how we analyze data-reuse, cache behavior and how we encode complex architecture-specific rules to give feedback. We evaluated our approach with several well-known, widely varying (compute-bound, bandwidth-bound, regular, and irregular) many-core programs on two different architectures: a GPU and a Xeon Phi. We demonstrated that our methodology provides programmers a well-defined hardware interface that gives them a trade-off in portability against performance. We showed that it is possible to obtain substantial speed-ups in almost all cases with the important benefit that programmers not only gain performance but also understand the performance they obtain.

Chapter 4

Cashmere: Heterogeneous many-core computing

New generations of many-core hardware become available frequently and are typically attractive extensions for data-centers because of power-consumption and performance benefits. As a result, supercomputers and clusters are becoming heterogeneous and start to contain a variety of many-core devices. Obtaining performance from a homogeneous cluster-computer is already challenging, but achieving it from a heterogeneous cluster is even more demanding. Related work primarily focuses on homogeneous many-core clusters.

In this chapter we present Cashmere, a programming system for heterogeneous many-core clusters. Cashmere is a tight integration of two existing systems: Satin from Chapter 2 is a programming system that provides a divide-and-conquer programming model with automatic load-balancing and latency-hiding, while Many-Core Levels (Chapter 3) is a programming system that provides a powerful methodology to optimize computational kernels for varying types of many-core hardware. We evaluate our system with several classes of applications and show that Cashmere achieves high performance and good scalability. The efficiency of heterogeneous executions is comparable to the homogeneous runs and is >90% in three out of four applications.

4.1 Introduction

Many-core devices offer enormous potential in compute-power and can increase the performance of supercomputer and data-center applications significantly. Developments in the many-core field move fast: new generations of many-core hardware become available frequently and are fitted in data-centers because of performance and power consumption benefits. However, older-generation accelerators may still be powerful for some applications. For instance, in our DAS-4 cluster, we have older-generation Fermi GTX480 GPUs that for some applications are as fast as the newer generation Kepler GTX680 GPUs [70].

The result is that supercomputers and data-centers will more and more contain multiple types of many-core hardware. As an example, our DAS-4 cluster [77] has a wide variety of many-core hardware, multiple generations of NVIDIA GPUs, AMD GPUs, and Intel Xeon Phi's. As another example, Table 4.1 shows several TOP500 supercomputers (as of November 2014) that contain more than one type of many-core device [78].

Extracting performance from many-core clusters is difficult in general, but heterogeneity makes it even more challenging. There is a variety of problems that need to be solved to obtain high-performance:

1. Since execution times are likely to vary among many-core devices, some form of load-balancing is needed.
2. Load-balancing is especially challenging for heterogeneous many-core clusters, because it requires communication between nodes. However, because many-cores are so fast, the network speed is relatively slow compared to clusters without many-cores, resulting in a skewed computation/communication ratio.
3. The computational kernels that will run on the many-core devices have to be identified, written, and optimized for each type of hardware.
4. The run-time system should know the type and number of available many-core devices in each node and map the right kernels to the devices.
5. The application needs logic to drive the execution on these many-core devices.

Related work focuses mostly on extracting high-performance from *homogeneous* many-core clusters, an already difficult task. Often, the overall solution is based on MPI combined with a programming language for many-core devices,

Table 4.1: TOP500 Supercomputers with heterogenous many-core devices.

name	institute	rank	configuration
Quartetto	Kyushu University	49	K20, K20X, Xeon Phi 5110P
Lomonosov	Moscow State University	58	2070, PowerXCell 8i
HYDRA	Max-Planck-Gesellschaft	77	K20X, Xeon Phi
SuperMIC	Louisiana State University	88	Xeon Phi 7110P, K20X
Palmetto2	Clemson University	89	K20m, M2075, M2070
Armstrong	Navy DSRC	103	Xeon Phi 5120D, K40
Loewe-CSC	Universitaet Frankfurt	179	HD5870, FirePro S10000
Inspur TS10000	Shanghai Jiaotong University	310	K20m, Xeon Phi 5110P
Tsubame 2.5	Tokyo Institute of Technology	392	K20X, S1070, S2070
El Gato	University of Arizona	465	K20, K20X, Xeon Phi 5110P

such as OpenCL [59] or CUDA [58]. However, because execution times vary among types of many-core hardware, a solution based on MPI with its rigid communication patterns is not the ideal solution.

In this chapter, we present Cashmere, a programming system that focuses on obtaining performance from *heterogeneous* many-core clusters. Cashmere is a tight integration of two existing systems: Satin (already discussed in Chapter 2), a programming system that provides a divide-and-conquer programming model with load-balancing and latency-hiding [7] and Many-Core Levels (MCL) providing the powerful methodology “Stepwise-refinement for performance” to optimize computational kernels for varying types of many-core hardware (Chapter 3). Satin addresses problems 1) and 2) described above, MCL addresses problem 3), and Cashmere integrates the two systems provides solutions for problems 4) and 5), and extends Satin’s load balancing for multiple many-core devices per node. We named our system Cashmere because it offers parallelism in the form of fine-grained threads.

Our contributions are the following:

- We seamlessly integrate coarse-grained divide-and-conquer parallelism with multiple fine-grained many-core parallelism levels for a variety of many-core devices with minimal changes to the original Satin programming model (Sec. 4.2).

- We describe several optimizations that are necessary to obtain high performance (Sec. 4.3).
- We demonstrate that our "stepwise-refinement for performance" methodology enables us to develop optimized many-core kernels for various devices. This scalable development is a prerequisite for achieving the heterogeneity that Cashmere targets (Sec. 4.4 and 4.5).
- We evaluate Cashmere with several classes of applications and show that we can achieve good scalability and performance despite the skewed communication and computation ratio (Sec. 4.4 and 4.5). Our heterogeneous runs are comparable to homogeneous runs and in three of four applications we achieve >90% efficiency with optimized kernels.
- We answer research questions 5 and 6 of this thesis.

After an overview of related work (Sec. 4.6) we present our conclusions (Sec. 4.7).

4.2 Cashmere Programming Model

Cashmere is a system for programming high-performance applications for clusters with many-core accelerators and forms a tight integration of two existing programming systems: Satin [7] and MCL (Chapter 3). Section 4.2.1 gives a brief overview of Satin and its programming model, Sec. 4.2.2 describes MCL, and Sec. 4.2.3 discusses how Satin and MCL interact and form the Cashmere programming model.

4.2.1 Satin

Satin [7] is a programming system that targets grids or clouds of clusters. It is inspired by Cilk [9] that targets multi-core processors. Similar to Cilk, Satin has a divide-and-conquer programming model that allows programmers to express computation in a hierarchical manner. Satin achieves very good scalability by mapping this computation to the hierarchical structure of grids or clouds of clusters.

The key features of Satin that we use are:

- **load-balancing** Satin uses random work-stealing to achieve load-balancing.
- **latency hiding** Overlap slow communication with computation.

```
 1  spawnable f(a) {
 2    if (small_enough_for_leaf(a)) {
 3      return do_leaf_computation(a)
 4    }
 5
 6    r1 = f(make_smaller(a)) // asynchronous
 7    r2 = f(make_smaller(a)) // asynchronous
 8    sync
 9
10    return combine(r1, r2)
11  }
```

Figure 4.1: Skeleton of a Satin program.

- **fault tolerance** Satin recovers from nodes that are no longer responding.
- **shared objects** Shared objects make the divide-and-conquer programming model less restrictive by allowing programmers to use a custom consistency model.

Other than in Chapter 2, we will show Satin and Cashmere code in pseudo-code as Java-specific code is irrelevant for the discussion and Cashmere could equally well be written in another language. Figure 4.1 shows a basic skeleton of a Satin program in pseudo-code. Line 1 shows a recursive function `f()` that is declared to be spawnable, which means that a function call will execute asynchronously. Lines 6 and 7 show recursive calls to `f()`. The Satin system creates a child job for each call that will be asynchronously computed on a compute node that Satin has available. This may be the same node, another node in the cluster, or a node on another cluster. The results of the two calls are stored temporarily in `r1` and `r2` but are not available until the `sync` statement has finished. The function blocks on the `sync` statement until all previous child jobs have finished. After the `sync` statement, the function can return the result based on `r1` and `r2`.

The above mechanism creates (possibly) recursive jobs that are spread among the compute nodes of clusters. Satin achieves good scalability since all nodes steal from each other using random work-stealing. At some point, the jobs are small enough to perform the actual computation. Lines 2 to 4 show the stop-condition based on an application defined parameter that decides to perform the leaf computation. Section 4.2.3 will show how we extend this model with MCL.

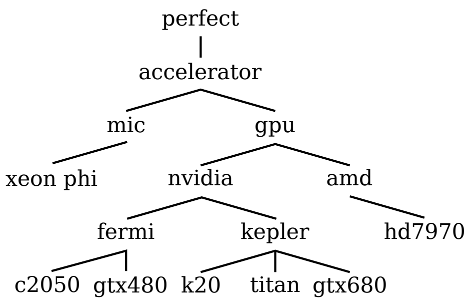


Figure 4.2: Hierarchy of hardware descriptions.

4.2.2 MCL

As discussed in Chapter 3, Many-Core Levels is a programming system that allows programmers to write computational kernels for many-core hardware. The programming system is designed such that programmers can choose their own abstraction-level for their program to trade-off performance, portability and ease of programming. MCL offers abstraction levels by providing a library of hardware descriptions that are organized in a hierarchy. Figure 4.2 shows the hierarchy of hardware descriptions used for evaluating Cashmere.

4.2.3 Cashmere programming model

Cashmere is targeted at leveraging the fine-grained parallelism that many-core hardware offers on a large scale, typically clusters with compute nodes that contain a variety of many-core hardware such as GPUs or Intel's Xeon Phi.

Adding many-cores to the original Satin programming model provides additional levels of parallelism:

1. *parallelism on a many-core device* A many-core device itself exposes many levels of parallelism: instruction-level parallelism, task parallelism, and SIMD parallelism.

2. *multiple many-core devices per node* A compute node in a cluster can contain multiple (heterogeneous) many-core devices.

3. *overlap in communication and computation* Typically, many-cores are connected through a PCI Express bus. Many-core devices can overlap communication between host and device with computation.

```
1 leaf(a, b)
2   try {
3     Kernel kernel = Cashmere.getKernel()
4     KernelLaunch kl = kernel.createLaunch()
5     MCL.launch(kl, a, b)
6   catch (exception) {
7     leafCPU(a, b)
8 } }
```

Figure 4.3: Calling an MCL kernel in Cashmere.

Cashmere aims to exploit these levels of parallelism with minimal changes to the original Satin programming model and leveraging MCL for writing the computational kernels.

parallelism on a many-core device

Expressing the parallelism on the device is completely handled by MCL. Programmers write a kernel in MCL's programming language MCPL targeting a hardware description from the library. The MCL compiler will generate code for each of the leaf hardware descriptions and glue-code for Cashmere. This results in a minimal and convenient way to call MCL code from within Cashmere while maintaining Satin's fault-tolerance. Figure 4.3 shows a possible leaf computation. In this figure, the MCL kernel needs parameters `a` and `b` (lines 1 and 5). On line 3 in the `try/catch` clause, we retrieve the MCL kernel from Cashmere. From the kernel we create a launch `kl` on line 4 that we launch with the MCL front-end with parameters `a` and `b`. The MCL front-end makes sure that all necessary data is copied to the many-core device, it selects the appropriate kernel(s) for the devices available on the node, executes the kernel, and copies the data back. In case something goes wrong with the kernel execution, the system raises an exception which will then start the leaf computation on the CPU (line 7).

Figure 4.3 shows the basic scheme, but more advanced schemes are possible:

- **multiple kernels** The above scheme works for an application with only one kernel. Cashmere will automatically find this kernel and load it. If there are more kernels, the `Cashmere.getKernel()` function should have a string parameter that identifies the kernel to be loaded.

- **multiple kernel-launches** It is possible to launch the kernel multiple times in succession. For example, it is possible to put a loop around lines 4 and 5.

- **device copies** If there are multiple kernel launches, it may be unnecessary to transfer all the parameters to the many-core device each time. Cashmere offers the functions `Kernel.getDevice()` and `Device.copy()` to copy data to and from the many-core device for multiple kernel launches.

multiple many-core devices per node

If there are multiple many-core devices on a node, the standard Satin programming model and calling MCL as explained in the above paragraph will not lead to parallel execution of MCL kernels on both devices, because a call to `MCL.launch()` is blocking. We solved this problem with a minimal change to the divide-and-conquer programming model. Since the `spawnable` functions already express parallelism, we reuse this to express parallelism for the many-core devices. Figure 4.4 shows a skeleton of a typical Cashmere program. Compared to the Satin skeleton in Fig. 4.1, lines 5 to 7 have been added. If on line 5 the job is small enough, then the function `Cashmere.enableManyCore()` disables generating jobs for the compute nodes in the cluster. Instead, it continues to create jobs using the same mechanism of spawnable functions and `sync`, only now for the many-core devices on a node. If in turn the jobs generated for the many-core devices are small enough to execute the leaf computation, governed by the stop-condition on line 2, then the leaf computation running the MCL kernels will be started. In conclusion, by adding one library function, the divide-and-conquer model of Cashmere expresses parallelism among nodes and among many-core devices on a node.

overlap in communication and computation

The above mechanism also overlaps computation and communication to the many-core device over the PCI Express bus. If a node has multiple jobs available, Cashmere can launch kernels for one job and copy data for another. Cashmere automatically manages the available memory on a device.

In summary, the Cashmere programming model is similar to the original Satin programming model. It is extended with a library call to express parallelism between many-core jobs and it provides a simple front-end to call MCL kernels. MCL is used to write the kernel code and is extended to generate glue code for Cashmere.

```
spawnable f(a) {
  if (small_enough_for_leaf(a)) {
    return do_leaf_computation(a)
  }
  else if (small_enough_for_many_core(a)) {
    Cashmere.enableManyCore()
  }

  r1 = f(make_smaller(a)) //asynchronous
  r2 = f(make_smaller(a)) //asynchronous
  sync

  return combine(r1, r2)
}
```

Figure 4.4: Skeleton of a Cashmere program.

4.3 Implementation

This section describes the Cashmere implementation. Section 4.3.1 explains the role of MCL in Cashmere while Sec. 4.3.2 describes how the MCL kernels fit in the divide-and-conquer system.

4.3.1 MCL

To write kernels for multiple many-core devices, Cashmere makes use of several capabilities of MCL:

translation between abstraction-levels Hardware descriptions consist of definitions for the physical device and for the programming abstractions that define how code is mapped to the physical device. MCL can automatically translate kernels written for the programming abstractions of hardware description x to the programming abstractions of a child level y. Since each lower-level hardware description contains more detailed information, the mapping between programming abstraction and physical device becomes more precise. During this translation process the compiler does not apply optimizations.

generating OpenCL code For each leaf node in the hierarchy, there is a configuration file that tells the compiler how the programming abstractions from the hardware description map onto the OpenCL constructs. This means, together with translation between abstraction levels, that MCL can generate code from each abstraction level.

Generating Cashmere code Applying the stepwise-refinement methodology leads to multiple files with different versions of the same kernel. For instance, given a kernel written on level *perfect*, suppose programmers know the AMD HD7970 GPU well and choose to apply optimizations on level *gpu*, *amd*, and *hd7970*. This leads to four different files: a file with a kernel on level *perfect* and files on levels *gpu*, *amd*, and *hd7970*. The programmers can select these files and generate Cashmere code for these devices.

MCL will generate OpenCL code for each of the seven leaf nodes in Fig. 4.2 and automatically chooses the most specific kernel version for a device. This means that in the above situation, the Xeon Phi has a kernel on level *perfect*, all NVIDIA GPUs have kernels on level *gpu* and the HD7970 GPU has a kernel on level *hd7970*.

Together with the OpenCL code, MCL automatically generates glue code that calls the kernels with the right configuration for OpenCL's work-groups and work-items (parameters that determine the available parallelism) for inclusion in the divide-and-conquer framework of Cashmere.

This is important because the different devices have different granularity needs. For example, the Xeon Phi needs more coarse-grained parallelism than a GPU. MCL determines the work-group and work-item configuration based on the kernel parameters and its hardware-descriptions.

4.3.2 Cashmere

The Cashmere runtime was built with the Satin runtime as basis and extended it with a module that enables many-core functionality. This module has three tasks:

- managing and discovering many-core devices,
- launching kernels and sending data between the devices and host,
- and keep detailed statistics and timing information about the kernel execution.

Since MCL generates OpenCL code, the Cashmere runtime uses OpenCL to discover devices and it makes heavy use of the OpenCL event system to schedule kernels and data transfers. Cashmere also extends this event system to provide detailed timing information and synchronize the timing between nodes in the cluster.

Finally, Satin was built on the Ibis library [8] that provides primitives for communication between nodes in a cluster. We have modified Ibis to use RDMA over Infiniband instead of IP over Infiniband to increase the network bandwidth. The Cashmere code adds about 3100 lines of code to Satin including comments and whitespace. The rest of this section explains the specifics about how Cashmere runs an application.

On initialization In the initialization phase of an application, Cashmere assigns one node to be the master; the others become slaves. The kernels for the many-core devices may depend on run-time information that only the master node has. Therefore, the master broadcasts this run-time information to each slave. On each node, Cashmere retrieves which devices are available and after receiving the run-time information, compiles the most specific kernels for its compute devices. If there is a device on a compute node that is not available in the hierarchy of hardware descriptions, Cashmere suggests to add a hardware description for this device, so that it can compile a kernel for this device.

spawning jobs to other nodes If the application encounters a spawnable function, Cashmere generates jobs that can be stolen by other nodes. The master is the first that generates jobs that other nodes can steal. As soon as a node has jobs, each node in the cluster can randomly steal from other nodes which contributes to scalability and load-balancing. Stealing a job encompasses transferring the input data to the requesting node, execution of the job on the requesting node (possibly generating new jobs that can be stolen) and transferring back the output data. This all happens automatically.

spawning jobs to the many-core devices If the application encounters `Cashmere.enableManyCore()` Cashmere switches to a new mode. On encounter of a spawnable function, Cashmere no longer generates jobs that other compute nodes can steal, but instead it creates a thread that executes the spawnable function. The main thread will continue executing, possibly creating new threads for spawnable functions and will block on the `sync` statement until all threads have finished.

These threads can either generate more threads or encounter a call to an MCL kernel. In the last case, the input data for the kernel is scheduled to be copied to the device, the kernel is scheduled to run after the copies have completed, and the data transfer back to the device is scheduled to be run after the kernel execution. Because multiple threads are scheduling transfers and kernels, the data transfers can be completely overlapped with kernel executions except for the first and last data transfers.

Cashmere automatically load-balances the jobs scheduled to run on the many-core devices of a compute node. Initially, Cashmere uses a heuristic based on a static table of relative many-core device speeds to schedule the first jobs. For example, the table states that a K20 GPU has speed 40 and a GTX480 speed 20. When these jobs have completed, we know the execution time for each kernel for a specific device. Based on this time Cashmere submits the jobs to the different queues for each device trying to minimize the overall execution time for all jobs. For example, if the queue for a K20 has 3 jobs with an execution time of 100ms and the queue for the GTX480 has a queue with one job of 125ms, then Cashmere submits the job to the GTX480 queue because the execution time of this scenario is less: $min(scenario1, scenario2)$ where $scenario1 = max(4 * 100ms, 1 * 125ms)$ and $scenario2 = max(3 * 100ms, 2 * 125ms)$. This is possible because leaf jobs in a divide-and-conquer application typically have the same size.

4.4 Methodology

In this section we describe our methodology to show that Cashmere obtains high performance on heterogeneous many-core clusters. Our test-bed is the VU DAS-4 cluster [77], consisting of 74 dual Xeon E-5620 quad-core nodes that communicate with a QDR Infiniband interconnect. We evaluated Cashmere with the seven many-core devices available on this cluster:

- 22 NVIDIA GTX480 GPUs
- 8 NVIDIA K20 GPUs
- 2 Intel Xeon Phi (each fitted in a K20 node)
- 2 NVIDIA C2050 GPUs
- 1 NVIDIA Titan GPU
- 1 NVIDIA GTX680 GPU
- 1 AMD HD7970 GPU

Table 4.2: The classes of applications that we use to evaluate Cashmere.

application	type	computation	communication
raytracer	irregular	heavy	light
matmul	regular	heavy	heavy
k-means	iterative	moderate	light
n-body	iterative	heavy	moderate

We use 4 applications to evaluate Cashmere. Each application has its own characteristics and represents a class of applications. Table 4.2 shows an overview of how we classify each application to evaluate Cashmere.

Raytracer This application is based on smallpt [79], and its GPU port SmallptGPU [80]. It is a path tracing raytracer that leads to very realistic images given that each pixel is computed with many random samples from each object in the scene. This is an interesting application for Cashmere because of two reasons: First, the application is highly compute intensive. The amount of data that is processed is $\mathcal{O}(no)$ where n is the number of pixels and o the number of objects in a scene. The computation is $\mathcal{O}(nods)$ where d is the depth (the number of times a ray is bounced off an object) and s is the number of random samples. The number of samples determines the quality of the picture. Raytracer is also interesting because, although the application is compute-intensive, the application is highly irregular because of the random samples. The application has much control-flow based on randomness, making it difficult to optimize. Raytracer represents the class of highly parallel and compute-intensive irregular applications.

Matrix Multiplication Matrix multiplication multiplies two dense matrices of single-precision floating-point numbers. For multiplying two $n \times n$ square matrices, the amount of data that is processed is $3n^2$. The amount of computation is about $2n^3$, which means that we have a factor n more computation than communication.

This application is interesting for evaluating Cashmere because although it is a compute-intensive application, the application is highly regular, making it relatively easy to optimize. However, this then results in relatively high communication costs making this application difficult to scale. This application represents the class of regular, compute- and communication-intensive applications.

K-means K-means clusters a set of n d-dimensional data-points in k clusters. Given an initial set of d-dimensional centroids that represent the k clusters, k-means assigns the n data-points based on the distance to the centroid of the cluster. Based on the set of data-points belonging to one of the centroids, a new value for the centroid is computed. This process is repeated until there are no changes.

This application is interesting for evaluating Cashmere because it is an iterative algorithm that needs to update k values after each iteration and distribute these back to the compute nodes. A single iteration is compute-intensive and spawns the jobs over the nodes. For one iteration, communication is $\mathcal{O}(k)$ and computation is $\mathcal{O}(kn + k)$.

This application represents the class of iterative applications with minimal (constant) communication between the iterations.

N-body simulation This application simulates the forces between n bodies over time. As K-means, it is an iterative application but has a different complexity for computation and communication. Each iteration, the effect of each body on each other body has to be computed, which makes the computation $\mathcal{O}(n^2)$. After each iteration, the positions and accelerations have to be updated for each body, making the communication $\mathcal{O}(n)$.

N-body represents the class of iterative applications with intensive communication and is interesting for evaluating Cashmere because of its communication pattern (all-to-all for each compute node).

We use MCL to write kernels for each application. First, we write a kernel on level *perfect* and generate code for each of the 7 leaf hardware descriptions in Fig. 4.2. This is a kernel written on a high level and we consider this the *unoptimized* kernel. For each application we also optimize for each device. We consider this the *optimized* version. In Chapter 3 we compared the performance of MCL applications against that of hand-optimized applications from the literature, showing that MCL performance overall is in line with other published results.

We then perform the following scalability studies on one type of hardware:

- **Satin** These measurements show the original performance of Satin and how well it scales. The results from these measurements help to put in perspective the performance that we obtain with many-core hardware and the scalability that we achieve.

- **Cashmere with non-optimized kernels** These measurements show the scalability and performance difference between Satin and Cashmere with minimal effort because the kernels are written on a high level.

- **Cashmere with optimized kernels** These measurements show the performance of Cashmere when the computational time is reduced several factors.

Finally, we evaluate heterogeneous runs using various types of hardware and compute the efficiency by dividing the measured performance by the maximum attainable performance. We determine this by summing the measured performance for one node for each node in the configuration. We compare the efficiency to the efficiency of the homogeneous execution. We use the optimized kernels to evaluate the heterogeneous runs. Each of these studies provides strong scalability.

4.5 Evaluation

The following subsection discusses the kernel performance differences between the optimized and non-optimized versions. Section 4.5.2 shows the scalability studies and the absolute performance difference between the applications. Finally, Sec. 4.5.3 presents our findings on heterogeneous runs.

4.5.1 Kernel performance

This section shows the effect of the *stepwise-refinement for performance* methodology applied to the computational kernels. Since the hardware descriptions are organized in a hierarchy, optimizing the kernels becomes scalable. For example, optimizations that have been applied on level *gpu* are used for both NVIDIA and AMD GPUs, so these optimizations are used for four different devices.

Fig. 4.5 shows the kernel performance. The performance numbers are based on the timings of kernel execution alone without any overhead such as copying data to the device. It is clear that optimizing has a drastic effect on the kernel performance for most devices except for Raytracer. This can be explained by the irregularity of the kernel. Raytracer has much control-flow overhead and since the control-flow is based on randomness, threads often diverge, which has severe performance penalties. To obtain better performance from the raytracer would mean a different algorithm, something MCL cannot suggest. Raytracing

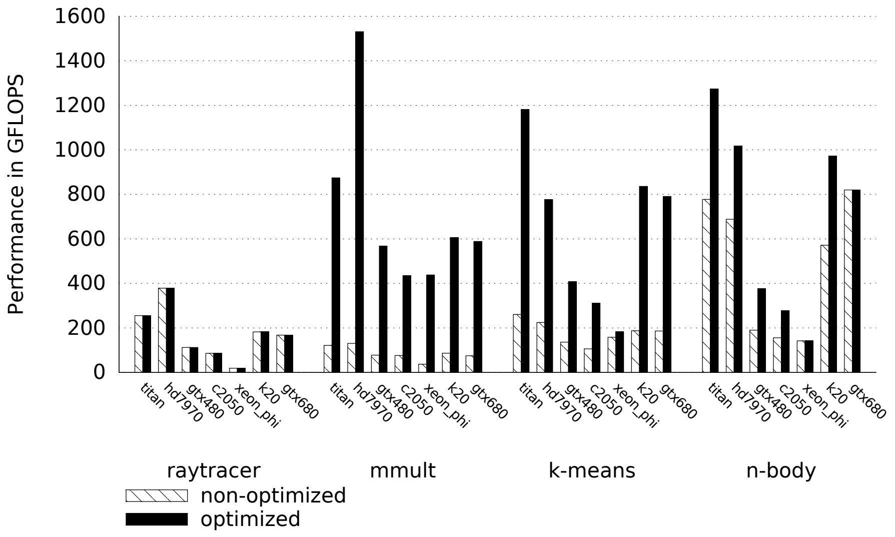


Figure 4.5: The performance of the kernels for the applications for the unoptimized version and the optimized version.

is known to be challenging on many-cores and Xiang et al. discuss hardware solutions for this kind of kernels [81].

4.5.2 Scalability

This section evaluates whether Cashmere is able to obtain similar scalability results as Satin, which is our aim. We also compare the absolute performance difference between Satin and Cashmere. To our surprise, Satin scales worse than Cashmere in most of the cases. We found that there are two factors that contribute to this reduced scalability. Firstly, Satin has more overhead in job creation because it needs to create 8 times more jobs to keep one node busy. This can be explained by the difference in programming models: In the Satin programming model, a leaf computation is single-threaded, whereas one node has two quad-core processors, which means that to keep one node busy, Satin has to run 8 jobs in parallel. In contrast, a leaf computation in Cashmere already exposes parallelism for the many-core device. Hence, Cashmere does not have to create as many jobs as Satin. The second factor that contributes to worse scaling is that since all cores on the CPUs are fully occupied with computation, communication and load-balancing tasks suffer from the lack of available compute-power.

Raytracer

Figure 4.6 shows that Raytracer scales better with Cashmere than with Satin. This is a good achievement as the absolute performance of Cashmere is an order of magnitude higher as shown in Fig. 4.7. Since the optimized and non-optimized kernels are similar in performance, the two Cashmere versions overlap. We performed our measurements on the Cornell scene [79, 80] with a resolution of 16384×8192 with 500 random samples.

Matrix Multiplication

Figure 4.8 shows the performance results of multiplying 2 32768×32768 single-precision floating point matrices. We can see that Matrix Multiplication does not scale very well, also for Satin. The graph makes clear that the scalability suffers from the relatively slower networking speed when the kernel is optimized. However, Fig. 4.9 shows that there is still a factor four absolute performance difference between optimized and non-optimized versions.

K-means

Figure 4.10 shows that K-means scales well, even for the optimized version and better than Satin. Figure 4.11 shows the absolute performance of the three versions. We computed 4096 clusters out of 268 million points with 4 features in three iterations.

N-body

Figure 4.12 and 4.13 show that N-body has similar results as K-means despite higher communication costs. We simulated two iterations of 2 million bodies.

4.5.3 Heterogeneity

In Table 4.3 we show the performance of the four applications with 2 configurations dependent on the availability of nodes on the cluster. For the K-means and N-body experiments all 7 types of hardware were available simultaneously.

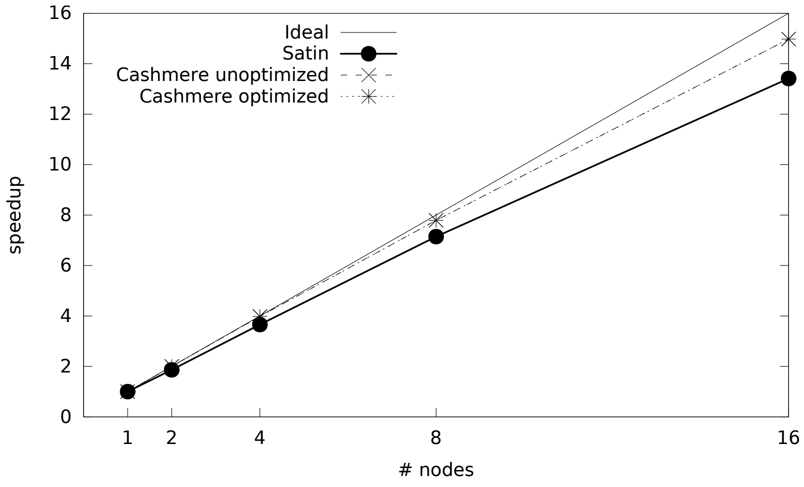


Figure 4.6: Scalability of Raytracer up to 16 GTX480 GPUs.

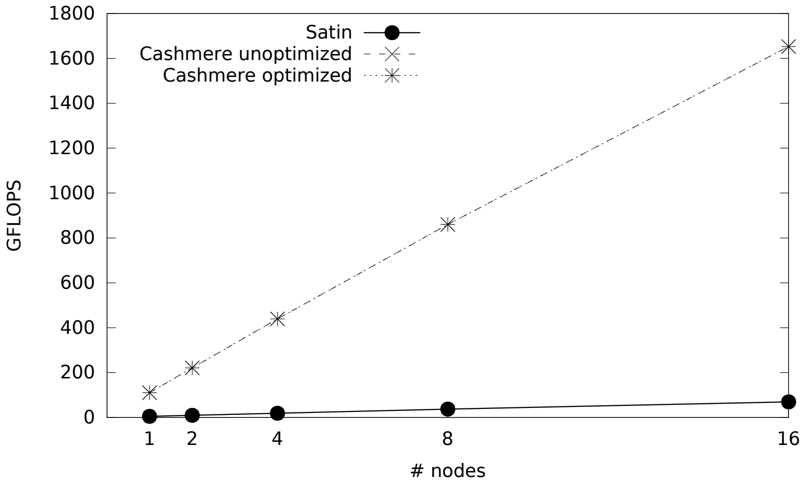


Figure 4.7: Absolute performance of Raytracer up to 16 GTX480 GPUs.

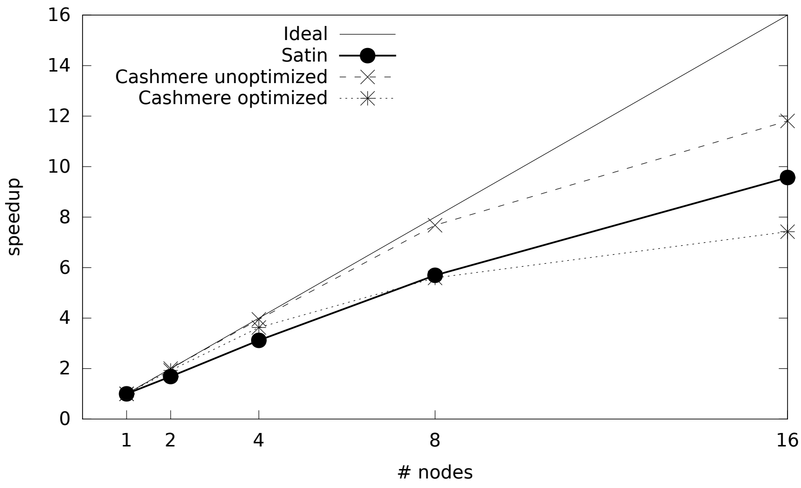


Figure 4.8: Scalability of Matrix Multiplication up to 16 GTX480 GPUs.

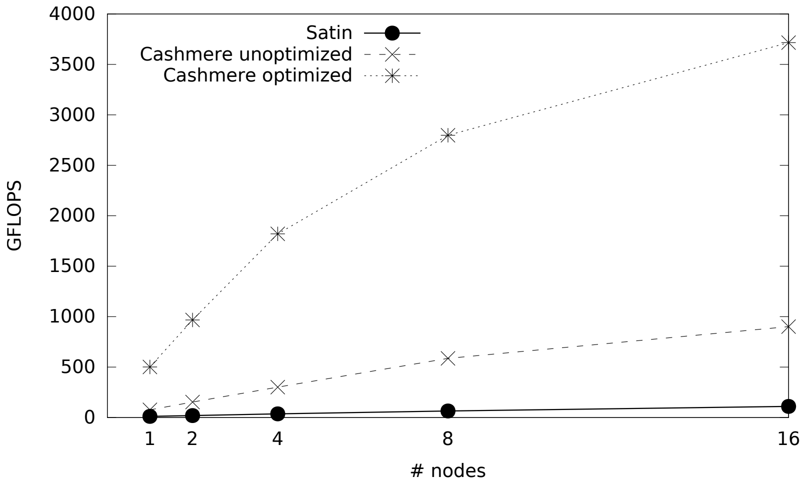


Figure 4.9: Absolute performance of Matrix Multiplication up to 16 GTX480 GPUs.

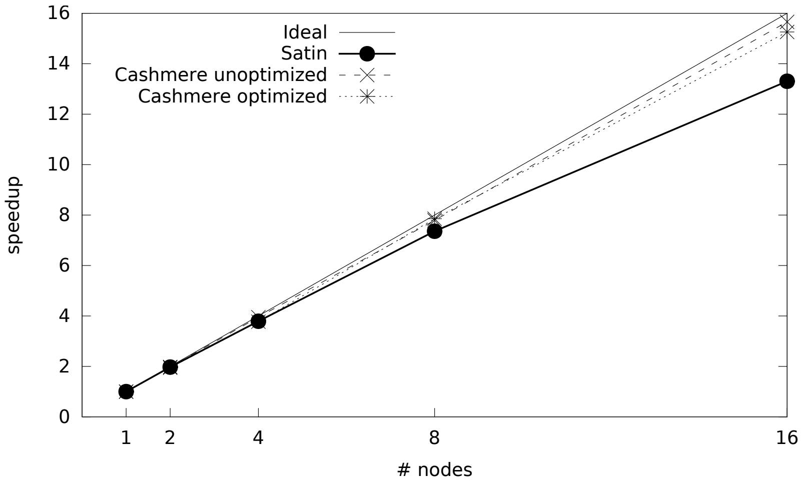


Figure 4.10: Scalability of K-means up to 16 GTX480 GPUs.

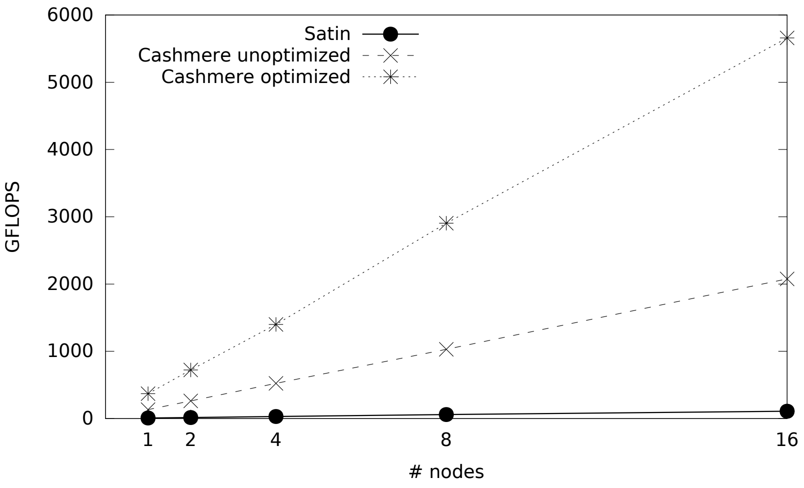


Figure 4.11: Absolute performance of K-means up to 16 GTX480 GPUs.

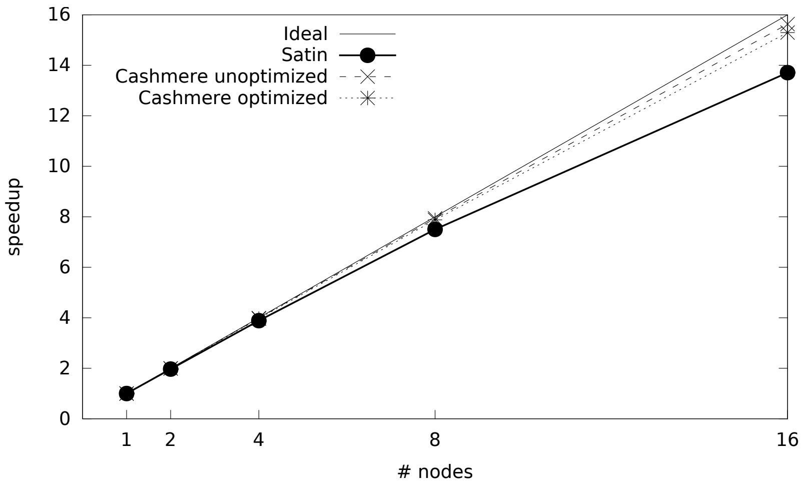


Figure 4.12: Scalability of N-body up to 16 GTX480 GPUs.

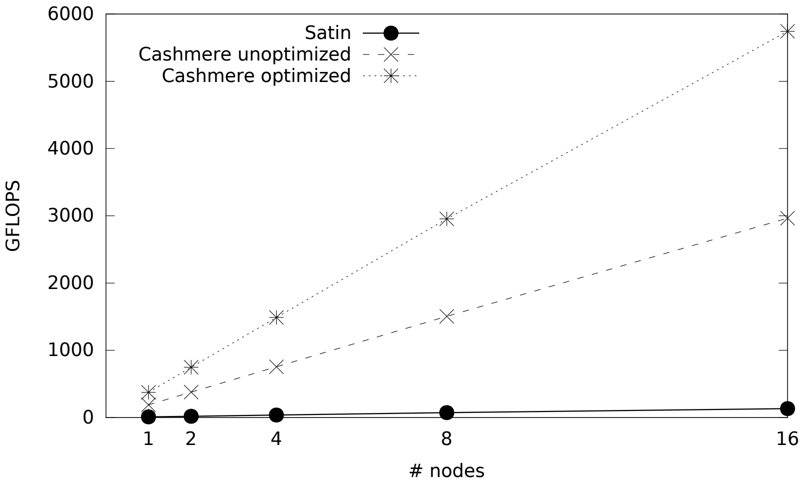


Figure 4.13: Absolute performance of N-body up to 16 GTX480 GPUs.

Table 4.3: Performance of the heterogeneous executions.

application	performance (GFLOPS)	configuration
raytracer	1883	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970
matmul	3927	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970
k-means	10644	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970, 7 k20, 1 xeon_phi
n-body	13517	10 gtx480, 2 c2050, 1 gtx680, 1 titan, 1 hd7970, 7 k20, 2 xeon_phi

We used the optimized kernels for our measurements. Fig. 4.14 shows the efficiency of the applications compared to the combined performance of one-node execution for each type hardware (i.e., the sum of 10× #GFLOPS of a one-node execution on the GTX480, 2× #GFLOPS of the one-node execution on the C2050, etc.). We compared this to the efficiency of the homogeneous executions for 16 GTX480 nodes from Sec. 4.5.2. We conclude that the efficiency for heterogeneous runs is similar to the homogeneous runs.

These results are especially impressive considering that in Matrix Multiplication the performance between the kernel versions varies widely (Fig. 4.5), thus showing that Cashmere prevents load-imbalance very well. The results for K-means and N-body are noteworthy as well because the efficiency is about the same as the homogeneous execution while using one third more nodes, achieving twice the performance with the heterogeneous runs, and after each iteration communication between all nodes is necessary.

Finally, to gain insight in these excellent results, we show a Gantt chart of the K-means run. Since there is so much parallelism, it is difficult to show all details. We therefore show a zoomed-in version of the Gantt chart (Fig. 4.15) where we can see two nodes, one with a GTX480 GPU and one with a Xeon Phi and a K20 GPU. The y-axis shows different queues denoted with ‘qn’. Each queue contains activities that can be overlapped with activities in other queues. The narrow bars are for CPU tasks, sending data from and to the device and to other nodes. The wider bars in q4 are kernel executions. The chart shows parallel execution of a Xeon Phi kernel, overlapped with (faster) K20 kernels and GTX480 kernels on node 3.

Figure 4.15 also shows our load balancing algorithm at work as explained in Sec. 4.3.2. Node 16 executes sets of 8 jobs and after each set synchronization

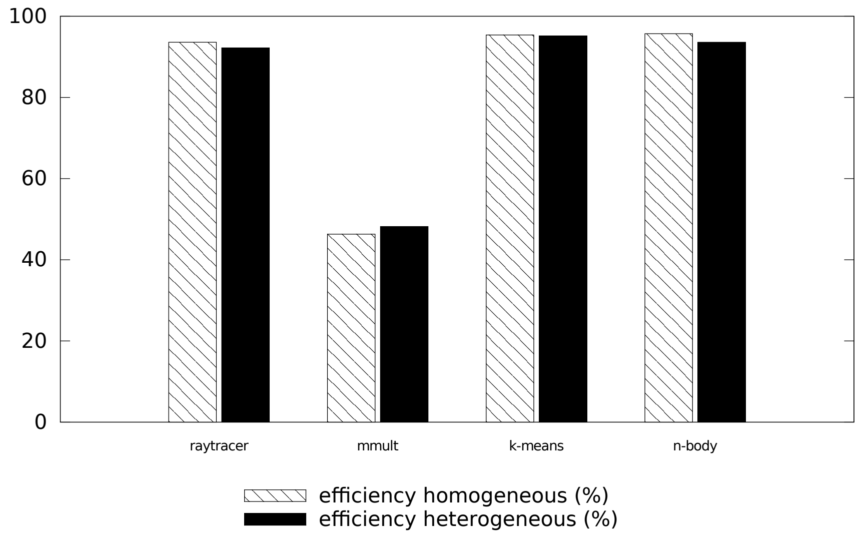


Figure 4.14: Efficiency of heterogeneous executions.

is required. Our load balancing algorithm schedules 1 job on the Xeon Phi and 7 on the K20 which is the fastest configuration. Since the Xeon Phi is about 4 times slower than the K20 (see Fig. 4.5), one job more on the Xeon Phi would lead to a longer overall execution time and not scheduling a job on the Xeon Phi would also be longer.

Figure 4.16 shows the zoomed-out version of the Gantt chart in which we left out all activities except kernel executions. It shows that this kind of execution can be maintained each iteration, thus showing Cashmere's effectiveness.

4.6 Related Work

In this section we focus on programming systems that – similar to Cashmere – aim to simplify many-core computing on clusters and supercomputers.

MapReduce is a framework that allows programmers to express computations in terms of map and reduce functions [82]. Several frameworks target many-core clusters. GPMR is a Map-Reduce framework for GPU clusters [83]. It fully relies on CUDA which makes the framework homogeneous. Their scal-

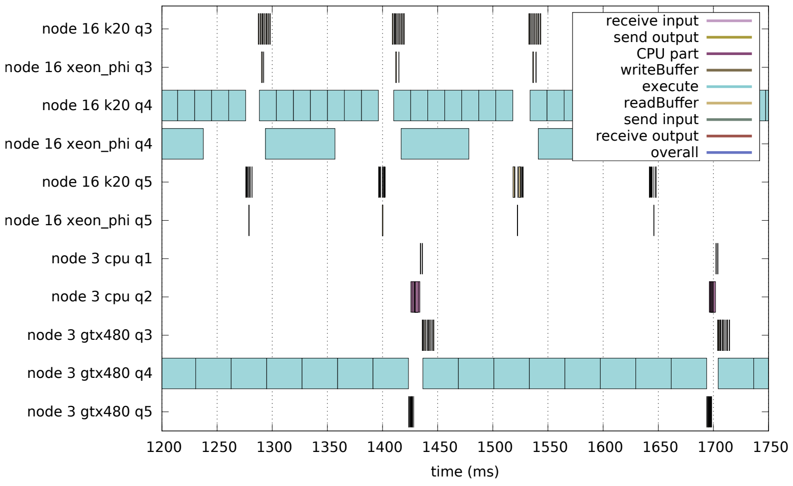


Figure 4.15: Zoomed-in view of the Gantt chart of heterogeneous K-means execution.

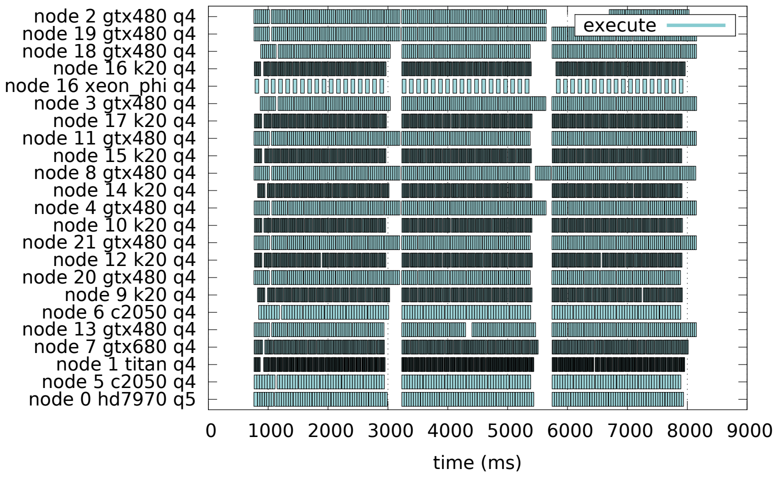


Figure 4.16: Gantt chart of heterogeneous K-means execution.

ability study does not take into account copying data back and forth between compute-nodes which is essential in practice.

HadoopCL [84] extends Hadoop [85] with OpenCL to make the computational power of many-cores available to Hadoop jobs. It uses APARAPI [86] to translate limited Java code to OpenCL. Cashmere offers similar functionality in a library to make use of MCL. HadoopCL performance is only compared to original Hadoop and obtains a speedup of 5 on 10 nodes with GPUs against 2 nodes of Hadoop without GPUs on the K-means application. Cashmere is an order of magnitude faster when compared to original Satin (a speedup of 186 on 8 GPU nodes compared to 2 Satin nodes).

Glasswing [87] is a MapReduce framework fully written in OpenCL and C++ and has a significant performance benefit over the previous two frameworks. Glasswing is set up on top of the Hadoop filesystem and gains performance by overlapping computation with I/O in several deep pipelines. Glasswing supports multiple architectures but was not evaluated with multiple different architectures at the same time. Glasswing also performs a $32k \times 32k$ Matrix Multiplication on the DAS-4 and obtains a performance of 2082 GFLOPS on 16 GTX480 GPUs while we obtain 3716 GFLOPS with the optimized version. However, Glasswing supports out-of-core data, data that is larger than the memory of a node, which Cashmere does not support yet.

OmpSs [88] combines OpenMP pragmas with StarSs [89] pragmas to program clusters of GPUs. It offers a sequential programming model in which programmers can annotate parallel regions with directives to indicate the data-dependencies with in, out, and inout statements between tasks and on what kind of device a task should run.

Planas et al. [90] extended the system with an adaptive scheduler that can choose multiple versions of compute-kernels and learns to choose the better performing version. In contrast to our system, the various versions are not organized in a hierarchy and our system does not have to learn which version to choose, but automatically chooses the most specific version for the many-core device.

Bueno et al. [88] report slightly higher performance on 8 GTX480 nodes for Matrix Multiplication (just above 3 TFLOPS compared 2.8 TFLOPS for the optimized kernel for Cashmere). However, they use a CUBLAS kernel that has higher initial performance than our optimized kernel.

StarPU [91] provides an execution model based on tasks. Programmers implement tasks in the native many-core programming system, for example CUDA or a BLAS routine and they annotate them with the tasks on which they depend. The run-time system handles data-dependencies, scheduling of tasks, and load balancing. Experiments in [91] have been performed on much older GPUs making a comparison difficult.

SkePU [45] is built on top of StarPU and provides a skeleton framework that allows programmers to express a program in terms of frequently used patterns that are encoded in generic constructs. The program remains sequential while the implementation of the patterns is parallel. Since the paper only reports relative speedups, comparing performance is difficult. However, SkePU does not appear to scale as well as Cashmere: for the best configuration of N-body, SkePU obtains a speedup of about 2.6 on 3 GPU nodes.

Ernsting and Kuchen do show absolute performance numbers for their skeleton framework Muesli [92] that supports GPU clusters for NVIDIA cards. They report the highest performance number for one node containing four NVIDIA GTX480 GPUs with an 8192×8192 matrix, which is 48.3 GFLOPS. This is low performance compared to Cashmere: With four nodes each having one GTX480 GPU (which requires communication over the network instead of within a node) Cashmere obtains 299 GFLOPS for an unoptimized kernel and 1102 GFLOPS for an optimized kernel for an 8192×8192 matrix.

PaRSEC [93, 94] is a framework with a task-based run-time to overcome the problem that MPI does not scale well for more dynamic applications. Programmers have to manually annotate data-flow dependencies between tasks with their granularity and PaRSEC does not have an integrated solution for programming kernels as Cashmere has. However, PaRSEC provides a more general programming model than divide-and-conquer. PaRSEC only has a CUDA back-end and was not evaluated with heterogeneous many-core devices.

4.7 Conclusion

Many-core programming is challenging because many-core hardware exposes complicated hardware interfaces that vary widely among devices. However, many-cores are becoming the norm in clusters and data-centers as a result of performance and power-consumption benefits. This chapter focused on the following two sub-questions:

5. How to achieve good scalability when programming *clusters* of many-cores?
6. How to program *heterogeneous* many-core clusters?

Sub-question 5 is a difficult problem because the high performance of many-cores makes the communication overhead between nodes in the cluster an even bigger problem than with traditional processors. Sub-question 6 focuses on how to optimize for the various devices in a cluster and how to achieve high performance with possibly large relative performance differences among the various devices.

Our solution, Cashmere, seamlessly integrates the many levels of parallelism that arise as a result of using many-cores in a cluster setting that are needed to achieve high performance. MCL allows programmers to write and optimize kernels for different many-core devices in which common optimizations can be shared thanks to the support for multiple abstraction levels. Our approach delivers high performance and automatic load balancing even when the many-core devices differ widely. Cashmere achieves high efficiency (>90% in three out of four applications) in heterogeneous executions by exploiting multiple levels of parallelism.

Chapter 5

Conclusions

In the mid 2000s, hardware manufacturers met the “energy wall”: although they could still increase the number of transistors, the increase of clock frequency came to a halt. This hardware limitation had severe consequences for software: to gain a similar performance increase, software had to be parallel.

In the coming years, it is very likely that we encounter more and more hardware limitations, resulting in hardware that exposes an increasingly complex interface to programmers, such as multiple levels of parallelism and complicated memory hierarchies.

We consider many-core processors as a first manifestation of this trend: its hardware trades logic targeted at optimizing sequential instruction streams, for logic that mostly performs computations in order to achieve as much as possible performance for the same energy footprint. These processors are solely targeting high performance and make no compromises in the hardware interface exposed to programmers. Because of this, there will be many different types of hardware, each with its own unique characteristics to achieve high performance. This makes many-core hardware very challenging to program.

5.1 Summary

In this thesis, we propose solutions for this programming problem. In Chapter 2 we presented an analysis to relieve programmers from the burden to place synchronization statements. Our understanding of the limitations of the compiler in relation to Satin’s programming model guided the design of Many-Core

Levels and helped us to answer the first sub-question:

1. What are important design considerations for parallel programming models and their compiler analyses?

We learned that aliasing is difficult to analyze, but that certain restrictions may have a positive effect. We also learned that the compiler may become part of the platform programmers are targeting, leading to programmers adapting their program for the compiler instead of adapting their program to real hardware which may be preferable for many-core hardware. Finally, it showed that compiler analysis can be imprecise because of the lack of application knowledge that programmers have.

This understanding guided the design of Many-Core Levels (Chapter 3), a programming system that provides solutions for the following sub-questions:

2. How to balance control over hardware with raising the level of abstraction?
3. How can we manage the many different types of many-core hardware that exist?
4. Can we provide programmers a structured approach with which a programming system can assist them to achieve high performance?

Our main answer to these questions is the *stepwise-refinement for performance* methodology that MCL supports. Inspired by the observation in Chapter 2 that a compiler can become part of the platform, MCL tries to target real many-core hardware. We do this by describing hardware formally and incorporate these hardware descriptions in the programming model. The hardware descriptions are organized in a hierarchy where each child hardware description exposes more hardware details to the programmer than its parent. As such, MCL provides a solution for sub-question 2 allowing programmers to start on a high-level of abstraction and optimize the program for each level guided by the compiler. The compiler can give performance feedback because the mapping between algorithm and hardware is made clear in MCL while programmers remain in control over the applied optimizations.

The hierarchy of hardware descriptions also provides a solution for sub-question 3. Because of the hierarchy of hardware descriptions, optimizations on a certain level impact all programs written for child hardware descriptions. In essence, programmers are offered a trade-off in the level of abstraction. A high level of abstraction provides code maintainability and portability, while lower levels provide programmers more detailed performance feedback and control over the hardware.

We provide a solution for sub-question 4 in the form of the methodology *stepwise-refinement for performance* and our programming system MCL. The compiler has much knowledge about the hardware, much knowledge about the programs, and about how the program is mapped to the hardware. With this information, the compiler can give programmers feedback on a level of detail that matches the level of abstraction the programmers are working on.

Chapter 4 combines Satin from Chapter 2 and MCL described in Chapter 3 to result in Cashmere, a programming system for programming heterogeneous many-core clusters. Cashmere succeeds in making the computational power accessible to clusters with minimal changes in the Satin programming model.

Chapter 4 provides solutions for the following sub-questions:

5. How to achieve good scalability when programming *clusters* of many-cores?

6. How to program *heterogeneous* many-core clusters?

Achieving good scalability on a cluster in a many-core context is very difficult because the computational power is much higher than for traditional clusters while the bandwidth of the network remains the same. However, Cashmere still achieves good scalability, even with a heterogeneous configuration (multiple different many-core devices) because the use of these devices relieves the CPU to use its resources to perform load-balancing and networking. Additionally, we reuse Satin's divide-and-conquer programming model to express parallelism for overlapping data transfers to and from the many-core device, and to use multiple devices per node.

Cashmere provides an integrated solution for sub-question 6: MCL is used to write and optimize kernels for multiple many-core devices. The MCL compiler generates Cashmere code which makes it very simple to include MCL kernels into Cashmere. At run-time, Cashmere automatically loads the most-specific kernels for the hardware devices that are available on the compute nodes of the cluster and automatically load-balances Cashmere's divide-and-conquer programs. Finally, Cashmere shows a detailed Gantt-chart to monitor the performance of the system.

5.2 Future Directions

Although MCL and Cashmere provide solutions for the programming problems at hand, there are still many opportunities to improve our work. This section shows a number of future directions for MCL and Cashmere.

Versioning One of the limitations of MCL is that programmers potentially have to maintain multiple versions of a kernel, which is more difficult than maintaining only one version. However, this problem can be mitigated by a system that provides versioning for the different kernels.

We envision a system that keeps track of the kernels and the feedback generated by the compiler and combines this information to provide provenance of optimizations. The system then allows programmers to retrieve what optimizations had been applied and which feedback they were based on, thus capturing the optimization knowledge. An additional possibility is to add this functionality to our Eclipse plugin such that editors can automatically hide or show the different versions of the kernels.

Finally, this system could provide functionality to automatically run and compare lower-level kernels against higher-level kernels. This can help programmers to assure that the kernels remain correct over optimizations.

Auto-tuning An often used technique in the context of many-cores is auto-tuning. Auto-tuning is the process of finding an optimal kernel by defining a set of parameters and generating a multitude of kernels based on those parameters. MCL could help in two ways: By providing feedback, programmers may be able to limit the number of parameters that have to be checked, thus limiting the search space for the kernels. Another direction is to enhance MCL with language constructs to automatically generate parameterized kernels. A concrete example could be an assignment with optional values `nrThreads={64,128}; nrBlocks={512,256};` which compiles to four kernels with Cartesian product of the provided options.

Performance models Since the hardware descriptions form a model of the hardware, a logical direction is to extend MCL with existing performance models in the hardware descriptions. Another option is to define an extra language tightly integrated with the hardware description language with which custom performance models can be defined to provide custom feedback to the programmer.

Kernel fusion A limitation of MCL is that the unit on which feedback is provided and on which optimizations are applied is a kernel function. However, on a high level of abstraction, one would want to write small kernels that are easy to compose and reuse, whereas on a lower levels one would want to compose those kernels into a larger one to apply optimizations over the boundaries of kernels. Currently, this is not well supported in MCL and it would be an interesting and challenging direction to investigate how composition of lower-level kernels or kernel fusion techniques can be applied in the context of MCL.

Polyhedral transformations Polyhedral techniques are usually applied to expose parallelism or transform programs for improved cache behavior. Since polyhedral analysis techniques can find precise dependencies over loop iterations, they may be a good starting point to provide more detailed feedback to programmers than is currently possible.

Visualization of data access Often, it is difficult to understand memory-access patterns of kernels in relation to how the memory hierarchy is organized on a many-core device. Especially for many-core devices, memory access patterns can have a large impact on the performance of the kernel. Additionally, understanding the memory access patterns of an algorithm can give programmers insight in how to reorganize the kernel for better performance. Because MCL contains hardware descriptions and knows the mapping from data to the hardware, it is particularly suited to provide programmers insight of the access patterns in the form of a visualization. Ideally, programmers could follow the memory access pattern per thread to discover communication patterns and data reuse.

FPGAs The current approach of MCL is to map an algorithm to hardware. However, FPGAs (Field-Programmable Gate Arrays) allow one to define the hardware circuits for a specific algorithm. A direction for future work would be to automatically generate a hardware specification for FPGAs from an MCL kernel. The restrictions on MCL kernels may provide an opportunity to translate them to CλaSH [95], a domain-specific language for specifying hardware designs that are subsequently mapped to FPGAs.

Fine-grained synchronization Section 2.6 explained a generalization on the divide-and-conquer model for Satin. In Sec. 4.5 we have seen that matrix multiplication scales worse than the other applications in Cashmere. One of the causes is the fact that divide-and-conquer matrix-multiplication synchronizes more than strictly necessary as explained in Sec. 2.6. A direction for future work is to investigate whether more fine-grained synchronization can improve Cashmere's scalability for applications such as matrix multiplication.

Wide-area distributed systems Finally, Cashmere is currently targeted at cluster computers, whereas original Satin also targets wide-area distributed systems. An interesting direction is to investigate whether Cashmere can obtain good results when deployed in a wide-area context where inter-cluster latencies are typically much higher than intra-cluster.

5.2.1 Conclusions

Although there are many directions for further improvements, we conclude that MCL and Cashmere form a promising solution to the main research question of this thesis:

- How can we support programmers in their responsibility to achieve high performance from many-core hardware?

MCL addresses the tension between control over hardware to reach high performance and providing high-level abstractions. It help programmers to achieve high performance by supporting the *stepwise-refinement for performance* methodology. Not only gives the methodology programmers control over the hardware, but it also gives them insight in the compiler and the performance of their applications in relation to the used hardware.

Cashmere helps programmers to achieve high performance on heterogeneous many-core clusters. MCL, as a part of Cashmere, helps in writing many different optimized kernels. The Cashmere system provides automatic load-balancing, hides communication to achieve high performance, and provides detailed feedback in the form of Gantt-charts to monitor the performance. Finally, we show that heterogeneous executions do not differ in efficiency from homogeneous runs.

All in all, With the MCL and Cashmere systems, we hope to provide solutions for the “Programming Wall” at hand and we hope to inspire other researchers to continue working on this very interesting problem.

Appendix A

Many-Core Levels Language Descriptions

This is an extension of Chapter 3 and discusses more details of the syntax and semantics of the two languages of Many-Core Levels (MCL). Section A.1 discusses the hardware description language and is an extension of Sec. 3.4.2. Section A.2 extends Sec. 3.4.3 and provides the syntax of the programming language.

In the syntax descriptions we use EBNF in a slightly modified form for readability. A capitalized identifier is a non-terminal, = defines a rule of the syntax, | an option and concatenation is implied using whitespace. A string enclosed in `""` is a literal in the syntax and we will use three suffixes for terms: ? means zero or one ([ ] in EBNF), * means zero or more ({ } in EBNF), and + means one or more. We will use ( ) for grouping, and { } for interspersing terms: for example, to allow a term A interspersed with B zero or more times, allowing A, A B A or A B A B A, etc., we can use {A B}*, which is equivalent to $\lambda | A(BA)*$ where λ is the empty string.

A.1 Hardware Description Language HDL

A hardware description is defined in a .hdl file, separate from .mcl files which contain programming modules. The top-level structure of HDL is very simple:

```
HWDescription   =  "hardware_description" Identifier
                   Specializes?
                   Block*
Specializes     =  "specializes" Identifier ";"
```

The start symbol of HDL is HWDescription. The `hardware_description` keyword indicates that this file is a hardware description. In HDL each keyword is reserved. The Identifier indicates which hardware description it represents. Following this declaration, a hardware description has an optional Specializes clause which specifies on which hardware description the current hardware description specializes. Finally, a hardware description has zero or more Blocks.

The Block is the main construct in HDL. Its semantics are determined by the specific BlockKeyword with which a Block is declared. The syntax is listed below:

```
Block          =  BlockKeyword Identifier "{" Statement* "}"
BlockKeyword   =  "parallelism"  |  "memory_space"
               |  "par_unit"  |  "par_group"  |  "device"
               |  "memory"  |  "interconnect"
               |  "device_group"  |  "device_unit"
               |  "execution_group"  |  "execution_unit"
               |  "instructions"  |  "cache"
               |  "simd_group"  |  "simd_unit"
               |  "load_store_group"  |  "load_store_unit"
```

A Statement can have several forms, listed below:

```
Statement           =  Block
                    |  PropertyStatement
PropertyStatement   =  PropertyKeyword "=" Expression PrefixUnit?
                       ";"
                    |  Expression ";"
                    |  ArgPropKeyword "(" {Expression ","}+ ")"
                       ";"
                    |  StatementKeyword ";"
```

The first production rule shows that Blocks can be either a nested Block or a PropertyStatement. A PropertyStatement has the following forms: some property that can be assigned expressions with optionally a PrefixUnit, expressions can be used as statements, there are properties that take expressions as arguments, and finally, there are statements that are just keywords. The keywords that can be used in statements are listed below:

```
PropertyKeyword    =  "nr_units"   |  "max_nr_units"
                   |  "capacity"   |  "latency"
                   |  "bandwidth"   |  "nr_banks"
                   |  "clock_frequency"   |  "addressable"
                   |  "cache_line_size"   |  "width"
ArgPropKeyword     =  "slots"   |  "connects"   |  "space"
                   |  "op"   |  "performance_feedback"
StatementKeyword   =  "default"   |  "read_only"
```

A property can be assigned an expression having an optional PrefixUnit. This allows us to define units for some expressions, for example GB/s:

```
PrefixUnit   =  Prefix? Unit
Prefix       =  "G" | "M" | "k"
Unit         =  BasicUnit "/" BasicUnit
             |  BasicUnit
BasicUnit    =  "B" | "bit" | "bits" | "cycle" | "cycles" | "s" | "Hz"
```

The grammar for an expression is:

```
Expression   =  IntExp
             |  QualIdentifier "[*]"?
             |  QualIdentifier "(" {Expression ","}+ ")"
             |  Operation
             |  ExpressionKeyword
```

An Expression can be an integer expression IntExp (not further shown in this grammar) with the usual operations +, -, etc. defined on them. An Expression can also be a qualified Identifier with optionally a "[*]" suffix, or with Expressions as arguments (line 3). A QualIdentifier uses dot notation to indicate identify properties that are nested:

```
QualIdentifier   =  Identifier
                 |  QualIdentifier "." Identifier
```

Finally, an expression can be an operation or an expression keyword:

```
Operation          =  "(+)"  |  "(-)"  |  "(/)"  |  "(%)"
                   |  "(*)"  |  StringExpression
ExpressionKeyword  =  "unlimited"  |  "true"  |  "false"
```

The syntax of an operation allows us to define some properties about the operations +, -, etc. in the programming language. An operation can also be a string (not further explained in this grammar), for example to indicate something about special hardware functionality such as special units that support transcendental functions.

Hardware descriptions define their parents using the `specializes` declaration which results in a hierarchy (examples are shown in Fig. 3.4 in Chapter 3 and Fig. 4.2 in Chapter 4). The edges of the hierarchy do not mean that a hardware description on a lower level inherits features of its parent. It means that a program written for hardware description x can be translated to a program adhering to the rules of hardware description y where y is a child of x. For example, it is possible to translate a program written for hardware description *perfect* to a program for hardware description *fermi* in Fig. 3.4 in Chapter 3. It is possible to reuse parts of other hardware descriptions by using direct references to the blocks in other hardware descriptions.

A valid hardware description with the name *name* has at least two blocks, a `parallelism` block and a `device` block. The `parallelism` block defines a hierarchy of programming abstractions to which programmers map their algorithm, and the `device` block describes the physical device and should have *name* as Identifier.

The nesting of Blocks is governed by rules which are summarized in Table A.1. The blocks in the second column are allowed to be nested inside the blocks in the first column. The third column specifies whether such a block is required and the fourth column specifies whether more than one of these blocks are allowed.

The first section "Common" shows the rule for grouping blocks. A block ending with `_group` specifies that a certain number of `_unit` blocks are grouped together. The `_unit` blocks are nested in the `_group` blocks and must have the same prefix. The second and third section show the rules for the blocks for programming abstractions and the physical device respectively.

Inside the blocks, several properties can be specified. Table A.2 shows which properties can be used where and what kinds of expressions and units are al-

Table A.1: Allowed nestings of Blocks

Block	Block	required	multiple
	Common		
*x*_group	*x*_unit	yes	no
	Programming abstractions		
parallelism	par_group	yes	no
	memory_space	yes	yes
par_unit	memory_space	no	yes
	par_group	no	no
	Physical device		
device	memory	no	yes
	cache	no	yes
	interconnect	no	yes
	execution_group	no	yes
	device_group	no	yes
execution_unit	memory	no	yes
	cache	no	yes
	execution_group	no	yes
	load_store_group	no	yes
	simd_group	no	yes
	instructions	no	no
device_unit	memory	no	yes
	cache	no	yes
	interconnect	no	yes
	execution_group	no	yes
	device_group	no	yes
load_store_unit	instructions	yes	no
simd_unit	instructions	yes	no

lowed. The second column specifies the property that can be specified in the block in the first column. The third and fourth column indicate which expressions and units are allowed. The last column indicates whether the property is required inside the block.

A `_group` block always needs to have a `max_nr_units` or `nr_units` with an integer expression or the keyword `unlimited`, which means that it may be a very large integer number. The sections in this table are the same as in Table A.1. In section "Physical Device" the expression (inherited) means that the rules for the properties are inherited from the device parent. The relation between device blocks has been specified in Fig. 3.5 in Chapter 3.

A.2 Programming Language

Since many syntactic forms are similar to C, we will only discuss what is different from C. Please note that below, we define a different grammar than the grammar in Sec. A.1 that is used in different files. We can therefore safely reuse the names for non-terminals, such as Statement and Block denoting different syntactic forms.

An MCPL module consists of a `"module"` keyword with an Identifier declaring which module it is, a list of Imports and a list of Functions. An import declares which hardware descriptions should be imported into the module.

```
Module    =   "module" Identifier
              Import*
              Function*
Import    =   "import" Identifier ";"
```

A Function starts with an Identifier that indicates which hardware description this function targets. This means that we can specify per function which hardware we target. It then specifies the return type, an Identifier that indicates the name of the function, and then a list of Declarations for parameter list and a Block:

```
Function   =   Identifier Type Identifier "(" {Declaration ","}* ")"
               Block
```

A Type is a primitive type or a Type with an ArrayExpression. This results in a multi-dimensional tiled array type. The syntax is listed below.

Table A.2: Allowed properties of Blocks

Block	property	expression	unit	required
	Common			
*x*_group	(max_)nr_units	IntExp or unlimited		yes
	Programming abstractions			
memory_space	default			no
	read_only			no
	Physical device			
memory	capacity	IntExp or unlimited	B, bit, bits	no
	space	Identifier		yes
	nr_banks	IntExp		no
	addressable	false		no
cache	(inherited)			
	cache_line_size	IntExp	B, bit, bits	yes
interconnect	connects	Identifier		yes
	latency	IntExp	cycle(s)	yes
	bandwidth	IntExp	{B, bit, bits}/s	no
	width	IntExp	bit, bits	no
	clock_frequency	IntExp	Hz	no
execution_group	slots	Identifier, IntExp		yes
execution_unit	slots	Identifier, IntExp		yes
	performance_feedback	String		no
simd_unit	(inherited)			
load_store_unit	(inherited)			

```
Type             =  "int"  |  "float"  |  "bool"  |  "void"
                 |  Type ArrayExpression
ArrayExpression  =  "[" {Expression ","}+ "]"
```

Having tiled multi-dimensional arrays means that we can have the following forms: The type `int[2][3]` has 2 tiles of 3 elements. The type `int[2,3]` is a two-dimensional array with 2 rows and 3 columns. The type `int[2,2][3,3]` is a two-dimensional array with 2×2 tiles of 3×3 elements each.

A Declaration has two forms. It can have BasicDeclarations interspersed with an `as` keyword or it can be an assignment Declaration.

```
Declaration       =  Modifier* {BasicDeclaration "as"}+
                  |  Modifier* BasicDeclaration "=" Expression
BasicDeclaration  =  Type Identifier
Modifier          =  "const"
                  |  Identifier
```

With the `as` keyword one can declare variables with more than one form, for example: `int[2,3]aasint[6]b`. A Modifier indicates whether the variable is constant or it can be an Identifier that has to refer to a memory space in the hardware description of the function.

A Block contains a list of Statements:

```
Block      =  "{" Statement* "}"
Statement  =  Block  |  Declaration ";"  |  Assignment ";"
           |  Increment ";"  |  Call ";"  |  Return ";"  |  If
           |  For  |  As ";"  |  Barrier ";"  |  ForEach
```

A Statement can be a Block and the Declaration that was defined above. The subsequent syntactic forms are not discussed, since they are very similar to any C-like language, except for the last three: As, Barrier, and ForEach.

An As statement is equivalent to a Declaration with keyword `as`. It declares new forms of a declaration, only not at the declaration itself but at a later point. The syntactic form is:

```
As             =  Variable "as" {BasicDeclaration "as"}+
Variable       =  BasicVariable ("." Variable)?
BasicVariable  =  Identifier ArrayExpression*
```

A Variable can be interspersed with dots and a BasicVariable has zero or more array expressions.

A Barrier statement creates a synchronization point for units of parallelism for a specific memory space. The syntax is:

Barrier = `"barrier"` `"("` Identifier `")"`

The Identifier has to refer to a memory space in the hardware description.

In MCPL, the ForEach statement expresses parallelism with an Identifier that indicates the specific `parallelism_group` that is targeted. The syntax is:

ForEach = `"foreach"` `"("` BasicDeclaration `"in"` Expression Identifier `")"` Statement

Considering the ForEach as a loop, then the BasicDeclaration declares a variable that will hold a unique value from 0 to Expression for each loop iteration. The Identifier has to refer to a `par_group` in the hardware description.

Appendix B

Translating an MCPL program to a lower level of abstraction

This appendix describes how an MCPL module, represented by a list of functions is translated to hardware description *target*, given a hierarchy of hardware descriptions. We have added type annotations to the pseudo code below to make the code more readable. The type list[T] is a list of type T, a list expression is $[1, 2, 3]$. A tuple[T, S] is a tuple with types T and S, a tuple expression is $< 1, 2 >$. The first and second field of a tuple can be reached with $t.first$ and $t.second$ where t is a tuple.

Throughout the following algorithms the type ParGroupMapping is an important type which is a list of tuples, where each tuple contains two fields with the first field being a ParGroup and the second field a list of ParGroups: list[tuple[ParGroup, list[ParGroup]]]. The type ParGroupMapping will be used as an alias for this type.

A ParGroup contains a ParUnit, and a ParUnit may contain a ParGroup. We consider the outer scope of a `parallelism` hierarchy a ParGroup as well, the top ParGroup *parallelism*. All ParGroups and ParUnits can contain MemorySpaces.

B.1 The top-level functions

The top-level function takes as input a list of functions and a string representing the hardware description to which we want to translate. Since a hardware description always knows its parent in the hierarchy, the function construct a path from hardware description $perfect$ to $target$.

```
Require: each hardware description in $hwds$ (except for "perfect") has
            defined its parent in the hierarchy
Ensure: each function that has a path to $target$ is translated to $target$
 1: function TRANSLATEFUNCTIONS(list[Function] $fs$, str $target$,
            list[HWDesc] $hwds$)
 2:     list[HWDesc] $path \leftarrow$ FINDPATH("perfect", $target$, $hwds$)
 3:     for all Function $f$ in $fs$ do
 4:         TRANSLATEFUNCTION($f$, $target$, $path$)
 5:     end for
 6: end function
```

The following function translates one function iteratively to $target$:

```
Ensure: if function is on the $path$ from "perfect" to $target$, then it is
            translated to $target$
 1: function TRANSLATEFUNCTION(Function $f$, str $target$, list[HWDesc]
            $path$)
 2:     for int $i \leftarrow 0$; $i < path.size - 1$; $i \leftarrow i + 1$ do
 3:         if $f.hwdesc = path[i]$ then
 4:             TRANSLATE($f$, $path[i + 1]$)
 5:         end if
 6:     end for
 7: end function
```

The following function translates a function, but only one step in the hierarchy. An executing ParUnit is the innermost ParUnit in a parallelism hierarchy, the unit that executes the instructions. In MCPL other ParUnits are not allowed to execute code. An executing ParGroup is a ParGroup that contains the executing ParUnit.

```
Require: $hwdescTo$ has as direct parent $f.hwdesc$
 1: function TRANSLATE(Function $f$, HWDesc $hwdescTo$)
 2:     HWDesc $hwdescFrom \leftarrow f.hwdesc$
 3:     ParUnit $executingParUnitFrom \leftarrow$
```

```
            getExecutingParUnit($hwdescFrom$)
 4:     ParUnit $executingParUnitTo \leftarrow$ getExecutingParUnit($hwdescTo$)
 5:     ParGroup $parGroupTo \leftarrow$ getParGroup($executingParUnitTo$)
 6:     ParGroup $parGroupFrom \leftarrow$
            getParGroup($executingParUnitFrom$)

 7:     ParGroupMapping $m \leftarrow [< parGroupFrom, [parGroupTo] >]$
 8:     $m \leftarrow$findEquivalentParGroups($m$)
          ▷ Sec. B.2

 9:     translateDeclarations($f$, $m$)
10:     translateBarriers($f$, $m$)
          ▷ Sec. B.3

11:     translateForEachStats($f$, $m$)
          ▷ Sec. B.4
12: end function
```

B.2 Finding equivalent ParGroups

The following function finds equivalence between ParGroups. For each ParGroup the compiler needs to know how to translate it to potential more ParGroups. Therefore the function needs a mapping from a ParGroup to lists of ParGroups. As this may occur at different levels in the parallelism hierarchy, the compiler maintains a list of those mappings, for each level an element in the list. A mapping is represented by a tuple with the first element denoting the "from" ParGroup and the second element the list of ParGroups "to". Below we explain the algorithm in more detail.

```
Require: $m$ contains at least a mapping from executing ParGroups
 1: function findEquivalentParGroups(ParGroupMapping $m$)
 2:     ParGroup $from \leftarrow m[0].first$
 3:     list[ParGroup] $to \leftarrow m[0].second$
 4:     if $from = parallelism \wedge to[0] = parallelism$ then
 5:         return $m$
 6:     else if $from = parallelism \vee to[0] = parallelism$ then
 7:         error("not equivalent")
 8:     end if
 9:     ParGroup $parentFrom \leftarrow$ getParentParGroup($from$)
10:     ParGroup $parentTo \leftarrow$ getParentParGroup($to[0]$)
11:     if nrUnits($from$) = nrUnits($to$) then
12:         return findEquivalentParGroups([<
              $parentFrom, [parentTo] >] + m$)
13:     else if nrUnits($from$) < nrUnits($to$) then
14:         error("from less than to")
15:     else if nrUnits($from$) > nrUnits($to$) then
16:         if $parentTo = parallelism$ then
17:             $m \leftarrow [< parentFrom, [parentTo] >] + m$
18:         else
19:             $m[0].second \leftarrow [parentTo] + to$
20:         end if
21:         return findEquivalentParGroups($m$)
22:     end if
23: end function
```

Line 4 is the stop condition that states that the function stop if it has reached the top of the parallelism hierarchy. Both $from$ and to must have reached the $parallelism$ to be a valid hardware description. In the conditions in lines 11-22 the flow of control is based on the number of units in $from$ and to. As to is a list, the number of units is the product of the ParGroup sizes contained in the list. Everything is less than *unlimited* except *unlimited* itself which is equal to *unlimited*.

If the number of units in $from$ and to are equal, the function prepends the parents to the mapping and continues recursively. In HDL it is not allowed to have less units of parallelism in $from$ than in to (line 13 and 14). If the number of parallelism units in $from$ is greater than in to, the compiler has to add the parent of $to[0]$ to the mapping. However, if the parent is $parallelism$, the function can assume that the parent of $from$ is also $parallelism$. If this is not true, then the stop-condition will detect the error and the hardware description is not valid.

B.3 Translating memory spaces

Both declarations and barrier statements contain memory-space expressions. For simplicity we do not handle declarations that use the `default` memory-space. Primitive constant declarations do not have memory spaces and are not considered. The functions translateDeclarations and translateBarri-

ERS are now shown. They visit the function and call TRANSLATEDECL and TRANSLATEBARRIER for each declaration and barrier in the function respectively.

```
function TRANSLATEDECL(Decl $d$, ParGroupMapping $m$)
    if $d$ is not primitive and constant then
        MemorySpace $from \leftarrow$ GETMEMORYSPACE($d$)
        MemorySpace $to \leftarrow$ FINDEQUIVALENTMEMORYSPACE($from$, $m$)
        $d \leftarrow$ SETMEMORYSPACE($d$, $to$)
    end if
end function
```

```
function TRANSLATEBARRIER(Barrier $b$, ParGroupMapping $m$)
    MemorySpace $from \leftarrow$ GETMEMORYSPACE($b$)
    MemorySpace $to \leftarrow$ FINDEQUIVALENTMEMORYSPACE($from$, $m$)
    $b \leftarrow$ SETMEMORYSPACE($b$, $to$)
end function
```

The function below requires that memory spaces do not disappear in moving from a higher level of abstraction to a lower level of abstraction. Therefore, this function defines the limitation on designing parallelism hierarchies. It is also required that memory spaces that are defined in ParUnits have a representative memory-space in a lowest-level ParUnit, and memory spaces in ParGroups in the highest-level ParGroup. We explain the algorithm below.

```
Require: In the mapping there should be a representative memory space for
         $ms$.
function FINDEQUIVALENTMEMORYSPACE(MemorySpace $ms$,
         ParGroupMapping $m$)
    $< level, inUnit > \leftarrow$ FINDMEMORYSPACE($ms$, $m$)
      ▷ $ms$ is in $m[level].first$
      ▷ $inUnit$ denotes whether $ms$ was found in a ParUnit or not
    list[ParGroup] $pgsTo \leftarrow m[level].second$
    if $inUnit$ then
        ParGroup $pg \leftarrow$ LAST($pgsTo$)
        ParUnit $pu \leftarrow$ GETPARUNIT($pg$)
        return FINDEQUIVALENTMEMORYSPACE($ms$, $pu.memorySpaces$)
    else
        ParGroup $pg \leftarrow pgsTo[0]$
        return FINDEQUIVALENTMEMORYSPACE($ms$, $pg.memorySpaces$)
```

```
11:     end if
12:     ERROR("no matching memory-space")
13: end function
```

On line 2, the *level* in the mapping for the memory space is found. It is also detected whether the memory-space is in a ParUnit or a ParGroup. The function then looks up the equivalent memory-space in the mapping, taking into account whether *ms* was found in a ParUnit or ParGroup. In the first case, the function retrieves the memory from the last, the lowest-level ParGroup. If *ms* was found in the ParGroup, then the function retrieves it from the first ParGroup.

The following function finds the right memory-space with a preference to match whether a memory space is read-only or not. If there is no match, then it returns a memory-space that is not read-only. It is an error if *ms* is not read-only and there are only read-only memory-spaces in *mss*.

```
function FINDEQUIVALENTMEMORYSPACE(MemorySpace $ms$,
        list[MemorySpace] $mss$)
    for all MemorySpace $msTo$ in $mss$ do
        if $(ms.readonly \wedge msTo.readonly) \vee (\neg ms.readonly \wedge$
              $\neg msTo.readonly)$ then
            return $msTo$
        end if
    end for
    for all MemorySpace $msTo$ in $mss$ do
        if $ms.readonly$ then
            return $msTo$
        end if
    end for
    ERROR("no equivalent memory space")
end function
```

B.4 Translating ForEach statements

This section discusses how ForEach statements are translated into ForEach statements of a lower-level abstraction. The following function translates a list of statements in a list of statements where each ForEach has been translated to the lower-level abstraction. Other supporting statements will also be generated.

Ensure: Each ForEach in *stats* has been translated, including the nested ForEach statements.

```
1: function TRANSLATEFOREACHSTATS(list[Stat] $stats$, ParGroupMapping $m$)
2:     list[Stat] $newStats \leftarrow []$
3:     for all Stat $s$ in $stats$ do
4:         if $s$ is ForEach then
5:             $newStats \leftarrow newStats +$ TRANSLATEFOREACHSTAT($s$, $m$)
6:         else
7:             $newStats \leftarrow newStats + [s]$
8:         end if
9:     end for
10:    return $newStats$
11: end function
```

The following function is called for ForEach statements. If the ForEach is not split into multiple ForEach statements, then it is a simple translation.

Require: s is a statement with a ForEach
Ensure: s and all its inner ForEach statements are translated into a list of Stats with ForEach, dimension, and indexing Stats.

```
1: function TRANSLATEFOREACHSTAT(Stat $s$, ParGroupMapping $m$)
2:     int $currentLevel \leftarrow$ GETLEVEL($s.foreach.parGroup$, $m$)
       ▷ $m[currentLevel].first = s.foreach.parGroup$
3:     if $m[currentLevel].second.size = 1$ then
4:         return TRANSLATEFOREACHSIMPLE($s$, $m$)
5:     else
6:         return TRANSLATEFOREACHADVANCED($s$, $m$)
7:     end if
8: end function
```

A simple translation of ForEach statements just modifies the ParGroup of the ForEach statement. It continues by calling TRANSLATEFOREACHSTATS on each inner ForEach statement.

Require: The list of ParGroups $m[currentLevel].second$ has only one element.
Ensure: A list of statements is returned.

```
1: function TRANSLATEFOREACHSIMPLE(Stat $s$, ParGroupMapping $m$)
2:     int $currentLevel \leftarrow$ GETLEVEL($s.foreach.parGroup$, $m$)
3:     $s.foreach.parGroup \leftarrow m[currentLevel].second[0]$
4:     $s.foreach.stats \leftarrow$ TRANSLATEFOREACHSTATS($s.foreach.stats$, $m$)
```

```
5:     return [s]
6: end function
```

The above function automatically handles multiple dimensions within the same ParGroup. This is not the case for the function below. First, the function has to find out how many dimensions the ForEach has. It then collect the *innerStats* for the last dimension with which it will continue.

The function generates three kinds of statements. The *dimensionStats* compute the dimensions of each ForEach statement, the *foreachStats* contain the translated ForEach Statements, and the *indexingStats* contain statements that translate the index of the old ForEach into indices for the new ForEeach statements. The `for`-loop on line 9 collects all three statements for each dimension, starting with the innermost. Variable *pgsTo* will be updated to reflect which ParGroups still need to be handled.

On line 16, the inner `foreach`-statements are translated that do not have *s.foreach.parGroup*. The statement on line 17 adds the indexing statements to the inner statements. On lines 18-21 the nesting of statements is organized.

```
Ensure: A list of statements is returned.
 1: function TRANSLATEFOREACHADVANCED(Stat $s$, ParGroupMapping $m$)
 2:     int $nrDimension \leftarrow$ GETNRDIMENSIONS($s.foreach$,
            $s.foreach.parGroup$)
        ▷ # dimensions in the same ParGroup
 3:     list[Stat] $innerStats \leftarrow$ GETSTATSDIMENSION($nrDimension - 1$,
            $s.foreach$)
 4:     list[Stat] $dimensionStats \leftarrow []$
 5:     list[Stat] $foreachStats \leftarrow []$
 6:     list[Stat] $indexingStats \leftarrow []$
 7:     int $currentLevel \leftarrow$ GETLEVEL($s.foreach.parGroup$, $m$)
 8:     list[ParGroup] $pgsTo \leftarrow m[currentLevel].second$
 9:     for int $dim \leftarrow nrDimension - 1$; $dim \geq 0$; $dim \leftarrow dim - 1$ do
10:         Stat $feDim \leftarrow$ GETFOREACHDIMENSION($dim$)
11:         $< ds, fes, is, pgsTo > \leftarrow$ CREATEFOREACH($feDim$, $pgsTo$, $m$)
12:         $dimensionStats \leftarrow dimensionStats + ds$
13:         $foreachStats \leftarrow foreachStats + fes$
14:         $indexingStats \leftarrow indexingStats + is$
15:     end for

16:     $innerStats \leftarrow$ TRANSLATEFOREACHSTATS($innerStats$, $m$)
```

```
17:     $innerStats \leftarrow indexingStats + innerStats$

18:     for all Stat $s$ in $foreachStats$ do
19:         $s.foreach.stats \leftarrow innerStats$
20:         $innerStats \leftarrow [s]$
21:     end for
22:     return $dimensionStats + innerStats$
23: end function
```

The createForEach function creates all three kinds of statements for the ForEach statements that have to be created for the ParGroups in *pgsTo*. Line 5 shows the simple case. If there is only one ParGroup in *pgsTo*, the existing ForEach in *s* obtains the new ParGroup. Otherwise, the function keeps track of *dimensionVars* and *indexingVars* of which the former list contains the variables that will be used in the ForEach statements to indicate the size and the latter the variables that are used to create indices for the ForEach statements.

The loop on line 13 treats the ParGroups in reversed order and creates dimension, indexing, and foreach statements. The indexing statement that expresses the old indexing variable in terms of the new dimensions and indices can only be computed when all indexing variables and dimension variables are known (line 23).

```
 1: function createForEach(Stat $s$, list[ParGroup] $pgsTo$,
            ParGroupMapping $m$)
 2:     list[Stat] $dimensionStats \leftarrow []$
 3:     list[Stat] $foreachStats \leftarrow []$
 4:     list[Stat] $indexingStats \leftarrow []$

 5:     if $pgsTo.size = 1$ then
 6:         $s.foreach.parGroup \leftarrow pgsTo[0]$
 7:         $foreachStats \leftarrow [s]$
 8:         return $< dimensionStats, foreachStats, indexingStats, pgsTo >$
 9:     else
10:         list[Var] $dimensionVars \leftarrow []$
11:         list[Var] $indexingVars \leftarrow []$
12:         list[ParGroup] $pgsReversed \leftarrow$ reverse($pgsTo$)

13:         for all ParGroup $pg$ in $pgsReversed$ do
```

```
14:             Exp $dimensionExp \leftarrow$
                createDimensionExp($dimensionVars$, $pg$,
                    $s.foreach.nrIterations$)
15:             Var $dimensionVar \leftarrow$ createVar
16:             Var $indexingVar \leftarrow$ createVar

17:             Stat $dimensionStat \leftarrow$ createAssignStat( $dimensionVar$,
                $dimensionExp$)
18:             Stat $foreachStat \leftarrow$ createForEachStat( $indexingVar$,
                $dimensionVar$, $pg$)

19:             $dimensionVars \leftarrow dimensionVar + dimensionVars$
20:             $indexingVars \leftarrow dimensionVar + dimensionVars$
21:             $dimensionStats \leftarrow dimensionStats + dimensionStat$
22:             $foreachStats \leftarrow foreachStats + foreachStat$

23:             if $pg = pgsReversed.last$ then
24:                 Stat $indexingStat \leftarrow$
                createIndexingStat($s.foreach.indexingVar$,
                    $indexingVars$, $dimensionVars$)
25:                 $indexingStats \leftarrow indexingStats + indexingStat$
26:                 $pgsTo \leftarrow [pg]$
27:             end if
28:         end for
29:         return $< dimensionStats, foreachStats, indexingStats, pgsTo >$
30:     end if
31: end function
```

Function createDimensionExp creates an expression for the number of parallelism units of a ForEach loop. The function uses the value from the hardware description on line 3, if there are no dimension variables or if the number of units of the ParGroup is definite. This means that a ParGroup has as property `nr_units` (denoting the exact number of units there has to be) instead of `max_nr_units`.

```
1: function createDimensionExp(list[Var] $dimensionVars$, ParGroup $pg$,
            Exp $nrIterations$)
2:     if $dimensionVars.size = 0 \vee pg.nrUnitsDefinite$ then
3:         return createNrUnitsExp($pg$)
```

```
 4:     else
 5:         Exp $productVars \leftarrow$ MULVARS($dimensionVars$)
        ▷ $productVars$ is 1 if $dimensionVars.size = 0$
 6:         return DIV($nrIterations$, $productVars$)
 7:     end if
 8: end function
```

The function below creates a statement that defines how the old index is computed from the newly generated index and dimension variables. The first dimension variable is not part of the computation.

```
 1: function CREATEINDEXINGSTAT(Var $oldIndexingVar$, list[Var]
            $indexingVars$, list[Var] $dimensionVars$)
 2:     $dimensionVars \leftarrow$TAIL($dimensionVars$)
 3:     Exp $e \leftarrow$CREATEINDEXINGEXP($indexingVars$, $dimensionVars$)
 4:     return CREATEASSIGNSTAT($oldIndexingVar$, $e$)
 5: end function
```

This function recursively creates an indexing expression from the existing dimension and indexing variables. The *size* expression is computed by multiplying all dimension variables.

```
 1: function CREATEINDEXINGEXP(list[Var] $indexingVars$, list[Var]
            $dimensionVars$)
 2:     if $indexingVars.size = 1$ then
 3:         return EXP($indexingVars[0]$)
 4:     end if
 5:     Exp $size \leftarrow$MULVARS($dimensionVars$)
 6:     Exp $indexingVar \leftarrow indexingVars[0]$
 7:     $indexingVars \leftarrow$TAIL($indexingVars$)
 8:     $dimensionVars \leftarrow$TAIL($dimensionVars$)
 9:     Exp $lowerDim \leftarrow$CREATEINDEXINGEXP($indexingVars$,
            $dimensionVars$)
10:     return ADD(MUL($indexingVar$, $size$), $lowerDim$)
11: end function
```

References

[1] S. Huang, S. Xiao, and W. Feng. **On the Energy Efficiency of Graphics Processing Units for Scientific Computing**. In *IEEE International Parallel and Distributed Processing Symposium, 2009. IPDPS 2009.*, pages 1–8, May 2009.

[2] Jeremy Enos, Craig Steffen, Joshi Fullop, Michael Showerman, Guochun Shi, Kenneth Esler, Volodymyr Kindratenko, John E. Stone, and James C. Phillips. **Quantifying the Impact of GPUs on Performance and Energy Efficiency in HPC Clusters**. *International Conference on Green Computing*, **0**:317–324, 2010.

[3] Wu-chun Feng and Kirk Cameron. **The Green500 List: Encouraging Sustainable Supercomputing**. *Computer*, **40**(12):50–55, December 2007.

[4] Ian Sommerville. *Software Engineering*. Addison Wesley Longman Publishing Co. Inc., Redwood City, CA, USA, 9th edition, 2010. ISBN: 978-0137035151.

[5] B. Randell. **Software Engineering in 1968**. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 1–10, Piscataway, NJ, USA, 1979. IEEE Press.

[6] E.A. Lee. **The Problem with Threads**. *Computer*, **39**(5):33 – 42, May 2006.

[7] Rob V. van Nieuwpoort, Gosia Wrzesińska, Ceriel J. H. Jacobs, and Henri E. Bal. **Satin: A High-Level and Efficient Grid Programming Model**. *ACM Trans. Program. Lang. Syst.*, **32**(3):1–39, 2010.

[8] Henri E. Bal, Jason Maassen, Rob V. van Nieuwpoort, Niels Drost, Roelof Kemp, Nick Palmer, Gosia Wrzesińska, Thilo Kielmann, Frank Seinstra, and Ceriel J. H. Jacobs. **Real-World Distributed Computing with Ibis**. *Computer*, **43**:54–62, August 2010.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. **Cilk: An Efficient Multithreaded Runtime System**. *SIGPLAN Not.*, **30**(8):207–216, 1995.

[10] *Byte Code Engineering Library*, 2006. `http://jakarta.apache.org/bcel`.

[11] Michael Hind. **Pointer Analysis: Haven't We Solved This Problem Yet?** In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[12] Kees van Reeuwijk, Rob V. van Niewpoort, and Henri E. Bal. **Developing Java Grid Applications with Ibis**. In *Proc. of the 11th International Euro-Par Conference*, pages 411–420, Lisbon, Portugal, September 2005.

[13] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. **Best-First Fixed-Depth Minimax Algorithms**. *Artificial Intelligence*, **87**(1–2):255–293, November 1996.

[14] Rob V. van Nieuwpoort, Jason Maassen, Andrei Agapi, Ana Oprescu, and Thilo Kielmann. **Experiences Deploying Parallel Applications on a Large-scale Grid**. In *EXPGRID - Experimental Grid testbeds for the assessment of large-scale distributed applications and tools, workshop in conjunction with HPDC-15*, june 2006.

[15] J. Rose. **LocusRoute: A Parallel Global Router for Standard Cells**. *Design Automation Conference*, pages 189–195, 1988.

[16] Gosia Wrzesińska, Jason Maassen, Kees Verstoep, and Henri E. Bal. **Satin++: Divide-and-Share on the Grid**. In *2nd IEEE International Conference on e-Science and Grid Computing*, Amsterdam, The Netherlands, 2007.

[17] Henry C. Baker, Jr. and Carl Hewitt. **The Incremental Garbage Collection of Processes**. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[18] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. **Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications**. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 34–43, New York, NY, USA, 2001. ACM.

[19] Frej Drejhammar, Christian Schulte, Per Brand, and Seif Haridi. **Flow Java: Declarative Concurrency for Java**. In Catuscia Palamidessi, editor, *Logic Programming*, **2916** of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin / Heidelberg, 2003.

[20] Ondřej Lhoták and Laurie Hendren. **Scaling Java Points-to Analysis Using Spark**. In G. Hedin, editor, *Compiler Construction*, **2622** of *Lecture Notes in Computer Science*, pages 153–169. Springer Berlin / Heidelberg, 2003.

[21] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. **Type-based Alias Analysis**. *SIGPLAN Not.*, **33**(5):106–117, May 1998.

[22] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. **Interprocedural Pointer Alias Analysis**. *ACM Trans. Program. Lang. Syst.*, **21**(4):848–894, July 1999.

[23] John Whaley and Martin Rinard. **Compositional Pointer and Escape Analysis for Java Programs**. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. ACM.

[24] Bruno Blanchet. **Escape Analysis for Object-Oriented Languages: Application to Java**. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 20–34, New York, NY, USA, 1999. ACM.

[25] P. Hijma. Available from: `https://github.com/pieterhijma/mcl`.

[26] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. **Brook for GPUs: Stream Computing on Graphics Hardware**. *ACM Trans. Graph.*, **23**(3):777–786, August 2004.

[27] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. **Compiling a High-Level Language for GPUs**. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM.

[28] Leslie G. Valiant. **A Bridging Model for Parallel Computation**. *Commun. ACM*, **33**(8):103–111, August 1990.

[29] Qiming Hou, Kun Zhou, and Baining Guo. **BSGP: Bulk-Synchronous GPU Programming**. *ACM Trans. Graph.*, **27**:19:1–19:12, August 2008.

[30] Guy E. Blelloch. **Programming Parallel Algorithms**. *Commun. ACM*, **39**(3):85–97, 1996.

[31] Lars Bergstrom and John Reppy. **Nested Data-Parallelism on the GPU**. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 247–258, New York, NY, USA, 2012. ACM.

[32] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. **Copperhead: Compiling an Embedded Data Parallel Language**. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56, 2011.

[33] Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. **Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language**. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, 2011.

[34] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Newnes, 2013. ISBN: 9780124104143.

[35] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. **GPU Programming in a High Level Language: Compiling X10 to CUDA**. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 8:1–8:10, New York, NY, USA, 2011. ACM.

[36] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. **Breaking the GPU Programming Barrier with the Auto-Parallelising SAC Compiler**. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 15–24, New York, NY, USA, 2011. ACM.

[37] Bradford Larsen. **Simple Optimizations for an Applicative Array Language for Graphics Processors**. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 25–34, New York, NY, USA, 2011. ACM.

[38] Sean Lee, Vinod Grover, Gabriele Keller, and Manuel M.T. Chakravarty. **GPU Kernels as Data-Parallel Array Computations in Haskell**. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM 2009)*, 2009.

[39] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. **Accelerating Haskell Array Codes with Multicore GPUs**. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, 2011.

[40] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. **Expressive Array Constructs in an Embedded GPU Kernel Programming Language**. In *Proceedings of the 7th workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, pages 21–30, New York, NY, USA, 2012. ACM.

[41] Geoffrey Mainland and Greg Morrisett. **Nikola: Embedding Compiled GPU Functions in Haskell**. *SIGPLAN Not.*, **45**(11):67–78, September 2010.

[42] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. **A GPGPU Compiler for Memory Optimization and Parallelism Management**. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, 2010.

[43] Chenxi Wang, Kang Kang, Maohua Zhu, and Yangdong Deng. **A Polyhedral Modeling Based Source-to-Source Code Optimization Framework for GPGPU**. In *2012 IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops & PhD Forum*, pages 1964–1970, 2012.

[44] D. Buono, M. Danelutto, S. Lametti, and M. Torquati. **Parallel Patterns for General Purpose Many-Core**. In *21st Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 131–139, Feb 2013.

[45] Mudassar Majeed, Usman Dastgeer, and Christoph Kessler. **Cluster-SkePU: A Multi-Backend Skeleton Programming Library for GPU Clusters**. In *Proc. of the Int. Conf. on Par. and Dist. Proc. Techn. and Appl. (PDPTA)*, Las Vegas, USA, July 2013.

[46] Cedric Nugteren and Henk Corporaal. **Introducing 'Bones': A Parallelizing Source-to-source Compiler Based on Algorithmic Skeletons**. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[47] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. **A Fusion-Embedded Skeleton Library**. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, **3149** of *Lecture Notes in Computer Science*, pages 644–653. Springer Berlin Heidelberg, 2004.

[48] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. **Towards Parallel Programming by Transformation: The FAN Skeleton Framework**. *Parallel Algorithms and Applications*, **16**(2):87–121, 2001.

[49] Shigeyuki Sato and Hideya Iwasaki. **A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming**. In Zhenjiang Hu, editor, *Programming Languages and Systems*, **5904** of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg, 2009.

[50] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. **A Domain-Specific**

Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, 2011.

[51] Luke Cartey, Rune Lyngsø, and Oege de Moor. **Synthesising Graphics Card Programs from DSLs**. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 121–132, 2012.

[52] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. **OpenACC - First Experiences with Real-World Applications**. In Christos Kaklamanis, Theodore Papatheodorou, and Paul Spirakis, editors, *Euro-Par 2012 Parallel Processing*, **7484** of *Lecture Notes in Computer Science*, pages 859–870. Springer Berlin / Heidelberg, 2012.

[53] Seyong Lee and Jeffrey S. Vetter. **OpenARC: Open Accelerator Research Compiler for Directive-based, Efficient Heterogeneous Computing**. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 115–120, New York, NY, USA, 2014. ACM.

[54] Tianyi David Han and Tarek S. Abdelrahman. **hiCUDA: High-Level GPGPU Programming**. *IEEE Transactions on Parallel and Distributed Systems*, **22**(1):78–90, January 2011.

[55] Seyong Lee and Rudolf Eigenmann. **OpenMPC: Extended OpenMP Programming and Tuning for GPUs**. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[56] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. **OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization**. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[57] Seyong Lee and Jeffrey S. Vetter. **Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing**. In *Proceedings of the International Conference on*

High Performance Computing, Networking, Storage and Analysis, SC ’12, pages 23:1–23:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[58] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. **Scalable Parallel Programming with CUDA**. *Queue*, **6**(2):40–53, 2008.

[59] J.E. Stone, D. Gohara, and G. Shi. **OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems**. *Comp. in Science & Eng.*, **12**(3):66–73, 2010.

[60] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. **A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing**. *IEEE Transactions on Parallel and Distributed Systems*, **26**(1):272–281, Jan 2015.

[61] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. **PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications**. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[62] Ashay Rane, Saurabh Sardeshpande, and James Browne. **Poster: Determining Code Segments That Can Benefit from Execution on GPUs**. In *Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion*, SC ’11 Companion, pages 55–56, New York, NY, USA, 2011. ACM.

[63] A Rane, J Browne, and L Koesterke. **PerfExpert and MACPO: Which code segments should (not) be ported to MIC**. In *TACC-Intel Highly Parallel Computing Symposium*, 2012.

[64] Leonardo Fialho and James Browne. **Framework and Modular Infrastructure for Automation of Architectural Adaptation and Performance Optimization for HPC Systems**. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, **8488** of *Lecture Notes in Computer Science*, pages 261–277. Springer International Publishing, 2014.

[65] N. Bell and J. Hoberock. **Thrust: A Productivity-Oriented Library for CUDA**. In *GPU Computing Gems*, pages 359–371. Morgan Kaufmann Publishers, 2011.

[66] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. **Sequoia: Programming the Memory Hierarchy**. In *Proceedings of the ACM/IEEE SC 2006 Conference*, nov. 2006.

[67] Kyle L. Spafford and Jeffrey S. Vetter. **Aspen: A Domain Specific Language for Performance Modeling**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, 2012.

[68] Reda A. Ammar. **Hierarchical Performance Modeling and Analysis of Distributed Software Systems**. In Sanguthevar Rajasekaran and John Reif, editors, *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter 12. Chapman & Hall, 1 edition, 2007. ISBN: 9781584886235.

[69] Paul Klint, Tijs van der Storm, and Jurgen Vinju. **EASY Metaprogramming with Rascal**. In Fernandes, João and Lämmel, Ralf and Visser, Joost and Saraiva, João, editor, *Generative and Transformational Techniques in Software Engineering III*, **6491** of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011.

[70] Ben van Werkhoven, Jason Maassen, Henri E. Bal, and Frank J. Seinstra. **Optimizing convolution operations on GPUs using adaptive tiling**. *Fut. Gen. Comp. Systems*, **30**:14 – 26, 2014.

[71] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. **Identifying the Key Features of Intel Xeon Phi: A Comparative Approach**. Technical report, Delft University of Technology, 2013.

[72] S. Grauer-Gray, Lifan Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. **Auto-tuning a high-level language targeted to GPU codes**. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[73] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. **Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors**. In *Proceedings of the 27th International*

ACM Conference on International Conference on Supercomputing, ICS '13, pages 273–282, 2013.

[74] Corinne Ancourt and François Irigoin. **Scanning Polyhedra with DO Loops**. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 39–50, New York, NY, USA, 1991. ACM.

[75] Samuel Williams, Andrew Waterman, and David Patterson. **Roofline: An Insightful Visual Performance Model for Multicore Architectures**. *Commun. ACM*, **52**:65–76, April 2009.

[76] Sabela Ramos and Torsten Hoefler. **Modeling Communication in Cache-Coherent SMP Systems: A Case-study with Xeon Phi**. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 97–108, New York, NY, USA, 2013. ACM.

[77] **DAS-4: Distributed ASCI Supercomputer 4** [online]. Available from: `http://www.cs.vu.nl/das4`.

[78] **TOP500 Supercomputer Sites** [online]. Available from: `http://www.top500.org`.

[79] Kevin Beason. **smallpt: Global illumination in 99 lines of C++** [online]. 2008. Available from: `http://www.kevinbeason.com/smallpt`.

[80] David Bucciarelli. **SmallptGPU** [online]. 2009. Available from: `http://davibu.interfree.it/opencl/smallptgpu/smallptGPU.html`.

[81] Ping Xiang, Yi Yang, and Huiyang Zhou. **Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation**. In *Int. Symp. on High Perf. Comp. Arch. (HPCA)*, pages 284–295, 2014.

[82] Jeffrey Dean and Sanjay Ghemawat. **MapReduce: Simplified Data Processing on Large Clusters**. *Commun. ACM*, **51**(1):107–113, January 2008.

[83] Jeff A. Stuart and John D. Owens. **Multi-GPU MapReduce on GPU Clusters**. In *Int. Par. and Dist. Proc. Sym. (IPDPS)*, pages 1068–1079, Los Alamitos, CA, USA, 2011. IEEE Comp. Society.

[84] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. **HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL**. In *IPDPSW '13*, pages 1918–1927, Washington, DC, USA, 2013. IEEE Computer Society.

[85] Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 1st edition, 2009.

[86] G. Frost. **APARAPI: API for data parallel Java** [online]. 2011. Available from: `https://code.google.com/p/aparapi`.

[87] Ismail El-Helw, Rutger Hofman, and Henri E. Bal. **Scaling MapReduce Vertically and Horizontally**. In *SC '14: Proc. of the 2014 ACM/IEEE conf. on Supercomputing.* ACM, 2014.

[88] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. **Productive Programming of GPU Clusters with OmpSs**. In *Int. Par. and Dist. Proc. Sym. (IPDPS)*, pages 557–568, Los Alamitos, CA, USA, 2012. IEEE Comp. Society.

[89] J.M. Perez, R.M. Badia, and J. Labarta. **A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures**. In *IEEE Int. Conf. on Cluster Computing*, pages 142 –151, Sep. 2008.

[90] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. **Self-Adaptive OmpSs Tasks in Heterogeneous Environments**. In *Int. Par and Dist. Proc. Sym. (IPDPS)*, pages 138–149, Washington, DC, USA, 2013. IEEE Computer Society.

[91] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. **StarPU: a unified platform for task scheduling on heterogeneous multicore architectures**. *Concurrency and Computation: Practice and Experience*, **23**(2):187–198, 2011.

[92] Steffen Ernsting and Herbert Kuchen. **Algorithmic skeletons for multi-core, multi-GPU systems and clusters**. *International Journal of High Performance Computing and Networking*, **7**(2):129–138, 2012.

[93] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J.J. Dongarra. **PaRSEC: Exploiting Heterogeneity to Enhance Scalability**. *Computing in Science Engineering*, **15**(6):36–45, Nov 2013.

[94] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Piotr Luszczek, and Jack J. Dongarra. **Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach**. In Samee U. Khan, Lizhe Wang, and Albert Y. Zomaya, editors, *Scalable Computing and Communications: Theory and Practice*, pages 699–733. John Wiley & Sons, Jan 2013.

[95] R. Wester, C. Baaij, and J. Kuper. **A two step hardware design method using CλaSH**. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 181–188, Aug 2012.

List of Publications

Cashmere: Heterogeneous Many-Core Computing. Pieter Hijma, Ceriel J.H. Jacobs, Rob V. van Nieuwpoort, and Henri E. Bal. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015), 25-29 May 2015, Hyderabad, India.*, 2015.

Stepwise-refinement for performance: a methodology for many-core programming. Pieter Hijma, Rob V. van Nieuwpoort, Ceriel J.H. Jacobs, and Henri E. Bal. *Concurrency and Computation: Practice and Experience*, 2015. `http://dx.doi.org/10.1002/cpe.3416`.

Programming Many-Cores on Multiple Levels of Abstraction. Pieter Hijma, Rob V. van Nieuwpoort, and Henri E. Bal. In *Proceedings of the 5th USENIX Conference on Hot Topics in Parallelism (Poster presentation)*, HotPar '13, pages 1–7, Berkeley, CA, USA, 2013. USENIX Association.

Generating synchronization statements in divide-and-conquer programs. Pieter Hijma, Rob V. van Nieuwpoort, Ceriel J.H. Jacobs, and Henri E. Bal. *Parallel Computing*, 38(1-2):75 – 89, January–February 2012.

Automatically Inserting Synchronization Statements in Divide-and-Conquer Programs. Pieter Hijma, Rob V. van Nieuwpoort, Ceriel J.H. Jacobs, and Henri E. Bal. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1233–1241, May 2011.

Towards an Effective Unified Programming Model for Many-Cores. Ana L. Varbanescu, Pieter Hijma, Rob V. van Nieuwpoort, and Henri E. Bal. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 681–692, May 2011.

Samenvatting

Het programmeren van many-cores op meerdere abstractieniveaus

De afgelopen decennia hebben we een exponentiële groei van het aantal transistors op een computerchip meegemaakt. Dit ging gepaard met een exponentiële groei van de kloksnelheid van processors. Dit zorgde ervoor dat we automatisch een hogere performance kregen zonder dat de software aangepast hoefde te worden.

Echter, rond 2005 bleek dat de kloksnelheid niet verder omhoog kon, terwijl het aantal transistors door kon blijven groeien. Dit betekende het einde van het "single-core" tijdperk waarin processors een enkele rekenkern of "compute-core" bevatte. Dit type processor was sterk geoptimaliseerd om een stroom van sequentiële instructies, instructies die na elkaar worden uitgevoerd, te verwerken.

Omdat het aantal transistors op een chip door kon blijven groeien, maar dit niet gebruikt kon worden om de kloksnelheid op te voeren, kregen we "multicore" processoren met meerdere rekenkernen. Echter, dit betekende een fundamentele verandering van hoe men een processor programmeert: Om een processor optimaal te benutten, is het nodig een parallel programma aan te leveren, met meerdere stromen van instructies die mogelijkerwijs met elkaar moeten synchroniseren.

Deze multi-core processors bevatten nog steeds veel optimalisaties om sequentiële instructiestromen te optimaliseren wat veel ruimte op de chip kost, ruimte die ook gebruikt zou kunnen worden om rekenkernen toe te voegen. Dit proefschrift gaat over many-core processoren, waarbij zoveel mogelijk ruimte wordt benut om te rekenen. Dit betekent dat deze processors volledig toegespitst zijn op snel rekenen, maar alleen voor programma's die veel parallellisme

uitdrukken.

Parallel programmeren is moeilijk, vooral als het gaat om many-core processoren omdat deze processoren veel specifieke hardware eigenschappen bevatten om nog meer performance te verkrijgen. Dit resulteert in een ingewikkelde hardware interface naar de programmeur toe, maar zorgt er ook voor dat een programma factors sneller kan draaien, mits men goed rekening houdt met alle hardware-specifieke eigenschappen.

In dit proefschrift nemen wij het volgende standpunt in: Wij beschouwen het *single-core* tijdperk als een fortuinlijke situatie waarin we automatisch snellere programma's kregen met nieuwere generaties processors. Het *multi-core* tijdperk beschouwen wij vervolgens als een overgangsperiode naar een tijdperk waarin we te maken krijgen met allerlei limieten van de hardware, zoals een limiet op de kloksnelheid of een limiet op de snelheid van het geheugen. Om toch performance te krijgen, is het vervolgens nodig om met allerlei hardware-specifieke details rekening te houden om de processor optimaal te benutten. Wij zien *many-core processors* als een eerste manifestatie van deze trend. In dit proefschrift beschouwen wij deze limieten en problemen als een *programmeer-probleem*: Uiteindelijk zullen we allerlei hardwarelimieten tegenkomen wat zal leiden tot een ingewikkelde hardwareinterface waardoor de processoren steeds moeilijker te programmeren zijn. De hoofdvraag van dit proefschrift is hoe we effectief many-core hardware kunnen programmeren terwijl we nog steeds goede performance halen.

In dit proefschrift presenteren wij twee programmeersystemen, Many-Core Levels (MCL) en Cashmere, die een bijdrage leveren aan het programmeerprobleem dat hierboven beschreven is. Cashmere bouwt voort op een al bestaand systeem genaamd Satin, maar integreert hierbij MCL. Hoofdstuk 2 gaat nog niet over many-cores, maar presenteert een analyse op Satin programma's. Deze analyse leidde tot een aantal belanrijke conclusies die het ontwerp van MCL en Cashmere beïnvloed hebben.

Hoofdstuk 3 presenteert Many-Core Levels. MCL is een programmeersysteem voor many-core processors dat het mogelijk maakt om deze processors op meerdere abstractieniveaus te programmeren.

Programmeren, zodanig dat je rekening houdt met allerlei hardware-specifieke details noemen wij programmeren op een *laag* niveau. Aan de andere kant, programmeren op een *hoog* niveau geeft een programmeur de mogelijkheid te abstraheren van allerlei details en het programma op te schrijven zonder rekening te houden met de onderliggende hardware. Dit heeft allerlei voordelen, zoals bijvoorbeeld portabiliteit. Het programma is algemeen genoeg om vertaald te worden naar verschillende soorten hardware. Daarnaast is het programma vaak eenvoudiger uit te drukken en daardoor beter te onderhouden. Echter, de programmeur verliest ook controle over de hardware. Nu is dit vaak niet een probleem, maar many-core processoren zijn enkel en alleen bedoeld om performance te behalen en in dat geval doen de hardware details er wel toe. Voor het gebied van many-cores vormt dit dan ook een interessante afweging. Hoofdstuk 3 presenteert oplossingen voor deze afweging.

Een andere belangrijke bijdrage van dit hoofdstuk is dat MCL de programmeur een gestructureerde methodologie biedt om many-core processors te programmeren. Wij noemen deze methodologie "stapsgewijs verfijnen voor performance".

Hoofdstuk 4 presenteert het programmeersysteem Cashmere. Cashmere integreert Satin en MCL om een systeem te verkrijgen voor heterogene clustercomputers van many-cores. Hieronder verstaan wij een computer die bestaat uit een groep processors met verschillende types many-cores die aaneengeschakeld zijn met een snel netwerk. Dit soort computers is bedoeld om grote rekenproblemen op te lossen. Een clustercomputer is al moeilijk te programmeren, een clustercomputer met many-cores is nog moeilijker te programmeren, maar een clustercomputer met verschillende typen many-cores nóg veel moeilijker. In dit hoofdstuk laten wij zien dat Cashmere zeer goed schaalt en zeer goede performance behaalt met een elegant programmeermodel dat nauwelijks aangepast is ten opzichte van Satin.

In hoofdstuk 5 concluderen wij dat MCL en Cashmere een belangrijke bijdrage leveren aan de programmeerproblemen die many-core hardware met zich meebrengt, hardware waarbij men rekening moet houden met allerlei hardware-specifieke details om hoge performance te behalen.