# 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.