INFORMATIK

IMPROVING PROGRAMMABILITY AND PERFORMANCE PORTABILITY ON MANY-CORE PROCESSORS

Inaugural-Disseration
zur Erlangung des Doktorgrades der
Naturwissenschaften im Fachbereich
Mathematik und Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Westfälischen Wilhelms-Universität Münster

vorgelegt von

MICHEL STEUWER
aus Duisburg

– 2015 –

Improving Programmability
and Performance Portability
on Many-Core Processors

Michel Steuwer

DEKAN:	Prof. Dr. Martin Stein
ERSTER GUTACHTER:	Prof. Dr. Sergei Gorlatch
ZWEITER GUTACHTER:	Lecturer Dr. Christophe Dubach
TAG DER MÜNDLICHEN PRÜFUNG:	.....
TAG DER PROMOTION:	.....

ABSTRACT

Computer processors have radically changed in the recent 20 years with multi- and many-core architectures emerging to address the increasing demand for performance and energy efficiency. Multi-core CPUs and Graphics Processing Units (GPUs) are currently widely programmed with low-level, ad-hoc, and unstructured programming models, like multi-threading or OpenCL/CUDA. Developing functionally correct applications using these approaches is challenging as they do not shield programmers from complex issues of parallelism, like deadlocks or non-determinism. Developing optimized parallel programs is an even more demanding task – even for experienced programmers. Optimizations are often applied ad-hoc and exploit specific hardware features making them non-portable.

In this thesis we address these two challenges of programmability and performance portability for modern parallel processors.

In the first part of the thesis, we present the SkelCL programming model which addresses the *programmability* challenge. SkelCL introduces three main high-level features which simplify GPU programming: 1) parallel container data types simplify the data management in GPU systems; 2) regular patterns of parallel programming (a. k. a., algorithmic skeletons) simplify the programming by expressing parallel computation in a structured way; 3) data distributions simplify the programming of multi-GPU systems by automatically managing data across all the GPUs in the system. We present a C++ library implementation of our programming model and we demonstrate in an experimental evaluation that SkelCL greatly simplifies GPU programming without sacrificing performance.

In the second part of the thesis, we present a novel compilation technique which addresses the *performance portability* challenge. We introduce a novel set of high-level and low-level parallel patterns along with a set of rewrite rules which systematically express high-level algorithmic implementation choices as well as low-level, hardware-specific optimizations. By applying the rewrite rules pattern-based programs are transformed from a single portable high-level representation into hardware-specific low-level expressions from which efficient OpenCL code is generated. We formally prove the soundness of our approach by showing that the rewrite rules do not change the program semantics. Furthermore, we experimentally confirm that our novel compilation technique can transform a single portable expression into highly efficient code for three different parallel processors, thus, providing performance portability.

PUBLICATIONS

This thesis is based on ideas and results which have been described in the following publications:

M. Steuwer, P. Kegel, S. Gorlatch. *SkelCL – A Portable Multi-GPU Skeleton Library*. Technical Report. University of Münster, Dec. 2010.

M. Steuwer, P. Kegel, S. Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. Anchorage, AK, USA: IEEE, May 2011, pp. 1176–1182.

M. Steuwer, S. Gorlatch, M. Buß, S. Breuer. "Using the SkelCL Library for High-Level GPU Programming of 2D Applications." In: *Euro-Par 2012: Parallel Processing Workshops, August 27-31, 2012. Revised Selected Papers*. Edited by Ioannis Caragiannis et al. Vol. 7640. Lecture Notes in Computer Science. Rhodes Island, Greence: Springer, Aug. 2012, pp. 370–380.

M. Steuwer, P. Kegel, S. Gorlatch. "A High-Level Programming Approach for Distributed Systems with Accelerators." In: *New Trends in Software Methodologies, Tools and Techniques New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Eleventh SoMeT*. Edited by Hamido Fujita and Roberto Revetria. Vol. 246. Frontiers in Artificial Intelligence and Applications. Genoa, Italy: IOS Press, Sept. 2012, pp. 430–441.

M. Steuwer, P. Kegel, S. Gorlatch. "Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library." In: *2012 26th IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. Shanghai, China: IEEE, May 2012, pp. 1858–1865.

P. Kegel, **M. Steuwer**, S. Gorlatch. "Uniform High-Level Programming of Many-Core and Multi-GPU Systems." In: *Transition of HPC Towards Exascale Computing*. Edited by Erik D'Hollander, Jack Dongarra, Ian Foster, Lucio Grandinetti, and Gerhard Joubert. Vol. 24. IOS Press, 2013, pp. 159–176.

M. Steuwer, S. Gorlatch. "High-Level Programming for Medical Imaging on Multi-GPU Systems using the SkelCL Library." In: *Proceedings of the International Conference on Computational Science (ICCS 2013)*. Edited by Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot. Vol. 18. Procedia Computer Science. Barcelona, Spain: Elsevier, June 2013, pp. 749–758.

M. Steuwer, S. Gorlatch. "SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems." In: *Parallel Computing Technologies - 12 International Conference (PaCT 2013)*. Edited by Victor Malyshkin. Vol. 7979. Lecture Notes in Computer Science. St. Petersburg, Russia: Springer, Sept. 2013, pp. 258–272.

S. Breuer, **M. Steuwer**, S. Gorlatch. "Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems." In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*. Edited by Armin Größlinger and Harald Köstler. HiStencils 2014. Vienna, Austria, Jan. 2014, pp. 15–21.

S. Gorlatch, **M. Steuwer**. "Towards High-Level Programming for Systems with Many Cores." In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. Edited by Andrei Voronkov and Irina Virbitskaite. Vol. 8974. Lecture Notes in Computer Science. Springer, 2014, pp. 111–126.

C. Kessler, S. Gorlatch, J. Emmyren, U. Dastgeer, **M. Steuwer**, P. Kegel. "Skeleton Programming for Portable Many-Core Computing." In: *Programming Multi-core and Many-core Computing Systems*. Edited by Sabri Pllana and Fatos Xhafa. Wiley Series on Parallel and Distributed Computing. Wiley-Blackwell, Oct. 2014.

M. Steuwer, M. Friese, S. Albers, S. Gorlatch. "Introducing and Implementing the Allpairs Skeleton for Programming Multi-GPU Systems." In: *International Journal of Parallel Programming* 42.4 (Aug. 2014), pp. 601–618.

M. Steuwer, S. Gorlatch. "SkelCL: A high-level extension of OpenCL for multi-GPU systems." In: *The Journal of Supercomputing* 69.1 (July 2014), pp. 25–33.

M. Steuwer, M. Haidl, S. Breuer, S. Gorlatch. "High-Level Programming of Stencil Computations on Multi-GPU Systems Using the SkelCL Library." In: *Parallel Processing Letters* 24.03 (Sept. 2014). Edited by Armin Größlinger and Harald Köstler, pp. 1441005/1–17.

M. Steuwer, C. Fensch, C. Dubach. *Patterns and Rewrite Rules for Systematic Code Generation (From High-Level Functional Patterns to High-Performance OpenCL Code)*. Technical Report. Feb. 9, 2015.

M. Steuwer, C. Fensch, S. Lindley, C. Dubach. "Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Patterns to High-Performance OpenCL Code." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP. accepted for publication. Vancouver, Canada: ACM, 2015.

ACKNOWLEDGEMENTS

This thesis is the result of 4 and a half years of research during which I have been supported by many helpful people.

First and foremost, I thank my supervisor Sergei Gorlatch how has given me the guidance and freedom which enabled my research. I'm especially thankful for this support of my numerous and fruitful research visits at the University of Edinburgh.

I thank Christophe Dubach for our joint research collaboration with many rich and extensive discussions. I'm especially grateful for this continuous enthusiasm for our collaborative work.

During my time in Münster I had the pleasure of a friendly and supportive group of colleagues, which includes: Sebastian Albers, Frank Glinka, Waldemar Gorus, Michael Haidl, Tim Humernbrum, Julia Kaiser-Mariani, Philipp Kegel Dominik Meiländer, Mohammed Nsaif, Alexander Ploss, and Ari Rasch.

I want to specifically thank Philipp Kegel for our numerous discussions and close collaboration on SkelCL and Sebastian Albers and Michael Haidl for their work on our joint publications.

I also want to mention and thank two friends and fellow PhD students from Münster Benjamin Risse and Sven Strothoff with whom I could share the day-to-day struggles of a PhD student.

During the work on this thesis I visited the University of Edinburgh numerous times and eventually started working there as a Research Associate. During my visits I have met many friendly, interesting, and inspiring people – too many to list all of them here. I want to especially thank Thibaut Lutz which whom I collaborated closely during my first stay in Edinburgh and his supervisors Christian Fensch and Murray Cole. I already mentioned Christophe Dubach which whom I started a fruitful research collaboration leading to the development of the code generation technique presented in the second half of this thesis.

I'm grateful for the financial support I received from the EU founded HPCEuropa2 program and the HiPEAC network of excellence which made my research visits to Edinburgh possible.

As I spend a large part of my time in Münster teaching, supervising, and working with students I want to thank all the students participating in the 14 courses I co-supervised. By teaching students computer science I gained many useful skills which have contributed to the quality of my research.

I particular want to mention and thank the students I have co-supervised during their Bachelor and Master theses. These are: Markus Blank-Burian, Stefan Breuer, Julian Buscher, Matthias Buss,

Matthias Droste, Malte Friese, Tobias Günnewig, Wadim Hamm, Kai Kientopf, Lars Klein. Sebastian Missbach, Michael Olejnik, Florian Quinkert, Patrick Schiffler, and Jan-Gerd Tenberge.

Last, but not least I thank my parents and my brothers for their support, love, and interest in me and my research. Especially, for their continuing engagement and enthusiasm about my research visits to universities and conferences around the world.

Michel Steuwer
April 2015
Edinburgh

CONTENTS

Part I

INTRODUCTION & BACKGROUND

1 INTRODUCTION

Computer architectures have radically changed in the last 20 years with the introduction of multi- and many-core designs in almost all areas of computing: from mobile devices and desktop computers to high-performance computing systems. Furthermore, recently novel types of specialized architectures, especially *Graphics Processing Units* (GPUs), have been developed featuring large amounts of processing units for accelerating computational intensive applications. Parallelism and specialization are seen by many as two crucial answers to the challenge of increasing performance and at the same time improving energy efficiency [70, 125].

Modern multi- and many-core processors are still programmed with programming languages developed in the 1980s and 1990s, like C++ or Java, or even older languages like C (from the 1970s) or Fortran (from the 1950s). These languages have a simplified view of the underlying hardware, often more or less directly resembling the Von Neumann architecture. Multi-core processors are programmed with low-level libraries providing *threads* where the programmer explicitly controls the computations on each core executing in parallel with the other cores. This style of programming has turned out to be extremely difficult, as threads running concurrently on distinct cores can modify shared data leading to serious problems like deadlocks, race conditions, and non-determinism. Even if programmers develop correct parallel implementations, optimizing these implementations for modern parallel processors is a challenging task even for experienced programmers. Due to the lack of better programming systems, programmers are forced to manually develop low-level hardware-specific implementations optimized towards a particular hardware architecture to achieve high performance, which limits portability.

This thesis describes our work to address these issues. In the first part of the thesis, we present a high-level programming model and its implementation which is designed to simplify the programming of modern parallel processors, especially systems comprising multiple GPUs with many cores. In the second part, we discuss a system which generates portable high-performance code for different modern parallel processors, particularly a multi-core CPU and two different types of GPUs, from a single high-level program representation. In the final part of the thesis, we outline how these two systems can be combined in the future to obtain a programming system which simplifies programming and achieves high performance on different hardware architectures.

We will start the thesis with an introduction on programming modern parallel processors. From this we will identify the two main challenges we address in this thesis: the *programmability* challenge and the *performance portability* challenge. We will then list our contributions before presenting the outline for the rest of the thesis.

1.1 MULTI-CORE PROCESSORS AND THEIR PROGRAMMING

Traditionally, the performance of microprocessors has been mainly increased by boosting the clock frequency, optimizing the execution flow, and improving caches [145]. Processor development drastically changed around 10 years ago, as shown in Figure 1.1. While the number of transistors continues to grow according to Moore's Law [116], the clock frequency and power consumption hit a plateau around 2005. The reason for this lies in multiple physical effects, especially the physical limit of signal speed and increased heat development, mainly due to the increased power consumption, both of which limit further increasing of the clock frequency. The predominant answer to this challenge has been the introduction of multiple distinct *cores* within one processor which can operate independently and concurrently. Parallelism has been exploited in computer architectures for a long time at the bit level and the instruction level, but different than before with the introduction of multiple cores, this new *thread-level parallelism* has been made explicit to the programmer. This has profound effects on the design and development of software running on modern multi-core processors. Most programming languages allow programmers to exploit multiple cores by the means of *threads*. A thread is a sequence of instructions defined by the programmer which can be scheduled by the runtime system to be executed on a particular core. Multiple threads can be executed concurrently and the programmer is responsible to use synchronization primitives, like locks or semaphores, for coordinating the execution of multiple threads. Programming with threads is widely regarded as extremely difficult [106] mainly because multiple threads can simultaneously modify a shared memory region. Even when programs are constructed carefully subtle problems can arise which are hard to detect but can have severe consequences. Nevertheless, threads are still the de-facto standard for programming multi-core CPUs.

While Moore's law still holds and transistor counts increase by further shrinking the transistor size, a related observation, known as Dennard scaling [52], has broken down. Dennard scaling suggested that power consumption is proportional to the area used for transistors on the chip. Combined with Moore's law this meant, that the energy efficiency of processors doubled roughly every 18 month. The primary reason for the breakdown of Dennard scaling around 2005 were physical effects appearing at small scale, especially current leakage

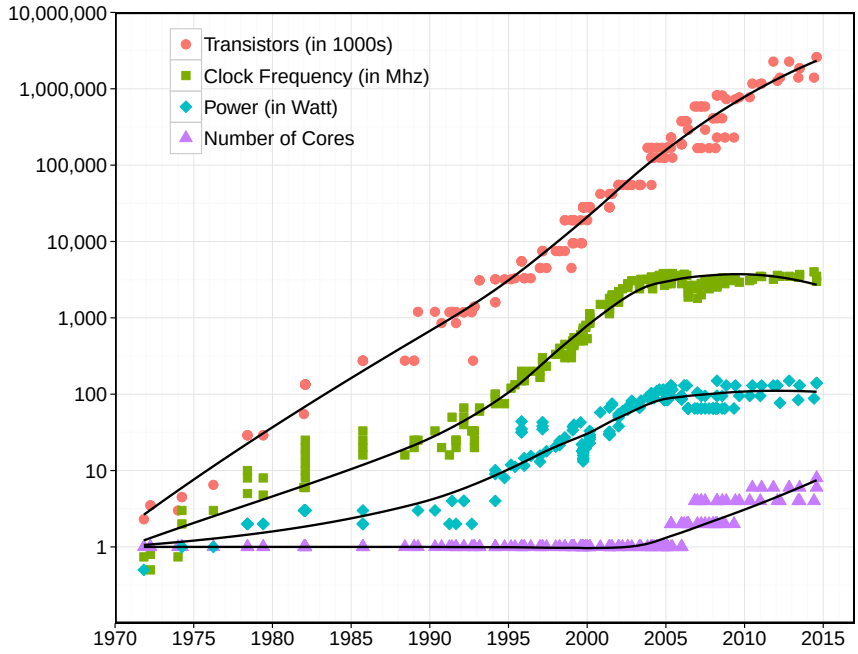


Figure 1.1: Development of Intel Desktop CPUs over time. While transistor count continues to grow, around 2005 clock frequency and power consumption have reached a plateau. As an answer multi-core processors emerged. Inspired by [145].

and increased heat development. This has led to architectures which particularly focus on their energy efficiency, the most prominent example of such architectures are modern graphics processing units (GPUs). Originally developed for accelerating the rendering of complex graphics and 3D scenes, GPU architectures have been recently generalized to support more types of computations. Some people refer to this development using the term general-purpose computing on graphics processing units (GPGPU).

Technically GPU architectures are multi-core architectures like modern multi-core CPUs, but each individual core on a GPU typically has dozens or hundreds of functional units which can perform computations in parallel following the *Single Instruction, Multiple Data* (SIMD) principle. These types of architectures are optimized towards a high throughput of computations, therefore, they focus on performing a large amount of operations in parallel and feature no, or only small, caches to prevent or mitigate latencies of the memory: if a thread stalls waiting for the memory, another thread takes over and keeps the core busy. For multi-core CPUs switching between threads is more expensive, therefore, CPUs are instead optimized to avoid long latencies when accessing the memory with a deep cache hierarchy and advanced architectural features, like long pipelines and out-of-order execution, all of which are designed to keep each core busy.

1.2 THE PROGRAMMABILITY CHALLENGE

Developing efficient programs for modern parallel processors is a challenging task. The current approach for programming multi-core CPUs is using multithreading, i. e., explicitly managing the individual execution paths (called *threads*) in a single program. Threads communicate via a shared memory, where all threads have the possibility to modify any data item. This fundamental principle gives the programmer full control over the interaction of the threads, but greatly complicates the reasoning about the programs behavior. The extensive and explicit control implies a low level of abstraction, where the programmer deals with many details of the execution explicitly. This makes the development of programs using threads complex and certain types of bugs likely, including deadlocks and race conditions, where the computed result depends on the order the threads are executed in. This ultimately leads to extensive problems during the development cycle as well as possible non-deterministic and undesirable behavior at runtime.

Programming GPUs requires a different programming approach, as it is obviously infeasible to manage thousands of threads individually. CUDA [44] and OpenCL [119] are the two most popular approaches for programming GPUs. Both approaches are quite similar and require the programmer to write a function, called *kernel*, which will be executed in parallel on the GPU. The kernel usually contains source code for identifying the thread executing the particular kernel instance. This information is used to coordinate on which data each thread operates on. While this *Single Program, Multiple Data* (SPMD) programming model makes the management of many threads feasible, it does not address most problems associated with the low-level programming using threads. For example, explicit synchronization of threads is still required and problems like deadlocks and race conditions can still easily occur. Furthermore, GPU programming currently involves extensive boilerplate code to manage the execution on the GPU, including the compilation of the kernel (in OpenCL), transferring data to and from the GPU, and launching the kernel by explicit specifying the number of threads to be started in parallel.

All of this leads to our first central research *challenge of programmability*: to design a programming approach which significantly raises the level of abstraction and, thus, greatly simplifies the development process of parallel applications, without sacrificing high-performance on multi-core CPUs and GPUs. We will underpin the necessity to address this challenge with an extensive study of state-of-the-art GPU programming using a concrete application example in Chapter 3. In Part II of this thesis, we introduce, discuss, and evaluate a novel high-level programming model which addresses this challenge.

1.3 THE PERFORMANCE PORTABILITY CHALLENGE

Developing functionally correct applications for multi-core CPUs or GPUs is very challenging as discussed in the preceding subsection. Unfortunately, the development of high-performing optimized applications is even more demanding – even for experienced programmers. Applying optimizations often requires substantial restructuring of existing code. Optimizations are often applied ad-hoc, following certain "rules of thumb" derived from small-scale profiling and experiments as well as previously gained experience. There exists no widely established approach for performing optimizations more systematically. Especially, the performance implications of optimizations are currently not predictable: sometimes supposed optimizations lead to actual slowdowns in certain circumstances.

Modern optimizing compilers perform very sophisticated low-level optimizations, including various loop optimizations and optimizations based on data-flow analysis. For all these type of optimizations, the compiler first has to recover parts of the high-level program semantics using code analysis before applying the optimization. As these analyses quickly become very complex, compilers are ultimately limited in their optimization capabilities. High-level optimizations which might restructure the entire source code are often out of the reach of low-level compiler optimizations and, therefore, require explicit and often extensive manual refactoring. The state-of-the-art optimizing compilers are lacking the high-level semantic information necessary for performing this kind of optimizations.

Many optimizations exploit particular hardware-specific features of the target architecture and are, thus, inherently not portable. In current low-level programing models the programmers often encode these optimizations directly. As a result programmers are confronted with the trade-off of either fully optimize their code for performance – at the expense of code portability – or implement a portable version – at the expense of missing out on the highest performance possible. Nowadays, programmers often choose a middle ground by introducing conditionals and branches in their code to select optimizations for the given hardware, while maintaining portability. This, obviously, complicates the development process and decreases maintainability.

This leads to our second central research *challenge of performance portability*: to design a systematic approach that generates an optimized, hardware-specific code for different target architectures from a single portable, high-level program representation. We will confirm the lack of portability of performance in current state-of-the-art programming approaches using a study comparing optimizations on three different hardware architectures in Chapter 5. In Part III of this thesis, we introduce, discuss, and evaluate a novel systematic code generation approach which addresses this challenge.

1.4 CONTRIBUTIONS OF THIS THESIS

This thesis makes the following four major contributions:

For addressing the programmability challenge:

A HIGH-LEVEL PROGRAMMING MODEL FOR MUTLI-GPU SYSTEMS

We present the design and implementation of our high-level programming model SkelCL for programming multi-GPU systems. SkelCL offers three high-level features for simplifying the programming of GPUs: container data types, algorithmic skeletons, and data distributions. We discuss our C++ library implementation of the SkelCL programming model which is deeply integrated with modern C++ features. Finally, we evaluate the level of abstraction and runtime performance of SkelCL using a set of application studies.

TWO NOVEL ALGORITHMIC SKELETONS

We introduce two novel algorithmic skeletons specialized towards *allpairs* and *stencil* computations. We present their formal definitions and discuss possible use cases in real-world applications. For the allpairs skeleton we identify and formally define an optimization rule which enables an optimized implementation on GPUs. We discuss and evaluate efficient implementations for both skeletons on a multi-GPU system.

For addressing the performance portability challenge:

A FORMAL SYSTEM FOR REWRITING PATTERN-BASED PROGRAMS

We introduce a novel system comprising a set of high-level and low-level patterns along with a set of rewrite rules. The rewrite rules express high-level algorithmic and low-level optimization choices which can be systematically applied to transform a program expressed with our functional patterns. We prove the soundness of our approach by showing that applying the rewrite rules does not change the program semantics.

A CODE GENERATOR OFFERING PERFORMANCE PORTABILITY

Based on our formal system of rewrite rules, we present a novel code generator which generates from a single pattern-based program highly efficient OpenCL code for three different hardware platforms: one multi-core CPU and two different GPU architectures. We discuss the design of our code generator and evaluate the performance of the generated OpenCL code as compared against highly optimized hand-written library codes.

1.5 OUTLINE OF THIS THESIS

The remainder of this thesis is structured as follows.

Part I – Introduction

CHAPTER 2 provides an introduction into the main aspects of programming modern multi-core CPUs and GPUs. We will start with a discussion of the hardware where we mainly focus on GPUs, discuss in detail their architectures, the state-of-the-art programming approaches, and optimization opportunities. We will then shift our focus and introduce the algorithmic skeleton parallel programming model including its origins in functional programming.

Part II – The SkelCL High-Level Programming Model

CHAPTER 3 addresses the programmability challenge identified in this chapter by introducing our novel SkelCL high-level programming model targeted towards multi-GPU systems and its implementation as a C++ library. We will start by further motivating the need for high-level abstractions to simplify the software development process. Then we present SkelCL's three high-level features: 1) parallel container data types, 2) algorithmic skeletons, and 3) data distributions. The combination of these features greatly simplifies the programming of multi-GPU applications. Finally, we will discuss the SkelCL library implementation and its integration within C++.

CHAPTER 4 evaluates the SkelCL programming model and library using multiple application studies. We will start with discussing typical benchmark applications like the Mandelbrot set computation and popular benchmarks from linear algebra, including matrix multiplication. We then continue with discussing more advanced applications from different areas of computing, including image processing, medical imaging, and physics. We evaluate the improvement of programmability and the performance of SkelCL as compared with implementations written using the state-of-the-art OpenCL programming approach.

Part III – A Novel Code Generation Approach using Patterns

CHAPTER 5 addresses the performance portability challenge identified in this chapter by introducing a novel approach for generating highly optimized hardware-specific code for different target architectures from a single portable, high-level program representation. We will start the chapter with a study showing that optimizations in OpenCL are not portable across different hardware architectures. This

emphasizes the motivation for performance portability in order to preserve maintainability while increasing performance. We continue by discussing the design of our code generation approach consisting of a set of parallel patterns combined with a set of rewrite rules. The rewrite rules allow to systematically rewrite pattern-based programs without changing their semantics which we will show formally. Finally, we discuss how the rewrite rules can be systematically applied to optimize pattern-based programs for a particular hardware architecture.

Chapter 6 evaluates our code generation approach using a set of benchmark applications. We evaluate the performance of the code generated using our approach and compare it against highly tuned library code on three different hardware architectures.

Part IV – Summary & Conclusions

Chapter 7 summarizes Part I and Part II and describes their relation. We will discuss how SkelCL, presented in Part II, could be combined in the future with the code generator, presented in Part III, to obtain a system offering both high-level abstractions to simplify programming and portable, high performance on different hardware architectures. We will then discuss possible future extensions of our research.

Chapter 8 concludes with a comparison against related work.

2 BACKGROUND

In this chapter we will introduce and discuss the technical background of this thesis. We will start by discussing the structure and functionality of multi-core CPUs and GPUs. We will focus on GPUs as their architecture is quite different to traditional CPUs, and because GPUs will be the main hardware target of our SkelCL programming model presented later. Then we will discuss the most common programming approaches for multi-core CPUs and GPUs, with a particular focus on OpenCL.

We will then introduce the idea of structured parallel programming and discuss its origin in functional programming. In this approach, predefined parallel patterns (a. k. a., *algorithmic skeletons*) hiding the complexities of parallelism are used for expressing parallel programs. This thesis builds upon this idea, developing a structured parallel programming model for GPUs and a novel compilation technique that transforms pattern-based programs to efficient hardware-specific implementations.

2.1 MODERN PARALLEL PROCESSORS

As discussed in the introduction in Chapter 1, virtually all modern processors feature multiple cores to increase their performance and energy efficiency. Here we will look at two major types of modern parallel processors: multi-core CPUs and GPUs. Multi-core CPUs are *latency-oriented* architectures [70], i. e., they are optimized to hide memory latencies with large caches and various other advanced architectural features, like out-of-order execution and extensive branch prediction. GPUs are *throughput-oriented* architectures [70], i. e., they are optimized to increase the overall computational throughput of many parallel tasks instead of optimizing single task performance.

We will discuss both architectures in the following two sections.

2.1.1 *Multi-Core CPUs*

Figure 2.1 shows an abstract representation of a typical multi-core CPU architecture, like Intel's latest CPU architecture: Haswell [32]. The CPU is divided into multiple cores each of which features two levels of caches. The smallest but fastest cache is called L1 cache and is typically divided into a distinct cache for instructions and a data cache, each of 32 kilobyte for the Haswell architecture. Each core

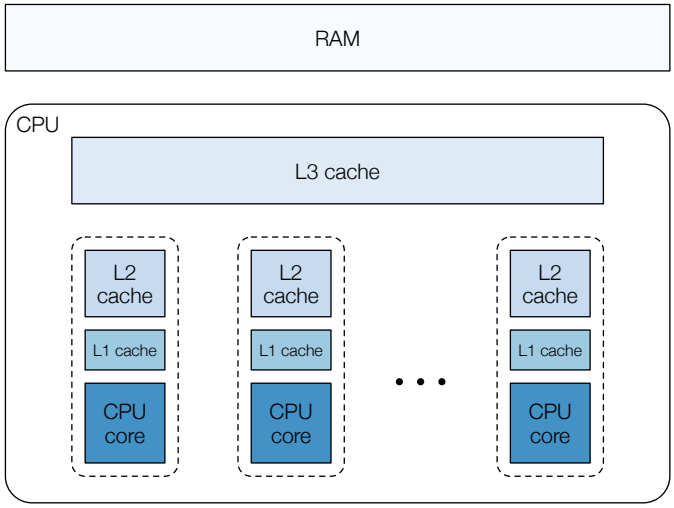


Figure 2.1: Overview of a multi-core CPU architecture

also features a second level cache (L2 cache) of 256 kilobyte for the Haswell architecture. The CPU has a large cache (L3 cache) which is shared by all cores. This cache is often several megabytes large, for Intel's Haswell architecture the L3 cache is at least 2 and up to 45 megabytes in size.

Each CPU core performs computations completely independently of the other cores, and it consists itself of multiple execution units which can perform computations in parallel. This happens transparently to the programmer even when executing a sequential program. CPUs use advanced branch prediction techniques to perform aggressive speculative execution for increased performance. The CPU core exploits *instruction level parallelism* (ILP) by performing out-of-order execution which re-orders instructions prior to execution while still respecting their dependencies: e. g., if two or more instructions have no dependencies then they can be executed in parallel. In the Intel Haswell architecture, 4 arithmetic operations and 4 memory operations can be performed in parallel in one clock cycle on every core. Furthermore, most modern CPU architectures support SIMD vector extensions like the *Advanced Vector Extensions* (AVX) or the older *Streaming SIMD Extensions* (SSE). These extensions add additional instructions to the instruction set architecture, allowing the compiler to generate code explicitly grouping data into vectors of small fixed sizes which can be processed in parallel. The current AVX extension allows vectors of up to 256 bits, e. g., a grouping of 8 single precision floating point numbers, to be processed in parallel. Most modern optimizing compilers perform some form of automatic vectorization, but often programmers vectorize programs manually when the compiler fails to generate sufficiently optimized code.

The design of multi-core CPUs is motivated by the overall goal to provide high performance when executing multi-threaded programs

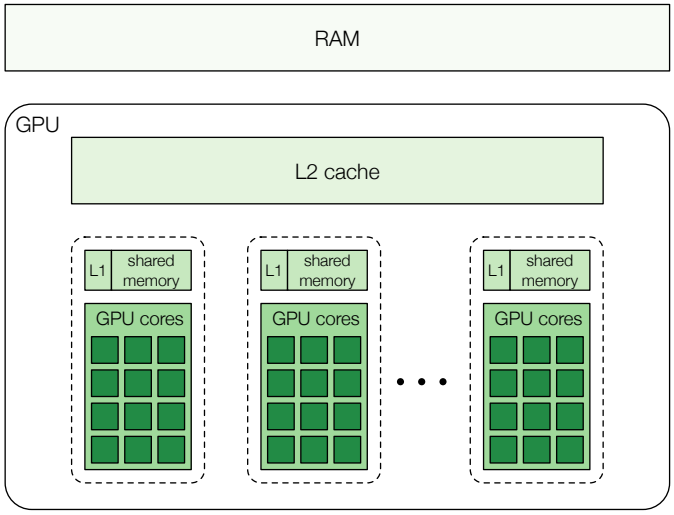


Figure 2.2: Overview of a GPU architecture

and still have a high single-core performance when executing sequential programs [70]. This is why the individual cores have a quite sophisticated design to achieve executing a series of instructions as fast as possible, by applying out-of-order execution and exploiting architectural features like a deep pipeline for decoding and executing instructions combined with advanced branch prediction. Unfortunately, the design of these complex cores makes switching between the threads running on the same core relatively expensive, as the execution pipeline has to be flushed and the register content for the first thread has to be saved before resuming the execution on the second thread. To address this issue, some multi-core architectures feature *Simultaneous Multi-Threading* (SMT), a. k. a., *hyper threading*: a single CPU core can execute multiple (usually 2 or 4) threads in parallel and switching between the threads is cheap. This technique is used to mitigate the effect which memory latency can have on the execution: if a thread is waiting for a memory request then another thread can continue executing. The other central component used to prevent waiting times for memory requests is the cache hierarchy. As we will see, multi-core CPUs feature considerably larger caches than GPUs. The caches help to keep the threads running in parallel on the multi-core CPU busy.

Among the most important optimizations to achieve high performance on modern multi-core CPUs are: exploiting the SIMD vector extensions and optimizing the cache usage [92].

2.1.2 *Graphics Processing Units*

Figure 2.2 shows an abstract representation of a typical GPU architecture, like Nvidia's latest high-performance computing GPU architecture: Kepler [158]. While the overall design looks fairly similar to the

design of the multi-core CPU, the architecture of the GPU is actually very different in its details. The lightweight GPU cores are grouped together into what Nvidia calls a *Streaming Multiprocessor* (SM). In the Kepler architecture, an SM features 192 single-precision cores, 64 double-precision units, 32 special function units, and 32 memory units. The 192 single-precision cores are very lightweight compared to the complex CPU cores we discussed in the previous subsection. A single Nvidia GPU core does not support SIMD vectorization: it performs the straightforward in-order execution, and does not perform complex branch prediction. This design makes the implementation of such a core in hardware cheap and enables the grouping of hundreds of cores in a single SM.

Modern GPUs feature two levels of caches: the first level (L1) is private to each SM and the second level cache (L2) is shared among all SMs. Compared to the cache sizes of the CPU, the caches on the GPU are fairly small. The Kepler architecture features an L1 cache of 48 kilobyte and an L2 cache of 1.5 megabyte. It is important to notice that these caches are shared by a vast amount of threads being executed on the GPU: the 48 kilobyte L1 caches are shared among up to 2048 threads being executed on a particular SM and the L2 cache is shared among up to about 30,000 threads.

Besides caches, each SM on a GPU features also a scratchpad memory, called *shared memory* in Nvidia's GPU architectures. This memory is small (16 or 48 kilobytes) but very fast (comparably to the L1 cache). While the L1 cache is automatically managed by a cache controller, the shared memory is explicitly managed by the programmer.

The design of GPUs is motivated by the overall goal to deliver maximal throughput even if this requires to sacrifice the performance of single threads [70]. This is why the GPU cores are lightweight offering poor sequential performance, but thousands of cores combined together offer a very high computational throughput. Memory latencies are mitigated by a high oversubscription of threads: in the Kepler architecture an SM can manage up to 2048 threads which can be scheduled for execution on the 192 cores. If a thread is waiting for a memory request, another thread continues its execution. This is the reason why caches are fairly small, as they play a less important role as in CPU architectures.

GPU THREAD EXECUTION The execution of threads on a GPU is very different as compared to the execution on a traditional multi-core CPU: multiple threads are grouped by the hardware and executed together. Such a group of threads is called a *warp* by Nvidia. In the Kepler architecture, 32 threads form a warp and an SM selects 4 warps and for each warp 2 independent instructions are selected to be executed in parallel on its single-precision cores, double-precision units, special function units, and memory units.

All threads grouped into a warp perform the same instructions in a lockstep manner. Is is possible that two or more threads follow different execution paths. In this case, all threads sharing a common execution path execute together while the other threads pause their execution. It is beneficial to avoid warps with divergent execution paths, as this reduces or eliminates the amount of threads pausing their execution. Nvidia calls this execution model: *single-instruction, multiple-thread* (SIMT).

Programmers are advised to manually optimize their code, so that all threads in a warp follow the same execution path. Threads in the same warp taking different branches in the code can substantially hurt the performance and should, therefore, be avoided.

GPU MEMORY ACCESSES Accessing the main memory of a GPU, called *global memory*, is an expensive operation. The GPU can optimize accesses when the global memory is accessed in certain fixed patterns. If the threads organized in a warp access a contiguous area of the memory which fits in a cache line, the hardware can *coalesce* the memory access, i. e., perform a single memory request instead of issuing individual requests for every thread. Several similar access patterns exist which are detected by the hardware to coalesce the memory accesses by multiple threads.

This is a major optimization opportunity for many GPU programs, as the available memory bandwidth can only be utilized properly if memory accesses are coalesced. Uncoalesced memory accesses may severely hurt the performance.

GPU SHARED MEMORY The usage of the shared memory featured in each SM can substantially increase performance – when used properly. The programmer is responsible for exploiting this memory by explicitly moving data between the global memory and the shared memory. All threads executing on the same SM have shared access to the shared memory at a high bandwidth and with low latency. When multiple threads need to access the same data item, it is often beneficial to first load the data item into the shared memory by a single thread and perform all successive accesses on the shared memory. This is similar to a cache with the difference that the programmer has to explicitly control its behavior. Shared memory can also be used by a thread to efficiently synchronize and communicate with other threads running on the same SM. On current GPUs, global synchronization between threads running on different SMs is not possible.

Summarizing, current GPU architectures are more sensitive to optimizations than CPU architectures. The most important optimizations on GPUs are: exploit the available parallelism by launching the right amount of threads, ensure that accesses to the global memory are coalesced, use the shared memory to minimize accesses to the global

memory, and avoid divergent branches for threads organized in a warp [151].
In addition to these optimizations, it is important for the programmer to keep in mind that the application exploiting the GPU is still executed on the CPU and typically only offloads computation-intensive tasks to the GPU. An often crucial optimization is to minimize the data transfer between the CPU and the GPU or overlap this transfer with computations on the GPU.

We now discuss the current approaches to program multi-core CPUs and GPUs in detail.

2.2 PROGRAMMING OF MULTI-CORE CPUS AND GPUS

Arguably the most popular programming approach for multi-core CPUs is *multithreading*. Threading libraries exist for almost all popular programming languages, including C [94, 129] and C++ [141]. Threads are conceptionally similar to processes in an operating system, with the major difference that threads share the same address space in memory while processes usually do not. This enables threads to communicate and cooperate efficiently with each other by directly using the same memory. Programmers have great flexibility and control over the exact behavior of the threads and their interaction, including the important questions of when and how many threads to use, how tasks and data are divided across threads, and how threads synchronize and communicate with each other. The disadvantage of this high flexibility is that the burden of developing a correct and efficient program lies almost entirely on the programmer. The interaction of threads executing concurrently can be very complex to understand and reason about, and traditional debugging techniques are often not sufficient as the execution is not deterministic any more.

OpenMP [127] is a popular alternative approach focusing on exploiting parallelism of loops in sequential programs. The focus on loops is based on the observation that loops often iterate over large data sets performing operations on every data item without any dependencies between iterations. Such workloads are traditionally called *embarrassingly parallel* and can be parallelized in a straightforward manner. In OpenMP, the programmer annotates such loops with compiler directives, and a compiler supporting the OpenMP standard automatically generates code for performing the computations of the loop in parallel. Recent versions of the OpenMP standard and the closely related OpenACC [149] standard support execution on GPUs as well. Here the programmer has to additionally specify the data regions which should be copied to and from the GPU prior to and after the computation. OpenMP and OpenACC allow for a fairly easy transition from sequential code to parallel code exploiting multi-core CPUs and GPUs. Performance is often not optimal as current

compilers are not powerful enough to apply important optimizations like cache optimizations, ensuring coalesced memory accesses, and the usage of the local memory.

CUDA [44] and OpenCL [119] are the most popular approaches to program GPU systems. While CUDA can only be used for programming GPUs manufactured by Nvidia, OpenCL is a standard for programming GPUs, multi-core CPUs, and other types of parallel processors regardless of their hardware vendor. All major processor manufactures support the OpenCL standard, including AMD, ARM, IBM, Intel, and Nvidia. We will study the OpenCL programming approach in more detail, as it is the technical foundation of the work presented later in Part II and Part III of this thesis.

2.2.1 *The OpenCL Programming Approach*

OpenCL is a standard for programming multi-core CPUs, GPUs, and other types of parallel processors. OpenCL was created in 2008 and has since been refined multiple times. In this thesis, we will use the OpenCL standard version 1.1 which was ratified in June 2010. This is the most commonly supported version of the standard with support from all major hardware vendors: AMD, ARM, Intel, and Nvidia. For example, are the newer OpenCL standards version 2.0 and 2.1 currently not supported on Nvidia GPUs at the time of writing this thesis.

OpenCL is defined in terms of four theoretical models: platform model, memory model, execution model, and programming model. We will briefly discuss all four models.

THE OPENCL PLATFORM MODEL Figure 2.3 shows the OpenCL platform model. OpenCL distinguishes between a *host* and multiple OpenCL *devices* to which the host is connected. In a system comprising of a multi-core CPU and a GPU, the GPU constitutes an OpenCL device and the multi-core CPU plays the dual rule of the host and an OpenCL device as well. An OpenCL application executes sequentially on the host and offloads parallel computations to the OpenCL devices.

OpenCL specifies that each device is divided into one or more *compute units* (CU) which are again divided into one or more *processing elements* (PE). When we compare this to our discussion of multi-core CPUs, there is a clear relationship: a CPU core corresponds to a compute unit and the functional units inside of the CPU core performing the computations correspond to the processing elements. For the GPU the relationship is as follows: a streaming multiprocessor corresponds to a compute unit and the lightweight GPU cores correspond to the processing elements.

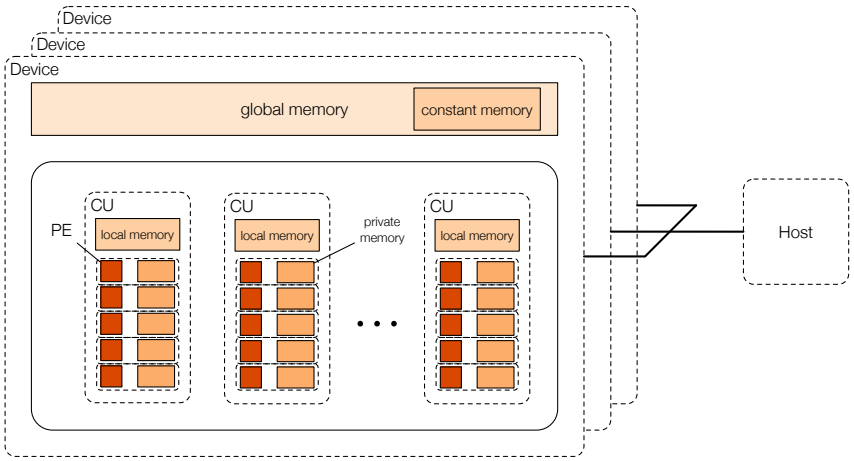


Figure 2.3: The OpenCL platform and memory model

THE OPENCL MEMORY MODEL Each device in OpenCL has its own memory. This memory is logically (but not necessarily physically) distinct from the memory on the host. Data has to be moved explicitly between the host and device memory. The programmer issues commands for copying data from the host to the device and vice versa.

On a device, OpenCL distinguishes four memory regions: global memory, constant memory, local memory, and private memory. These memory regions are shown in Figure 2.3.

The *global memory* is shared by all compute units of the device and permits read and write modifications from the host and device. The global memory is usually the biggest but slowest memory area on an OpenCL device.

The *constant memory* is a region of the global memory which remains constant during device execution. The host can read and write to the constant memory, but the device can only read from it.

The *local memory* is a memory region private to a compute unit. It permits read and write accesses from the device; the host has no access to tis memory region. The local memory directly corresponds to the fast per-SM shared memory of modern GPUs. On multi-core CPUs, the local memory is usually only a logical distinction: local memory is mapped to the same physical memory as the global memory.

The *private memory* is a memory region private to a processing element. The device has full read and write access to it, but the host has no access at all. The private memory contains all variables which are private to a single thread.

THE OPENCL EXECUTION MODEL The communication between the host and a particular device is performed using a *command queue*. The host submits commands into a *command queue* which by-default processes all commands in the first-in-first-out order. It is also pos-

```
1  kernel void matMultKernel(global float* a, global float* b,
2                            global float* c, int width) {
3   int colId = get_global_id(0); int rowId = get_global_id(1);
4
5   float sum = 0.0f;
6   for (int i = 0; i < width; ++i)
7     sum += a[rowId * width + k] * b[k * width + colId];
8
9   c[rowId * width + colId] = sum;
10 }
```

Listing 2.1: Example of an OpenCL kernel.

sible to configure command queues to operate out-of-order, i. e., no order of execution of commands is guaranteed.

There exist three types of commands which can be enqueued in a command queue. A *memory command* indicates to copy data from the host to the device or from the device to the host. A *synchronization command* enforces a certain order of execution of commands. Finally, a *kernel execution command* executes a computation on the device.

A *kernel* is a function which is executed in parallel on an OpenCL device. Listing 2.1 shows a simple OpenCL kernel performing matrix multiplication. When launching a kernel on a device, the programmer explicitly specifies how many threads will execute in parallel on the device. A thread executing a kernel is called a *work-item* in OpenCL. Work-items are grouped together in *work-groups* which allow a more coarse-grained organization of the execution. All work-items of a work-group share the same local memory, and synchronization between work-items of the same work-group is possible but it is forbidden for work-items from different work-groups. This is because all work-items from the same work-group are guaranteed to be executed on the same CU, but this guarantee does not hold for work-items from different work-groups.

OpenCL kernels are implemented in a dialect of C (starting with OpenCL 2.1 a dialect of C++) with certain restrictions as well as extensions. The most prominent restrictions are: recursive functions and function pointers are not supported, as well as system calls, including `malloc`, `printf`, file I/O, etc. The most important extensions are: qualifiers for pointers reflecting the four address spaces (e. g., the `global` qualifier in line 1 and line 2 of Listing 2.1) and vector data types, like `float4`. Moreover, OpenCL provides a set of functions which can be used to identify the executing work-item and work-group. One example is the `get_global_id` function used in line 3 of Listing 2.1. This is usually used so that different work-items operate on different sets of the input data. In Listing 2.1, each work-item computes one item of the result matrix `c` by multiplying and summing up a row and

column of the input matrices `a` and `b`. This kernel only produces the correct result if exactly one work-item per element of the result matrix is launched. If the host program launches a different configuration of work-items then only part of the computation will be performed or out-of-bound memory accesses will occur.

THE OPENCL PROGRAMMING MODEL OpenCL supports two programming models: data parallel programming and task parallel programming. The predominant programming model for OpenCL is the data parallel programming model.

In the *data parallel programming model*, kernel are executed by many work-items organized in multiple work-groups. Parallelism is mainly exploited by the work-items executing a single kernel in parallel. This programming model is well suited for exploiting modern GPUs and, therefore, widely used. This model is also well suited for modern multi-core CPUs.

In the *task parallel programming model*, kernels are executed by a single work-item. Programmers exploit parallelism by launching multiple tasks which are possibly executed concurrently on an OpenCL device. This programming model is not well suited for exploiting the capabilities of modern GPUs and, therefore, not widely used.

2.3 STRUCTURED PARALLEL PROGRAMMING

Structured programming emerged in the 1960s and 1970s as a reaction to the "software crises". Back then (and still today) programs were often poorly designed, hard to maintain, and complicated to reason about. Dijkstra identified the low-level `goto` statement as a major reason for programmers writing "spaghetti code". In his famous letter [53], he argued that programs should organize code more structurally in procedures and using higher-level control structures like `if` and `while`. Dijkstra's letter helped to eventually overcome unstructured sequential programming and to establish structured programming [45]. The new structures proposed to replace `goto` where suggested based on observations of common use cases – or *patterns* of use – of the `goto` statement in sequential programming. By capturing an entire pattern of usage, these patterns raise the abstraction level and make it easier to reason about them. An `if A then B else C` statement has a clear semantic which is easy to understand and reason about, for example is it clear to the programmer – and the compiler – that `B` and `C` cannot both be executed. This helps the programmer to understand the source code and the compiler to produce an optimized executable. The equivalent unstructured code containing multiple `goto` statements obtains the same high-level semantic by combining the low-level operations with their low-level semantics. Both, programmer and compiler, have to "figure out" the high-level

semantic by means of analyzing the sequence of low-level operations and reconstructing the overall semantic.

In recent years researchers have suggested similar arguments to Dijkstra's for addressing the challenges attached with traditional parallel programming by introducing *structured parallel programing*. For parallel programming using message passing, Gorlatch argues similar to Dijkstra that single send and receive statements should be avoided and replaced by collective operations [74]. Each collective operation, e. g., broadcast, scatter, or reduce, captures a certain common communication pattern traditionally implemented with individual send and receive statements. In [113], the authors argue that *structured parallel patterns*, each capturing a common computation and communication behavior, should be used to replace explicit thread programming to improve the maintainability of software. The book discusses a set of parallel patterns and their implementation in different recent programming approaches. The underlying ideas go far back to the 1980s when Cole was the first to introduced *algorithmic skeletons* to structure and, therefore, simplify parallel programs [36]. As with sequential structured programming, structured parallel programming raises the abstraction level by providing high-level constructs: collective operations, parallel patterns, or algorithmic skeletons. The higher level of abstraction both simplifies the reasoning about the code for the programmer and enables higher-level compiler optimizations.

As our work directly extends the ideas of Cole, we will discuss algorithmic skeletons in more detail next.

2.3.1 *Algorithmic Skeletons*

Cole introduced algorithmic skeletons as special higher-order functions which describe the "computational skeleton" of a parallel algorithm. Higher-order functions are a well-known concept in functional programming and describe functions accepting other functions as arguments or returning a function as result. This is often useful, as it allows to write more abstract and generic functions.

An example for an algorithmic skeletons is the *divide & conquer skeleton* which was among the original suggested skeletons by Cole [36]:

$$\mathrm{D_C}\ \mathit{indivisible}\ \mathit{split}\ \mathit{join}\ \mathit{f}$$

The algorithmic skeleton $\mathrm{D_C}$ accepts four functions as its arguments:

- *indivisible* is a function deciding if the given problem should be decomposed (divided) or not,
- *split* is a function decomposing a given problem into multiple sub-problems,

- `join` is a function combining multiple solved sub-problems into a larger solution,
- `f` is a function solving an indivisible problem, i. e., the base case.

Applications like the discrete Fourier transformation, approximate integration, or matrix multiplication can be expressed and implemented using this algorithmic skeleton. The application developer provides implementations for the functions required by the algorithmic skeleton to obtain a program which can be applied to the input data.

An algorithmic skeleton has a parallel implementation. In the example of the D_C skeleton, the implementation follows the well-known divide & conquer technique which divides problems into multiple sub-problems which can be solved independently in parallel. The implementation of the algorithmic skeleton hides the complexities of parallelism from the user. It, therefore, provides a higher-level interface abstracting away the details of the parallel execution, including low-level details like launching multiple threads, as well as synchronization and communication of threads.

A CLASSIFICATION OF ALGORITHMIC SKELETONS Algorithmic skeletons can broadly be classified into three distinct classes [72]:

- *data-parallel skeletons* transform typically large amounts of data,
- *task-parallel skeletons* operate on distinct tasks which potentially interact with each other,
- *resolution skeletons* capture a family of related problems.

Examples of data-parallel skeletons are *map* which applies a given function to each element of its input data in parallel, or *reduce* which performs a parallel reduction based on a given binary operator. Well known task-parallel skeletons are *farm (a. k. a., master-worker)* where independent tasks are scheduled for parallel execution by the workers, or *pipeline* where multiple stages are connected sequentially and the execution of all stages can overlap to exploit parallelism. Finally, the discussed D_C skeleton is an example of a resolution skeleton which captures the family of problems which can be solved by applying the divide & conquer technique.

In this thesis, we will mainly focus on data-parallel skeletons, as they are especially suitable for the data-parallel GPU architecture. We will also introduce two new resolution skeletons which capture two specific application domains for which we can provide efficient GPU implementations as well.

2.3.2 *Advantages of Structured Parallel Programming*

Structured parallel programming offers various advantages over the traditional unstructured parallel programming.

SIMPLICITY Structured parallel programming raises the level of abstraction by providing higher-level constructs which serve as basic building blocks for the programmer. Lower-level details are hidden from the programmer and handled internally by the implementation of the algorithmic skeletons. This simplifies the reasoning about programs and helps in the development process, as well as increases the maintainability of the software.

SAFETY AND DETERMINISM Potentially dangerous low-level operations are not used directly by the programmer in structured parallel programming. Therefore, issues like deadlocks and race conditions can be entirely avoided – given a correct and safe implementation of the provided algorithmic skeletons. Furthermore, the high-level semantic of the algorithmic skeletons can guarantee determinism, while the lacking of determinism due to the parallel execution is a major concern in low-level, unstructured parallel programming; this concern complicates development and debugging of the software.

PORTABILITY Algorithmic skeletons offer a single high-level interface but can be implemented in various ways on different hardware systems. Existing skeleton libraries target distributed systems [3, 103, 112], shared memory systems like multi-core CPUs [5, 35, 108], and – as we will discuss in this thesis – systems with multiple GPUs as well. In contrary, unstructured parallel programming commits to a low-level programming approach targeting a particular hardware architecture, thus, making portability a major issue. Furthermore, algorithmic skeletons evolved from functional programming and are, therefore, by-nature composabel and offer a high degree of re-use.

Another issue of portability is the portability of performance: can a certain level of performance be obtained when switching from one hardware architecture to another? This is virtually absent from low-level unstructured parallel programming as programmers apply low-level hardware-specific optimizations to achieve high performance. These optimizations are usually not portable, as we will show later in Chapter 5. Performance portability is a challenging and timely research topic which we address in this thesis by introducing a novel approach using structured parallel programming to achieve performance portability.

PREDICTABILITY The structure and regularity of structured parallel programming allows to build performance models which can be used in the development process to estimate the performance of the developed software. Many research projects are devoted to this topic, showing that it is possible to predict the runtime of programs expressed with algorithmic skeletons [8, 19, 47, 84, 142]. Examples for related work in this area include work on particular algorithmic skeletons [19], work targeted towards distributed and grid systems [8], and recently work targeting real-time systems [142].

PERFORMANCE AND OPTIMIZATIONS Many studies have shown that structured parallel programs can offer the same level of performance as programs implemented and optimized with traditional unstructured techniques. Examples include application studies on grid systems [8], distributed systems [34], as well as systems featuring multi-core CPUs [6]. In this thesis, we will investigate the performance of programs expressed with data-parallel algorithmic skeletons on systems with one or multiple GPUs.

The high-level semantic of algorithmic skeletons enable high-level optimizations like the rewrite rules presented in [75]. These optimizations exploit information about the algorithmic structure of a program which is often hard or impossible to extract from unstructured programs. In this thesis, we will present a system for encoding and systematically applying such high-level optimizations to generate highly optimized code from a high-level, skeleton-based program representation.

2.4 SUMMARY

In this chapter we have discussed the technical background of this thesis. We first introduced the design of modern parallel processors and discussed the differences between multi-core CPUs and GPUs. Then, we looked at how these processors are currently programmed identifying some problems of parallel programming, including nondeterminism, race conditions, and deadlocks. We particular looked at the OpenCL programming approach and how it can be used for programming multi-core CPUs and GPUs. Finally, we introduced structured parallel programming as an alternative approach which avoids many drawbacks of traditional parallel programming techniques.

In the next part of the thesis we introduce a novel structured parallel programming model for single- and multi-GPU systems addressing the *programmability* challenge.

Part II

THE SKELCL HIGH-LEVEL PROGRAMMING MODEL

3 HIGH-LEVEL PROGRAMMING FOR MULTI-GPU SYSTEMS

In this chapter we address the first main challenge identified in Chapter 1: *Programmability* of modern parallel systems. We will see how structured parallel programming significantly simplifies the task of programming for parallel systems. We will focus on programming of single- and multi-GPU systems throughout the thesis, but the observations made here are more generic and also valid when programming other parallel systems.

We will first motivate the need for high-level abstractions using a real-world OpenCL application from the field of medical imaging. Then we introduce the *SkelCL* programming model and its implementation as a C++ library which addresses the lack of high-level abstractions in state-of-the-art GPU programming models. The following Chapter 4 will provide several application studies to thoroughly evaluate the usefulness and performance of the abstractions and implementation presented in this chapter.

3.1 THE NEED FOR HIGH-LEVEL ABSTRACTIONS

We start the discussion of programming for parallel systems and in particular for GPU systems by looking thoroughly at an application example. By doing so we will identify challenges faced by application developers which arise from typical characteristics of parallel hardware architectures. We will then derive from these challenges requirements for a potential high-level programming model.

3.1.1 *Challenges of GPU Programming*

We choose to investigate a real-world application rather than a simple benchmark, in order to identify not only fundamental challenges every application developer targeting GPU systems faces, but also practical programming challenges which become only visible for more complex applications, e. g., managing the execution of multiple compute kernels.

Our example application is the LM OSEM algorithm [133, 137] for image reconstruction used in Positron Emission Tomography (PET). In PET, a radioactive substance is injected into a human or animal body, which is then placed inside a PET scanner that contains several arrays of detectors. As the particles of the applied substance de-

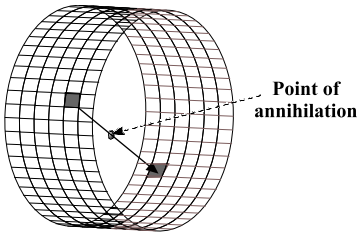


Figure 3.1: Two detectors register an event in a PET-scanner

cay, positrons are emitted (hence the name PET) and annihilate with nearby electrons, such that two photons are emitted in the opposite directions (see Figure 3.1). These "decay events" are registered by two opposite detectors of the scanner which records these events. Data collected by the PET scanner are then processed by a reconstruction algorithm to obtain a resulting image.

3.1.1.1 *The LM OSEM Algorithm*

List-Mode Ordered Subset Expectation Maximization [133] (called LM OSEM in the sequel) is a block-iterative algorithm for 3D image reconstruction. LM OSEM takes a set of events from a PET scanner and splits them into s equally sized subsets. Then, for each subset $S_l, l \in 0, \ldots, s-1$, the following computation is performed:

$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A_N^T \mathbf{1}} \sum_{i \in S_l} (A_i)^T \frac{1}{A_i f_l}. \tag{3.1}$$

Here $f \in \mathbb{R}^n$ is a 3D image in vector form with dimensions $n = (X \times Y \times Z)$, A it the so-called system matrix, element a_{ik} of row A_i is the length of intersection of the line between the two detectors of event i with voxel k of the reconstruction image, computed with Siddon's algorithm [139]. As storing the entire system matrix A is impossible due to memory constrains each row A_i is computed independently as it is needed. $1/A_N^T \mathbf{1}$ is the so-called normalization vector; since it can be precomputed, we will omit it in the following. The multiplication $f_l c_l$ is performed element-wise. Each subset's computation takes its predecessor's output image as input and produces a new, more precise image.

The structure of a sequential LM OSEM implementation is shown in Listing 3.1. The outermost loop iterates over the subsets. The first inner loop (step 1, line 7—line 11) iterates over subset's events to compute c_l, which requires three sub-steps: row A_i is computed from the current event using Siddon's algorithm; the local error for row A_i is computed and, finally, added to c_l. The second inner loop (step 2, line 14—line 16) iterates over all elements of f_l and c_l to compute f_{l+1}.

```
1  // input: subsets of recorded events
2  // output: image estimate f
3  for (int l = 0; l < subsets; l++) {
4    // read subset S
5
6    // step 1: compute error image c_l
7    for (int i = 0; i < subset_size; i++) {
8      // compute A_i from subset S
9      // compute local error
10     // add local error to c_l
11   }
12
13   // step 2: update image estimate f
14   for (int k = 0 ; k < image_size; k++) {
15     if (c_l[k] > 0.0) { f[k] = f[k] * c_l[k]; }
16   }
17 }
```

Listing 3.1: Sequential code for LM OSEM comprises one outer loop with two nested inner loops.

3.1.1.2 *Parallelization of LM OSEM in OpenCL*

LM OSEM is a rather time-consuming algorithm that needs parallelization: a typical 3D image reconstruction processing $6 \cdot 10^7$ input events for a $150 \times 150 \times 280$ voxel PET image takes more than two hours when executed sequentially on a modern PC.

The iterations of the outer loop in Listing 3.1 are inherently sequential, as in each iteration the image estimate computed by the previous iteration is refined. Within one iteration we can parallelize the two calculation steps across multiple GPUs as shown in Figure 3.2 for a system comprising two GPUs. Note that the two steps require different data distribution patterns:

> *Step 1:* Subset's events (S) are copied from the CPU to all GPUs (*upload*) to compute the summation parts of c_l concurrently. This step requires that the complete image estimate f_l is available on all GPUs.
>
> *Step 2:* For computing the next image estimate f_{l+1} in parallel, the current image estimate f_l and the error image c_l computed in step 1 have to be distributed in disjoint parts (blocks) among all GPUs.

Thus, the parallelization schema in Figure 3.2 requires a *data redistribution* phase between the two computation steps. During step 1, each GPU computes a partial sum of c_l which are then summed up and redistributed disjoinlty to all GPUs after step 1. Note that for

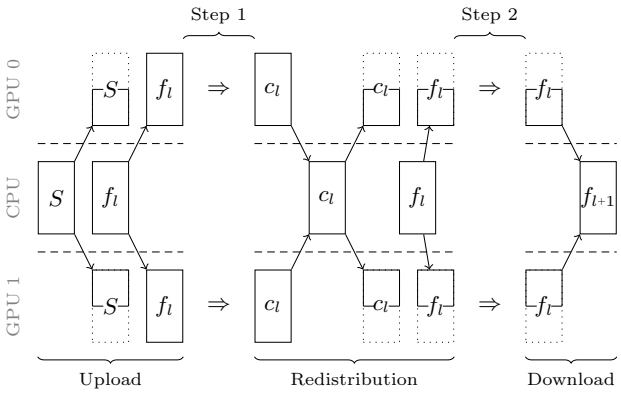


Figure 3.2: Parallelization schema of the LM OSEM algorithm.

step 1, each GPU requires a full copy of the image estimate, while in step 2 all GPUs update disjoint parts of it. After step 2, the disjoint parts of the image estimate are copied from all GPUs back to the CPU (*download*).

In the following, we describe how the phases in the parallelization schema in Figure 3.2 are implemented using OpenCL.

UPLOAD Listing 3.2 shows a simplified OpenCL implementation of the upload phase. Uploading of the event vector S is performed in line 3—line 7, while line 10—line 14 upload the image estimate f_l. In OpenCL, we have to manage each GPU explicitly, therefore, for each GPU, we manage an array of buffers (`s_gpu` in line 4 and `f_gpu` in line 11) and we use a loop (line 1) to repeat all memory operations for each GPU. For performance reasons, we use asynchronous copy operations, specified using the `CL_FALSE` flag (line 4 and line 11): this allows data transfers to multiple GPUs to overlap. We perform different operations with S (distribute among all GPUs) and f_l (copy to each GPU), therefore, there are differences when specifying the amount of bytes to copy (line 5 and line 12) and the offsets in the CPU memory (line 6 and line 13). Altogether eleven such memory operations – each with different amounts of bytes and offsets – appear in the OpenCL source code.

STEP 1 The implementation of step 1 performs the three sub-steps shown in Listing 3.1. Because of memory restrictions on the GPU, the OpenCL implementation is not straightforward, therefore, we will not show the detailed code here. An OpenCL kernel is launched where each work-item processes multiple events one after another. For each event first A_i is computed, then the local error for A_i is computed using the current image estimate f_l which is finally added to c_l.

In Equation (3.1) A_i represents the *path* of a detected line through the 3D image space. In mathematics it is convenient to think of this

```
1  for (int gpu = 0; gpu < gpu_count; gpu++) {
2    // upload S
3    clEnqueueWriteBuffer( command_queue[gpu],
4          s_gpu[gpu], CL_FALSE, 0,
5          sizeof(float) * size_of_s / gpu_count,
6          (void*)&s_cpu[ gpu * size_of_s / gpu_count ],
7          0, NULL, NULL );
8
9    // upload f_l
10   clEnqueueWriteBuffer( command_queue[gpu],
11         f_gpu[gpu], CL_FALSE, 0,
12         sizeof(float) * size_of_f,
13         (void*)&f_cpu[0],
14         0, NULL, NULL );
15 }
```

Listing 3.2: Implementation of the upload phase in OpenCL (omitting error checks for brevity).

as a sparse vector containing the length of the intersection of the line with a given voxel in the image space. As most voxels are not intersected by the line, most entries in the vector remain zero. Therefore, in the OpenCL code A_i is represented as an array of pairs, where the first entry of each pair is the index of a voxel in the image space and the second entry is the length of the intersection with this voxel.

Synchronization between work-items in OpenCL is restricted, i. e., synchronization is only allowed between work-items organized in the same work-group. Therefore, it is not possible to efficiently protect the writing operation to c_l to avoid race conditions. We have conducted studies and found that for this particular algorithm these race conditions are acceptable as they do not substantially decrease the numerical accuracy of the computed result [137].

REDISTRIBUTION Listing 3.3 shows an OpenCL pseudocode for the redistribution phase. To download the data from all GPUs, we use the `clEnqueueReadBuffer` function (line 4) and perform the operations asynchronously, but this time, we have to wait for the operations to finish. For such synchronizations, OpenCL uses *events*, associated with an operation (line 4) for waiting for the operation to finish (line 6). After all downloads have finished, we combine the different values of c_l to a new value of c_l on the CPU (line 9), and upload the blocks of c_l to the GPUs. Even if we only copied data between GPUs, without processing them on the CPU, we still would have to download them to the CPU because direct GPU-to-GPU transfers are currently not possible in any version of OpenCL.

```
1  // download all c_l values from the GPUs to the CPU
2  cl_event events[gpu_count];
3  for (int gpu = 0; gpu < gpu_count; gpu++) {
4    clEnqueueReadBuffer( ..., &events[gpu] );
5  }
6  clWaitForEvents(gpu_count, events);
7
8  // combine data on CPU
9  combine( ... );
10
11 // upload block of the new c_l version to each GPU
12 for (int gpu = 0; gpu < gpu_count; gpu++) {
13   clEnqueueWriteBuffer( ... );
14 }
```

Listing 3.3: OpenCL pseudocode for the redistribution phase

STEP 2 Listing 3.4 shows the implementation of step 2. Computations are specified as *kernels* in OpenCL which are created from the source code specifying the computation. The computation in step 2 is, therefore, described as a string in line 3—line 7. The operations used here are the same as in the sequential code.

For executing the computations of step 2, we have to perform the following steps for each GPU:

- create an OpenCL kernel from the source code (requires 50 lines of code in OpenCL);
- compile the kernel specifically for the GPU (requires 13 lines of code in OpenCL);
- specify kernel arguments one-by-one using the `clSetKernelArg` function (line 14—line 24);
- specify execution environment, i. e., how many instances of the kernel to start (line 27—line 29);
- launch the kernel (line 31—line 34).

DOWNLOAD The implementation of the download phase is similar to the upload phase as shown in Listing 3.2.

```
// step 2 (in Figure 3.2)
source_code_step_2 =
  "kernel void step2(global float* f, global float* c_l,
                     int offset, int size) {
    int id = get_global_id(0) + offset;
    if (id < size && c_l[id] > 0.0){ f[id] = f[id]*c_l[id];}
   }";

for (int gpu = 0; gpu < gpu_count; gpu++) {
  // create kernel (50 lines of code)
  // compile kernel (13 lines of code)

  // specifying kernel arguments:
  clSetKernelArg(kernel_step2[gpu], 0, sizeof(cl_mem),
                 (void*)&f_buffer[gpu]);
  clSetKernelArg(kernel_step2[gpu], 1, sizeof(cl_mem),
                 (void*)&c_l_buffer[gpu]);
  int offset = gpu * (size_of_f / gpu_count);
  clSetKernelArg(kernel_step2[gpu], 2, sizeof(int),
                 (void*)&offset);
  int size   = MIN( (gpu + 1) * (size_of_f / gpu_count),
                    size_of_f );
  clSetKernelArg(kernel_step2[gpu], 3, sizeof(int),
                 (void*)&size);

  // specify execution environment
  int local_work_size[1]  = { 32 };
  int global_work_size[1] =
        { roundUp(32, size_of_f / gpu_count) };
  // launch the kernel
  clEnqueueNDRangeKernel(
        command_queue[gpu], kernel_step2[gpu],
        1, NULL, &global_work_size, &local_work_size, 0,
        NULL, NULL); }
```

Listing 3.4: Implementation of step 2 in OpenCL (omitting error checks for brevity).

3.1.2 *Requirements for a High-Level Programming Model*

The described implementation of the example application reveals the main problems and challenges that application developers have to overcome when targeting GPU systems. Our analysis shows that to simplify programming for a system with multiple GPUs, at least the following three high-level abstraction are desirable:

PARALLEL CONTAINER DATA TYPES Compute-intensive applications typically operate on a (possibly big) set of data items. As shown in Listing 3.2, managing memory is error-prone because low-level details, like offset calculations, have to be programmed manually.

A high-level programming model should be able to make collections of data automatically accessible to all processors in a system and it should provide an easy-to-use interface for the application developer.

RECURRING PATTERNS OF PARALLELISM While each application performs (of course) different concrete operations, the general structure of parallelization often resembles parallel patterns that are commonly used in many applications. In step 1, for computing the error image c_l, the same sequence of operations is performed for every event from the input subset, which is the well-known *map* pattern of data-parallel programming [76]. In step 2, two images (the current image estimate f_l and the error image c_l) are combined element-wise into the output image (f_{l+1}), see line 6 of Listing 3.4, which is again a well-known pattern of parallelism commonly called *zip*.

It would be, therefore, desirable to express the high-level structure of an application using pre-defined common patterns, rather than describing the parallelism manually in much detail.

DISTRIBUTION AND REDISTRIBUTION MECHANISMS To achieve scalability of applications on systems comprising multiple GPUs, it is crucial to decide how the application's data are distributed across all available GPUs. Distributing and re-distributing data in OpenCL is cumbersome because data transfers have to be managed manually and performed via the CPU, as shown in Listing 3.2 and Listing 3.3.

Therefore, it is important for a high-level programming model to allow both for describing the data distribution and for changing the distribution at runtime.

3.2 THE SKELCL PROGRAMMING MODEL

SkelCL (Skeleton Computing Language) is a high-level programming model targeting multi-GPU systems. It is developed as an extension of the standard OpenCL programming model [119].

SkelCL adds three main high-level features to OpenCL which we identified as desirable in Section 3.1.2:

- *parallel container data types* for unified memory management between CPU and (multiple) GPUs;
- *recurring patterns of parallelism* (a. k. a. *algorithmic skeletons*) for easily expressing parallel computation patterns;
- *data distribution* and *redistribution* mechanisms for transparent data transfers in multi-GPU systems.

SkelCL inherits all advantageous properties of OpenCL, including its portability across different heterogeneous parallel systems. SkelCL is designed to be fully compatible with OpenCL: arbitrary parts of a SkelCL code can be written or rewritten in OpenCL, without influencing program's correctness.
In the remainder of this section we discuss the design of the SkelCL programming model in more detail. Its implementation as a C++ library will be discussed in the following section.

3.2.1 *Parallel Container Data Types*

SkelCL offers the application developer two container classes – vector and matrix – which are transparently accessible by both, CPU and GPUs (or using OpenCL's terminology, host and devices). The *vector* abstracts a one-dimensional contiguous memory area while the *matrix* provides an interface to a two-dimensional memory area. The SkelCL container data types have two major advantages as compared with OpenCL.

The first advantage is its automatic memory management. When a container is created on the host, memory is allocated on the devices automatically; when a container on the host is deleted, the memory allocated on the devices is freed automatically. In OpenCL memory has to be allocated and freed manually.

The second advantage is that the necessary data transfers between the host and devices are performed automatically and implicitly. Before performing a computation on container types, the SkelCL system ensures that all input containers' data is available on all participating devices. This may result in implicit data transfers from the host to device memory, which in OpenCL requires explicit programming, as we saw in Listing 3.2 in Section 3.1.1. Similarly, before any data is accessed on the host, the implementation of SkelCL implicitly ensures

that this data on the host is up-to-date by performing necessary data transfers automatically. Using the `clEnqueueWriteBuffer` (see line 3—line 7 in Listing 3.2) and `clEnqueueReadBuffer` (see Listing 3.3) functions nine arguments have to be specified to perform a single data transfer in OpenCL. The SkelCL container classes shield the programmer from these low-level operations like memory allocation (on the devices) and data transfers between host and device.

Developing applications working with two-dimensional data for modern parallel architectures is cumbersome and challenging, since efficient memory handling is essential for high performance. In case of GPUs, it is key for achieving good performance to exploit the memory hierarchy by using the fast but small on-chip memory. Therefore, in addition to the vector as a one-dimensional abstract data structure, SkelCL offers a specific abstract data type for handling two-dimensional data, the matrix. We will see later in Section 3.2.4.4 and Section 3.3 how SkelCL automatically exploits the memory hierarchy to improve performance.

3.2.2 *Algorithmic Skeletons*

To shield the application developer from the low-level programming issues discussed in the previous section, SkelCL extends OpenCL by introducing high-level programming patterns, called *algorithmic skeletons* [36]. Formally, a skeleton is a higher-order function that executes one or more user-defined (so-called *customizing*) functions in a predefined parallel manner hiding the details of parallelism and communication from the user [76].

SkelCL provides four basic data-parallel skeletons: *map*, *zip*, *reduce*, and *scan*, as well as two more advanced skeletons targeting specific application domains: *stencil* and *allpairs*. In this section we will look at the basic skeletons, the advanced skeletons will be discussed in Section 3.2.4. The four basic skeletons have been selected, because they have been proven to be useful for a broad range of applications. Moreover, these skeletons can be efficiently implemented on GPUs as their computation patterns match the data-parallel execution model implemented by GPUs.

NOTATION We will use a notation strongly inspired by the Bird-Meertens formalism [18]. Function application is written with a space between the function name and the argument, i. e., $f\ x$. We use parenthesis solely for enforcing precedence or for structuring complex expressions to make them easier to understand. Function application is left associative and functions are curried, i. e., $f\ x\ y$ means $(f\ x)\ y$ and not $f\ (x\ y)$. For sequential function composition we use the $\circ$ operator, i. e., $(f \circ g)\ x = f\ (g\ x)$. Function application binds stronger than any other operation, e. g., $f\ x \circ g\ y = (f\ x) \circ (g\ y)$.

We use various symbols, e. g., $\oplus$ and $\otimes$ as binary operators. We write these operators using infix notation, i. e., $x \oplus y$. Using parenthesis we can *section* a binary operator which we write using prefix notation:

$$(\oplus)\ x\ y = x \oplus y$$
$$(x \oplus)\ y = x \oplus y$$
$$(\oplus\ y)\ x = x \oplus y$$

This is a helpful notation to treat a binary operator syntactically like an ordinary function.

As our data structures we use vectors written as $\vec{x} = [x_1, x_2, \ldots, x_n]$ and matrices written as:

$$M = \begin{bmatrix} m_{1,1} & \cdots & m_{1,m} \\ \vdots & & \vdots \\ m_{n,1} & \cdots & m_{n,m} \end{bmatrix}.$$

THE MAP SKELETON The *map* skeleton is a well-known basic algorithmic skeleton, applying the customizing function to each element of a container in parallel. This skeleton originates from the functional programming world, where the `map` function is recognized as an important primitive for writing high-level code. In many programming languages an equivalent sequential function exists, either known under the same name, like in Haskell or Python, or by other names, like `transform` in C++.

In SkelCL, the *map* skeleton can operate on vectors as well as matrices. We start by formally defining the skeleton on vectors:

DEFINITION 3.1. *Let $\vec{x}$ be a vector of size n with elements x_i where $0 < i \leqslant n$. Let f be a unary customizing function defined on elements. The algorithmic skeleton map is then defined as follows:*

$$map\ f\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [f\ x_1, f\ x_2, \ldots, f\ x_n].$$

The definition for matrices is similar:

DEFINITION 3.2. *Let M be an $n \times m$ matrix with elements $m_{i,j}$ where $0 < i \leqslant n$ and $0 < j \leqslant m$. Let f be a unary customizing function. The algorithmic skeleton map is defined as follows:*

$$map\ f \begin{bmatrix} m_{1,1} & \cdots & m_{1,m} \\ \vdots & & \vdots \\ m_{n,1} & \cdots & m_{n,m} \end{bmatrix} \stackrel{def}{=} \begin{bmatrix} f\ m_{1,1} & \cdots & f\ m_{1,m} \\ \vdots & & \vdots \\ f\ m_{n,1} & \cdots & f\ m_{n,m} \end{bmatrix}.$$

The output container of the *map* skeleton, either vector or matrix, can be computed in parallel, because the computation of each single element is independent of each other.

A simple possible application of the *map* skeleton is negating all the values in a vector:

$$\mathtt{neg}\ \vec{x} = \mathit{map}\ (-)\ \vec{x}$$

THE ZIP SKELETON The *zip* skeleton operates on two containers and combines them into one. As the *map* skeleton it is defined for vectors and matrices as well.

DEFINITION 3.3. *Let $\vec{x}$ and $\vec{y}$ be vectors of size n with elements x_i and y_i where $0 < i \leqslant n$. Let $\oplus$ be a binary customizing operator. The algorithmic skeleton zip is defined as follows:*

$$\begin{aligned} &\mathit{zip}\ (\oplus)\ [x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] \\ &\quad \stackrel{def}{=} [x_1 \oplus y_1, x_2 \oplus y_2, \ldots, x_n \oplus y_n]. \end{aligned}$$

Again the definition for matrices is similar:

DEFINITION 3.4. *Let M and N be $n \times m$ matrices with elements $m_{i,j}$ and $n_{i,j}$ where $0 < i \leqslant n$ and $0 < j \leqslant m$. Let $\oplus$ be a binary customizing operator. The algorithmic skeleton zip is defined as follows:*

$$\begin{aligned} &\mathit{zip}\ (\oplus) \begin{bmatrix} m_{1,1} & \cdots & m_{1,m} \\ \vdots & & \vdots \\ m_{n,1} & \cdots & m_{n,m} \end{bmatrix} \begin{bmatrix} n_{1,1} & \cdots & n_{1,m} \\ \vdots & & \vdots \\ n_{n,1} & \cdots & n_{n,m} \end{bmatrix} \\ &\quad \stackrel{def}{=} \begin{bmatrix} m_{1,1} \oplus n_{1,1} & \cdots & m_{1,m} \oplus n_{1,m} \\ \vdots & & \vdots \\ m_{n,1} \oplus n_{n,1} & \cdots & m_{n,m} \oplus n_{n,m} \end{bmatrix}. \end{aligned}$$

This definitions require the two input containers to be of exactly the same size. The *zip* skeleton is parallelizeable in the same manner as *map*, as each element of the output container can be computed in parallel.

A possible application of the *zip* skeleton is performing pairwise addition of two vectors:

$$\mathtt{add}\ \vec{x}\ \vec{y} = \mathit{zip}\ (+)\ \vec{x}\ \vec{y}$$

THE REDUCE SKELETON The *reduce* skeleton computes a single value from a vector by successively applying the binary customizing function. In SkelCL, the *reduce* skeleton is only defined on vectors:

DEFINITION 3.5. *Let $\vec{x}$ be a vector of size n with elements x_i where $0 < i \leqslant n$. Let $\oplus$ be an associative and commutative binary customizing operator with the corresponding identity element $\oplus_{id}$. The algorithmic skeleton reduce is defined as follows:*

$$\mathit{reduce}\ (\oplus)\ \oplus_{id}\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

Requiring the operator to be associative and commutative enables efficient parallel implementations, which we will discuss in Section 3.3. The identity element $\oplus_{id}$ can be used by the implementation, e. g., to initialize intermediate variables or buffers.

A possible application of the *reduce* skeleton is to finding the maximum value of a vector:

$$\mathtt{maxValue}\ \vec{x} = \mathit{reduce}\ \mathtt{max}\ 0\ \vec{x}$$

$$\text{where:} \qquad \mathtt{max}\ a\ b = \begin{cases} a & \text{if } a \geqslant b \\ b & \text{if } a < b \end{cases}$$

THE *scan* SKELETON The *scan* skeleton (a. k. a., prefix-sum) yields an output vector with each element obtained by applying the customizing function to the elements of the input vector up to the current element's index. In SkelCL, the *scan* skeleton is only defined on vectors:

DEFINITION 3.6. *Let $\vec{x}$ be a vector of size n with elements x_i where $0 < i \leqslant n$. Let $\oplus$ be an associative binary customizing operator with the corresponding identity element $\oplus_{id}$. The algorithmic skeleton scan is defined as follows:*

$$\begin{aligned} &\mathit{scan}\ (\oplus)\ \oplus_{id}\ [x_1, x_2, \ldots, x_n] \\ &\quad \stackrel{def}{=} [\oplus_{id},\ x_1,\ x_1 \oplus x_2,\ \ldots\ ,\ x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}]. \end{aligned}$$

Even though the *scan* pattern seems inherently sequential, because each individual result contains the results of its predecessor, efficient parallel implementations do exist for this problem. Blelloch [21] studies this parallel pattern in great detail and efficient implementations for GPUs exist [83] following his algorithmic ideas.

A possible application of the *scan* skeleton is the computation of the prefix sum which can be used as part of the counting sort algorithm [101] or for solving the list ranking problem [38].

$$\mathtt{prefixSum}\ \vec{x} = \mathit{scan}\ (+)\ 0\ \vec{x}$$

3.2.2.1 *Parallel Programming with Algorithmic Skeletons*

In SkelCL, rather than writing low-level kernels, the application developer customizes suitable skeletons by providing application-specific functions which are often much simpler than kernels as they specify an operation on basic data items rather than containers.

Skeletons can be customized and composed to express complex algorithms. To demonstrate how to express computations with algorithmic skeletons let us consider three simple linear algebra applications: scaling a vector with a constant, computing the sum of absolute values of a vector, and computing the dot product (a. k. a., inner product)

of two vectors. These three applications are all part of the well-known *Basic Linear Algebra Subprograms* (BLAS) [55, 56] library.

For scaling a vector with a constant α we use the *map* skeleton:

$$\begin{aligned} & scal\ \alpha\ \vec{x} = map\ (f\ \alpha)\ \vec{x} \\ \text{where:} \quad & f\ \alpha\ x = \alpha \times x \end{aligned}$$

We use currying here to bind α to f, thus, producing a new unary function used to customize *map*.

For computing the sum of absolute values we combine a *map* and *reduce* skeleton.

$$\begin{aligned} & asum\ \vec{x} = reduce\ (+)\ 0\ (\ map\ (|\,.\,|)\ \vec{x}\ ) \\ \text{where:} \quad & |a| = \begin{cases} a & \text{if } a \geqslant 0 \\ -a & \text{if } a < 0 \end{cases} \end{aligned}$$

This definition can also be expressed in a more compositional style, also known as *point-free style*, without mentioning the input vector $\vec{x}$ and by using the $\circ$ operator:

$$asum = reduce\ (+)\ 0\ \circ\ map\ (|\,.\,|)$$

To compute the dot product of two vectors we compose a *zip* skeleton customized with multiplication and a *reduce* skeleton customized with addition:

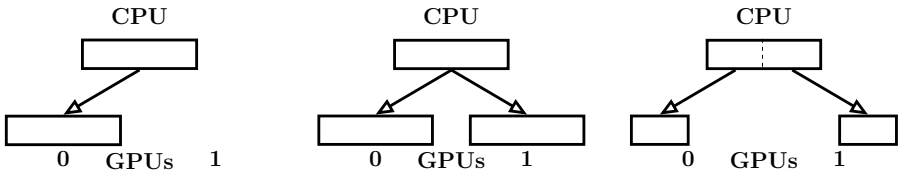
$$dot\ \vec{x}\ \vec{y} = reduce\ (+)\ 0\ (\ zip\ (\times)\ \vec{x}\ \vec{y}\ )$$

Algorithmic skeletons are not limited to linear algebra applications, but can be used to implement a boarded range of application types as we will discuss in Chapter 4. Among others, we will see an algorithmic skeleton based implementation of the real-world medial imaging application example discussed at the beginning of this chapter.

3.2.3 *Data Distribution and Redistribution*

For multi-device systems, SkelCL's parallel container data types (vector and matrix) abstract from the separate memory areas on multiple OpenCL devices, i. e., container's data is accessible by all devices. Each device may access different parts of a container or may even not access it at all. For example, when implementing work-sharing on multiple GPUs, the GPUs will access disjoint parts of input data, such that copying only a part of the vector to a GPU is more efficient than copying the whole data to each GPU.

To simplify the partitioning of a container on multiple devices, SkelCL introduces the concept of *distributions* which describe how the container's data is distributed among the available devices. It allows



(a) *single* (b) *copy* (c) *block*

Figure 3.3: Distributions of a vector in SkelCL.

the application developer to abstract from the challenges of managing memory ranges which are shared or partitioned across multiple devices: the programmer can think of a distributed container as of a self-contained entity.

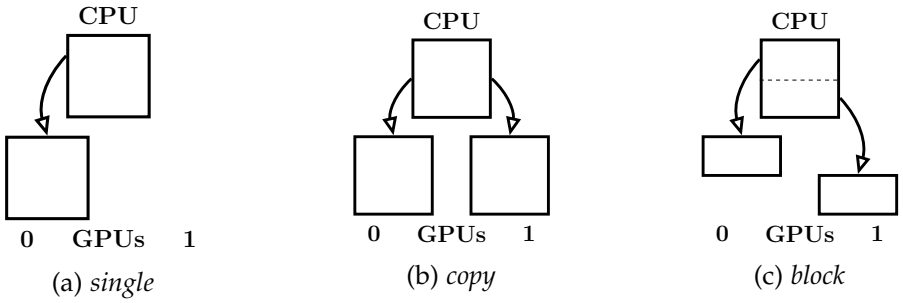
Four kinds of distributions are currently available in SkelCL: three basic distributions and one more advanced distribution. We will introduce and discuss the more advanced overlap distribution later in Section 3.2.4.2.

The three basic distributions in SkelCL are: *single*, *copy*, and *block* (see Figure 3.3 for distributing a vector on a system with two GPUs). If distribution is set to *single* (Figure 3.3a), then vector's whole data is stored on a single GPU (the first GPU if not specified otherwise). The *copy* distribution (Figure 3.3b) copies vector's entire data to each available GPU. With the *block* distribution (Figure 3.3c), each GPU stores a contiguous, disjoint chunk of the vector.

The same three distributions are provided also for the matrix container as shown in Figure 3.4. The *block* distribution (Figure 3.4c) splits the matrix into chunks of rows, which simplifies the implementation.

The application developer can set the distribution of containers explicitly, otherwise every skeleton selects a default distribution for its input and output containers. Container's distribution can be changed at runtime: this implies data exchanges between multiple GPUs and the CPU, which are performed by the SkelCL implementation implicitly. As seen earlier in this chapter, e. g., in Listing 3.3, implementing such data transfers in standard OpenCL is a cumbersome task: data has to be downloaded to the CPU before it is uploaded to the GPUs, including the corresponding length and offset calculations; this results in a lot of low-level code which becomes completely hidden when using SkelCL.

A special situation arises when the distribution is changed from the *copy* distribution to any other distribution. In this case each GPU holds its own full copy of the data which might have been modified locally on each GPU. In order to maintain SkelCL's concept of a self-contained container, these different versions are combined using a user-specified function when the distribution is changed. If no function is specified, the copy of the first GPU is taken as the new version of the container; the copies of the other GPUs are discarded.



(a) *single* (b) *copy* (c) *block*

Figure 3.4: Distributions of a matrix in SkelCL.

3.2.4 *Advanced Algorithmic Skeletons*

The basic algorithmic skeletons presented in Section 3.2.2 are long known in the functional and algorithmic skeleton communities. In this section we will introduce two new more advanced algorithmic skeletons, which are more restrictive. By limiting the use cases of these novel algorithmic skeleton we are able to make more assumptions in the implementation and provide advanced optimizations on modern multi-GPU systems.

The first new skeleton (*stencil*) is targeted towards *stencil* (a. k. a., nearest neighbor) computations, which are computations performed for every element of a container while including neighboring elements in the computation. The second new skeleton (*allpairs*) combines two matrix containers in an all-to-all fashion, which is a pattern used in applications like N-body simulation or matrix multiplication.

For both skeletons we will first formally define them before looking at possible use cases. Their implementations targeting multi-GPU systems will be described in Section 3.3.

3.2.4.1 *The Stencil Skeleton*

Many numerical and image processing applications deal with two-dimensional data and perform calculations on each data element while taking neighboring elements into account. These type of applications are also known as *stencil* or *nearest neighbor* applications. Figure 3.5 shows a visualization of a stencil application. For every pixel in the left image (one pixel is highlighted in red in the middle) a function is applied which computes a weighted average of the surrounding pixels (which are highlighted in orange) to produce the resulting image on the right side. To facilitate the development of such applications, we introduce the *stencil* skeleton that can be used with both the vector and matrix data type.

The *stencil* skeleton is customized with three parameters: a unary function f, an integer value d, and an out-of-bounds function h. The skeleton applies f to each element of an input container while taking the neighboring elements within the range d in each dimension

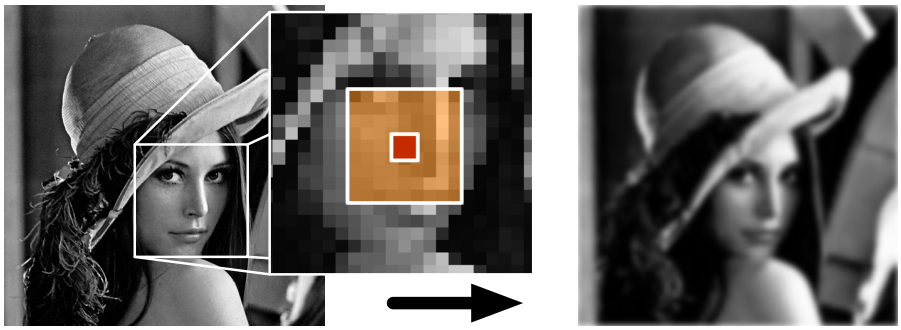


Figure 3.5: Visualization of the Gaussian blur stencil application.

into account. When neighboring elements are accesses at the boundaries of the container out-of-bound accesses occur. In these cases the function h is called with the index causing the out-of-bound access and returns a replacement value. We now formally define the *stencil* skeleton. We start with the definition for vectors:

DEFINITION 3.7. *Let $\vec{x}$ be a vector of size n with elements x_i where $0 < i \leqslant n$. Let f be an unary customizing function, d be a positive integer value, and h be an out-of-bound handling function. The algorithmic skeleton stencil is defined as follows:*

$$\textit{stencil}\ f\ d\ h\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [y_1, y_2, \ldots, y_n]$$

where

$$y_i = f\ [x_{i-d}, \ldots, x_{i+d}] \qquad \forall\, i : 0 < i \leqslant n$$

and

$$x_j = h\ j \qquad \forall\, j : -d < j \leqslant 0 \ \vee n < j \leqslant n + d.$$

The definition for matrices is similar:

DEFINITION 3.8. *Let M be an $n \times m$ matrix with elements $m_{i,j}$ where $0 < i \leqslant n$ and $0 < j \leqslant m$. Let f be an unary customizing function, d be an positive integer value, and h be an out-of-bound handling function. The algorithmic skeleton stencil is defined as follows:*

$$\textit{stencil}\ f\ d\ h \begin{bmatrix} m_{1,1} & \cdots & m_{1,m} \\ \vdots & & \vdots \\ m_{n,1} & \cdots & m_{n,m} \end{bmatrix} \stackrel{def}{=} \begin{bmatrix} n_{1,1} & \cdots & n_{1,m} \\ \vdots & & \vdots \\ n_{n,1} & \cdots & n_{n,m} \end{bmatrix}$$

where

$$n_{i,j} = f \begin{bmatrix} m_{i-d,j-d} & \cdots & m_{i-d,j+d} \\ \vdots & & \vdots \\ m_{i+d,j-d} & \cdots & m_{i+d,j+d} \end{bmatrix} \qquad \forall\, i,j \begin{array}{l} 0 < i \leqslant n, \\ 0 < j \leqslant m \end{array}$$

and

$$m_{i,j} = h\ i\ j \qquad \forall\, i,j \begin{array}{l} -d < j \leqslant 0 \ \vee n < j \leqslant n + d, \\ -d < j \leqslant 0 \ \vee m < j \leqslant m + d \end{array}.$$

SkelCL currently supports a fixed set of choices for the out-of-bound handling function h motivated by common cases of of bound handling in image processing applications. This restriction could easily be lifted in the future. The *stencil* skeleton can currently be configured to handle out-of-bound accesses in two possible ways:

1. a specified neutral value is returned (i. e., the out-of-bound function h is constant);
2. the nearest value inside the container is returned.

Possible applications for the *stencil* skeleton are image processing applications or physics simulations (see Section 4.5 and Section 4.7). A simple example application is the *discrete Laplacian operator* used in image processing, e. g., for edge detection [152]. It computes a new value for every pixel of an image by weighting and summing up its direct neighboring pixel values, as follows:

$$\mathtt{laplacian}\ M = \textit{stencil}\ f\ 1\ \bar{0}\ M$$

$$\text{where:} \quad f \begin{bmatrix} m_{i-1,j-1} & m_{i-1,j} & m_{i-1,j+1} \\ m_{i,j-1} & m_{i,j} & m_{i,j+1} \\ m_{i+1,j-1} & m_{i+1,j} & m_{i+1,j+1} \end{bmatrix} =$$

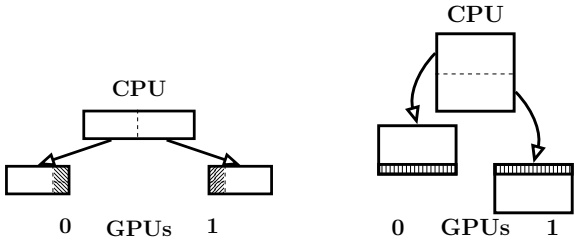
$$m_{i-1,j} + m_{i,j-1} - 4 \times m_{i,j} + m_{i,j+1} + m_{i+1,j}$$

and $\bar{0}$ is the constant function always returning 0.

3.2.4.2 *The overlap data distribution*

Together with the *stencil* skeleton we introduce a new advanced distribution called *overlap* (Figure 3.6). The overlap distribution splits the container into one chunk for each device, similarly to the *block* distribution. In addition, each chunk consists of a number of continuous elements (in case of the vector) or rows (in case of the matrix) which are copied to the neighboring devices, similar to the *copy* distribution. Therefore, the overlap distribution can be seen as a combination of the *block* and *copy* distribution. The number of elements or rows at the edges of a chunk which are copied to the neighboring devices are called the *overlap size*.

The *overlap* distribution ensures that neighboring elements are always accessible even if the container is split across multiple devices. The *overlap* distribution is, therefore, automatically selected as distribution by the *stencil* skeleton automatically setting the overlap size to the skeleton parameter d, which ensures that every device has access to the necessary neighboring elements.



(a) *overlap* (b) *overlap*

Figure 3.6: Overlap distribution of a vector and matrix in SkelCL.

3.2.4.3 *The Allpairs Skeleton*

Allpairs computations occur in a variety of applications, ranging from matrix multiplication and N-body simulations in physics [14] to pairwise Manhattan distance computations in bioinformatics [31]. These applications share a common computational pattern: for two sets of entities, the same computation is performed independently for all pairs in which entities from the first set are combined with entities from the second set. We propose the *allpairs* skeleton to simplify the development of such applications. We represent the entries of both sets as vectors of length d, where the cardinality of the first set is n and the cardinality of the second set is m. We model the first set as a $n \times d$ matrix A and the second set as a $m \times d$ matrix B. The allpairs computation yields an output matrix C of size $n \times m$ with $c_{i,j} = a_i \oplus b_j$, where a_i and b_j are row vectors of A and B, correspondingly, and $\oplus$ is a binary operator defined on vectors.

We formally define the *allpairs* skeleton as follows:

DEFINITION 3.9. *Let A be a $n \times d$ matrix, B be a $m \times d$ matrix, and C be a $n \times m$ matrix, with their elements $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$ respectively. The algorithmic skeleton allpairs with customizing binary function $\oplus$ is defined as follows:*

$$\texttt{allpairs}\ (\oplus) \begin{bmatrix} a_{1,1} & \cdots & a_{1,d} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,d} \end{bmatrix} \begin{bmatrix} b_{1,1} & \cdots & b_{1,d} \\ \vdots & & \vdots \\ b_{m,1} & \cdots & b_{m,d} \end{bmatrix} \stackrel{def}{=} \begin{bmatrix} c_{1,1} & \cdots & c_{1,m} \\ \vdots & & \vdots \\ c_{n,1} & \cdots & c_{n,m} \end{bmatrix}$$

where elements $c_{i,j}$ of the $n \times m$ matrix C are computed as follows:

$$c_{i,j} = [a_{i,1} \ \cdots \ a_{i,d}] \oplus [b_{j,1} \ \cdots \ b_{j,d}] .$$

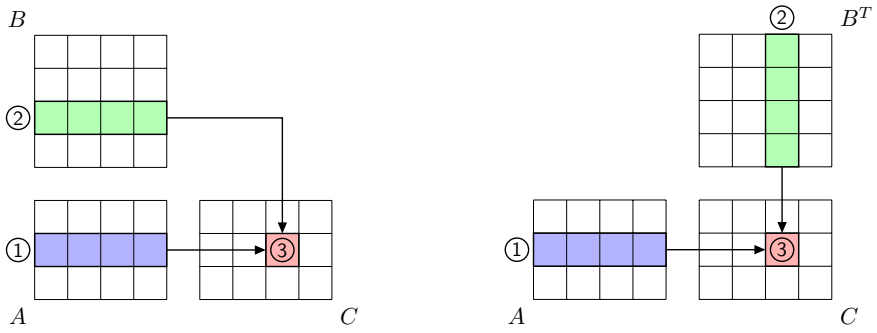


Figure 3.7: The allpairs computation schema. (a): element $c_{2,3}$ ③ is computed by combining the second row of A ① with the third row of B ② using the binary operator $\oplus$. (b): the same situation where the transpose of matrix B is shown.

Figure 3.7a illustrates this definition: the element $c_{2,3}$ of matrix C marked as ③ is computed by combining the second row of A marked as ① with the third row of B marked as ② using the binary operator $\oplus$. Figure 3.7b shows the same computation with the transposed matrix B. This visualization shows how the structure of matrix C is determined by the two input matrices A and B.

Let us consider two example applications which can be expressed by customizing the *allpairs* skeleton with a particular function $\oplus$.

EXAMPLE 1: The Manhattan distance (or L_1 distance) is a measure of distance which is used in many applications. In general, it is defined for two vectors, $\vec{x}$ and $\vec{y}$, of equal length d, as follows:

$$\mathit{ManDist}\ \vec{x}\ \vec{y} = \sum_{k=1}^{d} |x_k - y_k| \tag{3.2}$$

In [31], the so-called Pairwise Manhattan Distance (*PMD*) is studied as a fundamental operation in hierarchical clustering for data analysis. *PMD* is obtained by computing the Manhattan distance for every pair of rows of a given matrix. This computation for arbitrary matrix A can be expressed using the *allpairs* skeleton customized with the Manhattan distance defined in Equation (3.2):

$$\mathit{PMD}\ A = \mathtt{allpairs}\ \mathit{ManDist}\ A\ A \tag{3.3}$$

The $n \times n$ matrix computed by the customized skeleton contains the Manhattan distance for every pair of rows of the input $n \times d$ matrix A.

EXAMPLE 2: Matrix multiplication is a basic linear algebra operation, which is a building block of many scientific applications. A $n \times d$

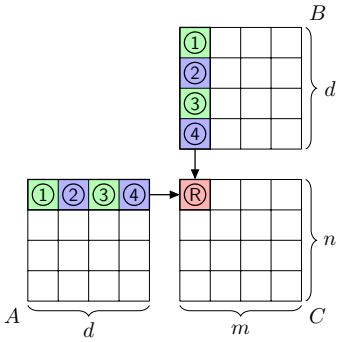


Figure 3.8: Memory access pattern of the matrix multiplication $A \times B = C$.

matrix A is multiplied with a $d \times m$ matrix B, producing a $n \times m$ matrix $C = A \times B$ whose element $c_{i,j}$ is computed as the dot product of the ith row of A with the jth column of B. The dot product of two vectors, $\vec{a}$ and $\vec{b}$ of length d, is computed as follows:

$$dotProduct\ \vec{a}\ \vec{b} = \sum_{k=1}^{d} a_k \times b_k \tag{3.4}$$

The matrix multiplication is expressed using the *allpairs* skeleton as:

$$A \times B = allpairs\ dotProduct\ A\ B^T \tag{3.5}$$

where B^T is the transpose of matrix B.

3.2.4.4 *The Specialized Allpairs Skeleton*

When targeting GPU architectures, implementing an optimized version of the *allpairs* skeleton is possible if certain assumptions about the customizing function f can be made. In this section, we will investigate the properties of f necessary for an optimized GPU implementation. In particular we will analyze the memory access pattern of the matrix multiplication as the memory accesses turns out to be crucial for the performance. We then observe that the identified memory access pattern can also be found in other allpairs computations and, therefore, define a specialized version of the allpairs skeleton, which is suitable for applications having this pattern.

MEMORY ACCESS PATTERN OF THE MATRIX MULTIPLICATION

In Figure 3.8 the memory access pattern of the matrix multiplication for 4×4 matrices is shown. To compute the element Ⓡ of the result matrix C, the first row of matrix A and the first column of matrix B are combined. In the skeleton-based code, these two vectors are used by the customizing function f, which is the dot product in case of the matrix multiplication. The dot product performs a pairwise multiplication of the two vectors and then sums up the intermediate result. In

the example, the two elements marked as ① are multiplied first and the intermediate result is stored; then, the next elements (marked as ②) are multiplied and the result is added to the intermediate result, and so forth.

A key observation is, that other applications share the same memory access pattern as the matrix multiplication shown in Figure 3.8. For example, the customizing function of the pairwise Manhattan distance as defined by Equation (3.2) follows obviously the same memory access pattern as the matrix multiplication. To find a common representation for a customizing function with this pairwise access pattern, we describe it as a sequential composition of two basic algorithmic skeletons: *zip* and *reduce*.

For the *reduce* skeleton customized with $\oplus$ and corresponding identity element $\oplus_{id}$, and the *zip* skeleton customized with $\otimes$ we can sequentially compose them as follows:

$$\begin{aligned} & \textit{reduce}\ (\oplus)\ \oplus_{id}\ \left(\ \textit{zip}\ (\otimes)\ \vec{a}\ \vec{b}\ \right) \quad = \\ \textit{reduce}\ (\oplus)\ \oplus_{id}\ & \left(\ \textit{zip}\ (\otimes)\ [a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n]\ \right) \quad = \\ & (a_1 \otimes b_1)\ \oplus\ \cdots\ \oplus\ (a_n \otimes b_n) \end{aligned}$$

This composition of the two customized skeletons yields a function which we denote *zipReduce*; it takes two input vectors and produces a single scalar value:

$$\textit{zipReduce}\ (\oplus)\ \oplus_{id}\ (\otimes)\ \vec{a}\ \vec{b} \stackrel{\text{def}}{=} \textit{reduce}\ (\oplus)\ \oplus_{id}\ \left(\ \textit{zip}\ (\otimes)\ \vec{a}\ \vec{b}\ \right)$$

Following the definition of *zipReduce*, we can express the customizing function of the Manhattan distance as follows. We use the binary operator $a \ominus b = |a - b|$ as the customizing function for *zip*, and addition as the customizing function for the *reduce* skeleton:

$$\begin{aligned} \textit{ManDist}\ \vec{a}\ \vec{b} \quad &= \quad \sum_{i=1}^{n} |a_i - b_i| = (a_1 \ominus b_1) + \cdots + (a_n \ominus b_n) \\ &= \quad \textit{zipReduce}\ (+)\ 0\ (\ominus)\ [a_1\ \cdots\ a_n]\ [b_1\ \cdots\ b_n] \end{aligned}$$

Similarly, we can express the dot product (which is the customizing function of matrix multiplication) as a zip-reduce composition, by using multiplication for customizing the *zip* skeleton and addition for customizing the *reduce* skeleton:

$$\begin{aligned} \textit{dotProduct}\ \vec{a}\ \vec{b} \quad &= \quad \sum_{i=1}^{n} a_i \times b_i = (a_1 \times b_1) + \cdots + (a_n \times b_n) \\ &= \quad \textit{zipReduce}\ (+)\ 0\ (\times)\ a\ b \end{aligned}$$

DEFINITION OF THE SPECIALIZED ALLPAIRS SKELETON We can now specialize the generic Definition 3.9 of the *allpairs* skeleton by employing the sequential composition of the customized *reduce* and *zip* skeletons for customizing the *allpairs* skeleton. From here on, we refer to this specialization as the *allpairs* skeleton *customized with zip-reduce*.

DEFINITION 3.10. *Let* A *be a* $n \times d$ *matrix,* B *be a* $m \times d$ *matrix, and* C *be a* $n \times m$ *matrix, with their elements* $a_{i,j}$, $b_{i,j}$, *and* $c_{i,j}$ *respectively. Let* $\oplus$ *be an associative binary customizing operator with the corresponding identity element* $\oplus_{id}$. *Let* $\otimes$ *be a binary customizing operator. The specialized algorithmic skeleton allpairs customized with zip-reduce is defined as follows:*

$$\textit{allpairs}\ (\oplus)\ \oplus_{id}\ (\otimes) \begin{bmatrix} a_{1,1} \cdots a_{1,d} \\ \vdots \quad\quad \vdots \\ a_{n,1} \cdots a_{n,d} \end{bmatrix} \begin{bmatrix} b_{1,1} \cdots b_{1,d} \\ \vdots \quad\quad \vdots \\ b_{m,1} \cdots b_{m,d} \end{bmatrix} \stackrel{def}{=} \begin{bmatrix} c_{1,1} & \cdots & c_{1,m} \\ \vdots & & \vdots \\ c_{n,1} & \cdots & c_{n,m} \end{bmatrix}$$

where elements $c_{i,j}$ *of the* $n \times m$ *matrix* C *are computed as follows:*

$$c_{i,j} = \textit{zipReduce}\ (\oplus)\ \oplus_{id}\ (\otimes)\ [a_{i,1}\ \cdots\ a_{i,d}]\ \left[b_{j,1}\ \cdots\ b_{j,d}\right].$$

While not every allpairs computation can be expressed using this specialization, many real-world problems can. In addition to the matrix multiplication and the pairwise Manhattan distance examples are the pairwise computation of the Pearson correlation coefficient [31] and estimation of Mutual Informations [50]. The composition of *zip* and *reduce* is well known in the functional programming world. The popular *MapReduce* programming model by Google has been inspired by a similar composition of the *map* and *reduce* skeletons. [104] extensively discusses the relation of MapReduce to functional programming.

3.3 THE SKELCL LIBRARY

In this section we discuss the SkelCL library – our implementation of the SkelCL programming model. It provides a C++ API that implements the features of the SkelCL programming model, and thus liberates the application developer from writing low-level code. In addition, the library provides some commonly used utility functions, e. g., for program initialization. The SkelCL library is open source software and available at: `http://skelcl.uni-muenster.de`.

```
1  #include <SkelCL/SkelCL.h>
2  #include <SkelCL/Zip.h>
3  #include <SkelCL/Reduce.h>
4  #include <SkelCL/Vector.h>
5
6  float dotProduct(const float* a, const float* b, int n) {
7    using namespace skelcl;
8    skelcl::init( 1_device.type(deviceType::ANY) );
9
10   auto mult = zip([](float x, float y) { return x*y; });
11   auto sum = reduce([](float x, float y){ return x+y; }, 0);
12
13   Vector<float> A(a, a+n); Vector<float> B(b, b+n);
14
15   Vector<float> C = sum( mult(A, B) );
16
17   return C.front();
18 }
```

Listing 3.5: Implementation of the dot product computation in SkelCL.

We start our discussion with an example showing how to use the SkelCL library. We describe the syntax used to represent the SkelCL programming model introduced in the previous section. This will include a discussion of C++ techniques used to implement the library. We then shift the focus to the implementations of the memory management, algorithmic skeletons, and distributions.

3.3.1 *Programming with the SkelCL Library*

Listing 3.5 shows the implementation of the dot product computation, discussed in the previous section, in SkelCL. After including the appropriate SkelCL headers (line 1—line 4) the SkelCL library can be initialized as shown in line 8. This will perform the initializations required by OpenCL. The argument provided to the `init` function specifies how many and which OpenCL devices should be used by SkelCL. Here a single device should be used which can be of any type, i. e., it can be either a CPU or a GPU. The dot product is specified using the *zip* and *reduce* skeletons. The skeletons are created using `zip` (line 10) and `reduce` (line 11) functions which expect the customizing functions of the skeletons as arguments. In this example we use C++ lambda expressions (line 10 and line 11) to specify the customizing functions of the skeletons. We create the two input `Vectors` from C style pointers (line 13). In line 15 we perform the computation by applying the customized skeletons to the data, before we finally return the computed result in line 17.

Listing 3.5 shows that the SkelCL library integrates nicely with C++: the interface of the `Vector` class looks familiar to C++ programmers and the usage of modern C++ features like lambda expressions, type deduction (`auto`), and user-defined literals simplify the programming as functions can be defined directly inline, type information can be omitted, and the specification of which devices to use can be written intuitively.

We will now discuss how our implementation enables this comfortable level of integrating SkelCL with C++.

3.3.2 *Syntax and Integration with C++*

The SkelCL library is built on top of OpenCL. This offers clear benefits as well as introduces technical challenges. One of the technical challenges is, that OpenCL requires the kernel to be specified as a string in the host program. While this enables portability across different hardware architectures, it also introduces a burden on the application developer as strong typing cannot be guaranteed statically when the host program is compiled. For the implementation of SkelCL, we choose to address this issue using a two-step implementation, which is visualized in Figure 3.9. In the first step, a custom compiler transforms the source code as seen in Listing 3.5 to a representation where the kernel computations are represented as strings as required by OpenCL. In the second step, the transformed program is compiled using a traditional C++ compiler to produce the final executable.

This allows ourselves to free the users from writing strings in their application code and maintain the usual type safety guarantees from C++ at compile time. Furthermore, we implemented type inference capabilities in our source-to-source compiler to free the application developer from specifying type information explicitly. Our two-step design also allows application developers to compile their application code into a form which then can be deployed on systems where it might be difficult to install a custom compiler.

In the next paragraph, we will start the discussion of our source-to-source compiler. We will discuss how our custom compiler, together with our template-based implementation, helps to maintain strong type safety at compile time. Finally, we will look at issues regarding the integration with C++ like sharing code between host and device code.

THE CUSTOM SKELCL COMPILER To allow a deep integration of SkelCL code with C++, we implemented a custom compiler: `skelclc`. We leverage the LLVM infrastructure [105] for building this compiler. LLVM and the related Clang project offer well-defined application programming interfaces for writing C and C++ related compiler tools. In particular the *LibTooling* API allows for writing tools which search

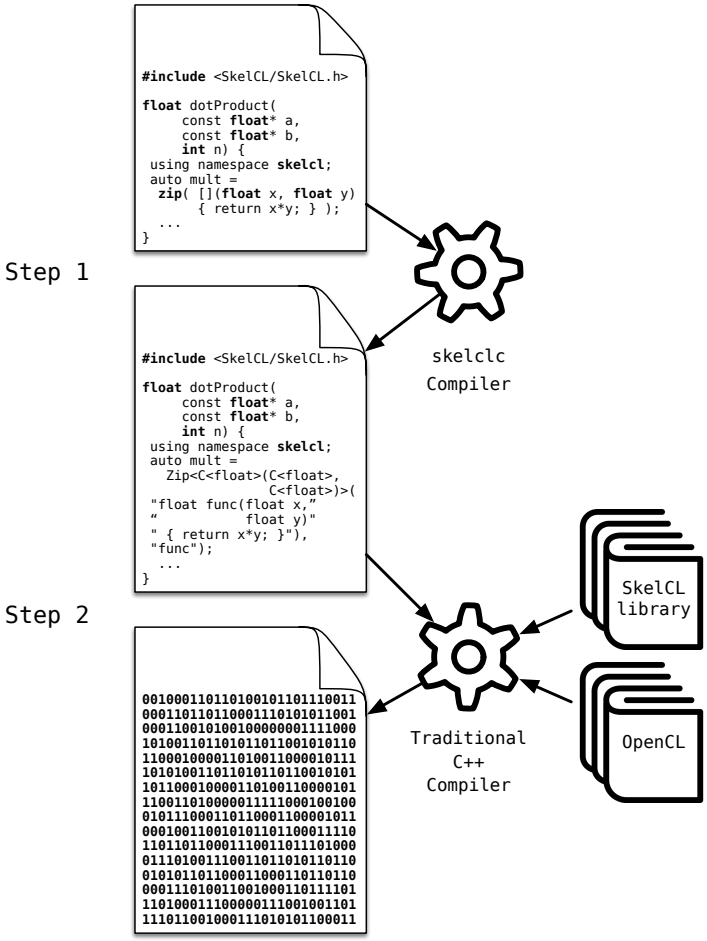


Figure 3.9: Overview of the SkelCL implementation. In the first step, the custom `skelclc` compiler transforms the initial source code into a representation where kernels are represented as strings. In the second step, a traditional C++ compiler generates an executable by linking against the SkelCL library implementation and OpenCL.

for particular patterns in the abstract syntax tree (AST) which is representing the source code and perform actions on the found code fragments. The `skelclc` custom compiler does exactly this. For example, the custom compiler searches for an expression like this:

```
auto mult = zip( [](float x, float y){ return x*y; } );
```

Listing 3.6: SkelCL code snippet before transformation.

and replaces it with:

```
auto mult = Zip<Container<float>(Container<float>,
                                 Container<float>)>(
              Source("float func(float x, float y)"
                     " { return x*y; }"), "func");
```

Listing 3.7: SkelCL code snippet after transformation.

In this example, the `zip` function has been transformed into a call of the constructor of the `Zip` class and the lambda expression has been transformed into a string. Furthermore, type information that has been inferred from the lambda expression, has been added to the templated `Zip` class.

The user is free to write the expression in Listing 3.7 explicitly, but it is arguably more convenient to write the expression as in Listing 3.6 and let `skelclc` perform the transformation automatically.

For every skeleton available in SkelCL, there exists a function like `zip` which is transformed by `skelclc` to a call of the constructor of the corresponding skeleton class.

MAINTAINING TYPE SAFETY Each skeleton is represented by a templated class, as seen in Listing 3.7 for the *zip* skeleton. The template arguments of the skeleton define the types which can be used when executing the skeleton. For a skeleton of type `skeleton<T(U)>`, an object of type `U` has to be provided on execution to receive an object of type `T`. In this respect skeletons behave exactly like functions in C++ which can be represented using the `std::function<T(U)>` template class. Skeletons taking more then one argument (like the `zip` skeleton) are represented as `skeleton<T(U, V)>`.

When performing the transformations, `skelclc` infers the types used as template arguments of the skeleton class. To do so, it determines the types of the lambda expression arguments and skeleton's result type. For Listing 3.6 this is: `float` and `float` as argument types and `float` as the result type. The result type is inferred following standard C++ rules. Based on these types, the template arguments of the skeleton are constructed.

The skeletons *map*, *zip*, and *stencil* can operate on either `Vector` or `Matrix`. For each of these skeletons there exist three kinds of fac-

tory functions: 1) for creating a skeleton operating on vectors, e. g., `mapVector`; 2) for creating a skeleton operating on matrices, e. g., `mapMatrix`; 3) for creating a skeleton capable of operating on both vectors and matrices, e. g., `map`. To allow the latter case, the templated class `Container` was introduced. Using the template specialization mechanism, the mentioned skeletons have a special implementation when `Container` is used as a template argument. In this case, the classes provide a templated function call operator which can be used either with `Vector` or `Matrix`.

Type safety is guaranteed, as the templated skeleton classes can only be executed with a container of compatible type and the type inference ensures that the user function has exactly the same type as the template argument of the skeleton class. Therefore, applying the user function to the input data is always a type-safe operation.

Because skeletons are strongly typed, the composition of skeletons is type-safe as well, i. e., in Listing 3.5 type safety between the two skeletons is guaranteed. If the result type of `zip` does not match the input type of `reduce`, a compile time error would occur.

INTEGRATION WITH C++ To allow a deep integration with C++, a customizing function is allowed to make use of user-defined types and make calls to other functions. The `skelclc` compiler detects these behaviors and ensures that the definition of the user defined type and the source code of all called functions is included in the source code passed to the skeleton. This frees the user from providing type definitions and source code twice, which would be required when using OpenCL directly or using SkelCL without the `skelclc` compiler. Code duplication should almost always be avoided as it can easily lead to hardly maintainable code and subtle bugs.

ADDITIONAL ARGUMENTS: In real-world applications (e. g., the LM OSEM discussed in detail in Section 4.6), user-defined functions often operate not only on a skeleton's input vector, but may also take additional inputs. With only a fixed number of input arguments, traditional skeletons would not be applicable for the implementation of such applications. In SkelCL, skeletons can accept an arbitrary number of additional arguments which are passed to the skeleton's customizing function.

Listing 3.8 shows an example implementation in SkelCL of the *single-precision real-alpha x plus y* (saxpy) computation – a commonly used BLAS routine. The saxpy computation is a combination of scalar multiplication of a with vector $\vec{x}$ followed by vector addition with $\vec{y}$. In Listing 3.8, the computation is implemented using the *zip* skeleton: vectors $\vec{x}$ and $\vec{y}$ are passed as input, while factor a is passed to the customizing function as an additional argument (line 5). The additional argument is simply appended to the argument list when the skeleton

```
1  float a = initScalar();
2
3  /* create skeleton Y <- a * X + Y */
4  auto saxpy = zip(
5        [](float x, float y, float a) { return a*x+y; } );
6
7  Vector<float> X(SIZE); initVector(X);
8  Vector<float> Y(SIZE); initVector(Y);
9
10 Y = saxpy( X, Y, a );      /* execute skeleton */
```

Listing 3.8: The BLAS saxpy computation using the *zip* skeleton with additional arguments

is executed (line 10). Besides scalar values, like shown in the example, vectors and matrices can also be passed as additional arguments to a skeleton. When vectors or matrices are used as additional arguments, the user is responsible to ensure that no race conditions occur when writing to or reading from the container. If the container is only used for reading, it is guaranteed that no race conditions will occur.

The additional argument feature is implemented in SkelCL using the variadic template feature from C++.

LAMBDA CAPTURES: In C++, lambda expressions can make use of variables which have been accessible at the time the lambda expression is created. The user has to explicitly indicate how these variable are *captured*, i. e., how they will be accessed once the lambda is executed. A variable can either be captured *by-value* or *by-reference*. If a variable is captured by-value, a copy of the variable is made at the time the lambda expression is created and accessed later when the lambda expression is executed. If a variable is captured by-reference, the original variable is accessed when the lambda expression is executed through a reference. Capturing a variable by reference allows the user to change the content of the variable from inside the lambda expression, furthermore, in a parallel setting the user has to ensure that the lifetime of the variable exceeds the time when the lambda expression is executed.

In SkelCL, the customizing function can be expressed as a lambda expression capturing variables. The by-value capturing of variables is fully supported by the `skelclc` compiler and the capturing by-reference is supported with the following restrictions: by-reference capturing is only allowed if the reference is used to read from the variable only, and writing to the captured variable is forbidden. The reason for these restrictions is that when executing the lambda expression on the GPU, a write to a by-reference captured variable requires a costly data transfer to ensure that the variable stored in CPU memory

```
1  float a = initScalar();
2
3  /* create skeleton Y <- a * X + Y */
4  auto saxpy = skelcl::zip(
5        [a](float x, float y) { return a*x+y; });
6
7  Vector<float> X(SIZE); initVector(X);
8  Vector<float> Y(SIZE); initVector(Y);
9
10 Y = saxpy( X, Y );      /* execute skeleton */
```

Listing 3.9: The BLAS saxpy computation using a *zip* skeleton customized with a lambda expression capturing the variable a.

is modified. Furthermore, as the lambda expression will be executed multiple times in parallel as part of the skeletons execution, it is likely that race conditions occur when writing to the captured reference. We can rewrite the saxpy example using a lambda expression capturing the variable a as shown in Listing 3.9.

The variable a is now captured by the lambda expression in line 5. Note that a is no longer passed as an argument when executing the skeleton in line 10.

Additional arguments and lambda captures are related techniques which both can be used to make additional data available inside of the user function. There is one main technical differences between both: for using lambda captures the variable to be captured has to be declared and available when *declaring* the user function (in line 5 in Listing 3.9) which is not the case for the additional argument feature where the variable has to available when *executing* the skeleton (in line 10 in Listing 3.8). By supporting the capturing feature of lambda expressions SkelCL source code becomes more C++ idiomatic, but ultimately the user has the choice of using either feature as it fits his or her needs and personal taste.

3.3.3 *Skeleton Execution on OpenCL Devices*

The process of executing a skeleton on an OpenCL device, e. g., a GPU, follows always the same steps independently of the skeleton involved. This process is described in this subsection, before we will look at the individual skeleton implementations in the next subsection.

The `skelclc` compiler transforms the SkelCL application code so that the source code of the customizing function of a skeleton is available to the SkelCL library implementation as a string. When a skeleton instance is created, the SkelCL library implementation performs the following steps to create an OpenCL kernel: 1) the source code

```
1  float a = initScalar();
2  /* create skeleton Y <- a * X + Y */
3  auto saxpy = skelcl::Zip<Vector<float>(Vector<float>,
4                                         Vector<float>,
5                                         float)>(
6      skelcl::Source("float func(float x, float y, float a)"
7                     " { return a*x+y; }"), "func");
8  /* create input vectors */
9  Vector<float> X(SIZE); initVector(X);
10 Vector<float> Y(SIZE); initVector(Y);
11 Y = saxpy( X, Y, a );      /* execute skeleton */
```

Listing 3.10: Source code for the saxpy application emitted by the `skelclc` compiler.

of the customizing function is merged with the OpenCL kernel implementation of the skeleton; 2) the merged source code is adapted into the final OpenCL kernel. This is done to support additional arguments and so that names and types of the customizing function and skeleton implementation match; 3) the created OpenCL kernel is stored to a file to avoid performing steps 1) and 2) multiple times for the same kernel.

On execution of the skeleton, the following steps are performed to execute the computation on an OpenCL device: 1) the data of the input containers is uploaded to the OpenCL device, if it is not already there; 2) the OpenCL kernel arguments are set, including the additional arguments; 3) the OpenCL kernel is executed.

In the next two paragraphs, we will describe these two processes and their individual steps.

CREATING AN OPENCL KERNEL We will use the saxpy example from the previous section as a running example to explain how an OpenCL kernel is created. Listing 3.10 shows the code for the saxpy application emitted by the `skelclc` compiler after receiving the code shown in Listing 3.8 as input.

To create the corresponding OpenCL kernel, the source code of the customizing function in line 6 and line 7 is combined with the prototype implementation of the *zip* skeleton which is shown in Listing 3.11. The prototype implementation created the OpenCL kernel `ZIP` receiving three pointers to global memory and one integer as arguments. A boundary check is performed in line 8 before a function with the name `USER_FUNC` is called (line 9) with a pair of elements read from the two input arrays `left` and `right`. The result of the function call is stored in the output array `out`.

The combined source codes clearly do not yet form a valid OpenCL kernel as there exists no function called `USER_FUNC`. This *prototype* im-

```
1 typedef int T0; typedef int T1; typedef int T2;
2
3 kernel void ZIP(const global T0* left,
4                 const global T1* right,
5                       global T2* out,
6                 const unsigned int size) {
7   unsigned int id = get_global_id(0);
8   if (id < size) {
9     out[id] = USER_FUNC(left[id], right[id]); } }
```

Listing 3.11: Prototype implementation of the *zip* skeleton in OpenCL.

```
1  float USER_FUNC(float x, float y, float a) { return a*x+y; }
2  typedef float T0; typedef float T1; typedef float T2;
3
4  kernel void ZIP(const global T0* left,
5                  const global T1* right,
6                        global T2* out,
7                  const unsigned int size, float a) {
8    unsigned int id = get_global_id(0);
9    if (id < size) {
10     out[id] = USER_FUNC(left[id], right[id], a); } }
```

Listing 3.12: OpenCL implementation of the *zip* skeleton customized for performing the *saxpy* computation.

plementation has to be adapted to become a valid OpenCL kernel. To make this adaption, the SkelCL library implementation makes use of the same LLVM and Clang infrastructure used to build the `skelclc` compiler. Three steps are performed to create a valid OpenCL kernel: 1) the customizing function is renamed into `USER_FUNC`; 2) the types of the customizing functions are used to change the `typedefs` in line 1 so that the types of the kernel function `ZIP` match; 3) the additional arguments of the customizing function are appended to the parameter list of the kernel function `ZIP` and the call to the customizing function is adapted to forward the arguments.

After these steps the source code has been transformed into a valid OpenCL kernel as shown in Listing 3.12.

Performing the adaption of the source code involves parsing the code and building an abstract syntax tree for it by the Clang compiler library. Performing this every time a skeleton is created is wasteful as this task can take up to several hundreds of milliseconds, which can be a considerable overhead for small kernels. Therefore, the SkelCL library implements a caching of already transformed kernels to disk.

If the same kernel is used again, the transformed code is loaded from the cache.*

EXECUTE A SKELETON ON AN OPENCL DEVICE On execution of the skeleton, the created OpenCL kernel is executed on possibly multiple OpenCL devices. The data distribution of the input containers determine which OpenCL devices are used for the execution. If the copy, block, or overlap distribution is used, this means that all available devices are involved in the computation as defined by the distributions. If the single distribution is used then just a single device is used. If no distribution is set for the input containers, each skeleton selects a default distribution.

The first step for the execution is to upload the data of the input containers to the OpenCL devices. Before performing a data transfer, the SkelCL implementation checks if the transfer is necessary or if the data is already up to date on the receiving side. The details how this is implemented in the memory management part of SkelCL are described in Section 3.3.5.

Before enqueuing the OpenCL kernel in the OpenCL command queue, its kernel arguments have to be set. First, the regular arguments are set correspondingly to the order of arguments as defined in the skeleton's prototype implementation of the OpenCL kernel. Afterwards, the additional arguments are set.

Finally, the OpenCL kernel is enqueued with a global size corresponding to the size of the input containers.

3.3.4 *Algorithmic Skeleton Implementations*

The SkelCL library implements each skeleton of the SkelCL programming model using one or more OpenCL kernels. In this subsection, we discuss the implementation of each skeleton targeting single- and multi-device systems.

3.3.4.1 *The Map Skeleton*

The OpenCL kernel implementing the *map* skeleton is straightforward and similar to the implementation of the *zip* skeleton shown in Listing 3.11. After a boundary check is performed, the customizing function is called with an item of the input container. The return value is stored in the output container. On execution, the size of the input container determines the number of work-items launched for executing the kernel code. The same kernel code is used on single- and multi-device systems. If the input container has no distribution declared for it, the block distribution is used as default, ensuring that all devices participate in the execution of the *map* skeleton.

* We observed that loading kernels from disk is in some cases at least five times faster than creating them from the source.

3.3.4.2 *The Zip Skeleton*

The OpenCL kernel of the *zip* skeleton was presented in the previous section in Listing 3.11. For multi-device systems the block distribution is used as default. The SkelCL library implementation enforces that the distributions of the two input vectors are the same. If this is not the case, the distribution of the second input is changed to meet this requirement.

3.3.4.3 *The Reduce Skeleton*

The *reduce* skeleton is implemented as a parallel tree-based reduction. Two OpenCL kernels are used for the implementation. The first kernel implements a reduction where each work-group independently reduces, depending on the input size, a large number of several hundred thousand elements to a single element. This kernel is executed by multiple work-groups in parallel to fully exploit the GPU. The second kernel is only executed by a single work-group and continues the reduction of the intermediate results from the first kernel. This two-kernel strategy is required for an efficient implementation in OpenCL as no synchronization between work-items from different work-groups is possible.

We will discuss different optimized implementations of the parallel reduction in OpenCL in great detail in Chapter 5. The SkelCL implementation is based on the most optimized implementation discussed there and shown in Listing 5.7 on page 128.

On a multi-device system, when using the block distribution, each device performs a partial reduction of the data available in its memory. Then the intermediate results are transferred to the first device which performs the final reduction. The output of the *reduce* skeleton is a vector with a single element which is single distributed on the first device.

3.3.4.4 *The Scan Skeleton*

The implementation of the *scan* skeleton follows the implementation presented in [83]. This is a two-phase algorithm where each phase is implemented as an OpenCL kernel.

The execution of the *scan* skeleton on multi-device systems is visualized in Figure 3.10. Here an example is shown where the prefix sum is computed on a four-device system using the *scan* skeleton. The input vector with the values $[1, \ldots, 16]$ is distributed using the *block* distribution by default. This is shown in the top line. After performing the scan algorithm on all devices (second line of the figure), *map* skeletons are built implicitly using the marked values and executed on all devices except the first one. This produces the final result, as shown in the bottom line.

The output vector is block-distributed among all GPUs.

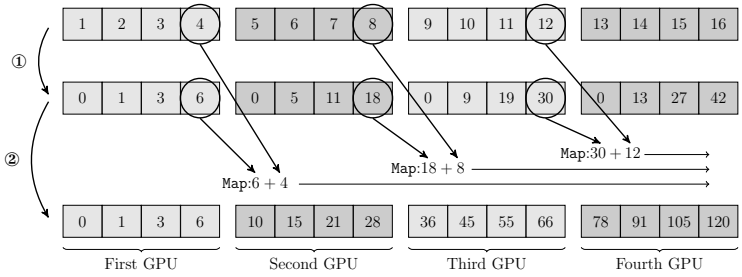


Figure 3.10: Implementation of the *scan* skeleton visualized for four GPUs: (1) All GPUs scan their parts independently. (2) *map* skeletons are created automatically and executed to produce the result.

3.3.4.5 *The Stencil Skeleton*

In stencil applications, the same computation is performed for each element of a container where the computation depends upon neighboring values. In Section 3.2 we defined the *stencil* skeleton to simplify the development of stencil applications. The SkelCL library provides two implementations of the *stencil* skeleton. The first one, called `MapOverlap`, supports simple stencil computations; the second one, called `Stencil`, provides support for more complex stencil computations possibly executed iteratively. In order to achieve high performance, both implementations `MapOverlap` and `Stencil` use the GPU's fast local memory. Both implementations perform the same basic steps on the GPU: first, the data is loaded from the global memory into the local memory; then, the customizing function is called for every data element by passing a pointer to the element's location in the local memory; finally, the result of the customizing function is copied back into the global memory. Although both implementations perform the same basic steps, different strategies are implemented for loading the data from the global into the local memory.

In this subsection, we will start by discussing the `MapOverlap` implementation and then discuss the `Stencil` implementation. Finally, we will discuss the implementation of the *stencil* skeleton for multi-device systems.

THE MAPOVERLAP IMPLEMENTATION We will use the Gaussian blur as an example application to discuss the `MapOverlap` implementation. The Gaussian blur is a standard image processing algorithm used, among other things, for noise reduction. The color of every pixel of an image is modified by computing a weighted average of the neighboring pixel color values. The application will be discussed in more detail in Section 4.5.1.

Listing 3.13 shows how the `MapOverlap` skeleton implementation is used to express the Gaussian blur. The `mapOverlap` function in line 1 is customized with a lambda expression which uses its argument, a `Neighborhood` object (line 2), to access the neighboring elements with

```
1 auto gauss = mapOverlap(
2   [](Neighborhood<char>& in_img) {
3       char ul = in_img[{-1, -1}];
4       ...
5       char lr = in_img[{+1, +1}];
6       return computeGaussianBlur(ul, ..., lr); },
7   1, BorderHandling::NEUTRAL(0));
```

Listing 3.13: Implementation of Gaussian blur using the `MapOverlap` implementaion of the *stencil* skeleton.

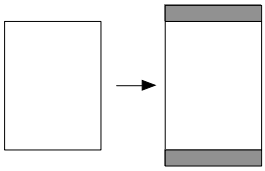


Figure 3.11: The `MapOverlap` implementation of the *stencil* skeleton prepares a matrix by copying data on the top and bottom

relative indices (line 3 and line 5). The second and third argument of the factory function define the range of the stencil shape and the border handling method (line 7). The actual computation of the Gaussian blur is omitted from Listing 3.13 for brevity reasons.

Listing 3.14 shows a sketch of the OpenCL kernel created by the `MapOverlap` implementation. To represent the `Neighborhood` object in OpenCL, a C `struct` is created (line 2—line 3). A helper function `get` (line 5—line 6) is created which handles the read access to the data hold by the neighborhood object. The kernel function `mapoverlap`, defined in line 14, first loads the data required by a work-group into a local buffer stored in the fast local GPU memory (line 16). It then prepares a neighborhood object and passes it to the customizing function after a boundary check was performed. A barrier (line 19) ensures that the loading into the fast local memory has been completed before any work-item executes the customizing function.

Loading of the data into the fast local memory is lengthy and involves performing the boundary checks, therefore, it is not shown in Listing 3.14. To minimize the overhead on the GPU, the `MapOverlap` implementation prepares the input matrix on the CPU before uploading it to the GPU: padding elements are appended; they are used to avoid out-of-bounds memory accesses to the top and bottom of the input matrix, as shown in Figure 3.11. This slightly enlarges the input matrix, but it reduces branching on the GPU due to avoiding some out-of-bound checks. In SkelCL a matrix is stored row-wise in memory on the CPU and GPU, therefore, it would be complex and costly to add padding elements on the left and right of the matrix. To avoid out-of-bound accesses for these regions, the boundary checks are performed on the GPU.

```
1  #define RANGE (1)   #define NEUTRAL (0)
2  struct {
3    local char* data; int row; int col; } char_neighborhood_t;
4
5  char get(char_neighborhood_t* m, int x, int y) {
6    return m->data[/*computed from RANGE, row, col, x, y*/]; }
7
8  char USER_FUNC(char_neighborhood_t* in_img) {
9    char ul = get(in_img, -1, -1);
10   ...
11   char lr = get(in_img, +1, +1);
12   return computeGaussianBlur(ul, ..., lr); }
13
14 kernel void mapoverlap(global char* in, global char* out,
15           local char* buffer, int numCols, int numRows) {
16   ... // load part of in into local buffer
17   char_neighborhood_t M;   M.data = buffer;
18   M.row = get_local_id(1); M.col = get_local_id(0);
19   barrier(CLK_LOCAL_MEM_FENCE);
20   if (/* not out of bound */)
21     out[index] = USER_FUNC(&M); }
```

Listing 3.14: OpenCL kernel created by the `MapOverlap` implementation for the Gaussian blur application.

THE STENCIL IMPLEMENTATION We use an iterative stencil application simulating heat flow to discuss the `Stencil` implementation. The application simulates heat spreading from one location and flowing throughout a two-dimensional simulation space. Let us assume that the heat flows from left to right as indicated by the arrows in Figure 3.12. The heat value of a cell is updated based on its (left) neighboring cells. Multiple iteration steps are required to simulate the flow of heat over a longer distance.

Listing 3.15 shows the implementation of an iterative stencil application simulating heat transfer in SkelCL. The application developer specifies the customizing function (line 2—line 6), as well as the extents of the stencil shape (line 7) and the out-of-bound handling (line 8). The `Stencil` implementation allows the stencil shape's extents to be specified using four values for each of the directions: up, right, down, and left. In the example in Listing 3.15, the heat flows from left to right, therefore, no accesses to the elements to the right are necessary and the stencil space's extents are specified accordingly (note the 0 in line 7 representing the extent to the right). Figure 3.12 illustrates this situation: the dark gray element is updated by using the values from the left. The specified stencil shape's extent is highlighted in light gray. In our current implementation, the user has to

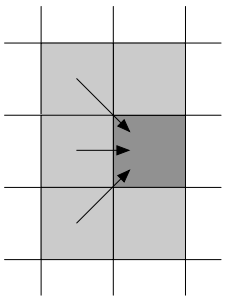


Figure 3.12: Stencil shape for heat simulation

```
1  auto heatSim = skelcl::stencil(
2   [](Neighborhood<char>& in) {
3    char lt = in[{-1, -1}];
4    char lm = in[{-1,  0}];
5    char lb = in[{-1, +1}];
6    return computeHeat(lt,lm,lb);},
7   StencilShape(1, 0, 1, 1),
8   BorderHandling::NEUTRAL(255));
9  heatSim(100_iterations,
10         simulationSpace);
```

Listing 3.15: Heat simulation with the *stencil* skeleton

explicitly specify the stencil shape's extents, which is necessary for performing the out-of-bound handling on the GPU. In future work, the stencil shape could be inferred from the customizing function using source code analysis in many common cases. This would help to avoid inconsistencies and free the user from specifying this information explicitly.

Stencil applications often perform stencil computations iteratively, like the heat transfer example which performs multiple iteration steps in order to simulate the transfer of heat over time. The `Stencil` implementation supports iterative execution, which is especially challenging to implement on multi-device systems as we will see later. On execution of the skeleton, the number of iterations to be performed is specified by the user as the first argument (line 9). This could be extended in the future, so that the user specifies a function to check if a application specific condition is met and stop the iteration.

The OpenCL kernel created by the `Stencil` implementation looks similar to the `MapOverlap` implementation presented in Listing 3.14. The `Stencil` implementation uses a slightly different strategy than the `MapOverlap` implementation in order to enable the usage of different out-of-bound modes and stencil shapes when using several *stencil* skeletons in a sequence, which we discuss in the next paragraph. To understand why the strategy used by the `MapOverlap` implementation is not sufficient for stencil sequences let us consider a situation where two stencil computations are performed one after another and the two stencil shapes used are different. This cannot be implemented efficiently using `MapOverlap`'s implementation strategy, as the input matrix is extended on the CPU as specified by the first stencil shape. Therefore, data would have to be downloaded to the CPU between executions and the data layout would have to be changed. To avoid this problem, the `Stencil` implementation does not append padding elements on the CPU, but rather manages all out-of-bounds accesses on

```
1 auto gauss     = stencil(...);
2 auto sobel     = stencil(...);
3 auto nms       = stencil(...);
4 auto threshold = stencil(...);
5
6 StencilSequence<Pixel(Pixel)>
7     canny(gauss, sobel, nms, threshold);
```

Listing 3.16: Structure of the Canny algorithm implemented by a sequence of skeletons.

the GPU, which slightly increases branching in the code, but enables a more flexible usage of the skeleton.

SEQUENCE OF STENCIL OPERATIONS Many real-world applications perform different stencil operations in sequence, like the popular *Canny algorithm* [120] used for detecting edges in images. For the sake of simplicity, we consider a version which applies the following steps: 1) a noise reduction operation is applied, e. g., a Gaussian blur; 2) an edge detection operator like the Sobel filter is applied; 3) the so-called non-maximum suppression is performed, where all pixels in the image are colored black except pixels being a local maximum; 4) a threshold operation is applied to produce the final result. A more complex version of the algorithm performs the edge tracking by hysteresis as an additional step.

Using the SkelCL library, each single step of the Canny algorithm can be expressed using the *stencil* skeleton, as shown in Listing 3.16. The threshold operation performed as the last step, does not need access to, neighboring elements, because the user function only checks the value of a single pixel. The `Stencil` implementation automatically uses the implementation of the simpler (and thus faster) *map* skeleton when the user specifies a stencil shape whose extents are 0 in all directions. The single steps are combined into a single object of type `StencilSequence` which can be executed like a *stencil* skeleton. On execution, it passes its input data to the first stencil defined in the sequence, which passes its output to the next stencil, and so forth.

TARGETING MULTI-GPU SYSTEMS The implicit and automatic support of systems with multiple OpenCL devices is one of the key features of SkelCL. By using distributions, SkelCL completely liberates the user from error-prone and low-level explicit programming of data (re)distributions on multiple GPUs.

The `MapOverlap` implementation uses the overlap distribution with *border regions* in which the elements calculated by a neighboring device are located. When it comes to iteratively executing a skeleton, data has to be transferred among devices between iteration steps,

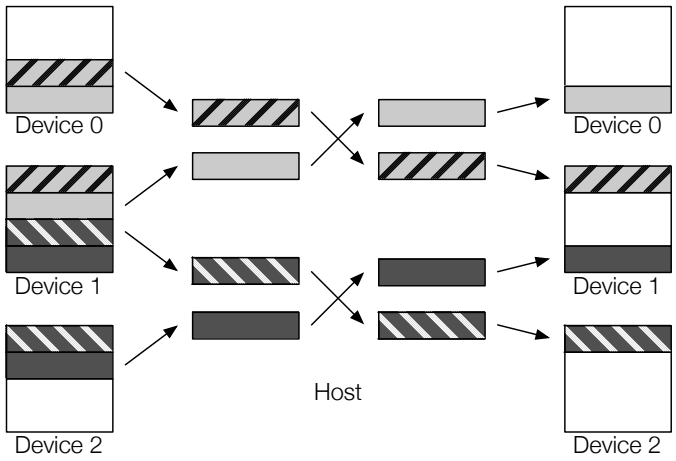


Figure 3.13: Device synchronization for three devices. Equally patterned and colored chunks represent the border regions and their matching inner border region. After the download of the appropriate inner border regions, they are swapped pair-wise on the host. Then the inner border regions are uploaded in order to replace the outdated border regions.

in order to ensure that data for the next iteration step is up-to-date. As the `MapOverlap` implementation does not explicitly supports iterations, its implementation is not able to exchange data between devices besides a full down- and upload of the matrix.

The `Stencil` implementation explicitly supports iterative execution and, therefore, only exchanges elements from the border region and does not perform a full down- and upload of the matrix, as the `MapOverlap` implementation does. Figure 3.13 shows the *device synchronizations*, i. e., the data exchange performed between two iterations by the `Stencil` implementation. Only the appropriate elements in the *inner border region*, i. e., the border regions adjacent to two OpenCL devices, are downloaded and stored as `std::vectors` in a `std::vector`. Within the outer vector, the inner vectors are swapped pair-wise on the host, so that the inner border regions can be uploaded in order to replace the out-of-date border regions.

By enlarging the number of elements in the border regions, multiple iteration steps can be performed on each device before exchanging data. However, this introduces redundant computations, such that a trade-off between data exchange and redundant computations has to be found. For the `Stencil` implementation, the user can specify the number of iterations between device synchronizations. In [23] the effects of exploring this parameter space is discussed. For the investigated applications, the effect of choosing different numbers of iterations between device synchronization was not very large.

```
1 auto mm = allpairs([](const Vector<float>& a,
2                       const Vector<float>& b) {
3                         float c = 0.0f;
4                         for (int i = 0; i < a.size(); ++i)
5                           c += a[i] * b[i];
6                         return c; });
```

Listing 3.17: Matrix multiplication expressed using the generic *allpairs* skeleton.

3.3.4.6 *The allpairs Skeleton*

The *allpairs* skeleton defined in Section 3.2 applies a customizing function to all pairs of vectors from two matrices. There exist two version of the skeleton: the generic *allpairs* skeleton introduced in Section 3.2.4.3 and the specialized *allpairs* skeleton introduced in Section 3.2.4.4. In the specialized version the customizing function is expressed as a composition of the *zip* and *reduce* skeleton.

In this subsection, we will first discuss the generic and then the specialized implementation. We will also estimate the performance benefit gained by the specialized implementation. Finally, we will discuss the implementation of the *allpairs* skeleton on multi-device systems.

THE GENERIC ALLPAIRS SKELETON Listing 3.17 shows the program for computing matrix multiplication using the generic *allpairs* skeleton. The implementation of the customizing function is straightforward. It is expressed as a lambda expression that receives a pair of vectors, multiplies their elements and sums them up.

Listing 3.18 shows the OpenCL kernel created after adding and adapting the generic allpairs implementation. The customizing function (line 4—line 9) has been transformed by the `skelclc` compiler. The vector class has been replaced by an OpenCL representation (defined in line 1 and line 2) and the element access to the vectors has been replaced by computations of the matching indices. This implementation assumes that the rows of the first matrix are combined with the columns of the second matrix, as it is required for the matrix multiplication.

The `allpairs` kernel function prepares instances of the `struct` replacing the vector class in line 16 and line 17. After performing a boundary check, the customizing function is called in line 19. This OpenCL kernel is executed once for every element of the output matrix C.

This generic implementation makes no assumption about the order in which the customizing function (`USER_FUNC`) accesses the elements of its two input vectors. In this generic case, we cannot assume that the two vectors fit entirely into the fast but restricted GPU local

```
1  struct {
2    global float* data; int size; int index; } float_matrix_t;
3
4  float USER_FUNC(float_matrix_t* a, float_matrix_t* b) {
5     float c = 0.0f;
6     for (int i = 0; i < a->size; ++i) {
7       c +=   a->data[a->index * a->size + i]
8            * b->data[i * b->size + b->index]; }
9     return c; }
10
11 kernel void allpairs(const global float* Ap,
12                       const global float* Bp,
13                             global float* Cp,
14                       int n, int d, int m) {
15   int col = get_global_id(0); int row = get_global_id(1);
16   float_matrix_t A; A.data = Ap; Am.size = d; A.index = row;
17   float_matrix_t B; B.data = Bp; Bm.size = m; B.index = col;
18   if (row < n && col < m)
19     Cp[row * m + col] = USER_FUNC(&A, &B); }
```

Listing 3.18: OpenCL kernel used in the implementation of the generic *allpairs* skeleton.

memory. Therefore, we have to use only the slower global memory in the generic implementation. On modern GPUs, accesses to the global memory are very expensive, taking up to 800 processor cycles, as compared to only few cycles required to access the local memory [44].

Let us assume targeting GPU architectures and estimate the number of global (and, therefore, expensive) memory accesses required for computing an element of the matrix multiplication in the generic case. One global memory read access for every element of both input vectors is performed, and a single global memory write access is required to write the result into the output matrix. Therefore,

$$n \cdot m \cdot (d + d + 1) \tag{3.6}$$

global memory accesses are performed in total, where n and m are the height and width of matrix C and d is the width of A and the height of B. By using the fast but small local memory, this number of global memory accesses can be reduced and, thus, performance can be improved, as we will see in the next paragraph. Using the local memory for matrix multiplication is a well-known optimization which we systematically apply to a generic skeleton, rather than to a particular application as usually done in the literature.

```
auto mult  = zip([](float x, float y){return x*y;});
auto sumUp = reduce([](float x, float y){return x+y;}, 0);
auto mm    = allpairs(sumUp, mult);
```

Listing 3.19: Matrix multiplication expressed using the specialized *allpairs* skeleton.

THE SPECIALIZED ALLPAIRS SKELETON Listing 3.19 shows matrix multiplication expressed using the specialized *allpairs* skeleton. By expressing the customizing function of the *allpairs* skeleton as a *zip-reduce* composition, we provide to the skeleton implementation additional semantic information about the memory access pattern of the customizing function, thus allowing for improving the performance. Our idea of optimization is based on the OpenCL programming model that organizes *work-items* in *work-groups* which share the same GPU local memory. By loading data needed by multiple work-items of the same work-group into the local memory, we can avoid repetitive accesses to the global memory.

For the *allpairs* skeleton with the *zip-reduce* customizing function, we can adopt the implementation schema for GPUs from [136], as shown in Figure 3.14. We allocate two arrays in the local memory, one of size $r \times k$ ($r = 2$, $k = 3$ in Figure 3.14) for elements of A and one of size $k \times c$ ($c = 3$ in Figure 3.14) for elements of B. A work-group consisting of $c \times r$ work-items computes s blocks ($s = 2$ in Figure 3.14) of the result matrix C. In Figure 3.14, the two blocks marked as ⑦ and ⑧ are computed by the same work-group as follows. In the first iteration, the elements of blocks ① and ② are loaded into the local memory and combined following the *zip-reduce* pattern. The obtained intermediate result is stored in block ⑦. Then, the elements of block ③ are loaded and combined with the elements from ② which still reside in the local memory. The intermediate result is stored in block ⑧. In the second iteration, the algorithm continues in the same manner with blocks ④, ⑤, and ⑥, but this time, the elements of the blocks

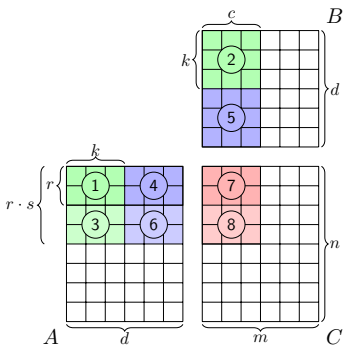


Figure 3.14: Implementation schema of the specialized *allpairs* skeleton.

are also combined with the intermediate results of the first iteration, which are stored in blocks ⑦ and ⑧. The advantage of computing multiple blocks by the same work-group is that we keep the elements of B in the local memory when computing the intermediate results, i. e., we do not reload block ② twice for the computation of blocks ⑦ and ⑧.

Every element loaded from the global memory is used by multiple work-items: e. g., the upper left element of block ① is loaded only once from the global memory, but used three times: in the computation of the upper left, upper middle, and upper right elements of ⑦. In general, every element loaded from A is reused c times, and every element from B is reused $r \cdot s$ times. As the intermediate results are stored in the global memory of matrix C, we perform two additional memory accesses (read/write) for every iteration, i. e., $2 \cdot \frac{d}{k}$ in total. Therefore, instead of $n \cdot m \cdot (d + d + 1)$ (see Equation (3.6)) global memory accesses necessary when using the non-specialized skeleton only

$$n \cdot m \cdot \left(\frac{d}{r \cdot s} + \frac{d}{c} + 2 \cdot \frac{d}{k}\right) \tag{3.7}$$

global memory accesses are performed. By increasing the parameters s and k, or the number of work-items in a work-group (c and r), more global memory accesses can be saved. However, the work-group size is limited by the GPU hardware. While the parameters can be chosen independently of the matrix sizes, we have to consider the amount of available local memory. [68] and [136] discuss how suitable parameters can be found by performing runtime experiments. In [68] the parameters $c = 32$, $r = 8$, $s = 32$, and $k = 64$ are used on modern GPU hardware showing good performance.

We will report measurements of the performance difference for the two skeleton implementations on real hardware in Chapter 4.

THE ALLPAIRS SKELETON USING MULTIPLE GPUS The *allpairs* skeleton can be efficiently implemented not only on systems with

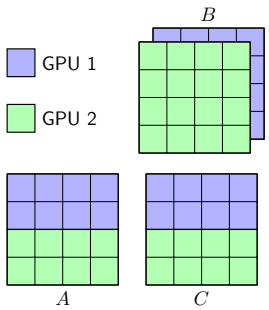


Figure 3.15: Data distributions used for a system with two GPUs: matrices A and C are *block* distributed, matrix B is *copy* distributed.

a single GPU, but on multi-GPU systems as well. The necessary data distribution can be easily expressed using two of SkelCL's *distributions*, as shown in Figure 3.15: Matrix B is *copy* distributed, i. e., it is copied entirely to all GPUs in the system. Matrix A and C are *block* distributed, i. e., they are row-divided into as many equally-sized blocks as GPUs are available; each block is copied to its corresponding GPU. Following these distributions, each GPU computes one block of the result matrix C. In the example with two GPUs shown in Figure 3.15, the first two rows of C are computed by GPU 1 and the last two rows by GPU 2. The *allpairs* skeleton automatically selects these distributions, therefore, the same source code can be used when using a single GPU or multiple GPUs.

3.3.5 *Memory Management Implementation*

In the SkelCL programming model, the user manages memory using *container data types*. In the SkelCL library, the two container data types – vector and matrix – are implemented as template classes. This generic implementation allows for storing data items of any primitive C/C++ data type (e. g., `int`), as well as user-defined data structures (`structs`). The implementations follow the *resource acquisition is initialization* (RAII) idiom, which means that they automatically allocate needed resources and free them automatically when the lifetime of the container ends.

THE SKELCL VECTOR The SkelCL vector replicates the interface of the vector from the C++ Standard Template Library (STL), i. e., it can be used as a drop-in replacement of the standard vector. Internally, a vector comprises pointers to the corresponding areas of main memory (accessible by the host) and device memory. The vector holds one pointer for the host and one pointer for each device available. Memory on the devices is allocated automatically, according to the distribution of the vector: while for a single distributed vector only memory on a single device is allocated, for a vector distributed with the copy, block, or overlap distribution memory on all devices is allocated. The selected distribution obviously also influences how big the buffers allocated on the devices will be.

Before the execution of a skeleton, the input vector's implementation ensures that all of its data is available on the devices. This might result in implicit data transfers from the host memory to device memory. The data of the output vector is not copied back to the host memory but rather resides in the device memory. Before every data transfer, the vector implementation checks whether the data transfer is necessary; only then the data is actually transferred. Hence, if an output vector is used as the input to another skeleton, no further data transfer is performed. This *lazy copying* in SkelCL defers data transfers as

long as possible or avoids them completely and, thus, minimizes the costly data transfers between host and device. While all data transfers are performed implicitly by SkelCL, we understand that experienced application developers may want to have a fine-grained control over the data transfers between host and devices. For that purpose, SkelCL offers a set of APIs which developers can use to explicitly initiate and control the data transfer to and from the devices.

THE SKELCL MATRIX The SkelCL matrix offers an easy to use interface similar to the interface of the vector. Data is stored in the row-major order and iterators are provided to iterate first over rows and then inside of a single row to access a particular element. For the copy, block, and overlap distributions, the matrix is divided across rows. A single row is never split across multiple devices, which simplifies the memory management. Besides offering an interface to access elements on the host, the matrix also offers an interface for accessing elements on the device by using two-dimensional indices. This frees the application developer from performing cumbersome index calculations manually.

3.3.6 *Data Distribution Implementation*

The data distributions determine how the data of a container is distributed across multiple devices. In the SkelCL library implementation, there exists a class for each data distribution encapsulating the behavior of the distribution. Every container stores its current data distribution as a member variable. When the data of the container has to be transferred to or from the devices, the data distribution object is invoked to perform the data transfer operation.

3.4 CONCLUSION

In this chapter we introduced the SkelCL programming model and its implementation as a C++ library. We started the chapter with a case study showing the drawbacks of the low-level OpenCL programming approach. Then, we introduced the SkelCL programming model with its three high-level features: 1) container data types which simplify the memory management as data is automatically allocated and transfered to and from GPUs; 2) algorithmic skeletons which capture common programming patterns and hide the complexities of parallel programming from the user; and 3) data distributions which allow the user to specify how data should be distributed across GPUs and, thus, simplifies the programming of multi-GPU systems.

In this chapter we also introduced two novel algorithmic skeletons targeted towards stencil and allpairs computations. For both skeletons we provided a formal definition, as well as implementations for

single- and multi-GPU systems. For the allpairs skeleton we identified a specialization rule, which enables an optimized implementation on GPUs as it reduces the amount of global memory accesses.

The SkelCL programming model and library address the programmability challenge. In the next chapter we will evaluate the raised level of programmability and the performance of SkelCL by implementing a set of benchmark applications and performing runtime experiments with them.

4 APPLICATION STUDIES

In this chapter we present various application studies evaluating the usefulness and performance of the abstractions introduced by the SkelCL programming model which were presented in the previous chapter. We start with a brief discussion of the metrics used to evaluate the SkelCL programming model and discuss the experimental setup used throughout the chapter. We will then look at applications from a wide range of domains ranging from simple benchmark applications like the computation of the Mandelbrot set, over linear algebra and image processing applications, to real-world applications in medical imaging, and physics.

For all source codes we only show the relevant code sections and omit implementation details like initializing SkelCL.

4.1 EXPERIMENTAL SETUP

This section briefly discusses the evaluation metrics used in this chapter which were chosen to measure the quality of the abstractions introduced in the SkelCL programming model and their implementation in the SkelCL library. We will also discuss the hardware used in the experiments.

4.1.1 *Evaluation Metrics*

We want to evaluate programmability, i. e., the ease of programming, and the performance of the SkelCL programming model and library.

Measuring programmability is difficult. Various studies [87, 88] have been conducted and metrics [153] have been proposed to measure how convenient a certain style of programming or a certain programming model is. None of these metrics is widely established in the scientific or industrial communities. We chose to use one of the simples metrics possible to quantify programming effort: counting the *Lines Of source Code* (LOC). The author wants to emphasize that this metric is not always a good representation of programmability. A shorter program does not necessarily mean that the development of the program has been more convenient. We will, therefore, alongside presenting the lines of code argue *why* SkelCL simplifies the programming of parallel devices, like GPUs, as compared to the state-of-the-art approach OpenCL.

As the metric for performance we use absolute and relative runtime. We only make comparisons using software executed on the same hardware. We perform comparisons using published application and benchmark software from researchers or officially provided by Nvidia or AMD. In addition, we compare self developed and optimized OpenCL code versus code written using the SkelCL library.

For all measurements we performed 100 runs and report the median runtime.

4.1.2 *Hardware Setup*

For performing our runtime experiments we used a PC equipped with a quad-core CPU (Intel Xeon E5520, 2.26 GHz) and 12 GB of main memory. The system is connected to a Nvidia Tesla S1070 computing system consisting of four Nvidia Tesla GPUs. The S1070 has 16 GB of dedicated memory (4 GB per GPU) which is accessed with up to 408 GB/s (102 GB/s per GPU). Each GPU comprises 240 streaming processor cores running at up to 1.44 GHz. The Linux based Ubuntu operating system was used. The runtime experiments where conducted at different times from 2010 until 2015. We used the following GPU driver versions and versions of the CUDA toolkit: 258.19 and CUDA toolkit 3.2 for the Mandelbrot set computation and the medical imaging application, 304.54 and CUDA toolkit 5.0 for the Matrix Multiplication application, 319.37 and CUDA toolkit 5.5 for the image processing applications, and, finally, 331.62 and CUDA toolkit 6.0 for the linear algebra applications and the physics simulation.

This hardware setup represents a common heterogeneous system comprising four CPU cores and 960 GPU streaming processor cores with a total of 28 GB of memory. Overall this system has a theoretical single-precision floating point performance of 4219.52 GFLOPS ($4 \times$ 1036.8 GFLOPS for the GPUs and 72.32 GFLOPS for the CPU).

4.2 COMPUTATION OF THE MANDELBROT SET

The Mandelbrot [111] set includes all complex numbers $c \in \mathbb{C}$ for which the sequence

$$z_{i+1} = z_i^2 + c, \qquad i \in \mathbb{N} \tag{4.1}$$

starting with $z_0 = 0$ does not escape to infinity. When drawn as an image with each pixel representing a complex number, the boundary of the Mandelbrot set forms a fractal. Figure 4.1 shows an image visualizing part of the Mandelbrot set. The software producing the image was implemented using the SkelCL library. The calculation of such an image is a time-consuming task, because the sequence given by Equation (4.1) has to be calculated for every pixel. If this sequence does not

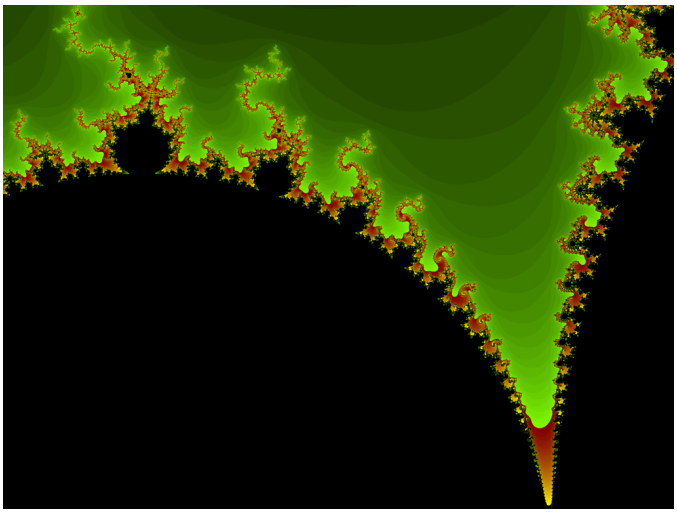


Figure 4.1: Visualization of a part of the Mandelbrot set. The image was produced using the SkelCL library.

cross a given threshold for a given number of iteration steps, it is presumed that the sequence will converge. The respective pixel is thus taken as a member of the Mandelbrot set, and it is displayed black. Other pixels outside are assigned a color that corresponds to the number of iterations of the sequence given by Equation (4.1). Computing a Mandelbrot fractal is easily parallelizable, as all pixels of the fractal can be computed simultaneously.

SkelCL Implementation

The Mandelbrot set computation expressed in the SkelCL programming model uses the *map* skeleton as shown in Equation (4.2).

$$\begin{aligned} &\mathtt{mandelbrot}\ w\ h = \\ &\quad \mathtt{map}\ \mathtt{computeColorOfPixel}\ (\mathtt{generateIndex}\ w\ h) \end{aligned} \tag{4.2}$$

Here a function `mandelbrot` is defined taking two arguments – the width w and height h of the image to compute. The function is defined in terms of the *map* skeleton which is customized with the `computeColorOfPixel` function computing the color of a single pixel of the image following Equation (4.1) (not shown here) and operating on an input matrix of size $w \times h$ consisting of indices. This input matrix is generated by the `generateIndex` function which – given a width w and height h – produces a matrix containing all combinations of index-pairs (x, y) with $x < w$ and $y < h$.

The implementation of the Mandelbrot set computation using the SkelCL library is shown in Listing 4.1. A user-defined data type is introduced to represent the color of a pixel (line 1). An instance of the *map* skeleton is created in line 6 and applied to `IndexMatrix` (line 7). The `IndexMatrix` represents all indices up to a given width

```
1 typedef struct { char r; char g; char b; } Pixel;
2
3 Pixel computeColorOfPixel(IndexPoint) { ... };
4
5 void mandelbrot(const int width, const int height) {
6   auto m = map(computeColorOfPixel);
7   auto image = m(IndexMatrix{w, h});
8   writeToDisk(image); }
```

Listing 4.1: Implementation of the Mandelbrot set computation in SkelCL

and height. It is implemented as a special representation of the more generic `Matrix` class avoiding the explicit storing of the indices in memory. Instead when accessing an element the index value is computed on the fly. This implementation avoids allocation of memory for storing the indices and transferring them to the GPU.

We created a similar parallel implementation for computing a Mandelbrot fractal using OpenCL. We compare the programming effort and performance for this implementation against our SkelCL-based implementation.

Programming effort

SkelCL require a single line of code for initialization in the host code, whereas OpenCL requires a lengthy initialization of different data structures which takes about 20 lines of code.

The host code differs significantly between both implementations. In OpenCL, several API functions are called to load and build the kernel, pass arguments to it and to launch it using a specified work-group size. In SkelCL, the *map* skeleton is used to compute the color of all pixels in the image. An `IndexMatrix` representing complex numbers, each of which is processed to produce a pixel of the Mandelbrot fractal, is passed to the *map* skeleton upon execution. Specifying the work-group size is mandatory in OpenCL, whereas this is optional in SkelCL.

PROGRAM SIZE The OpenCL-based implementation has 118 lines of code (kernel: 28 lines, host program: 90 lines) and is thus more than twice as long as the SkelCL versions with 57 lines (26, 31) (see Figure 4.2).

KERNEL SIZE The kernel function is similar in both implementations: it takes a pixel's position (i. e., a complex number) as input, performs the iterative calculation for this pixel, and returns the pixel's color. However, while the input positions are given explicitly when using the *map* skeleton in SkelCL, no positions are passed to the ker-

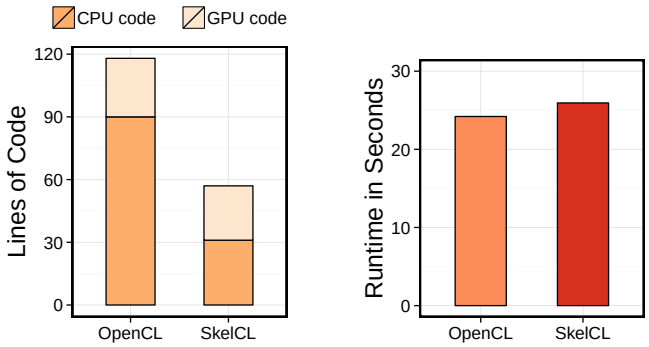


Figure 4.2: Runtime and program size of the Mandelbrot application.

nel in the OpenCL-based implementation. The positions are implicitly determined based on the work-item's index.

Performance experiments

We tested our implementations on a single GPU of our test system to compute a Mandelbrot fractal of size 4096×3072 pixels. In OpenCL, work-groups of 16×16 are used; SkelCL uses its default work-group size of 256 work-items.

The results are shown in Figure 4.2. As compared to the runtime of the SkelCL-based implementation (26 seconds), the implementation based on OpenCL (25 seconds) is faster by 4%. Since SkelCL is built on top of OpenCL, the performance difference of SkelCL and OpenCL can be regarded as the overhead introduced by SkelCL. The Mandelbrot application demonstrates that SkelCL introduces a tolerable overhead of less than 5% as compared to OpenCL. A clear benefit of this overhead is the reduced programming effort required by the SkelCL program.

4.3 LINEAR ALGEBRA APPLICATIONS

In this section we are going to evaluate SkelCL using two basic linear algebra applications:

- the sum of absolute values of a vector and
- the dot product of two vectors.

Both applications are included in BLAS [55, 56], the well known library of basic linear algebra functions. The BLAS functions are used as basic building blocks by many high-performance computing applications.

Here we want to investigate how easily these applications can be expressed with skeletons in SkelCL. Furthermore, we are interested in the runtime performance of the SkelCL implementations.

Sum of Absolute Values

Equation (4.3) shows the mathematical definition of the sum of absolute values (short *asum*) for a vector $\vec{x}$ of length n with elements x_i:

$$\mathrm{asum}\ \vec{x} = \sum_{i=0}^{n} |x_i| \tag{4.3}$$

For all elements of the vector the absolute values are added up to produce the final scalar result.

SkelCL Implementation

In the SkelCL programming model we can express *asum* using the *map* and *reduce* skeletons as follows:

$$\mathrm{asum}\ \vec{x} = \mathrm{reduce}\ (+)\ 0\ (\ \mathrm{map}\ (|\,.\,|)\ \vec{x}\ ) \tag{4.4}$$

$$\text{where:} \quad |a| = \begin{cases} a & \text{if } a \geqslant 0 \\ -a & \text{if } a < 0 \end{cases}$$

The *map* skeleton applies the $(|\,.\,|)$ function to each element of the input vector before the *reduce* skeleton is used to sum up the elements.

The implementation of asum using the SkelCL library is shown in Listing 4.2. In line 3—line 6 the customized skeletons are defined. The *map* skeleton in Equation (4.4) corresponds directly to line 3 and line 4 in Listing 4.2 where the $|\,.\,|$ function is represented using a C++ lambda expression. The line 5 and line 6 correspond directly to the *reduce* skeleton in Equation (4.4). By applying the skeletons to the input vector (line 7) the result is computed and accessed in line 7. In the SkelCL library implementation the *reduce* skeleton returns a vector containing a single element, called `result` in this particular case. The containers in the SkelCL library are implemented as *futures* [67, 85]. This allows the computation of all skeletons to be performed asynchronously, i. e., when executing a skeleton the computation is launched and the called function returns immediately. When accessing values of the returned container, e. g., via the array subscript operator as shown in line 7, the call will block until the accessed value has been computed. Here we could have also used the equivalent `front` member function for accessing the first element of the result vector.

```
1 float asum(const Vector<float>& x) {
2   skelcl::init();
3   auto absAll = map(
4       [](float a){ if (a >= 0) return a; else return -a; });
5   auto sumUp = reduce(
6       [](float a, float b){return a+b;}, 0);
7   auto result = sumUp( absAll( x ) ); return result[0]; }
```

Listing 4.2: Implementation of the *asum* application in SkelCL

```
1 float dotProduct(const Vector<float>& x,
                   const Vector<float>& y) {
2   skelcl::init();
3   auto mult  = zip(
4       [](float x, float y){return x*y;});
5   auto sumUp = reduce(
6       [](float x, float y){return x+y;}, 0);
7   return sumUp( mult( x, y ) ).front(); }
```

Listing 4.3: Implementation of the dot product application in SkelCL

Dot Product

The computation of the dot product, a. k. a., scalar product, is a common mathematical operation performed on two input vectors $\vec{x}$ and $\vec{y}$ of identical length n as defined in Equation (4.5):

$$\text{dotProduct}\ \vec{x}\ \vec{y} = \sum_{i=0}^{n} x_i \times y_i \tag{4.5}$$

SkelCL Implementation

In the SkelCL programming model we can express dotProduct using the *zip* and *reduce* skeletons as follows:

$$\text{dotProduct}\ \vec{x}\ \vec{y} = \text{reduce}\ (+)\ 0\ (\ \text{zip}\ (\times)\ \vec{x}\ \vec{y}\ ) \tag{4.6}$$

The *zip* skeleton performs pairwise multiplication of the input vectors before the *reduce* skeleton is used to sum up the intermediate results.

Listing 4.3 shows the implementation using the SkelCL library. The structure of the implementation is very similar to the *asum* application. Here we use the `front` member function to access the first (and only) element of the computed result vector (line 7).

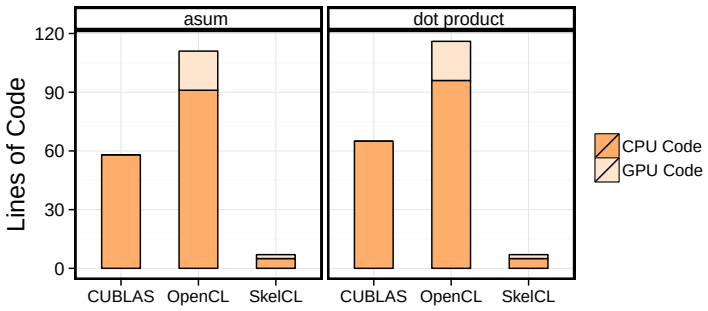


Figure 4.3: Lines of code for both basic linear algebra applications

We now compare the SkelCL implementations shown in Listing 4.2 and Listing 4.3 against naïve OpenCL implementations and implementations using the CUBLAS library respectively.

Programming effort

Figure 4.3 shows the lines of code for the *asum* application on the left and for the dot product application on the right. The CUBLAS implementations requires 58 respectively 65 lines in total, whereby no GPU code is written directly. The code contains a lengthy setup, memory allocations, memory transfers and launching the computation on the GPU including all necessary error checks. The naïve OpenCL implementations require over 90 lines each (*asum*: 91, dot product: 96) for managing the GPU execution and 20 lines of GPU code. By far the shortest implementations are the SkelCL implementations with 7 lines each, as shown in Listing 4.2 and Listing 4.3. Here we count the lambda expressions customizing the algorithmic skeletons as GPU code and all other lines as CPU code. The code is not just shorter but each single line is clear and straightforward to understand, where, e. g., the CUBLASAPI call for launching the dot product computation alone expects 7 arguments requiring the programmer to look up the documentation for using it correctly.

Performance experiments

Figure 4.4 shows the runtime for the *asum* application and the runtime for the computation of the dot product. The figure uses a logarithmic scale on the vertical axis and shows the runtime for different sizes of the input vectors: from 4 megabyte to 512 megabyte. These results include data transfers to and from the GPU as well as the computation performed on the GPU.

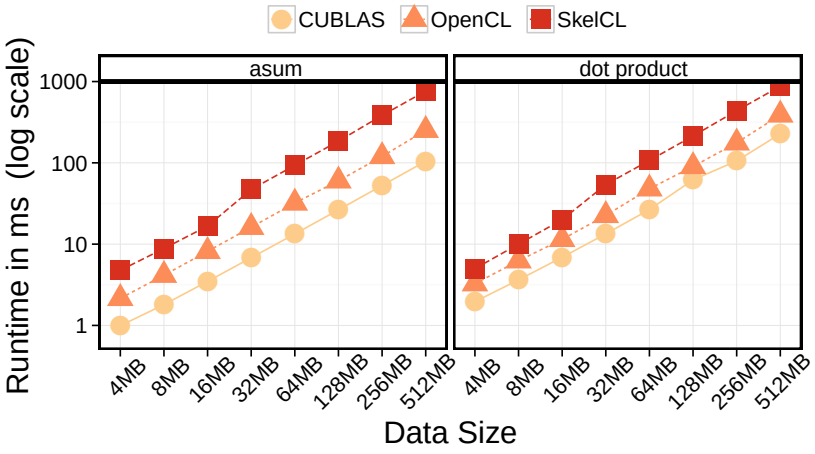


Figure 4.4: Runtime for both basic linear algebra applications

The results clearly show that the SkelCL version is the slowest, followed by the naïve OpenCL implementation, and, finally, the implementation using CUBLAS which is the fastest. For both applications the performance difference between the versions is roughly similar independent of the input size. For *asum* SkelCL is between 2 and 3.2 times slower than the naïve OpenCL version and up to 7 times slower than the CUBLAS version. For dot product SkelCL is on average about 2.25 times slower than the naïve OpenCL version and between 3.5 and 4 times slower than CUBLAS. We will discuss the reasons for the bad performance of the SkelCL implementation in the next subsection.

The CUBLAS version is about 2 times faster than the naïve OpenCL version for *asum* and 1.7 times for the dot product application. One has to keep in mind, that these benchmark are mostly dominated by the memory transfers moving data between CPU and GPU which are the same in both version. This explains the rather small difference in runtime between the naïve OpenCL version and the optimized CUBLAS version.

Discussion

As discussed in Chapter 3 the SkelCL library implementation generates one OpenCL kernel for each skeleton (and two for the *reduce* skeleton). This procedure makes it difficult to *fuse* multiple skeleton implementations into a single OpenCL kernel, which would be required to achieve a competitive performance for the *asum* and dot product benchmark.

To validate this explanation and quantify the effect of launching additional kernels where a single kernel would be sufficient we in-

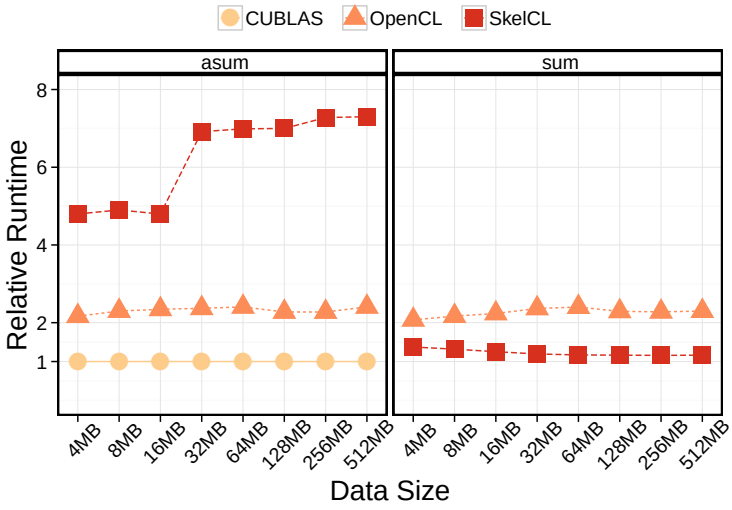


Figure 4.5: Runtime of the naïve OpenCL and SkelCL versions of *asum* and *sum* compared to CUBLAS.

vestigate the performance of SkelCL's *reduce* skeleton on its own. This is similar to the *asum* benchmark, but without applying the *map* skeleton for obtaining the absolute value of every element, i. e., we measure the *sum* benchmark:

$$\text{sum}\ \vec{x} = \text{reduce}\ (+)\ 0\ \vec{x} \tag{4.7}$$

For comparison we also modified the naïve OpenCL implementation of *asum* to not apply the absolute value function to every element of the input vector. Figure 4.5 shows the performance results of the naïve OpenCL version and the SkelCL version for *asum* on the left and *sum* on the right. The plots show the performance relative to the performance of the CUBLAS implementation of *asum*. We can see, that the performance difference for *asum* and *sum* it is very small (only up to 4%) for the naïve OpenCL version. This makes sense, as the most part of the computation is spend for reducing the elements and not for applying a simple function to each element in parallel. For the SkelCL implementations the difference between the two benchmarks is very large. This is due to the fact that for *asum* a separate OpenCL kernel is launched which reads each value from the global memory, applies the absolute value function to it, and stores the result back to global memory. The reduction kernel then starts by reading each element back from global memory. This leads to a huge performance penalty, where the *asum* benchmark is up to 6.2 times slower than the *sum* benchmark. Furthermore, we can see that for the *sum* benchmark the SkelCL implementation outperforms the naïve OpenCL implementa-

tion and is only between 16% and 37% slower than the CUBLAS *asum* implementation.

In Chapter 5, we will discuss a novel compilation technique which addresses this drawback of SkelCL. This technique supports the generation of a single efficient OpenCL kernel for applications like the *asum* and dot product examples.

4.4 MATRIX MULTIPLICATION

The multiplication of matrices is a fundamental building block for many scientific applications. A $n \times d$ matrix A is multiplied with a $d \times m$ matrix B to produce a $n \times m$ matrix C, where the elements of C are computed as:

$$C_{ij} = \sum_{k=0}^{d} A_{ik} \times B_{kj}, \qquad \forall\, i \in 1,\ldots,n \wedge j \in 1,\ldots,m$$

Here A_{i*} refers to the ith row of A and B_{*j} to the jth column of B. Figure 4.6 visualizes this computation. To compute the highlighted element in matrix C, the highlighted row of matrix A is combined with the highlighted column of matrix B. For computing the entire matrix C, *all pairs* of rows from A and columns of B have to be processed. Therefore, in SkelCL the *allpairs* skeleton can be used to express matrix multiplication.

SkelCL Implementation

Equation (4.8) shows how matrix multiplication can be expressed using the *allpairs* skeleton in the SkelCL programming model:

$$mm\ A\ B = allpairs\ f\ A\ B^T \tag{4.8}$$

$$\text{where:} \qquad f\ \vec{a}\ \vec{b} = \sum_{k=0}^{d} a_k \times b_k$$

When looking back at Equation (4.5) in the previous section, we can see, that f is actually the dot product computation, therefore, we can write:

$$mm\ A\ B = allpairs\ dotProduct\ A\ B^T \tag{4.9}$$

We know that we can express the dot product as a sequential composition of the *zip* and *reduce* skeletons as we saw in the previous section. In Chapter 3, Section 3.2.4, we discussed a specialized implementation of the *allpairs* skeleton for computations which can be expressed in this way. Therefore, we can use the SkelCL library to develop two implementations: 1) using the generic *allpairs* skeleton; and 2) using the specialized *allpairs* skeleton.

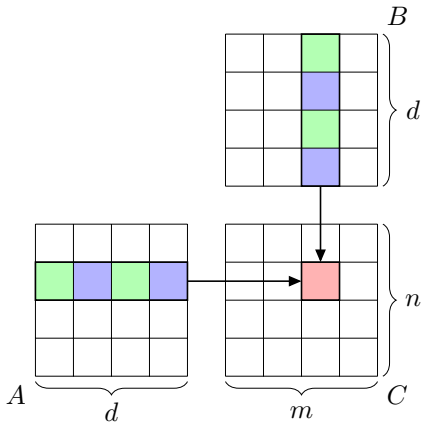


Figure 4.6: Matrix multiplication A × B = C. The red highlighted element in matrix C is computed by combining the highlighted row of matrix A with the highlighted column of matrix B.

```
1  Matrix<float> mm(const Matrix<float>& A,
2                   const Matrix<float>& B) {
3    skelcl::init();
4    auto mm = allpairs(
5      [](const Vector<float>& a, const Vector<float>& b) {
6        float c = 0.0f;
7        for (int i = 0; i < a.size(); ++i)
8          c += a[i] * b[i];
9        return c; });
10   return mm(A, B); }
```

Listing 4.4: Implementation of matrix multiplication using the generic *allpairs* skeleton in SkelCL.

Listing 4.4 shows the implementation of matrix multiplication using the generic *allpairs* skeleton. The skeleton is customized with a lambda expression processing two vectors: `a` is a row vector of matrix A and `b` is a column vector of matrix B. In this generic implementation the dot product computation is implemented using a `for` loop iterating over the vectors, multiplying elements pairwise and summing them up in the accumulation variable `c`.

Listing 4.5 shows the implementation of matrix multiplication using the specialized *allpairs* skeleton. Here the *allpairs* skeleton is customized with *zip* and *reduce* skeletons defined in line 4 and line 6. This implementation corresponds more closely to Equation (4.9): as we express the dot product using these two skeletons (as shown in Equation (4.6)). Therefore, we reuse the definitions of `mult` and `sumUp` as used in Listing 4.3.

```
Matrix<float> mm(const Matrix<float>& A,
                 const Matrix<float>& B) {
  skelcl::init();
  auto mult  = zipVector(
      [](float x, float y){return x*y;});
  auto sumUp = reduce(
      [](float x, float y){return x+y;}, 0);
  auto mm    = allpairs(sumUp, mult);
  return mm(A, B); }
```

Listing 4.5: Implementation of matrix multiplication using the specialized *allpairs* skeleton in SkelCL.

```
kernel void mm(global float* A, global float* B,
               global float* C, int m, int d, int n) {
  int row = get_global_id(0);
  int col = get_global_id(1);
  float sum = 0.0f;
  for (int k = 0; k < d; k++)
    sum += A[row * d + k] * B[k * n + col];
  C[row * n + col] = sum; }
```

Listing 4.6: OpenCL kernel of matrix multiplication without optimizations [99].

Implementations used for comparison

We compare six different implementations of matrix multiplication:

1. the OpenCL implementation from [99] without optimizations,
2. the optimized OpenCL implementation from [99] using GPU local memory,
3. the optimized BLAS implementation by AMD [2] written in OpenCL (clBLAS version 1.10),
4. the optimized BLAS implementation by Nvidia [43] written in CUDA (CUBLAS version 5.0),
5. the SkelCL implementation using the generic *allpairs* skeleton shown in Listing 4.4,
6. the SkelCL implementation using the specialized *allpairs* skeleton shown in Listing 4.5.

1. OPENCL IMPLEMENTATION Listing 4.6 shows the kernel of the first, unoptimized OpenCL implementation from [99].

```
1  #define T_WIDTH 16
2  kernel void mm(global float* A, global float* B,
                  global float* C, int m, int d, int n) {
3    local float Al[T_WIDTH][T_WIDTH];
4    local float Bl[T_WIDTH][T_WIDTH];
5    int row = get_global_id(0);
6    int col = get_global_id(1);
7    int l_row = get_local_id(0);
8    int l_col = get_local_id(1);
9    float sum = 0.0f;
10   for (int m = 0; m < d / T_WIDTH; ++m {
11     Al[l_row][l_col] = A[row * d + (m * T_WIDTH + l_col)];
12     Bl[l_row][l_col] = B[(m * T_WIDTH + l_row) * d + col];
13     barrier(CLK_LOCAL_MEM_FENCE);
14     for (int k = 0; k < T_WIDTH; k++)
15       sum += Al[l_row][k] * Bl[k][l_col];
16     barrier(CLK_LOCAL_MEM_FENCE); }
17   C[row * n + col] = sum; }
```

Listing 4.7: OpenCL kernel of the optimized matrix multiplication using local memory [99].

2. OPTIMIZED OPENCL IMPLEMENTATIONS The kernel of the optimized OpenCL implementation from [99] using local memory is shown in Listing 4.7. Two fixed-sized arrays of local memory are allocated in line 3 and line 4. Matrix multiplication is carried out in the loop starting in line 10. In each iteration, data is loaded into the local memory (line 11 and line 12) before it is used in the computation in line 15. Note that two synchronization barriers are required (line 13 and line 16) to ensure that the data is fully loaded into the local memory and that the data is not overwritten while other work-items are still using it.

Both OpenCL implementations 1. and 2. from [99] are restrictive: they are only capable of performing matrix multiplication for square matrices.

3. BLAS IMPLEMENTATION BY AMD The implementation offered by AMD is called clBLAS, written in OpenCL and is part of their Accelerated Parallel Processing Math Libraries (APPML) [2].

4. BLAS IMPLEMENTATION BY NVIDIA CUBLAS [43] is implemented using CUDA and, therefore, can only be used on GPUs built by Nvidia.

Programming effort

Figure 4.7 shows the comparison regarding the number of lines of code (LOCs) required for each of the six implementations. Figure 4.1 presents the detailed numbers. We did not count those LOCs which are not relevant for parallelization and are similar in all six implementations, like initializing the input matrices with data and checking the result for correctness. For every implementation, we distinguish between CPU (host) code and GPU (kernel) code.

In the OpenCL implementations, the GPU code includes the kernel definition, as shown in Listing 4.6 and Listing 4.7; the CPU code includes the initialization of OpenCL, memory allocations, explicit data transfer operations, and management of the execution of the kernel.

In the BLAS implementations, the CPU code contains the initialization of the corresponding BLAS library, memory allocations, as well as a library call for performing the matrix multiplication; no separate definition of GPU code is necessary, as the GPU code is defined inside the library function calls.

For the implementation based on the generic *allpairs* skeleton (Listing 4.4), we count line 1—line 4 and line 10 as the CPU code, and the definition of the customizing function in line 5—line 9 as the GPU code. For the implementation based on the specialized *allpairs* skeleton (Listing 4.5), line 5 and line 7 are the GPU code, while all other lines constitute the CPU code.

Both skeleton-based implementations are clearly the shortest, with 10 and 9 LOCs. The next shortest implementation is the CUBLAS implementation with 65 LOCs – 7 times longer than the SkelCL-based implementations. The other implementations using OpenCL require even 9 times more LOCs than the SkelCL-based implementations.

Besides their length, the OpenCL-based implementations require the application developer to explicitly implement many low-level, error-prone tasks, like dealing with pointers and offset calculations. Furthermore, the skeleton-based implementations are more general, as they can be used for arbitrary allpairs computations, while all other implementations can compute matrix multiplication only.

Performance experiments

We performed experiments with the six different implementations of matrix multiplication on two different computer systems with GPUs:

System A: Our general testing system already described in Section 4.1: an Nvidia S1070 equipped with four Nvidia Tesla GPUs, each with 240 streaming processors and 4 GByte memory.

System B: An AMD Radeon HD 6990 card containing two GPUs, each with 1536 streaming processors and 1 GByte memory.

We include the data transfer times to and from the GPU in the results.

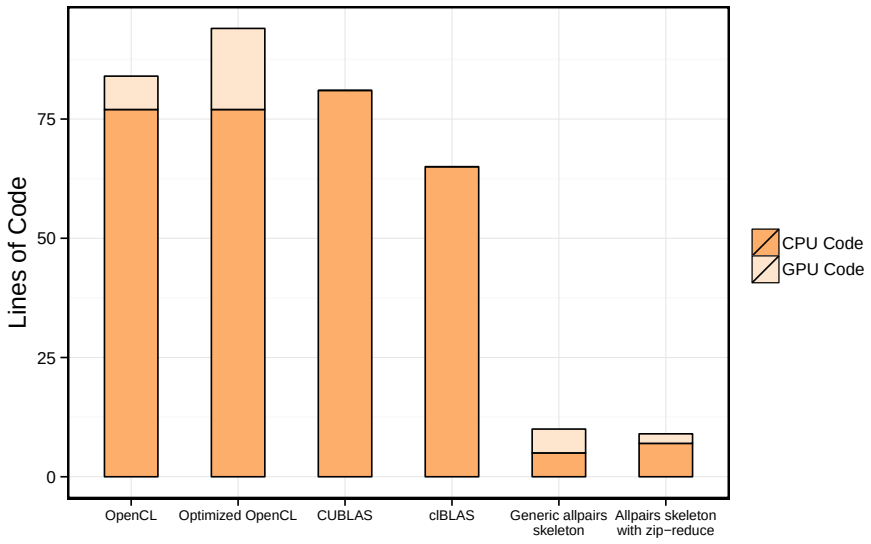


Figure 4.7: Programming effort (Lines of Code) of three OpenCL-based, and one CUDA-based vs. two SkelCL-based implementations.

	Lines of Code		
Implementation	CPU	GPU	Total
OpenCL	77	7	84
Optimized OpenCL	77	17	94
CUBLAS	81	–	81
clBLAS	65	–	65
Generic *allpairs* skeleton	5	5	10
Specialized *allpairs* skeleton	7	2	9

Table 4.1: Lines of Code of all compared implementations.

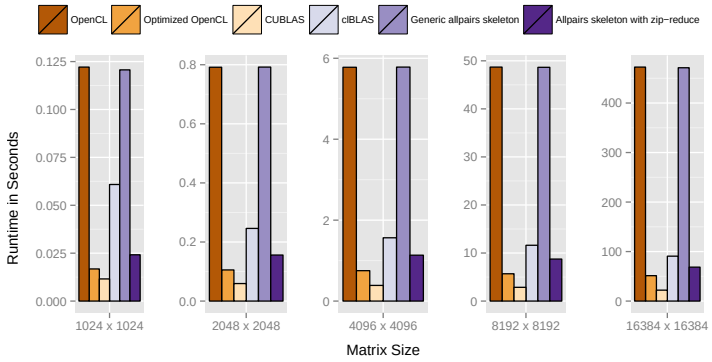


Figure 4.8: Runtime of different matrix multiplication implementations on the Nvidia system for different sizes of the matrices.

Implementation	Runtimes in Seconds				
	1024 ×1024	2048 ×2048	4096 ×4096	8192 ×8192	16384 ×16384
OpenCL	0.122	0.791	5.778	48.682	472.557
Optimized OpenCL	0.017	0.105	0.752	5.683	51.337
CUBLAS	0.012	0.059	0.387	2.863	22.067
clBLAS	0.061	0.246	1.564	11.615	90.705
Generic *allpairs* skeleton	0.121	0.792	5.782	48.645	471.235
Specialized *allpairs* skeleton	0.024	0.156	1.134	8.742	68.544

Table 4.2: Runtime results for matrix multiplication on the Nvidia system.

SYSTEM A (ONE GPU) Figure 4.8 shows the runtime of all six implementations for different sizes of the matrices (for readability reasons, all charts are scaled differently). For detailed numbers, see Table 4.2.

Clearly, the naive OpenCL implementation and the implementation using the generic *allpairs* skeleton are the slowest, because both do not use local memory, in contrast to all other implementations.

The implementation using the specialized *allpairs* skeleton performs 5.0 to 6.8 times faster than the generic allpairs skeleton, but is 33% slower on the largest matrices than the optimized OpenCL-based implementation. However, the latter implementation can only be used for square matrices and, therefore, it benefits from omitting many conditional statements and boundary checks.

CUBLAS is the fastest of all implementations, as it is highly tuned specifically for Nvidia GPUs using CUDA. The clBLAS implementation by AMD using OpenCL performs not as well: presumably, it is optimized for AMD GPUs and performs poorly on other hardware. Our optimized *allpairs* skeleton implementation outperforms the clBLAS implementation for all matrix sizes tested.

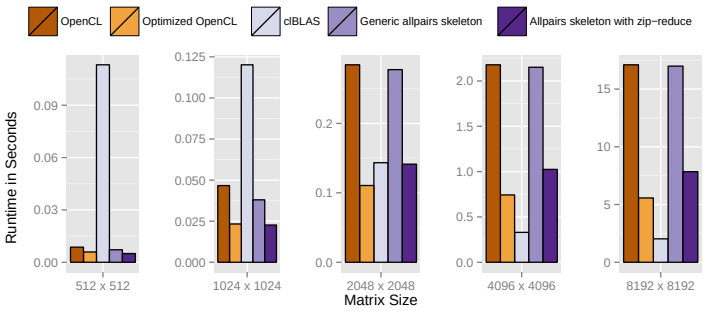


Figure 4.9: Runtime of all compared implementations for a matrix multiplication on the AMD system using one GPU.

Implementation	Runtimes in Seconds				
	512 ×512	1024 ×1024	2048 ×2048	4096 ×4096	8192 ×8192
OpenCL	0.008	0.046	0.284	2.178	17.098
Optimized OpenCL	0.006	0.023	0.111	0.743	5.569
clBLAS	0.113	0.120	0.143	0.329	2.029
Generic *allpairs* skeleton	0.007	0.038	0.278	2.151	16.983
Specialized *allpairs* skeleton	0.005	0.023	0.141	1.025	7.842

Table 4.3: Runtime results for all tested implementations of matrix multiplication on the AMD system.

SYSTEM B (ONE GPU) Figure 4.9 shows the measured runtime in seconds for five of the six implementations for different sizes of the matrices. Detailed numbers can be found in Table 4.3. We could not use the Nvidia-specific CUBLAS implementation as it does not work on the AMD GPU.

For bigger matrices, the slowest implementations are, again, the unoptimized OpenCL implementation and the implementation using the generic *allpairs* skeleton.

The optimized OpenCL implementation and the specialized *allpairs* skeleton perform similarly. For matrices of size 8192×8192, the optimized OpenCL implementation is about 30% faster.

The clBLAS implementation performs very poorly for small matrices, but is clearly the fastest implementation for bigger matrices. Similar to the CUBLAS implementation on the Nvidia hardware, it is not surprising that the implementation by AMD performs very well on their own hardware.

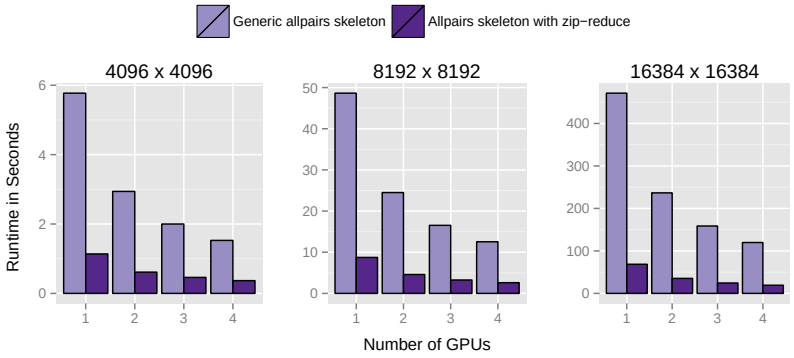


Figure 4.10: Runtime of the *allpairs* based implementations using multiple GPUs.

Implementation	Number of GPUs	Runtimes in Seconds 4096 ×4096	8192 ×8192	16384 ×16384	GFlops 16384 ×16384
Generic *allpairs* skeleton	1 GPU	5.772	48.645	471.328	18.72
	2 GPUs	2.940	24.495	236.628	37.43
	3 GPUs	2.000	16.532	158.611	56.17
	4 GPUs	1.527	12.540	119.786	74.90
Specialized *allpairs* skeleton	1 GPU	1.137	8.740	68.573	130.93
	2 GPUs	0.613	4.588	35.294	262.18
	3 GPUs	0.461	3.254	24.447	392.87
	4 GPUs	0.368	2.602	19.198	523.91

Table 4.4: Runtime of the *allpairs* based implementations of matrix multiplication using multiple GPUs. For the matrices of size 16384 × 16384 the results are also shown in GFlops.

SYSTEM A (MULTIPLE GPUS) Figure 4.10 shows the runtime behavior for both *allpairs* skeleton-based implementations when using up to four GPUs of our multi-GPU system. The other four implementations are not able to handle multiple GPUs and would have to be specially rewritten for such systems. Newer versions of Nvidia's CUBLAS implementation support the execution on multiple GPUs as well. We observe a good scalability for both of our skeleton-based implementations, achieving speedups between 3.09 and 3.93 when using four GPUs. Detailed numbers can be found in Table 4.4. For the matrices of size 16384 × 16384, performance is also provided in GFlops; to compute this value we excluded the data-transfer time (as usually done in related work) to enable better comparison with related work.

(a) The Lena image with noise.

(b) The Lena image after applying the Gaussian blur.

Figure 4.11: Application of the Gaussian blur to an noised image.

4.5 IMAGE PROCESSING APPLICATIONS

Many image processing applications are inherently parallel as they often independently process the pixels of an image. Common examples range from simple thresholding over noise reduction applications to techniques used in edge detection and pattern recognition [152]. In this section we will study three application examples from image processing and how they can be implemented using SkelCL.

We start by looking at the Gaussian blur application which can be used to reduce noise in images and is often used as a preprocessing step in more complex algorithms. We will then discuss two algorithms used for edge detection in images: the Sobel edge detection application and the more complex Canny edge detection algorithm.

These three applications can all be expressed using the *stencil* skeleton introduced in Section 3.2.4, but they have different characteristics. The Gaussian blur applies a single stencil computation, possibly iterated multiple times, for reducing the noise in images. The Sobel edge detection applies a stencil computation once to detect edges in images. The more advanced Canny edge detection algorithm consists of a sequence of stencil operations which are applied to obtain the final result. For each of the three applications, we compare the performance using our two implementations of the *stencil* skeleton: `MapOverlap` and `Stencil` with native OpenCL implementations using an input image of size 4096×3072.

4.5.1 *Gaussian Blur*

The Gaussian blur is a standard algorithm used in image processing [152]. One common application is reduction of image noise as shown in Figure 4.11. The image on the left has some noise as it is

typically produced by halftone printing used to print newspapers. The Gaussian blur has been applied to reduce the noise and produce the image on the right.

The Gaussian blur computes a weighted average for every pixel based on the neighboring pixels color values. Using SkelCL this application can easily be expressed using the *stencil* skeleton.

SkelCL Implementation

Equation (4.10) shows the Gaussian blur expressed in the SkelCL programming model using the *stencil* skeleton.

$$
\begin{aligned}
&\mathrm{gauss}\ M = \mathrm{stencil}\ f\ 1\ \bar{0}\ M \qquad \text{where:}\\
&f\ \begin{bmatrix} M_{i-1,j-1} & M_{i-1,j} & M_{i-1,j+1} \\ M_{i,j-1} & M_{i,j} & M_{i,j+1} \\ M_{i+1,j-1} & M_{i+1,j} & M_{i+1,j+1} \end{bmatrix} = \\
&\qquad \frac{\sum_{k=-1}^{1} \sum_{l=-1}^{1} (G\ k\ l) \cdot M_{i+k,j+k}}{9},\\
&G\ x\ y = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}
\end{aligned}
\tag{4.10}
$$

and $\bar{0}$ is the constant function always returning zero.

The function G is the two-dimensional Gaussian function which is used in the customizing function f to weight the neighboring values $M_{i,j}$. The values obtained by applying G can be precomputed, as G is only evaluated with values in the interval $[-1, 1]$ for x and y.

Listing 4.8 shows the SkelCL-based implementation of the Gaussian blur using the `MapOverlap` implementation of the *stencil* skeleton. Here the immediate neighboring pixels are accessed (line 4—line 6) and used to compute a weighted value for each pixel. The function computing the weighted sum is omitted here. It is also possible to extend the *range* of the Gaussian blur and include more neighboring pixel values in the computation.

Programming effort

Figure 4.12 shows the program sizes (in lines of code) for the four implementations of the Gaussian blur. The application developer needs 57 lines of OpenCL host code and 13 LOCs for performing a Gaussian blur only using global memory. When using local memory, some more arguments are passed to the kernel, thus, increasing the host-LOCs to 65, while the LOCs for the kernel function, which copies all necessary elements for a work-group's calculation into local memory, requires 88 LOCs including explicit out-of-bounds handling and complex index calculations. The `MapOverlap` and `Stencil` versions

```
1 Matrix<char> gaussianBlur(const Matrix<char>& image) {
2   auto gauss = mapOverlap(
3     [](Neighborhood<char>& in) {
4       char ul = in[{-1, -1}];
5       ...
6       char lr = in[{+1, +1}];
7       return computeGaussianBlur(ul, ..., lr); },
8     1, BorderHandling::NEUTRAL(0));
9   return gauss(image); }
```

Listing 4.8: Implementation of the Gaussian blur in SkelCL using the `MapOverlap` implementation of the *stencil* skeleton.

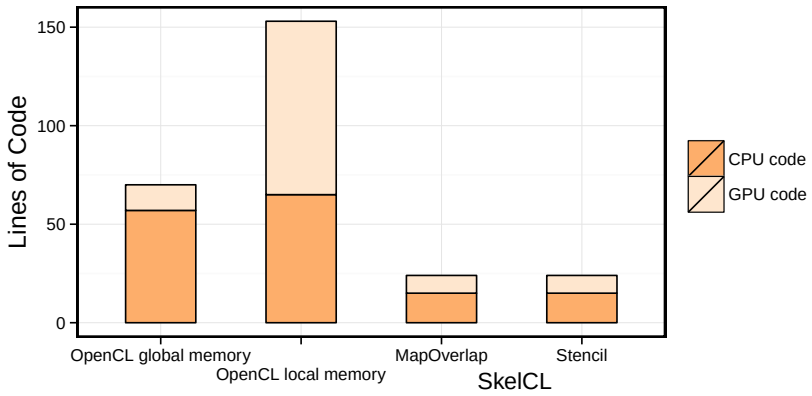


Figure 4.12: Lines of code of the Gaussian blur using a naïve OpenCL implementation with global memory, an optimized OpenCL version using local memory, and SkelCL's `MapOverlap` and `Stencil` implementations of the *stencil* skeleton.

are similar and both require only 15 LOCs host code and 9 LOCs kernel code to perform a Gaussian blur. The support for multi-GPU systems is implicitly provided when using SkelCL's skeletons, such that the kernel remains the same as for single-GPU systems. This is an important advantage of SkelCL over the OpenCL implementations of the Gaussian blur which are single-GPU only and require additional LOCs when adapting them for multi-GPU environments.

Performance experiments

Figure 4.13 shows the measured total runtime including data transfers of the Gaussian blur using:

1) a naïve OpenCL implementation using global memory,
2) an optimized OpenCL version using local memory,
3) the `MapOverlap`-based implementation, and

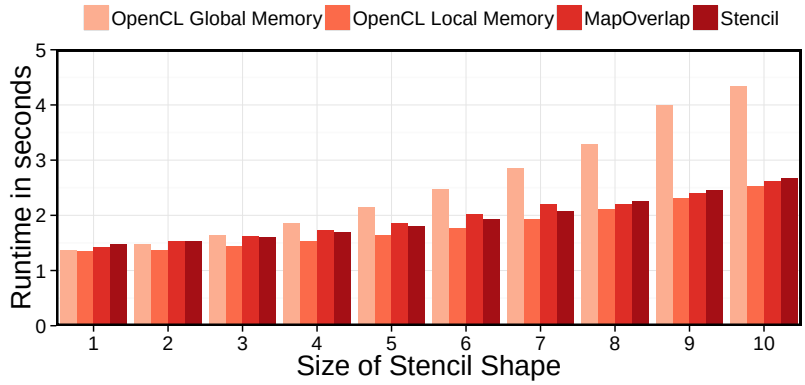


Figure 4.13: Runtime of the Gaussian blur using a naïve OpenCL implementation with global memory, an OpenCL version using local memory and SkelCL's MapOverlap and Stencil skeletons.

4) the `Stencil`-based implementation, correspondingly.

We can observe that for larger stencil shape sizes, the `MapOverlap` and `Stencil`-based versions outperform the naïve OpenCL implementation by 65% and 62%, respectively. The optimized OpenCL version, which copies all necessary elements into local memory prior to calculation, is 5% faster than `MapOverlap` and 10% faster than `Stencil` for small stencil shapes. When increasing the stencil shape size, this disadvantage is reduced to 3% for `MapOverlap` and 5% for `Stencil` with stencil shape's extent of 10 in each direction.

The `Stencil` implementation is slower for small stencil shapes than the `MapOverlap` implementation, up to 32% slower for an stencil shape size of 1. This is due to the increased branching required in the `Stencil` implementation, as discussed in more detail in Section 3.3.4.5. However, this disadvantage is reduced to 4.2% for the stencil shape size of 5 and becomes negligible for bigger stencil shape sizes. As the ratio of copying into local memory decreases in comparison to the number of calculations when enlarging the stencil shape's size, the kernel function's runtime in the `Stencil` implementation converges to the `MapOverlap` implementation's time. The `Stencil` implementation's disadvantage is also due to its ability to manage multiple stencil shapes and explicitly support the use of iterations. While both features are not used in this application example, they incur some overhead for the implementation as compared to the `MapOverlap` implementation for simple stencil computations.

Figure 4.14 shows the speedup achieved on the Gaussian blur using the `Stencil` implementation on up to four devices. The higher the computational complexity for increasing size of stencil shape, the better the overhead is hidden, leading to a maximum speedup of 1.90 for two devices, 2.66 for three devices, and 3.34 for four devices, for a large size of the stencil shape of 20.

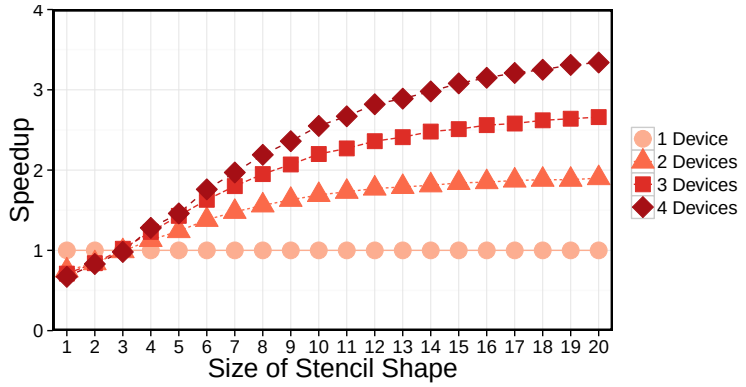


Figure 4.14: Speedup of the Gaussian blur application on up to four GPUs.

(a) Original image.

(b) Image after Sobel edge detection.

Figure 4.15: The Lena image [71] often used as an example in image processing before (left) and after (right) applying Sobel edge detection.

4.5.2 *Sobel Edge Detection*

The Sobel edge detection produces an output image in which the detected edges in the input image are marked in white and plain areas are shown in black. The effect is shown in Figure 4.15, where the original image is shown on the left and the output of Sobel edge detection applied to it on the right.

Listing 4.9 shows the sequential algorithm of the Sobel edge detection in pseudo-code, with omitted boundary checks for clarity. In this version, for computing an output value `out_img[i][j]` the input value `img[i][j]` and the direct neighboring elements are weighted and summed up horizontally and vertically. The *stencil* skeleton is a perfect fit for implementing the Sobel edge detection.

```
1 for (i = 0; i < width; ++i)
2   for (j = 0; j < height; ++j)
3     h = -1*img[i-1][j-1] +1*img[i+1][j-1]
4         -2*img[i-1][j  ] +2*img[i+1][j  ]
5         -1*img[i-1][j+1] +1*img[i+1][j+1];
6     v = ...;
7     out_img[i][j] = sqrt(h*h + v*v);
```

Listing 4.9: Sequential implementation of the Sobel edge detection.

SkelCL Implementation

Equation (4.11) shows the implementation of the Sobel edge detection in the SkelCL programming model.

$$\text{sobel } M = \text{stencil } f\ 1\ \bar{0}\ M \qquad \text{where:} \tag{4.11}$$

$$f \begin{bmatrix} M_{i-1,j-1} & M_{i-1,j} & M_{i-1,j+1} \\ M_{i,j-1} & M_{i,j} & M_{i,j+1} \\ M_{i+1,j-1} & M_{i+1,j} & M_{i+1,j+1} \end{bmatrix} = \sqrt{h^2 + v^2}$$

$$h = \sum_{k=0}^{2} \sum_{l=0}^{2} Gx_{k,l} \cdot M_{i+k-1,j+k-1}$$

$$v = \sum_{k=0}^{2} \sum_{l=0}^{2} Gy_{k,l} \cdot M_{i+k-1,j+k-1}$$

$$Gx = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad Gy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

and $\bar{0}$ is the constant function always returning 0.

The formula resembles the sequential implementation shown in Listing 4.9 where the final result is computed as the square root of the sum of two squared terms h and v. These are computed as weighted sums of the neighboring values $M_{i,j}$. The weights are given by the two matrices Gx and Gy.

Listing 4.10 shows the SkelCL implementation using the `MapOverlap` implementation of the *stencil* skeleton. The implementation is straightforward and very similar to the formula in Equation (4.11) and the sequential version in Listing 4.9.

Programming effort

Listing 4.11 shows a part of the rather simple OpenCL implementation for Sobel edge detection provided by AMD as an example application in their software development kit [11]. The actual computation is omitted in the listing, since it is quite similar to the sequential ver-

```
Matrix<char> sobelEdge(const Matrix<char>& image) {
  auto sobel = mapOverlap(
    [](Neighborhood<char>& in) {
      short h = -1*in[{-1,-1}] +1*in[{+1,-1}]
                -2*in[{-1, 0}] +2*in[{+1, 0}]
                -1*in[{-1,+1}] +1*in[{+1,+1}];
      short v = ...;
      return sqrt(h*h + v*v); },
    1, BorderHandling::NEUTRAL(0));
  return soble(img); }
```

Listing 4.10: SkelCL implementation of the Sobel edge detection.

```
kernel void sobel_kernel( global const uchar* img,
                          global       uchar* out_img) {
 uint i = get_global_id(0);   uint j = get_global_id(1);
 uint w = get_global_size(0); uint h = get_global_size(1);
 // perform boundary checks
 if(i >= 1 && i < (w-1) && j >= 1 && j < (h-1)) {
  char ul = img[((j-1)*w)+(i-1)];
  char um = img[((j-1)*w)+(i+0)];
  char ur = img[((j-1)*w)+(i+1)];
  // ... 6 more
  out_img[j * w + i] = computeSobel(ul, um, ur, ...); }}
```

Listing 4.11: Additional boundary checks and index calculations for Sobel algorithm, necessary in the standard OpenCL implementation.

sion in Listing 4.9. The listing shows that extra low-level code is necessary to deal with technical details, like boundary checks and index calculations, which are arguably quite complex and error-prone.

We also compare against a more optimized OpenCL implementation by Nvidia which makes use of the fast local GPU memory.

The SkelCL implementation is significantly shorter than the two OpenCL-based implementations. The SkelCL program only comprises few lines of code as shown in Listing 4.10. The AMD implementation requires 37 lines of code for its kernel implementation and the more optimized Nvidia implementation requires even 208 lines of code. Both versions require additional lines of code for the host program which manages the execution of the OpenCL kernel. No index calculations or boundary checks are necessary in the SkelCL version whereas they are crucial for a correct implementation in OpenCL.

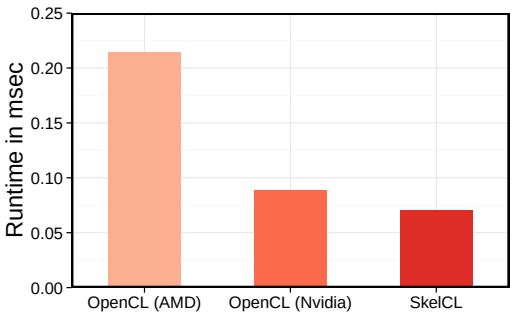


Figure 4.16: Performance results for Sobel edge detection

Performance experiments

Figure 4.16 shows the measured runtime of the two OpenCL versions (from the AMD and Nvidia SDKs) vs. the SkelCL version with the *stencil* skeleton presented in Listing 4.10. Here only the kernel runtimes are shown, as the data transfer times are equal for all versions. We used the popular Lena image [71] with a size of 512×512 pixel as an input. The AMD version is clearly slower than the two other implementations, because it does not use the fast local memory which the Nvidia implementation and the `MapOverlap` implementation of the *stencil* skeleton of SkelCL do. SkelCL completely hides the memory management details inside its implementation from the application developer. The Nvidia and SkelCL implementations perform similarly fast. In this particular example, SkelCL even slightly outperforms the implementation by Nvidia.

4.5.3 *Canny Edge Detection*

The Canny edge detection algorithm is a more complex algorithm to detect edges in images than the Sobel edge detection presented in the previous section. For the sake of simplicity we consider a slightly simplified version, which applies the following stencil operations in a sequence: 1), a noise reduction operation is applied, e. g., a Gaussian blur; 2), an edge detection operator like the Sobel edge detection is applied; 3), the so-called non-maximum suppression is performed, where all pixels in the image are colored black except pixels being a local maximum; 4), a threshold operation is applied to produce the final result. A more complex version of the algorithm performs the edge tracking by hysteresis, as an additional step. This results in detecting some weaker edges, but even without this additional step the algorithm usually achieves good results.

```
1 Matrix<char> sobelEdge(const Matrix<char>& image) {
2   auto gauss     = stencil(...);
3   auto sobel     = stencil(...);
4   auto nms       = stencil(...);
5   auto threshold = stencil(...);
6   StencilSequence<Pixel(Pixel)>
7       canny(gauss, sobel, nms, threshold);
8   return canny(image); }
```

Listing 4.12: Structure of the Canny algorithm expressed as a sequence of skeletons.

SkelCL Implementation

In SkelCL, each single step of the Canny algorithm can be expressed using the *stencil* skeleton. The last step, the threshold operation, can be expressed using SkelCL's simpler *map* skeleton, as the user threshold function only checks the value of the current pixel. In the SkelCL library the `Stencil` skeleton's implementation automatically uses the simpler *map* skeleton's implementation when the user specifies a stencil shape which extents are 0 in all directions.

To implement the Canny algorithm in SkelCL, the single steps can be combined as shown in Listing 4.12. The individual steps are defined in line 2—line 5 and then combined to a sequence of stencils in line 7. During execution (line 8), the stencil operations are performed in the order which is specified when creating the `StencilSequence` object.

Performance experiments

Figure 4.17 shows the measured runtime of the Canny algorithm using the two presented implementations. As the `MapOverlap` implementation appends padding elements to the matrix representing the image, the matrix has to be downloaded, resized and uploaded again to the GPU between every two steps of the sequence. This additional work leads to an increased time for data transfers. The Gaussian blur with a stencil shape extent of 2, as well as the Sobel edge detection and the non-maximum suppression with a stencil shape of 1, are 2.1 to 2.2 times faster when using `MapOverlap`. However, the threshold operation, which is expressed as the *map* skeleton in the Stencil sequence, is 6.8 times faster than `MapOverlap`'s threshold operation. Overall, we observe that when performing sequences of stencil operations, the `Stencil` implementation reduces the number of copy operations and, therefore, leads to a better overall performance. When performing the Canny algorithm, the `Stencil` implementation outperforms the `MapOverlap` implementation by 21%.

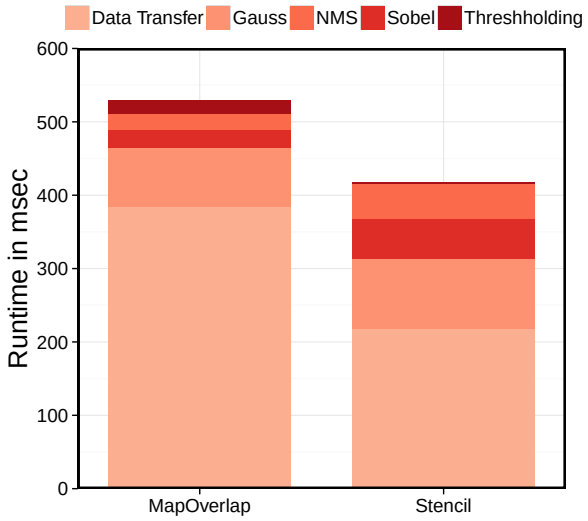


Figure 4.17: Runtime of the Canny edge detection algorithm comparing the `MapOverlap` and `Stencil` skeleton implementations.

4.6 MEDICAL IMAGING

At the beginning of Chapter 3 we used the LM OSEM medical imaging application as our motivational example and application study to identify requirements for a high-level programming model. In this section we will now study how we can express the LM OSEM application using algorithmic skeletons and how the parallel container data types and SkelCL's redistribution feature simplify the programming of multi-GPU systems. We will start by briefly reintroducing the application and its sequential implementation before moving to the parallel implementation using first traditional OpenCL and then SkelCL for comparison. A particular focus of this section will be on multi-GPU systems and how SkelCL drastically simplifies their programming.

The LM OSEM Algorithm

List-Mode Ordered Subset Expectation Maximization (LM OSEM) [133, 137] is a time-intensive, production-quality algorithm for medical image reconstruction. LM OSEM takes a set of events from a PET scanner and splits them into s equally sized subsets. Then, for each subset $S_l, l \in 0, \ldots, s-1$, the following computation is performed:

$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A_N^T \mathbf{1}} \sum_{i \in S_l} (A_i)^T \frac{1}{A_i f_l}. \tag{4.12}$$

Here f is the 3D reconstruction image which is refined over time. A is a matrix where element a_{ik} of row A_i represents the length of

```
1  for (int l = 0; l < subsets; l++) {
2    // read subset
3
4    // step 1: compute error image c_l
5    for (int i = 0; i < subset_size; i++) {
6      // compute A_i
7      // compute local error
8      // add local error to c_l
9    }
10
11   // step 2: update reconstruction image f
12   for (int k = 0 ; k < image_size; k++) {
13     if (c_l[k] > 0.0) { f[k] = f[k] * c_l[k]; }
14   }
15 }
```

Listing 4.13: Sequential code for LM OSEM comprises one outer loop with two nested inner loops.

intersection of the line between two PET detectors for a measured event i with voxel k of the reconstruction image. The factor in front of the sum can be precomputed and is, therefore, omitted from here on.

SEQUENTIAL IMPLEMENTATION Listing 4.13 shows the sequential code for LM OSEM as already presented in Section 3.1.1. The sequential LM OSEM is an iterative algorithm refining the reconstruction image f over time. At each iteration two major steps are performed:

> *Step 1:* the error image c_l is computed by performing three substeps: 1) computation of the row A_i; 2) computing the local error for row A_i; 3) adding the local error to c_l;
>
> *Step 2:* update the reconstruction image f using the error image c_l computed in Step 1.

PARALLELIZATION STRATEGY For parallelization two possible decomposition strategies can be considered for the LM OSEM algorithm as initially suggested in [96]: Projection Space Decomposition (PSD) and Image Space Decomposition (ISD).

In PSD, the subsets S_l are split into sub-subsets that are processed simultaneously while all processing units access a common reconstruction image f and error image c. Using this approach, we are able to parallelize *Step 1* of the algorithm, but *Step 2* is performed by a single processing unit. On a multi-GPU system, we have to copy the reconstruction image to all GPUs before each iteration, and we have

to merge all GPUs' error images computed in *Step 1* before proceeding with *Step 2*. While both steps are easy to implement, *Step 2* does not efficiently use the available processing units.

In ISD, the reconstruction image f is partitioned, such that each processing unit processes the whole subset S_l with respect to a single part of the reconstruction image f. Thus we are able to parallelize both steps of LM OSEM, but each processing unit still accesses the whole reconstruction image f in order to compute the error value for each path before merging it with the error image c. On a multi-GPU system, the whole subset S_l has to be copied to each GPU in *Step 1*. ISD requires large amounts of memory (up to several GB in practically relevant cases) to store all paths computed in *Step 1*. Summarizing, it is hard to implement *Step 1* on the GPU, while *Step 2* can be parallelized easily.

Therefore, we use a hybrid strategy for implementing LM OSEM: *Step 1* is parallelized using the PSD approach, while we use ISD for *Step 2*. This results in the sequence of five phases shown in Figure 4.18:

1. *Upload:* the subset (S) is divided into sub-subsets (one per GPU). One sub-subset and the reconstruction image (f) are uploaded to each GPU;

2. *Step 1:* each GPU computes the local error image (c_l) for its sub-subset;

3. *Redistribution:* the local error images that are distributed on all GPUs are downloaded and combined into a single error image on the host by performing element-wise addition. Afterwards, the combined error image and reconstruction image are partitioned, in order to switch the parallelization strategy from PSD to ISD. The corresponding parts of both images are distributed to the GPUs again;

4. *Step 2:* each GPU updates its part of the reconstruction image;

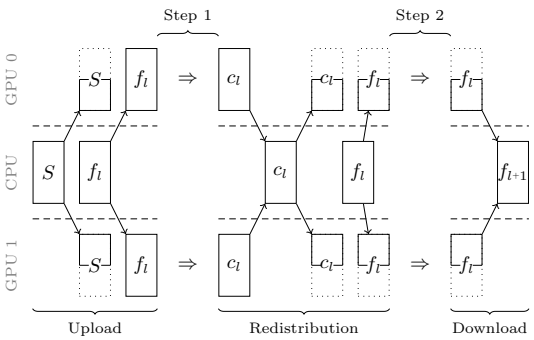


Figure 4.18: Parallelization schema of the LM OSEM algorithm.

5. *Download:* finally, all parts of the reconstruction image are downloaded from the GPUs to the host and merged into a single reconstruction image.

SkelCL Implementation

The SkelCL program in Listing 4.14 reflects the described five phases in a concise, high-level manner, as shown by the corresponding comments. The subset `s`, the error image `cl`, and the reconstruction image `f` are declared as SkelCL vectors which enables an easy and automatic data transfer between GPUs. As data transfers are performed implicitly by SkelCL, the upload phase is implemented by simply setting vector distributions (line 17—line 19), while the download phase is performed implicitly when the SkelCL containers are accessed or redistributed. The redistribution phase is implemented by changing the distributions of the corresponding SkelCL containers (line 25 and line 26).

The two computational steps are implemented using the *map* and *zip* skeleton from SkelCL, correspondingly, as follows.

The first step – the computation of the error image c_l – is implemented using the *map* skeleton. For each event `e` of the currently processed subset, the row A_i is computed (line 3). As A_i is sparsely populated it is stored as a special data structure, called `Path`, to reduce its memory footprint. Next, the local error is computed using A_i together with the current reconstruction image `f` which is passed to the *map* skeleton as an additional argument. Finally, the error image `cl` is updated with the local error. The error image is also provided as an additional argument, but when executing the *map* skeleton `cl` is wrapped using the `out` helper function (line 22). This marks the additional argument as output parameter, i. e., the SkelCL implementation is notified that this argument will be modified by the customizing function. It is interesting to point out that the *map* skeleton does not return a value, i. e., its return type is `void`. The skeleton is only executed for its side effects on `cl`.

The second step – the update of the reconstruction image `f` – is implemented using the *zip* skeleton. Here the customizing function operates on pairs of the voxels of the reconstruction and the error image, following the image space decomposition (ISD) strategy. If the voxel of the error image is greater than zero, the voxel of the reconstruction image is updated with the product of the pair of voxels from the reconstruction and the error image.

Programming effort

The lengthy and cumbersome OpenCL implementation of the LM OSEM was already discussed in Chapter 3. It is based on the work presented in [137]. OpenCL requires a considerable amount of boiler-

```
auto computeCl = mapVector(
  [](Event e, const Vector<float>& f, Vector<float>& cl) {
    Path Ai = computeAi(e);
    float c = computeLocalError(f, Ai);
    addLocalErrorToCl(cl, c, Ai); });

auto updateF = zipVector(
  [](float f_i, float cl_i) {
    if (cl_i > 0.0) return f_i * cl_i; else return f_i; });

Vector<float> f   = readStartImage();
for (l = 0; l < subsets; l++) {
  Vector<Event> s = read_subset();
  Vector<float> cl(image_size);

  /* Upload */
  s.setDistribution(block);
  f.setDistribution(copy);
  cl.setDistribution(copy, add);

  /* Step 1: compute error image cl */
  computeCl(s, f, out(cl));

  /* Redistribution */
  f.setDistribution(block);
  cl.setDistribution(block);

  /* Step 2: update image estimate f */
  f = updateF(f, cl);

  /* Download (implicit) */}
```

Listing 4.14: SkelCL code of the LM OSEM algorithm

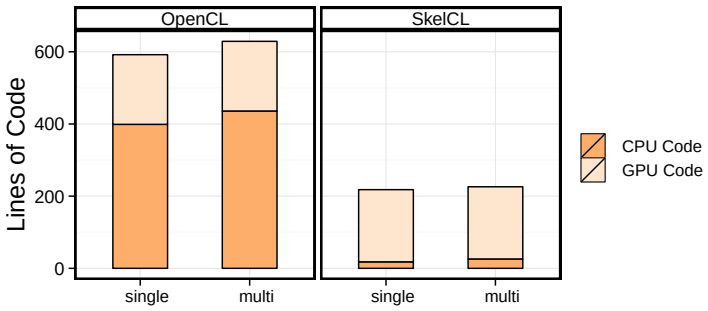


Figure 4.19: Lines of code for CPU and GPU of the LM OSEM implementations on single- and multi-GPU systems.

plate code for running a kernel on multiple GPUs, in particular for uploading and downloading data to and from the GPUs.

The parallelization strategies are the same for both versions. However, when using SkelCL's vector data type, we avoid additional programming effort to implement data transfer between host and GPU or between multiple GPUs, and we obtain a multi-GPU-ready implementation of LM OSEM for free.

Figure 4.19 shows the lines of code required for both implementation. The amount of lines required on the GPU is similar. This is not surprising, as these code describes the computation performed on the GPU which is similar for both implementations. The host code required for implementing the management of the GPU execution differs significantly across implementations. For a single GPU, the OpenCL-based implementation requires 206 LOCs, i. e., more than 11 times the number of lines than the SkelCL program which has 18 LOCs.

Using multiple GPUs in OpenCL requires explicit code for additional data transfers between GPUs. This accounts for additional 37 LOCs for the OpenCL-based implementation. In SkelCL, only 8 additional LOCs are necessary to describe the changes of data distribution. These lines are easily recognizable in the SkelCL program (line 17—line 19 and line 25—line 26 in Listing 4.14, plus 3 lines during the initialization) and make this high-level code arguably better understandable and maintainable than the OpenCL version.

Performance experiments

We evaluated the runtimes of our two implementations of LM OSEM by reconstructing an image of $150 \times 150 \times 280$ voxels from a real-world PET data set with about 10^8 events. From this data set, about 10^2 equally sized subsets are created. In our experiments, we measured the average runtime of processing one subset. To perform a full reconstruction producing a detailed reconstruction image, all subsets

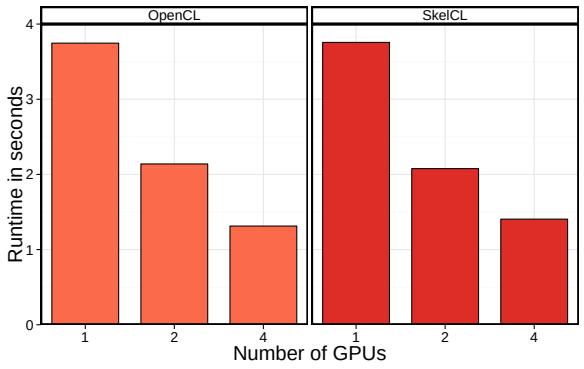


Figure 4.20: Average runtime of one iteration of the LM OSEM algorithm using OpenCL and SkelCL.

are processed multiple times, making LM OSEM a time-intensive application that runs several hours on a single-core CPU.

Figure 4.20 shows the runtime of both implementations of LM OSEM using up to four GPUs. While the differences in the programming effort to implement the SkelCL and OpenCL versions are significant, the differences in runtime are quite small. When running on a single GPU, both implementations take the same time (3.66 seconds) to complete. With two and four GPUs, the OpenCL implementation slightly outperforms the SkelCL implementation, being 1.2% and 4.7% faster. We presume that the slightly increasing overhead of SkelCL is caused by the more complex data distribution performed when using more GPUs. Comparing to the significant reduction in programming effort, the runtime overhead of less than 5% is arguably a moderate one. In conclusion, this example shows that SkelCL is suitable for implementing a real-world application and provides performance close to a native OpenCL implementation.

4.7 PHYSICS SIMULATION

Physics simulations are a very important class of scientific applications. We study one representative: the *Finite-Difference-Time-Domain (FDTD) Method* used for Random Lasing Simulations. This simulation from the field of optical physics simulates the propagation of light through a medium.

In the simulation two fields, the electric field $\vec{E}$ and the magnetic field $\vec{H}$, are iteratively updated using stencil computations. We use the Maxwell's equations which are the basic equations describing electrodynamic processes in nature to describe the light propagating through a non-magnetic (*dielectric*) medium.

Equation (4.13)—Equation (4.16) show the Maxwell's equations consisting of four coupled partial differential equations (PDEs).

$$\vec{\nabla}\vec{E}(\vec{r},t) = 0, \tag{4.13}$$

$$\vec{\nabla}\vec{H}(\vec{r},t) = 0, \tag{4.14}$$

$$\frac{\partial\vec{H}(\vec{r},t)}{\partial t} = -\frac{1}{\mu_0}\vec{\nabla}\times\vec{E}(\vec{r},t), \tag{4.15}$$

$$\frac{\partial\vec{D}(\vec{r},t)}{\partial t} = \frac{1}{\epsilon_0}\vec{\nabla}\times\vec{H}(\vec{r},t), \tag{4.16}$$

To couple the polarisation of a medium $\vec{P}$ to the electric field, Equation (4.17) is introduced:

$$\vec{E}(\vec{r},t) = \frac{\vec{D}(\vec{r},t) - \vec{P}\left(\vec{r},t,\vec{N}\right)}{\epsilon_0\epsilon_r(\vec{r})} \tag{4.17}$$

Here $\vec{N}$ is the induced energy distribution in the medium using the model proposed in [95]. The parameters μ_0, ϵ_0 and ϵ_r describe the permeability and permittivity of free space and the relative permittivity of the dielectric medium.

To solve this set of coupled PDEs, a method called Finite-Difference-Time-Domain (FDTD) [160] can be used. Here we use a form where the electric and magnet field are discretized within a n-dimensional regular grid. $\vec{E}$ and $\vec{H}$ are shifted against each other by a half grid-cell. This allows the calculation of the new values by computing finite differences between two values of the grid. Using the FDTD method, we implemented a simulation of the effect of random lasing on a nano-meter scale [27] for our evaluation.

Figure 4.21 shows a visualization of the electric field (and the field intensity) after about 1 ps of simulation time equal to 60 000 iterations. The shown field distribution can be found also in [138, 159], however the simulation parameters are different.

SkelCL Implementation

We implemented a two-dimensional version using SkelCL as well as a manually tuned OpenCL implementation. To solve the PDEs in Equation (4.15) and Equation (4.16), two separated three-point stencil computations are performed and one map computation for the gain-model is necessary. Equation (4.13) and Equation (4.14) are implicitly solved by the FDTD method [160]. Listing 4.15 shows the SkelCL code of the application: in every iteration first the energy distribution is updated (line 11) using a map skeleton (defined in line 1); then the first stencil (defined in line 2) updates the electric field $\vec{E}$ by combining a single element of $\vec{E}$ with three elements of the magnetic field $\vec{H}$

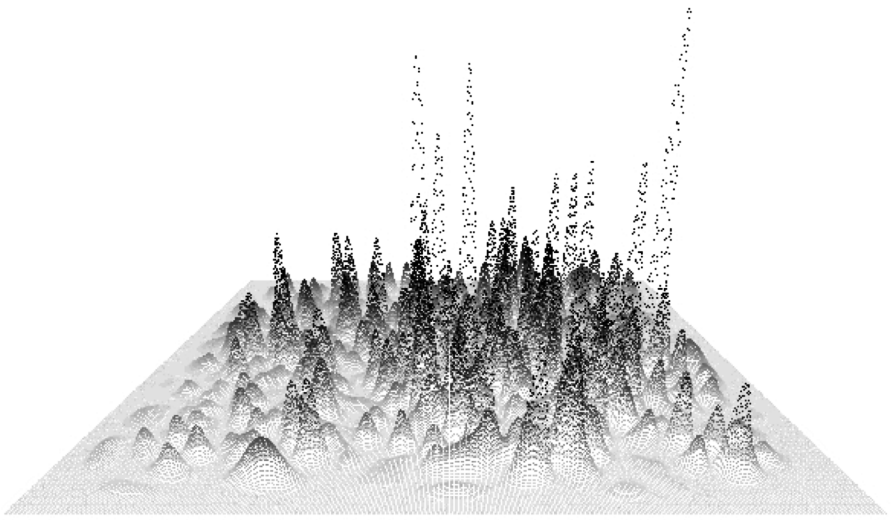


Figure 4.21: The image shows a 3D representation of the intensity for the 2D electric field as computed by the SkelCL FDTD implementation after 60 000 iterations.

(line 12); and finally the second stencil (defined in line 3) updates $\vec{H}$ by combining a single element of $\vec{H}$ with three elements of $\vec{E}$ (line 13).

Please note that the two stencil computations require both fields ($\vec{E}$ and $\vec{H}$) as input. To implement this, we use the *additional argument* feature of SkelCL which allows the additional field to be passed to skeletons on execution (see line 12 and line 13). The additional arguments are passed unchanged to the customizing function of the skeleton, therefore, the function customizing the stencil in line 4 now accepts $\vec{H}$ as a second parameter. This feature greatly increases the flexibility of applications written in SkelCL.

Performance experiments

In the evaluation we used a 2048×2048 sized matrix with a spatial resolution of 100 cells per µm. This matrix corresponds to a square-shaped medium with the edge length of 20.1 µm. The medium size is actually smaller than the matrix size because of the border handling. To provide a physically correct simulation, the borders of the magnet field must be treated specially. The Stencil skeleton in SkelCL provides sufficient functionality to allow for such border handling in the computation code.

We compared our SkelCL based implementation to a handwritten, fine-tuned OpenCL implementation which is based on [100] and was initially developed and described in [80]. The OpenCL version is specifically designed for modern Nvidia GPUs. In particular, it exploits the L1 and L2 caches of the Nvidia Fermi and Kepler architecture and does not explicitly make use of the local memory. We performed the experiments on a system with a modern Nvidia K20c Kepler GPU with 5GB memory and 2496 compute cores. Fig-

```
auto updateEnergyDist = map(...);
auto updateEField     = stencil(...);
auto updateHField     = stencil(
 [](Neighborhood<float4>& E, Matrix<float4>& H) { ... });

Matrix<float4> N; // energy distribution in the medium
Matrix<float4> E; // E (electric) field
Matrix<float4> H; // H (magnetic) field

for (...) { // for each iteration
  updateEnergyDist(out(N), N, out(E));
  updateEField(out(E), H, E);
  updateHField(out(H), E, H); }
```

Listing 4.15: Source code of the FDTD application in SkelCL.

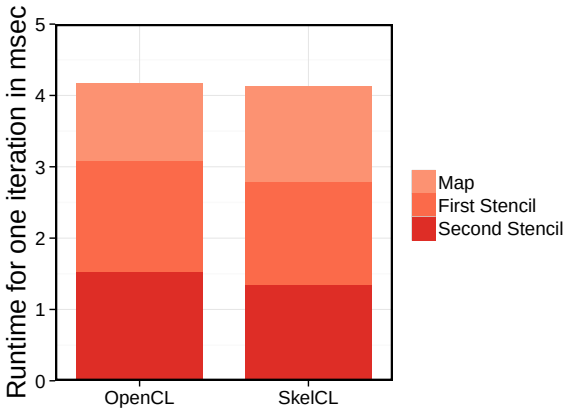


Figure 4.22: Runtime for one iteration of the FDTD application.

ure 4.22 shows the median runtimes of a simulation time of 1 ps equal to 60 000 iterations. The SkelCL version slightly outperforms the OpenCL version by 2%. The two stencil skeletons achieve ~10% faster runtimes than the corresponding OpenCL kernels but the map skeleton is ~20% slower, because it reads and writes all elements exactly once, while the customized OpenCL kernel does not write back all elements. For this application it seems beneficial to make explicit usage of the local memory as our implementation of the Stencil skeleton does, instead of relying on the caches of the hardware, as the OpenCL implementation does.

4.8 SUMMARY

Figure 4.23 and Figure 4.24 summaries the findings of this chapter.

Figure 4.23 shows the lines of code required for implementing five of the applications examples in OpenCL (on the left) and SkelCL (on

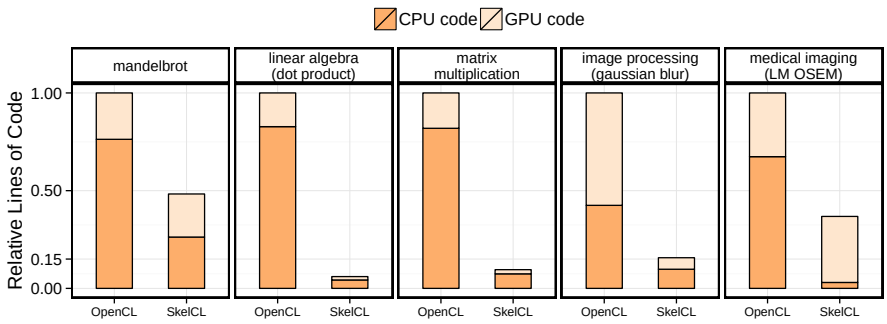


Figure 4.23: Relative lines of code for five application examples discussed in this chapter comparing OpenCL code with SkelCL code.

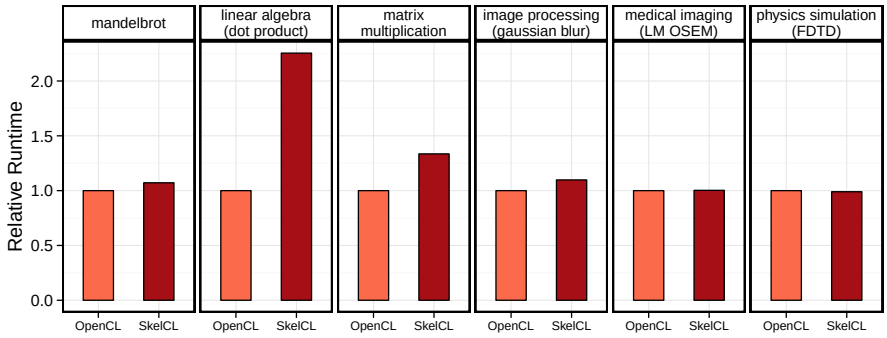


Figure 4.24: Relative runtime for six application examples discussed in this chapter comparing OpenCL-based implementations with SkelCL-based implementations.

the right). We scaled all graphs relative to the lines of code required by the OpenCL implementation. The SkelCL code is significant shorter in all cases, requiring less than 50% of the lines of code of the OpenCL-based implementation. For the linear algebra application, matrix multiplication, and image processing application even less than 15% of lines of code are required when using SkelCL.

Figure 4.24 shows the runtime results for six of the application examples presented in this chapter. We compare the runtime of optimized OpenCL implementations against SkelCL-based implementations. For all shown application examples – except the dot product application – we can see that SkelCL is close to the performance of the OpenCL implementations. For most applications the runtime of the SkelCL-based implementations are within 10% of the OpenCL implementations. For the matrix multiplication SkelCL is 33% slower than the optimized OpenCL implementation which only operates on squared matrices. The dot product application is significantly slower, as SkelCL generates two separate OpenCL kernels instead of a single optimized kernel.

4.9 CONCLUSION

In this chapter we have thoroughly evaluated the SkelCL programming model and its C++ library implementation. We have seen that SkelCL successfully addresses the programmability challenge and indeed greatly simplifies GPU programming as compared to OpenCL. For all investigated benchmarks we could see a reduction in the amount of code written by the programmer of up to 9 times for some benchmarks. The performance results show that SkelCL is able to achieve runtime performance on par with native written OpenCL code for most benchmarks. For two benchmarks, *asum* and dot product, SkelCL performs bad compared to native written code as unnecessarily multiple OpenCL kernels are generated and executed. We will present a compilation technique which addresses this performance drawback in Part III of this thesis.

For the *stencil* skeleton we saw, that the two presented implementations have slightly different performance characteristics. Their overall performance is comparable to native written stencil code.

For the *allpairs* skeleton we saw, that the specialized implementation taking advantage of the regular *zip-reduce* patterns offers large performance benefits as compared to the generic implementation. For matrix multiplication the specialized implementation performs close to an optimized OpenCL implementation, but still about 30% slower than the highly optimized BLAS implementations.

In the next part of the thesis we introduce a novel compilation technique which addresses the performance portability challenge of generating high performing code from a single high-level presentation.

Part III

A NOVEL CODE GENERATION APPROACH OFFERING PERFORMANCE PORTABILITY

5 CODE GENERATION USING PATTERNS

In the previous two chapters we discussed how regular parallel patterns, referred to as *algorithmic skeletons*, help to simplify programming of modern parallel systems. Parallel programming of multi-GPU systems is considerably simplified without sacrificing performance as shown by the evaluation in Chapter 4.

In this chapter, we address the second main challenge identified in Chapter 1: *Performance portability*. We will present a novel approach for generating efficient and hardware-specific code from compositions of high-level *patterns*. This approach is based on a system of rewrite-rules providing performance portability across different types of modern parallel processors. In the following Chapter 6 we will use a set of example applications to show that our approach generates code matching the performance of highly tuned implementations on CPUs and GPUs.

We will start this chapter by looking at optimizations in OpenCL and how applying them changes the source code of applications. We will show that these optimizations are often hardware-specific thus breaking performance portability: optimizations for one particular hardware architecture can lead to poor performance on other hardware architectures. This will motivate the necessity, when aiming for performance portability, for generating optimized and specialized code from a pattern-based high-level representation. We will discuss the performance benefits this code generation approach offers over a library-based approach like SkelCL presented in Chapter 3. Then we will give an overview of our approach and present it in more detail in the following sections. Chapter 6 will present an evaluation of the approach using a set of application studies.

5.1 A CASE STUDY OF OPENCL OPTIMIZATIONS

To understand the problems of performance and portability in the context of modern parallel processors, we will study a simple application example: parallel reduction. This discussion is based on the presentation *"Optimizing Parallel Reduction in CUDA"* by Harris [82] where optimizations for implementing the parallel reduction using CUDA and targeting Nvidia GPUs are presented. Optimization guidelines like this exist from almost every hardware vendor, including AMD [10], Intel [1, 126], and Nvidia [44], giving developers advice on how to most efficiently exploit their hardware.

In Chapter 3 we saw that we can express a parallel reduction using a single algorithmic skeleton: *reduce*. Here we look at how efficient OpenCL implementations of this algorithm look like. More precisely we will investigate the parallel summation of an array as a concrete example of the generic *reduce* algorithm. We are especially interested in gradually optimizing this application to see how beneficial the single optimization steps are and how they change the source code.

We will first start by looking at the implementation and optimizations on one particular hardware architecture, using a Nvidia GPU as our example. Then we will see how the optimized implementations perform on an AMD GPU and Intel CPU, to evaluate their performance portability. Finally, we will use our observations to motivate the need for a pattern-based code generator for achieving performance portability.

5.1.1 *Optimizing Parallel Reduction for Nvidia GPUs*

For implementing the parallel summation of an array in OpenCL usually an approach with two OpenCL kernels is used. We start with the elements to be reduced in the leaves at the top of the reduction tree shown in Figure 5.1. The first OpenCL kernel is executed in parallel by multiple OpenCL work-groups, four work-groups in the example shown in Figure 5.1. Each work-group produces a temporary result, then the second OpenCL kernel is executed by a single OpenCL work-group producing the final result. This strategy is applied as synchronization across work-groups is prohibited inside a single OpenCL kernel, but the parallel tree-based reduction requires synchronization at each level of the reduction tree (indicated by the bold lines in Figure 5.1). An implementation with a single OpenCL kernel would, therefore, be restricted to launch a single OpenCL work-group and, thus, limit the exploited parallelism.

By using the two-kernel approach, massive parallelism can be exploited in the first phase as multiple work-groups operate concurrently on independent parts of the input array. The second kernel is launched with a single work-group using synchronization inside

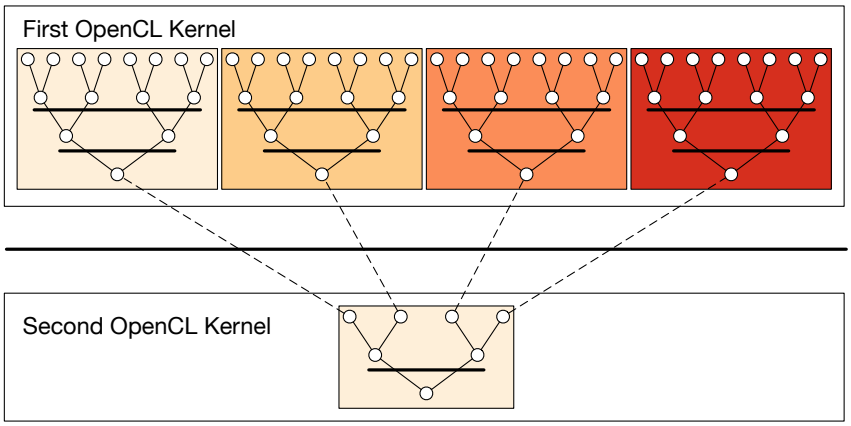


Figure 5.1: The first OpenCL kernel is executed by four work-groups in parallel: work-group 0, work-group 1, work-group 2, work-group 3. The second OpenCL kernel is only executed by the first work-group. The bold lines indicate synchronization points in the algorithm.

the work-group to compute the final result. The vast majority of the work is done in the first phase and the input size to the second phase is comparably small, therefore, the limited exploitation of parallelism in the second phase does not effect overall performance much. For this reason we will discuss and show only the differences and optimizations in the first OpenCL kernel.

We will follow the methodology established in [82] and evaluate the performance of the different versions using the measured GPU memory bandwidth as our metric. The memory bandwidth is computed by measuring the runtime in seconds and dividing it by the input data size which is measured in gigabytes. As we use the same input data size for all experiments, the bandwidth results shown in this section directly correspond to the inverse of the measured runtime. By investigating the memory bandwidth of the GPU memory, we can see which fraction of the maximum memory bandwidth available has been utilized. Using the memory bandwidth as evaluation metric for the parallel reduction is reasonable as the reduction has a very low arithmetic intensity and its performance is, therefore, bound by the available GPU memory bandwidth.

All following implementations are provided by Nvidia as part of their software development kit and presented in [82]. These implementations have originally been developed for Nvidia's Tesla GPU architecture [109] and not been updated by Nvidia for more recent GPU architectures. Nevertheless, the optimizations discussed are still beneficial on more modern Nvidia GPUs– as we will see. All performance numbers in this section have been measured on a Nvidia GTX 480 GPU featuring the Nvidia Fermi architecture [157].

```
1  kernel
2  void reduce0(global float* g_idata, global float* g_odata,
3               unsigned int n, local float* l_data) {
4    unsigned int tid = get_local_id(0);
5    unsigned int i   = get_global_id(0);
6    l_data[tid] = (i < n) ? g_idata[i] : 0;
7    barrier(CLK_LOCAL_MEM_FENCE);
8    // do reduction in local memory
9    for(unsigned int s=1; s < get_local_size(0); s *= 2) {
10     if ((tid % (2*s)) == 0) {
11       l_data[tid] += l_data[tid + s]; }
12     barrier(CLK_LOCAL_MEM_FENCE); }
13   // write result for this work-group to global memory
14   if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Listing 5.1: First OpenCL implementation of the parallel reduction achieving 6.64% of the memory bandwith limit.

FIRST OPENCL IMPLEMENTATION Listing 5.1 shows the first version of the parallel reduction in OpenCL. First each work-item loads an element into the local memory (line 6). After a synchronization (line 7) all work-items of a work-group execute a for loop (line 9—line 12) to perform a collective tree-based reduction. In every iteration the if statement (line 10) ensures that a declining number of work-items remain active performing partial reductions in the shrinking reduction tree. The second barrier in line 12 ensures that no race conditions occur when accessing the shared local memory. Finally, the work-item in the work-group with id $= 0$ writes back the computed result to the global memory in line 14.

The implementation presented in Listing 5.1 is not straightforward to develop. The application developer has to be familiar with the parallel execution model of OpenCL to avoid race conditions and deadlocks. For example, it is important that the second barrier in line 12 is placed *after* and not *inside* the if statement. This is true, even though work-items not entering the if statement will never read from or write to memory and, therefore, can never be influenced by a race condition. Nevertheless, OpenCL requires all work-items of a work-group to execute all barrier statements in a kernel exactly the same number of times. The application developer is responsible to ensure that this condition is met, otherwise a deadlock will occur.

Despite being difficult to program, this implementation does not provide high performance either. Only 6.64% (11.78 GB/s) of the available bandwidth is utilized on the GTX 480.

```
1  kernel
2  void reduce1(global float* g_idata, global float* g_odata,
3               unsigned int n, local float* l_data) {
4    unsigned int tid = get_local_id(0);
5    unsigned int i   = get_global_id(0);
6    l_data[tid] = (i < n) ? g_idata[i] : 0;
7    barrier(CLK_LOCAL_MEM_FENCE);
8
9    for(unsigned int s=1; s < get_local_size(0); s *= 2) {
10       // continuous work-items remain active
11       int index = 2 * s * tid;
12       if (index < get_local_size(0)) {
13           l_data[index] += l_data[index + s]; }
14       barrier(CLK_LOCAL_MEM_FENCE); }
15
16   if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Listing 5.2: OpenCL implementation of the parallel reduction avoiding divergent branching. This implementation utilizes 9.73% of the memory bandwidth limit.

AVOID DIVERGENT BRANCHING Listing 5.2 shows the second implementation. The differences from the previous implementation are highlighted in the code.

When performing the collective tree-based reduction in a work-group, a shrinking number of work-items remain active until the last remaining work-item computes the result of the entire work-group. In the previous version the modulo operator was used to determine which work-item remains active (see line 10 in Listing 5.1). This leads to a situation were not the consecutive work-items remain active, but rather work-items which id is divisible by 2, then by 4, then by 8, and so on. In Nvidia's GPU architectures, 32 work-items are grouped into a *warp* and executed together, as described in Chapter 2. It is highly beneficial to program in a style where all 32 work-items grouped into a warp perform the same instructions to avoid divergent branching between work-items of a warp. Using the modulo operator to determine the active work-items leads to highly divergent branching. The second implementation in Listing 5.2, therefore, uses a different formula (line 11) for determining the active work-items, which avoids divergent branching.

The runtime performance improves by a factor of 1.47 as compared to the first implementation. However, still only 9.73% (17.26 GB/s) of the theoretically available memory bandwidth are used by this version.

```
1  kernel
2  void reduce2(global float* g_idata, global float* g_odata,
3               unsigned int n, local float* l_data) {
4    unsigned int tid = get_local_id(0);
5    unsigned int i   = get_global_id(0);
6    l_data[tid] = (i < n) ? g_idata[i] : 0;
7    barrier(CLK_LOCAL_MEM_FENCE);
8
9    // process elements in different order
10   // requires commutativity!
11   for(unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
12       if (tid < s) {
13           l_data[tid] += l_data[tid + s]; }
14       barrier(CLK_LOCAL_MEM_FENCE); }
15
16   if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Listing 5.3: OpenCL implementation of the parallel reduction avoiding local memory bank conflicts. This implementation utilizes 12.61% of the memory bandwidth limit.

AVOID INTERLEAVED ADDRESSING Listing 5.3 shows the third implementation. The differences from the previous implementation are highlighted in the code.

On modern GPUs the fast local memory is organized in multiple *banks*. When two, or more, work-items simultaneously access memory locations in the same bank a *bank conflict* occurs which means that all memory requests are processed sequentially and not in parallel as usual. The previous two implementations use an access pattern for the local memory which makes bank conflicts likely. The third implementation in Listing 5.3 avoids this problematic local memory access pattern. Instead an access pattern is used where bank conflicts are unlikely and, thus, performance is improved. This better access pattern requires the reduction operation to be commutative, as the order of element is not respected when reading from local memory.

The performance improves by a factor of 1.30 as compared to the previous implementation and 1.90 to the initial implementation. With this version 12.61% (22.37 GB/s) of the theoretically available memory bandwidth are used.

```
1  kernel
2  void reduce3(global float* g_idata, global float* g_odata,
3               unsigned int n, local float* l_data) {
4    unsigned int tid = get_local_id(0);
5    unsigned int i = get_group_id(0) * (get_local_size(0)*2)
6                                     + get_local_id(0);
7    l_data[tid] = (i < n) ? g_idata[i] : 0;
8    // performs first addition during loading
9    if (i + get_local_size(0) < n)
10       l_data[tid] += g_idata[i+get_local_size(0)];
11   barrier(CLK_LOCAL_MEM_FENCE);
12
13   for(unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
14       if (tid < s) {
15           l_data[tid] += l_data[tid + s]; }
16       barrier(CLK_LOCAL_MEM_FENCE); }
17
18   if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Listing 5.4: OpenCL implementation of the parallel reduction. Each work-item performs an addition when loading data from global memory. This implementation utilizes 23.71% of the memory bandwidth limit.

INCREASE COMPUTATIONAL INTENSITY PER WORK-ITEM Listing 5.4 shows the fourth implementation. The differences from the previous implementation are highlighted in the code.

In the previous versions, each work-item loads one element from the global into the local memory before the work-items of the work-group collectively perform a tree-based reduction. That means that half of the work-items are idle after performing a single copy operation, which is highly wasteful. The fourth implementation in Listing 5.4 avoids this by having each work-item load two elements from global memory, perform an addition, and store the computed result in local memory (line 10). Assuming the same input size this reduces the number of work-items to start with by half and, therefore, increases the computational intensity for every work-item.

The performance improves by a factor of 1.88 as compared to the previous implementation and 3.57 to the initial implementation. With this version, 23.71% (42.06 GB/s) of the theoretically available memory bandwidth is used.

AVOID SYNCHRONIZATION INSIDE A WARP Listing 5.5 shows the fifth implementation. The differences from the previous implementation are highlighted in the code.

Wraps are the fundamental execution unit in Nvidia's GPU architectures, as explained in Chapter 2: All work-items grouped in a warp are guaranteed to be executed together in a lock-step manner, i. e., all work-items in the same warp execute the same instruction simultaneously. Because of this hardware behaviour, no barrier synchronization is required between instructions inside a single warp. The fifth implementation in Listing 5.5 takes advantage of this. The for loop performing the tree-based reduction is exited early at the stage when only 32 work-items remain active (see line 14). The extra code in line 22 up to line 28 performs the rest of the tree-base reduction without any barrier synchronization. The code shown here effectively unrolled the last six iterations of the for loop in line 14. As warps are specific to Nvidia's GPU architectures, this implementation is not portable and might produce incorrect results on other OpenCL devices.

The performance improves by a factor of 1.37 as compared to the previous implementation and 4.91 to the initial implementation. With this version, 32.59% (57.81 GB/s) of the theoretically available memory bandwidth is used.

COMPLETE LOOP UNROLLING Listing 5.6 on page 126 shows the sixth implementation. The differences from the previous implementation are highlighted in the code.

In the previous implementation we made a special case for the last six iterations of the for loop and provided special code handling for each iteration separately. This is a general optimization strategy known as *loop unrolling*. Loop unrolling can be beneficial because variables and branches required by a loop can be avoided. Furthermore, instruction level parallelism can be increased. In the sixth implementation, shown in Listing 5.6, the for loop has been removed entirely and replaced by three if statement (line 13, line 16, and line 19). Each if statement replaces one iteration of the loop. This code assumes that `WG_SIZE` is a compile-time constant and, therefore, the if statements will be evaluated at compile time, avoiding costly branches at runtime. Different to the previous optimization (Listing 5.5), we still have to provide a barrier to ensure correct synchronization, as multiple warps are involved here.

The performance improves by a factor of 1.13 as compared to the previous implementation and 5.54 times as compared to the initial implementation. With this version, 36.77% (65.23 GB/s) of the theoretically available memory bandwidth are used.

```
kernel
void reduce4(global float* g_idata, global float* g_odata,
             unsigned int n,local volatile float* l_data){
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
      l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  // prevent further unrolling (see next version)
  #pragma unroll 1
  for(unsigned int s=get_local_size(0)/2; s>32; s>>=1) {
      if (tid < s) {
          l_data[tid] += l_data[tid + s]; }
      barrier(CLK_LOCAL_MEM_FENCE); }

  // unroll for last 32 active work-items
  // no synchronization required on Nvidia GPUs
  // this is not protable OpenCL code!
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }

  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Listing 5.5: OpenCL implementation of the parallel reduction. Synchronization inside a warp is avoided by unrolling the loop for the last 32 work-items. This implementation utilizes 32.59% of the memory bandwidth limit.

```
1  kernel
2  void reduce5(global float* g_idata, global float* g_odata,
3               unsigned int n,local volatile float* l_data){
4    unsigned int tid = get_local_id(0);
5    unsigned int i = get_group_id(0) * (get_local_size(0)*2)
6                                     + get_local_id(0);
7    l_data[tid] = (i < n) ? g_idata[i] : 0;
8    if (i + get_local_size(0) < n)
9        l_data[tid] += g_idata[i+get_local_size(0)];
10   barrier(CLK_LOCAL_MEM_FENCE);
11
12   // unroll for loop entirely
13   if (WG_SIZE >= 512) {
14       if (tid < 256) { l_data[tid] += l_data[tid+256]; }
15       barrier(CLK_LOCAL_MEM_FENCE); }
16   if (WG_SIZE >= 256) {
17       if (tid < 128) { l_data[tid] += l_data[tid+128]; }
18       barrier(CLK_LOCAL_MEM_FENCE); }
19   if (WG_SIZE >= 128) {
20       if (tid <  64) { l_data[tid] += l_data[tid+ 64]; }
21       barrier(CLK_LOCAL_MEM_FENCE); }
22
23   if (tid < 32) {
24     if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
25     if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
26     if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
27     if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
28     if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
29     if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
30
31   if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Listing 5.6: OpenCL implementation of the parallel reduction with a completly unrolled loop. This implementation utilizes 36.77% of the memory bandwidth limit.

FULLY OPTIMIZED IMPLEMENTATION Listing 5.7 on page 128 shows the final and fully optimized implementation. The differences from the previous implementation are highlighted in the code.

One of the optimizations applied earlier was to increase the computational intensity for each single work-item by performing two loads and an addition instead of a single load. This final version applies the same idea, but performing multiple additions per work-item before the collective tree-based reduction in the entire work-group. This has indeed two advantages: first, the algorithmic intensity is increased, i. e., each work-item is doing more work, and, second, performing the summation sequentially by a single work-item does not require costly synchronizations. The fully optimized implementation is shown in Listing 5.7 with the changes highlighted. A while loop has been introduced (see line 11) which, in every iteration, loads two elements from the global memory and adds them to the local memory. No synchronization is required here as each work-item operates independently on different memory locations. The way the memory is accessed ensures that memory accesses will be coalesced (see Chapter 2) when reading from global memory.

The performance improves by a factor of 1.78 as compared to the previous implementation and 9.85 times as compared to the initial implementation. With this version, 65.44% (116.09 GB/s) of the theoretically available memory bandwidth are used.

CONCLUSIONS We can draw several valuable conclusions from studying these OpenCL source codes.

The first main conclusion is, that implementing these optimizations is not intuitive and straightforward. It requires experience as well as knowledge and reasoning about the target hardware architecture, in this case the Fermi GPU architecture: Listing 5.2 requires the understanding of the problem of *branch divergence*, Listing 5.3 requires knowledge about the organization of local memory and *bank conflicts*, Listing 5.4 and Listing 5.7 require reasoning about the *computational intensity* of work-items, Listing 5.5 requires understanding of *warps* and their execution by the hardware, Listing 5.6 requires experience with *loop unrolling* techniques, and, finally, Listing 5.7 required knowledge about the organization of global memory and *memory coalescing*. These are *additional* burdens for the application developer on top of implementing a functionally correct version where the programmer is facing widely recognized correctness problems of parallel programming like race conditions and deadlocks.

Furthermore, the source code changes necessary for individual optimization steps are not obvious either. The source code of the first implementation in Listing 5.1 is very different as compared to the final implementation shown in Listing 5.7. For example, the code size has more then doubled (from 16 to 35 lines of code) and many state-

```
1  kernel
2  void reduce6(global float* g_idata, global float* g_odata,
3               unsigned int n,local volatile float* l_data){
4    unsigned int tid = get_local_id(0);
5    unsigned int i = get_group_id(0) * (get_local_size(0)*2)
6                                     + get_local_id(0);
7    unsigned int gridSize = WG_SIZE*2*get_num_groups(0);
8    l_data[tid] = 0;
9
10   // multiple elements are reduced per work-item
11   while (i < n) { l_data[tid] += g_idata[i];
12                   if (i + WG_SIZE < n)
13                     l_data[tid] += g_idata[i+WG_SIZE];
14                   i += gridSize; }
15   barrier(CLK_LOCAL_MEM_FENCE);
16
17   if (WG_SIZE >= 512) {
18       if (tid < 256) { l_data[tid] += l_data[tid+256]; }
19       barrier(CLK_LOCAL_MEM_FENCE); }
20   if (WG_SIZE >= 256) {
21       if (tid < 128) { l_data[tid] += l_data[tid+128]; }
22       barrier(CLK_LOCAL_MEM_FENCE); }
23   if (WG_SIZE >= 128) {
24       if (tid <  64) { l_data[tid] += l_data[tid+ 64]; }
25       barrier(CLK_LOCAL_MEM_FENCE); }
26
27   if (tid < 32) {
28     if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
29     if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
30     if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
31     if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
32     if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
33     if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
34
35   if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Listing 5.7: Fully optimized OpenCL implementation of the parallel reduction. This implementation utilizes 65.44% of the memory bandwidth limit.

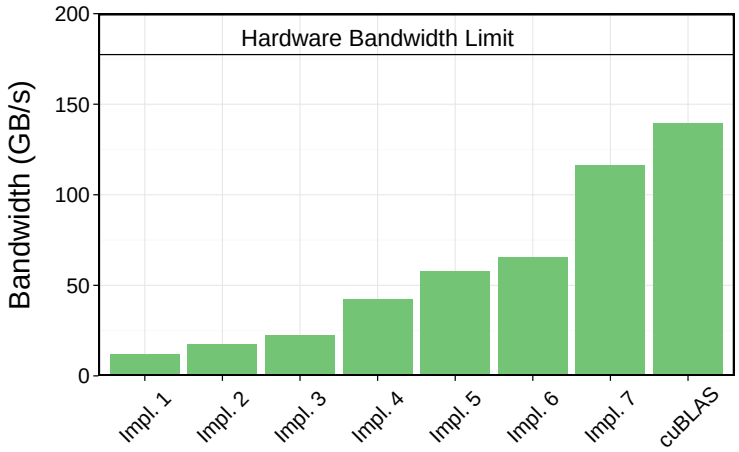
ments have been added or replaced by others. It is not obvious, neither to a human being nor to an optimizing compiler, that these two pieces of code have the same semantics (assuming an associative and commutative binary reduction operator, like +).

The second main conclusion we can draw is, that performing these optimizations on modern parallel architectures is highly beneficial. The first unoptimized version did only utilize about 6.64% of the available memory bandwidth, while the fully optimized version utilizes a more reasonable 65.44% on our GeForce GTX 480. Applying all optimizations improved the performance by a factor of ≈10 while utilizing the exactly same hardware. For an input array of size of 256 MB the runtime reduces from 95.7 ms to 9.1 ms when using the optimized kernel over the unoptimized one. Harris [82] reports an even higher improvement factor of ≈30 for the GeForce 8800 GTX used in his experiments. Modern parallel processors are often chosen as target architecture because of their high theoretical performance. Turning the theoretical performance into practical performance by applying such optimizations is, therefore, *essential* for most users.

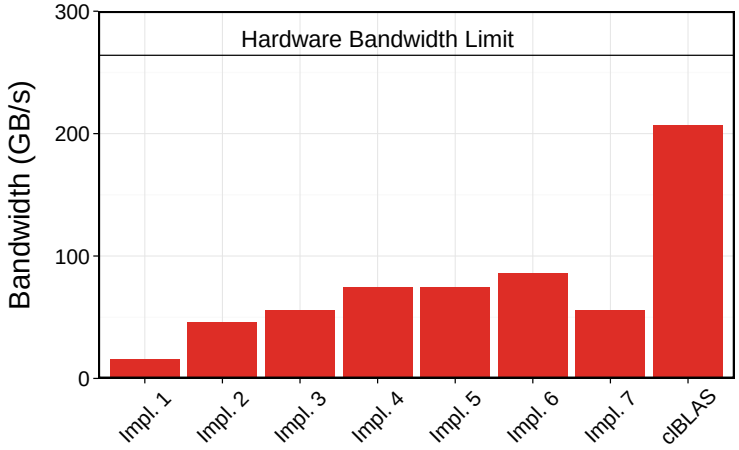
5.1.2 *Portability of the Optimized Parallel Reduction*

After we have established how crucial but hard to achieve optimizations are, we now will investigate their portability. To do so, we did run the code shown in Listing 5.1–Listing 5.7 on three different hardware devices: Nvidia's GTX 480 (Fermi GPU architecture [157]) which we have used in our analysis in the previous section, AMD's Radeon HD7970 (Graphics Core Next GPU architecture [155]), and Intel's E5530 CPU (Nehalem CPU architecture [156]). Figure 5.2 shows the performance numbers for each device. We use, as before, the memory bandwidth as our metric and show the hardware memory bandwidth limit of each respective hardware architecture at the top. As a practical comparison we also show bandwidth numbers for the parallel reduction using highly tuned, architecture specific implementations of the BLAS library. We use the CUBLAS library [43] for the Nvidia GPU, the clBLAS library [150] for the AMD GPU, and the MKL library [93] for the Intel CPU. Each library is implemented and provided by the corresponding hardware vendor.

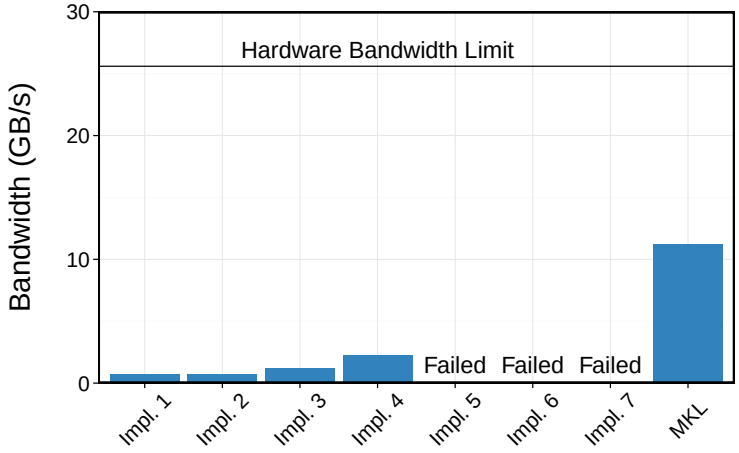
PORTABILITY OF OPTIMIZATIONS The results show that the optimizations discussed in the previous section are not portable. On each architecture a different version of the optimized implementations performs best: Implementation 7 (shown in Listing 5.7) on the Nvidia GPU, Implementation 6 (shown in Listing 5.6) on the AMD GPU, and Implementation 4 (shown in Listing 5.4) on the Intel CPU. Some implementations actually produce incorrect results on the Intel CPU due to the warp-specific optimization introduced in Implementa-



(a) Nvidia's GTX 480 GPU.



(b) AMD's HD 7970 GPU.



(c) Intel's E5530 dual-socket CPU.

Figure 5.2: Performance of differently optimized implementations of the parallel reduction

tion 5 (shown in Listing 5.5). Interestingly, this optimization happens to be valid on AMD's GPU architecture as well, as there exists a similar concept to warps called *wavefronts* [155].

PORTABILITY OF RELATIVE PERFORMANCE *Relative performance* refers to the performance of an implementation relative to the best theoretical performance possible or best existing implementation on a given hardware architecture. The theoretical performance of an architecture is given by its hardware limitations, like the number of arithmetic logical units, the width of the memory bus, or the maximum clock frequency. Practical issues like work load and utilization or the cache miss rate are ignored. Possible metrics for measuring the theoretical performance are number of floating point operation in GFLOP/s or the memory bandwidth in GB/s.

The best practical performance is given by the best implementation available for a particular hardware platform. It is not always possible to determine which implementation is the best. Here we consider the BLAS library implementations tuned by the respective hardware vendors as the best available.

By investigating relative performance we can compare the consequences of applying optimizations across different hardware architectures. The *relative performance* shown in Figure 5.2 differs widely across architectures.

On the Nvidia GPU, the best optimized implementation achieves 83.3% of the performance of the vendor-provided CUBLAS implementation utilizing 65.4% of the theoretical memory bandwidth limit.

On the AMD GPU, the gap between the manual and library implementation is much larger: the manually optimized implementation achieves only 41.3% of the clBLAS library implementation. Only 32.4% of the theoretical memory bandwidth limit is achieved.

On the Intel CPU, implementation 4 achieves only 16.6% of the MKL performance. That means, that MKL is over 5 times faster than the best of the discussed implementations. The hardware bandwidth limit is only utilized to 8.6%. Interestingly, the MKL implementation also only provides 43.8% of the maximum memory bandwidth. This is due to the combination of the implementation of the parallel reduction in MKL and the particular machine used in this experiment. The test machine used is configured as a dual-socket machine, i. e., two E5530 CPUs each with their own memory controller are available. Therefore, the hardware bandwidth available is doubled as compared to a single E5530 CPU. While the implementation of the parallel reduction in the MKL library is optimized using vector instructions, it does not exploit thread-level parallelism. Therefore, the second E5530 CPU can not be used by the implementation, thus, limiting the available bandwidth by half.

CONCLUSIONS Studying the performance of the optimizations presented by Harris [82] on different hardware architectures gained some interesting insides. First, optimizations are not portable across hardware architectures and can even result in incorrect programs on some architectures. Second, the relative performance achieved with optimizations on one architecture is not always achieved on other hardware architectures as well. This let us conclude that performance is *not* portable when using OpenCL or similar low-level approaches.

As a positive result we can see from Figure 5.2 that there exist implementations on the other hardware architectures which offer similar relative performance as the most optimized implementation on the Nvidia GPU. For the AMD GPU, the clBLAS implementation achieves 78.5% of the hardware bandwidth limit and Intel's MKL implementation achieves 87.6% of the hardware limit, considering just the bandwidth of a single CPU socket.

We aim at developing an approach which can systematically apply optimizations and generate code matching the performance on all three architectures, thus, offering performance portability.

5.1.3 *The Need for a Pattern-Based Code Generator*

Our main goal in this chapter is to develop a systematic approach to achieve *performance portability*, i. e., to achieve high relative performance for a given application across a set of different parallel processors. As we saw in the previous section, traditional approaches, like OpenCL, are not performance portable. Currently, programmers often tune their implementations towards a particular hardware using hardware-specific optimizations to achieve the highest performance possible. For example, in Listing 5.5 the Nvidia specific execution behavior of grouping work-items in warps is exploited. This reduces portability, maintainability, and clarity of the code: multiple versions have to be maintained, and non-obvious optimizations make the code hard to understand and to reason about.

We argue that *parallel patterns* can help overcome the tension between achieving the highest possible performance and preserving code portability and maintainability. Parallel patterns declaratively specify the desired algorithmic behavior, rather than encode a particular implementation which might offer suboptimal performance on some hardware architectures. A parallel pattern can be implemented in different ways, optimized towards particular hardware architectures. If the underlying hardware is changed, the optimal implementation for the new hardware can be chosen.

While a compelling idea in theory, existing approaches have fallen short of providing and selecting highly optimized implementations on different architectures. Previous work has been limited to ad-hoc

solutions for specific hardware architectures. This limitation has three main reasons:

First, providing optimized implementations of a pattern on every new hardware platform is a challenging task. Nowadays, dozens of parallel architectures and hundreds of variations of them exist and new architectures are released every year. Therefore, it is often not feasible to provide optimized implementations for all available hardware architectures and existing library approaches have focused on particular hardware architectures. For example, Nvidia GPUs have been the main target for the SkelCL library and, therefore, the skeleton implementations have been optimized appropriately. The *stencil* skeleton implementation, for example, makes heavy use of the local memory feature of OpenCL which is usually not beneficial to be used on a CPU, as CPUs do not feature the corresponding memory area in hardware. An approach using code generation could overcome this drawback, because instead of fixed implementations collected in a library rather possible optimizations are systematically described and then automatically applied by the code generator.

Second, most existing approaches are library-based, which makes the optimization of composition and nesting of patterns extremely complex. In SkelCL, for example, each pattern is implemented as a separate OpenCL kernel. When composing patterns, multiple kernels are executed, but often a better solution would be to fuse multiple kernels into a single kernel, thus avoiding costly operations to write and read the intermediate result into/from the global memory. As fusion of OpenCL kernels in general is complicated and requires code analysis, SkelCL currently cannot execute a fused, and, thus, fully optimized, implementation of composed patterns. Our envisaged approach using code generation should overcome this issue, as the code generator processes the entire pattern-based expression instead of focusing on individual patterns.

Finally, the optimal implementation of a parallel pattern usually depends very much on the application and context the pattern is used in. For example, the algorithmic skeleton *reduce* can be used on its own to implement a parallel summation, as discussed in Section 5.1, but it can also be used as part of the dot product computation which itself is a building block of matrix multiplication, as we saw in Chapter 3. The optimal implementation of *reduce* will most certainly differ across these use cases. Indeed, for the parallel summation the entire parallel processor should be exploited using many OpenCL work-items simultaneously, while as part of the matrix multiplication *reduce* should possibly only exploit thread level parallelism to a limited degree – if at all. An approach using code generation could overcome this issue, as specialized code can be generated for patterns in different contexts instead of providing a single fixed implementation.

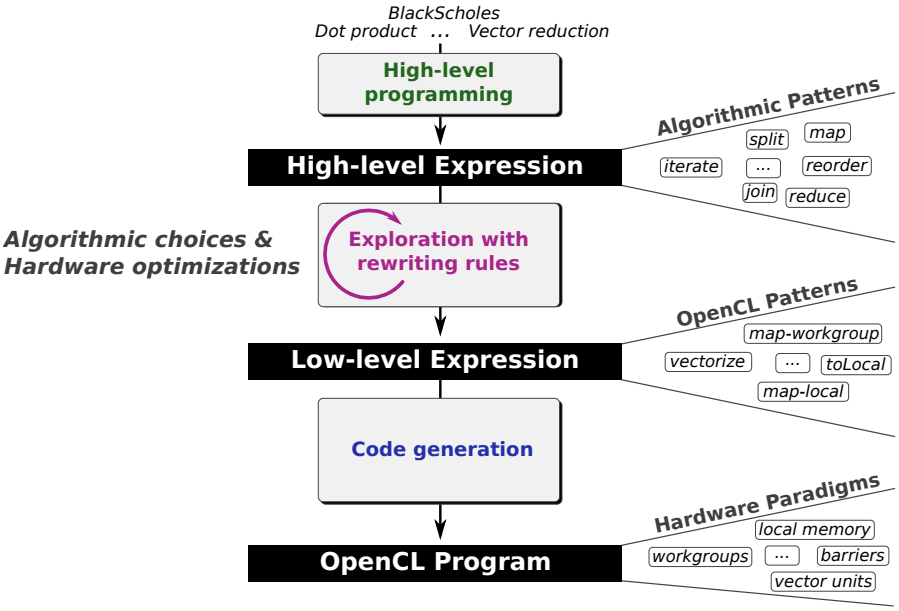


Figure 5.3: Overview of our code generation approach. Problems expressed with high-level algorithmic patterns are systematically transformed into low-level OpenCL patterns using a rule rewriting system. OpenCL code is generated by mapping the low-level patterns directly to the OpenCL programming model representing hardware paradigms.

We argue that the root of the problem lies in a gap in the system stack between the high-level algorithmic patterns on the one hand and low-level hardware optimizations on the other hand. We propose to bridge this gap using a novel pattern-based code generation technique. A set of rewrite rules systematically translates high-level algorithmic patterns into low-level hardware patterns. The rewrite rules express different algorithmic and optimization choices. By systematically applying the rewrite rules semantically equivalent, low-level expressions are derived from high-level algorithm expressions written by the application developer. Once derived, high-performance code based on these expressions can be automatically generated. The next section introduces an overview of our approach.

5.2 OVERVIEW OF OUR CODE GENERATION APPROACH

The overview of our pattern-based code generation approach is presented in Figure 5.3. The programmer writes a *high-level expression* composed of *algorithmic patterns*. Using a rewrite rule system, we transform this high-level expression into a *low-level expression* consisting of *OpenCL patterns*. At this rewrite stage, algorithmic and optimization choices in the high-level expression are explored. The generated low-level expression is then fed into our code generator that emits an *OpenCL program* which is, finally, compiled to machine code

by the vendor-provided OpenCL compiler. A major advantage of our approach is that there is no analysis or optimizations performed in the code generator: optimization decisions are made earlier in the rule rewrite system. This results in a clear separation of concerns between the high-level patterns used by the programmer and the low-level hardware paradigms that enable performance portability.

5.2.1 *Introductory Example*

We illustrate the advantages of our approach using a simple vector scaling example shown in Figure 5.4 on page 136. The user expresses the computation by writing a high-level expression using the algorithmic *map* pattern as shown in Figure 5.4a. This coding style is similar to functional programming as we saw with the SkelCL programming model introduced in Chapter 3. As common in functional programming, the $\circ$ operator represents sequential function composition.

Our technique first rewrites the high-level expression into a representation closer to the OpenCL programming model. This is achieved by applying rewrite rules which are presented later in Section 5.4. Figure 5.4b shows one possible derivation of the original high-level expression. Other derivations are possible with different optimizations applied. The particular expression shown in Figure 5.4b features vectorization as one possible optimization technique. In the derived low-level expression, starting from the last line, the input is split into chunks of a fixed number of elements, 1024 in the example. Each chunk is then mapped onto an OpenCL work-group with the *map-workgroup* low-level pattern (line 2). Within a work-group (line 3—line 5), we vectorize the elements (line 5), each mapped to a local work-item inside a work-group via the *map-local* pattern (line 3). Each local work-item now processes 4 elements, enclosed in a vector type. Finally, the *vect-4* pattern (line 4) vectorizes the user-defined function `mul3`. The exact meaning of our patterns will be given in Section 5.3.

The last step consists of traversing the low-level expression and generating OpenCL code for each low-level pattern encountered (Figure 5.4c). Each map patterns generates a for-loop (line 5—line 6 and line 9—line 10) that iterate over the input array assigning work to the work-groups and local work-items. The information of how many chunks each work-group and work-items processes comes from the corresponding *split*. In line 13 the vectorized version of the user-defined `mul3` function (defined in line 1) is applied to the input array.

To summarize, our approach consists of generating OpenCL code starting from a single portable high-level program representation of a program. This is achieved by systematically transforming the high-level expression into a low-level form suitable for code generation. The next three sections present our high-level and low-level patterns, the rewrite rules, and the code generation mechanism.

```
1 mul3 x     = x * 3    // user-defined function
2 vectorScal = map mul3          // map pattern
```

(a) **High-level expression** written by the programmer.

rewrite rules

```
1 mul3 x     = x * 3
2 vectorScal = join ∘ map-workgroup (
3                     asScalar ∘ map-local (
4                       vect 4 mul3
5                     ) ∘ asVector 4
6                   ) ∘ split 1024
```

(b) **Low-level expression** derived using rewrite rules.

code generator

```
1  float4 mul3(float4 x) { return x * 3; }
2
3  kernel vectorScal(global float* in,
4                    global float* out, int len) {
5   for (int i =get_group_id(0); i < len/1024;
6            i+=get_num_groups(0)) {
7    global float* grp_in  = in+(i*1024);
8    global float* grp_out = out+(i*1024);
9    for (int j =get_local_id(0); j < 1024/4;
10            j+=get_local_size(0)) {
11     global float4* in_vec4 =(float4*)grp_in+(j*4);
12     global float4* out_vec4=(float4*)grp_out+(j*4);
13     *out_vec4 = mul3(*in_vec4);
14 } } }
```

(c) **OpenCL program** produced by our code generator.

Figure 5.4: Pseudo-code representing vector scaling. The user maps the `mul3` function over the input array (a). This high-level expression is transformed into a low-level expression (b) using rewrite rules. Finally, our code generator turns the low-level expression into an OpenCL program (c).

5.3 PATTERNS: DESIGN AND IMPLEMENTATION

In this section, we will introduce the *patterns* which form the expressions written by the programmer and used for code generation. As we saw in the previous section there exist two type of patterns: high-level algorithmic patterns and low-level OpenCL patterns. Some of the high-level algorithmic patterns directly correspond to algorithmic skeletons we have introduced in Chapter 3. As we will also introduce patterns which we have not seen so far and which do not oblige to the common definition of algorithmic skeletons, we will use the more generic term *pattern* throughout this and the next chapter.

The key idea of our approach is to expose algorithmic choices and hardware-specific program optimizations as rewrite rules (discussed later in Section 5.4) which systematically transform pattern-based expressions. The high-level algorithmic patterns represent structured parallelism. They can either be used by the programmer directly as a stand-alone language, but could also be used as a domain specific language embedded in a general purpose programming language, or used as an intermediate representation targeted by the compiler of another programming language. Once a program is represented with our high-level patterns, we systematically transform the program into low-level patterns. The low-level patterns represent hardware-specific concepts expressed by a programming model such as OpenCL, which is the target chosen for this thesis. Following the same methodology, a different set of low-level patterns could be designed to target other low-level programming models such as Pthreads or MPI.

5.3.1 *High-level Algorithmic Patterns*

We define our patterns as functions. To simplify our implementation, we encode all types as arrays with primitives represented by arrays of length 1. The only exceptions are the user-defined functions, such as the `mul3` function in Figure 5.4a that operates on primitive types.

Table 5.1 presents our high-level patterns used to define programs at the algorithmic level. Most of the patterns are well known in functional programming, like *map* and *reduce*. The *zip*, *split* and *join* patterns transform the shape of the data. The *iterate* pattern iteratively applies a function multiple times. Finally, the *reorder* pattern lets our system know that it is safe to reorder the elements of an array arbitrarily, which enables additional optimizations – as we will see later in Chapter 6.

In the following, we discuss each high-level algorithmic pattern in more detail including their formal definitions. As in Chapter 3 we use the Bird-Meertens formalism [18] as inspiration for our notation of the patterns. For full details on the notation see Section 3.2.2 on page 36. Here is a short reminder of our notation: we write function

Pattern	Description
map	Apply a given function to every element of an input array.
reduce	Perform a reduction of an input array using a user-defined binary function and an initial value.
zip	Build an array of pairs by pairwise combining two arrays.
split	Produce a multi-dimensional array by splitting an array in chunks of a given size.
join	Join the two outer most dimensions of an multi-dimensional array.
iterate	Iterate a given function over an input array a fixed number of times.
reorder	Reorder the element of the input array.

Table 5.1: High-level algorithmic patterns used by the programmer.

application using a space and functions are often curried; binary operators (e. g., $\oplus$) are written using infix notation and can be written using prefix notation when being sectioned by using parenthesis, i. e.,: $a \oplus b = (\oplus)\ a\ b$. For an array xs of length n with elements x_i we write $[x_1, \ldots, x_n]$.

In this chapter we are especially interested in how patterns can be composed and nested. As *types* formally specify which compositions and nesting of patterns are legal, we will define the type of each pattern. We write $e : \sigma$ to denote that expression e has type σ. For a function mapping values of type α to values of type β we write its type as $(\alpha \to \beta)$. Tuple types are written as $\langle \alpha, \beta \rangle$. Finally, arrays have their length denoted as part of their type: for an array with elements of type α and length n, we write $[\alpha]_n$.

A formal description of a core subset of the patterns described in this section can also be found in our paper [143], where formal syntax, typing rules, and formal semantics using a denotational style are given.

MAP The *map* pattern is well known in functional programming and applies a given unary function f to all elements of an input array. In Chapter 3, we defined the *map* pattern as an algorithmic skeleton (see Definition 3.1). The same definition holds here. We repeat it here for completeness and we add the type information:

DEFINITION 5.1. *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* f *be a unary customizing function defined on elements. The map pattern is then defined as follows:*

$$map\ f\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

The types of f, xs, *and map are as follows:*

$$\begin{aligned} f &: (\alpha \to \beta), \\ xs &: [\alpha]_n, \\ map\ f\ xs &: [\beta]_n. \end{aligned}$$

In Chapter 3 we also defined the *map* skeleton for operating on matrices (see Definition 3.2). In this chapter we represent matrices as nested arrays, therefore, performing an operation on each element of a matrix can be represented by nesting two *map* patterns:

$$mapMatrix\ \ f\ \ X = map\ (map\ f)\ X$$

Let us assume that X represents an $n \times m$ matrix with elements of type α, then its type is $\left[[\alpha]_m\right]_n$. The outer *map* applies its customizing function to every row of matrix X. The customizing function is defined by currying *map* and f, thus, producing a function which applies f to every element of its argument array. Therefore, f will be applied to every element of matrix X.

REDUCE The *reduce* pattern (a. k. a., fold or accumulate) uses a binary operator $\oplus$ to combine all elements of the input array. We require the operator $\oplus$ to be associative and commutative which allows for an efficient parallel implementation. By requiring commutativity, our system can also generate vectorized implementations of the reduction and utilize the efficient coalesced memory access pattern. These are crucial optimizations on modern GPUs as we saw in Section 5.1. In Chapter 3 we defined *reduce* as an algorithmic skeleton (see Definition 3.5). The same definition holds here. We repeat it here for completeness and we add the type information:

DEFINITION 5.2. *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* $\oplus$ *be an associative and commutative binary customizing operator with the identity element* $id_\oplus$. *The reduce pattern is then defined as follows:*

$$reduce\ (\oplus)\ id_\oplus\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [x_1 \oplus x_2 \oplus \cdots \oplus x_n]$$

The types of $(\oplus)$, $id_\oplus$, xs, *and reduce are as follows:*

$$\begin{aligned}(\oplus) &: ((\alpha, \alpha) \to \alpha),\\ id_\oplus &: \alpha,\\ xs &: [\alpha]_n,\\ \mathit{reduce}\ (\oplus)\ id_\oplus\ xs &: [\alpha]_1.\end{aligned}$$

This definition is unambiguous and well-defined even without explicit parenthesis as we require the binary operator to be associative and commutative.

ZIP The *zip* pattern and the *split*/*join* patterns transform the shape of the data. The *zip* pattern fuses two arrays into a single array of pairs.

DEFINITION 5.3. *Let* xs *and* ys *be arrays of size* n *with elements* x_i *and* y_i *where* $0 < i \leqslant n$. *The zip pattern is then defined as follows:*

$$\mathit{zip}\ [x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] \stackrel{\mathit{def}}{=} [\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots, \langle x_n, y_n\rangle]$$

The types of xs, ys, *and zip are as follows:*

$$\begin{aligned}xs &: [\alpha]_n,\\ ys &: [\beta]_n,\\ \mathit{zip}\ xs\ ys &: [\langle \alpha, \beta\rangle]_n.\end{aligned}$$

This definition significantly differs from the definition of the *zip* skeleton in Chapter 3: While in Definition 3.3, *zip* applies a given function to pairs of elements, there is no function to be applied in Definition 5.3 of the *zip* pattern. The behavior of the *zip* skeleton from SkelCL can be achieved by composing the *zip* pattern with the *map* pattern:

$$\mathtt{zipWith}\ f\ xs\ ys = \mathit{map}\ f\ (\mathit{zip}\ xs\ ys)$$

SPLIT AND JOIN The *split* pattern, which is most often combined with the *join* pattern, partitions an array into chunks of specific size, resulting in an extra dimension in the output array.

We start with the definition of the *split* pattern.

DEFINITION 5.4. *Let* n *be an integer value. Let* xs *be an array of size* m *with elements* x_i *where* $0 < i \leqslant m$. *Let us assume that* m *is evenly divisible by* n. *The split pattern is then defined as follows:*

$$\begin{aligned}&\mathit{split}\ n\ [x_1, x_2, \ldots, x_m] \stackrel{\mathit{def}}{=}\\ &\quad \big[[x_1, \ldots, x_n], [x_{n+1}, \ldots, x_{2n}], \ldots, [x_{m-n+1}, \ldots, x_m]\big]\end{aligned}$$

The types of n, xs, *and split are as follows:*

$$\begin{aligned} n &: \mathrm{int}, \\ xs &: [\alpha]_m, \\ \mathit{split}\ n\ xs &: \left[[\alpha]_n\right]_{\frac{m}{n}}. \end{aligned}$$

The corresponding *join* pattern does the opposite: it reassembles an array of arrays by merging dimensions.

DEFINITION 5.5. *Let* xs *be an array of size* $\frac{m}{n}$ *whose elements are arrays of size* n. *We denote the elements of the* i*th inner array as* $x_{((i-1)\times n)+j}$ *where* $0 < i \leqslant \frac{m}{n}$ *and* $0 < j \leqslant n$. *The join pattern is then defined as follows:*

$$\begin{aligned} &\mathit{join}_n\ \left[[x_1,\ldots,x_n],[x_{n+1},\ldots,x_{2n}],\ldots,[x_{m-n+1},\ldots,x_m]\right] \stackrel{\mathit{def}}{=} \\ &\qquad [x_1,x_2,\ldots,x_m] \end{aligned}$$

The types of xs *and* join_n *are as follows:*

$$\begin{aligned} xs &: \left[[\alpha]_n\right]_{\frac{m}{n}}, \\ \mathit{join}_n\ xs &: [\alpha]_m. \end{aligned}$$

We will almost always omit the subscript, as n can be inferred from the length of the input array. From these definitions it follows, that the compositions of *split* and *join*: $\mathit{join}_n \circ \mathit{split}\ n$ and $\mathit{split}\ n \circ \mathit{join}_n$ for any value n yields the same type and also does not change any element in the array, i. e., it is equivalent to the identify function *id*.

ITERATE The *iterate* pattern corresponds to the mathematical definition of iteratively applying a function, which is defined as: $f^0 = id$ and $f^{n+1} = f^n \circ f$.

DEFINITION 5.6. *Let* n *be an integer value with* $n \geqslant 0$. *Let* f *be a unary function on arrays. Let* xs *be an array of arbitrary size. We define the iterate pattern recursively:*

$$\begin{aligned} &\mathit{iterate}\ 0\ f\ xs \stackrel{\mathit{def}}{=} xs, \\ &\mathit{iterate}\ n\ f\ xs \stackrel{\mathit{def}}{=} \mathit{iterate}\ (n-1)\ f\ (f\ xs) \end{aligned}$$

The types of n, f, xs, *and iterate are as follows:*

$$\begin{aligned} n &: int, \\ f &: ([\alpha]_k \to [\alpha]_{F(k)}), \forall k \textit{ and where} \\ & \qquad F : (int \to int) \textit{ describes the change} \\ & \qquad \textit{of array length when applying } f, \\ xs &: [\alpha]_m, \\ \textit{iterate } n\ f\ xs &: [\alpha]_{F^n(m)}. \end{aligned}$$

The type of the *iterate* pattern is interesting as its result type depends on the effect f has on the size of its argument. The index function F describes the effect the function f has on the length of its input array. This index function is used to compute the effect the *iterate* function has when applying f n times on its input array. Please note, that the length of the input array of f, i. e., k, possibly changes every time f is applied by *iterate*. A formal treatment of the topic using a denotational semantics can be found in our paper [143].

REORDER The *reorder* pattern is used to specify that the ordering of the elements of an array does not matter.

DEFINITION 5.7. *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* σ *be an arbitrary permutation of* $[1, \ldots, n]$. *The reorder pattern is then defined as follows:*

$$\textit{reorder}\ [x_1, \ldots, x_n] \overset{def}{=} [x_{\sigma(1)}, \ldots, x_{\sigma(n)}]$$

The types of xs *and reorder are as follows:*

$$\begin{aligned} xs &: [\alpha]_n, \\ \textit{reorder } xs &: [\alpha]_n \end{aligned}$$

This definition allows to pick any permutation, i. e., a bijective function mapping values from the domain $[1, \ldots, n]$ to the image $[1, \ldots, n]$, for reordering the elements of an array arbitrarily which may enable optimizations, as we will see later.

5.3.2 *Low-level, OpenCL-specific Patterns*

In order to achieve the highest performance, programmers often use a set of intuitive "rules of thumb" to optimize their applications. We extensively discussed one application example in Section 5.1. Each hardware vendor provides own optimization guides [1, 10, 44, 126] that extensively cover vendor's hardware particularities and optimizations. The main idea behind our approach is to identify common optimiza-

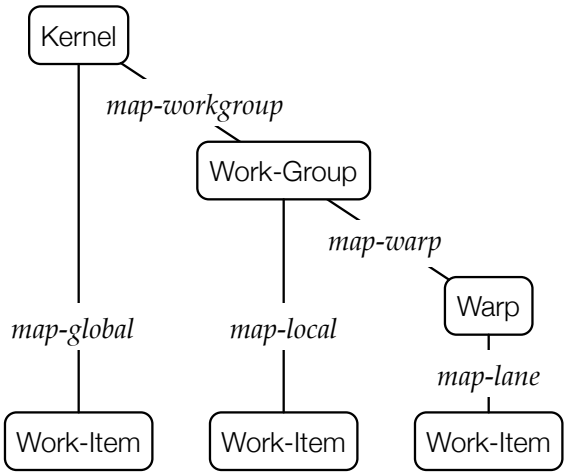


Figure 5.5: The OpenCL thread hierarchy and the corresponding parallel *map* patterns.

tions and express them systematically rather than intuitively, using low-level patterns coupled with a rewrite-rule system.

Table 5.2 gives an overview of the OpenCL-specific patterns we have identified.

PARALLEL MAPS The different low-level OpenCL *map* patterns represent possible ways of mapping computations to the hardware and exploiting thread-level parallelism in OpenCL. The execution semantics and types of all these low-level OpenCL *map* patterns are the same as of the high-level *map* pattern shown in Definition 5.1.

Figure 5.5 shows the OpenCL thread hierarchy together with the corresponding parallel *map* patterns:

- The *map-global* pattern assigns work to work-items independent of their work-group, as shown on the left in Figure 5.5. Following the definition of *map*, each OpenCL work-item executes its customizing function on a different part of the input array.

- The *map-workgroup* pattern assigns work to an OpenCL work-group and the *map-local* pattern assigns work to a local work-item inside a work-group. This is shown in the middle of Figure 5.5. The *map-local* pattern only makes sense in the context of a work-group and is, therefore, only correctly used when nested inside a *map-workgroup* pattern, e. g., *map-workgroup* (*map-local* f).

- There are two additional patterns which are only valid when generating code for Nvidia GPUs: *map-warp* and *map-lane*. These are shown on the right of Figure 5.5. These two patterns capture the notion of *warps* present in Nvidia's GPU architectures, i. e., 32 work-items are grouped and executed together (see Chapter 2 for details). Barrier synchronizations between work-items

Pattern	Description
map-workgroup	Each OpenCL **work-group** applies the given function to an element of the input array.
map-local	Each **local work-item** of a work-group applies the given function to an element of the input array.
map-global	Each **global work-item** applies the given function to an element of the input array.
map-warp	Each **warp** applies the given function to an element of the input array. Only available for Nvidia GPUs.
map-lane	Each **work item inside a warp** applies the given function to an element of the input array. Only available for Nvidia GPUs.
map-seq	Apply the given function to every element of the input array **sequentially**.
reduce-seq	Perform the reduction using the given binary reduction function and initial value on the input array **sequentially**.
reorder-stride	Access input array with a given stride to maintain **memory coalescing**.
toLocal	Store the results of a given function to **local memory**.
toGlobal	Store the results of a given function to **global memory**.
asVector	Turns the elements of an array into **vector type** of a given width.
asScalar	Turns the elements of an array into **scalar type**.
vectorize	**Vectorize** a given function by a given width.

Table 5.2: Low-level OpenCL patterns used for code generation.

in the same warp can be avoided because a warp executes in a lock-step manner. To exploit this optimization we provide the *map-warp* pattern which assigns work to a warp, i. e., a group of 32 work-items. The *map-lane* pattern is used to assign work to an individual work-item inside a warp.

SEQUENTIAL MAP AND REDUCE The *map-seq* and *reduce-seq* patterns perform a sequential map and reduction, respectively, within a single work-item.

For the *map-seq* pattern, the semantics and type are the same as for the high-level *map* pattern shown in Definition 5.1. This is not the case for the *reduce-seq* and *reduce* patterns, where their types differ. For the high-level *reduce* pattern, we require the customizing binary operator to be associative and commutative in order to allow for an efficient parallel implementation. As the *reduce-seq* pattern performs a sequential reduction, we can relax these requirements, therefore, we define *reduce-seq* separately.

DEFINITION 5.8. *Let* xs *be an array of size* n *with elements* x_i *where* $0 < i \leqslant n$. *Let* $\oplus$ *be a binary customizing operator with the identity element* $id_\oplus$. *The reduce-seq pattern is then defined as follows:*

$$\textit{reduce-seq}\ (\oplus)\ id_\oplus\ [x_1, x_2, \ldots, x_n] \overset{def}{=} [(\ldots((id_\oplus \oplus x_1) \oplus x_2)\ldots \oplus x_n)]$$

The types of $(\oplus)$, $id_\oplus$, $\vec{x}$, *and reduce are as follows:*

$$\begin{aligned} (\oplus) &: ((\alpha, \beta) \to \alpha), \\ id_\oplus &: \alpha, \\ xs &: [\beta]_n, \\ \textit{reduce-seq}\ (\oplus)\ id_\oplus\ xs &: [\alpha]_1. \end{aligned}$$

REORDER-STRIDE The *reorder-stride* pattern enforces a special reordering of an array. In our code generator (see Section 5.5) no code is produced for this pattern, but instead it affects how the following patterns reads it input from memory. This pattern, therefore, indirectly produces a strided-access pattern which ensures that when each work-item accesses multiple elements two consecutive work-items access consecutive memory elements at the same time. This corresponds to the *coalesced memory accesses*, which are beneficial on modern GPUs as discussed in Chapter 2.

DEFINITION 5.9. *Let* s *be an integer value. Let* xs *be an array of size* m *with elements* x_i, *where* $0 < i \leqslant m$. *Let us assume that* m *is evenly divisible by*

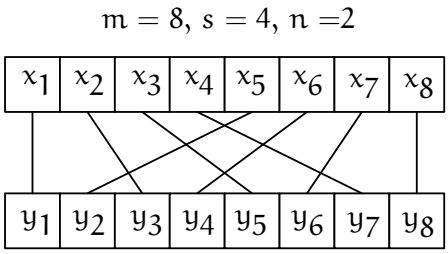


Figure 5.6: Visualization of the *reorder-stride* pattern for an array of size 8 and a stride of 4

s *and that* $m = s \times n$ *for some integer value* n*. The reorder-stride pattern is then defined as follows:*

$$\textit{reorder-stride}\ s\ [x_1, x_2, \ldots, x_m] \overset{def}{=} [y_1, y_2, \ldots, y_m],\ \textit{where}$$
$$y_i \overset{def}{=} x_{((i-1)\ \text{div}\ n + s \times ((i-1)\ \text{mod}\ n)) + 1}$$

Where div *is integer division and* mod *is the modulo operation. The types of* s*,* xs*, and reorder-stride are as follows:*

$$s : \text{int},$$
$$xs : [\alpha]_m,$$
$$\textit{reorder-stride}\ s\ xs : [\alpha]_m.$$

Figure 5.6 visualizes the reordering for an array xs with 8 elements and a stride of 4. In the reordered array the first two elements y_1 and y_2 correspond to x_1 and x_5.

TOLOCAL AND TOGLOBAL The *toLocal* and *toGlobal* patterns are used to specify where the result of a given function f should be stored. As explained in more detail in Chapter 2, OpenCL defines two distinct address spaces: global and local. Global memory is the commonly used large but slow memory. On GPUs, the comparatively small local memory has a high bandwidth with low latency and is used to store frequently accessed data. With these two patterns, we can exploit the memory hierarchy defined in OpenCL.

First, we define *toLocal*:

DEFINITION 5.10. *Let* f *be a function. The toLocal pattern is then defined as follows:*

$$\textit{toLocal}\ f \overset{def}{=} f',\ \textit{where}\ f'\ x \overset{def}{=} f\ x,\ \forall x,\ \textit{and}\ f'\ \textit{is guaranteed to store its result in local memory.}$$

The types of f*, and toLocal are as follows:*

$$\begin{aligned} f &: (\alpha \to \beta), \\ \mathit{toLocal}\ f &: (\alpha \to \beta). \end{aligned}$$

The definition of *toGlobal* is correspondent:

DEFINITION 5.11. *Let* f *be a function. The toGlobal pattern is then defined as follows:*

$$\mathit{toGlobal}\ f \overset{def}{=} f', \quad \textit{where } f'\ x \overset{def}{=} f\ x,\ \forall x \textit{ and } f' \textit{ is guaranteed to store its result in global memory.}$$

The types of f*, and toGlobal are as follows:*

$$\begin{aligned} f &: (\alpha \to \beta), \\ \mathit{toGlobal}\ f &: (\alpha \to \beta). \end{aligned}$$

ASVECTOR, ASSCALAR, AND VECTORIZE The OpenCL programming model supports vector data types such as `float4` where operations on this type will be executed in the hardware vector units. In the absence of vector units in the hardware, the OpenCL compiler generates automatically a version of the code using scalar data types.

The *asVector* and *asScalar* patterns change the data type into vector elements and scalar elements, correspondingly. For instance, an array of `float` is transformed into an array of `float4` as seen in the motivation example (Figure 5.4).

We start by defining the *asVector* pattern.

DEFINITION 5.12. *Let* n *be a positive integer value. Let* xs *be an array of size* m *with elements* x_i *where* $0 < i \leqslant m$*. Let us assume, that* m *is evenly divisible by* n*. The asVector pattern is then defined as follows:*

$$\mathit{asVector}\ n\ [x_1, x_2, \ldots, x_m] \overset{def}{=} \left[\{x_1, \ldots, x_n\}, \{x_{n+1}, \ldots, x_{2n}\}, \ldots, \{x_{m-n+1}, \ldots, x_m\}\right],$$

where $\{x_1, \ldots, x_n\}$ *denotes a vector of width* n*.*
The types of n*,* xs*, and asVector are as follows:*

$$\begin{aligned} n &: \mathit{int} \\ xs &: [\alpha]_m, \\ \mathit{asVector}\ n\ xs &: [\alpha_n]_{\frac{m}{n}}. \end{aligned}$$

Here α *is required to be a basic scalar type, e.g.,* `int` *or* `float`*, and* α_n *denotes the vectorized version of that type with a vector width of* n*.*

The corresponding *asScalar* pattern is defined as follows.

DEFINITION 5.13. *Let* xs *be an array of size* $\frac{m}{n}$ *whose elements are vectors of width* n*. We denote the individual vector elements of the* i*th element of* xs *as* $x_{((i-1)\times n)+j}$ *where* $0 < i \leqslant \frac{m}{n}$ *and* $0 < j \leqslant n$*. The asScalar pattern is then defined as follows:*

$$\begin{aligned} &\mathit{asScalar}_n\ \left[\{x_1,\ldots,x_n\},\{x_{n+1},\ldots,x_{2n}\},\ldots,\{x_{m-n+1},\ldots,x_m\}\right] \\ &\quad \stackrel{\mathit{def}}{=} [x_1, x_2, \ldots, x_m], \end{aligned}$$

where $\{x_1,\ldots,x_n\}$ *denotes a vector of width* n*.*
The types of xs*, and* $\mathit{asScalar}_n$ *are as follows:*

$$\begin{aligned} xs &: [\alpha_n]_{\frac{m}{n}}, \\ \mathit{asScalar}_n\ xs &: [\alpha]_m. \end{aligned}$$

Here α *is required to be a basic scalar type, e.g.,* `int` *or* `float`*, and* α_n *denotes the vectorized version of that type with a vector width of* n*.*

We will almost always omit the subscript for $\mathit{asScalar}_n$, as n can be inferred from the input array.

Finally, we define the *vectorize* pattern.

DEFINITION 5.14. *Let* n *be a positive integer value. Let* f *be a function. The vectorize pattern is then defined as follows:*

$$\mathit{vectorize}\ n\ f \stackrel{\mathit{def}}{=} f_n, \text{ where } f_n\ \{x_1,\ldots,x_n\} = \{f\ x_1,\ldots,f\ x_n\}$$

and $\{x_1,\ldots,x_n\}$ *denotes a vector of width* n*.*
The types of n*,* f*, and vectorize are as follows:*

$$\begin{aligned} n &: \mathit{int}, \\ f &: (\alpha \to \beta), \\ \mathit{vectorize}\ n\ f &: (\alpha_n \to \beta_n). \end{aligned}$$

Here α *and* β *are required to be basic scalar types, e.g.,* `int` *or* `float`*,* α_n *and* β_n *denote vectorized versions of these types with a vector width of* n*.*

5.3.3 *Summary*

In this section we introduced two type of *patterns*: high-level algorithmic patterns and low-level, OpenCL-specific patterns. While all of these patterns can be used by the application programmer to describe the solution for a particular problem, we expect the programmer to focus on the algorithmic patterns which should be used to express a high-level algorithmic implementation of the problem solution. We will see in the next section how such an implementation composed of our high-level algorithmic patterns can be systematically rewritten

using *rewrite rules*. During this process the original implementation will be modified and OpenCL-specific patterns will be used.

5.4 REWRITE RULES

This section introduces our set of rewrite rules that transform high-level expressions written using our algorithmic patterns into semantically equivalent expressions. One goal of our approach is to keep each rule as simple as possible and only express one fundamental concept at a time. For instance the vectorization rule, as we will see, is the only rule expressing the vectorization concept. This is different from most previous library or compiler approaches which provide or produce special vectorized versions of different algorithmic patterns such as map or reduce. The advantage of our approach lies in the power of composition: many rules can be applied successively to produce expressions that compose hardware concepts or optimizations and that are provably correct by construction.

Similarly to our patterns, we distinguish between algorithmic and OpenCL-specific rules. Algorithmic rules produce derivations that represent different algorithmic choices. Our OpenCL-specific rules transform expressions to OpenCL patterns. Once the expression consists only of OpenCL patterns, it is possible to produce OpenCL code for each single pattern straightforwardly with our code generator as described in the following section.

We write $a \to b$ for a rewrite rule which allows to replace the occurrence of term a in an expression with term b. Sometimes multiple rewrites are valid then we write $a \to b \mid c$ to indicate the choice to replace term a either with term b or term c.

For each rule we provide a proof of its correctness, i. e., that applying the rewrite rule does not change the semantic of the expression the rule is applied to. We will discuss some proofs directly here in this section. All proofs can be found in Appendix A.

5.4.1 *Algorithmic Rules*

Each algorithmic rule formulates a provably correct statement of the relationship of multiple algorithmic patterns. Applying the rules allows to rewrite an expression and, by doing so, explore different implementations. As the algorithmic rules are separated from the OpenCL rules, these rules can explore valid implementations regardless of their concrete implementation in a low-level programming model like OpenCL.

IDENTITY The identity rule in Equation (5.1) specifies that it is always valid to compose any function f with the identity function *id*. As we always operate on arrays, we technically compose f with *map* id.

$$f \rightarrow f \circ map\ id \mid map\ id \circ f \tag{5.1}$$

The *id* function can act as a copy operation; this is, e. g., useful for expressing copies of an array to local memory when composed with the *toLocal* OpenCL pattern: *toLocal* (*map id*).

We show here just the proof for the first option. The proof for the second option is shown in Appendix A.

Proof of Equation (5.1); option 1.
Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned}
(f \circ map\ id)\ xs &= f\ (map\ id\ xs) \\
&\{\text{definition of } map\} \\
&= f\ [id\ x_1, id\ x_2, \ldots, id\ x_n] \\
&\{\text{definition of } id\} \\
&= f\ [x_1, x_2, \ldots, x_n] \\
&\{\text{definition of } map\} \\
&= f\ xs
\end{aligned}$$

□

ITERATE DECOMPOSITION The rule in Equation (5.2) expresses the fact that an iteration can be decomposed into several iterations.

$$\begin{aligned}
&iterate\ 1\ f &&\rightarrow f \\
&iterate\ (m+n)\ f &&\rightarrow iterate\ m\ f \circ iterate\ n\ f
\end{aligned} \tag{5.2}$$

This rule can be proven by induction as shown in Appendix A.

REORDER COMMUTATIVITY The following Equation (5.3) shows that if the data can be reordered arbitrarily, as indicated by the *reorder* pattern, then, it does not matter if we apply a function f to each element before or after the reordering.

$$\begin{aligned}
&map\ f \circ reorder &&\rightarrow reorder \circ map\ f \\
&reorder \circ map\ f &&\rightarrow map\ f \circ reorder
\end{aligned} \tag{5.3}$$

The proof can be found in Appendix A.

SPLIT-JOIN The split-join rule expressed by Equation (5.4) partitions a map into two maps.

$$map\ f \rightarrow join \circ map\ (map\ f) \circ split\ n \tag{5.4}$$

This allows us to nest map patterns in each other and, thus, maps the computation to the thread hierarchy of the OpenCL programming model: using the OpenCL-specific rules (discussed in Section 5.4.2) we can rewrite *map* (*map* f) for example to *map-workgroup* (*map-local* f). This is an expression we have seen in our motivation example (Figure 5.4) for mapping a computation to OpenCL work-group and work-item.

We show the proof here, which employs the definitions of *split*, *map*, and *join*.

Proof of Equation (5.4). We start from the right-hand side and show the equality of both sides. Let $xs = [x_1, \ldots, x_m]$.

$$
\begin{aligned}
&(\mathit{join} \circ \mathit{map}\ (\mathit{map}\ f) \circ \mathit{split}\ n)\ xs = \mathit{join}\ (\mathit{map}\ (\mathit{map}\ f)\ (\mathit{split}\ n\ xs)) \\
&\quad \{\text{definition of } \mathit{split}\} \\
&\quad = \mathit{join}\ (\mathit{map}\ (\mathit{map}\ f)\ [[x_1, \ldots, x_n], \ldots, [x_{m-n+1}, \ldots, x_m]]) \\
&\quad \{\text{definition of } \mathit{map}\} \\
&\quad = \mathit{join}\ [\mathit{map}\ f\ [x_1, \ldots, x_n], \ldots, \mathit{map}\ f\ [x_{m-n+1}, \ldots, x_m]] \\
&\quad \{\text{definition of } \mathit{map}\} \\
&\quad = \mathit{join}\ [[f\ x_1, \ldots, f\ x_n], \ldots, [f\ x_{m-n+1}, \ldots, f\ x_m]] \\
&\quad \{\text{definition of } \mathit{join}\} \\
&\quad = [f\ x_1, \ldots, \ldots, f\ x_m] \\
&\quad \{\text{definition of } \mathit{map}\} \\
&\quad = \mathit{map}\ f\ xs
\end{aligned}
$$

□

REDUCTION We seek to express the reduction function as a composition of other primitive functions, which is a fundamental aspect of our work. From the algorithmic point of view, we first define a partial reduction pattern *part-red*:

DEFINITION 5.15. *Let* xs *be an array of size* m *with elements* x_i *where* $0 < i \leqslant m$. *Let* $\oplus$ *be an associative and commutative binary customizing operator with the identity element* $id_\oplus$. *Let* n *be an integer value where* m *is evenly divisible by* n. *Let* σ *be a permutation of* $[1, \ldots, m]$. *The part-red pattern is then defined as follows:*

$$
\begin{aligned}
&\mathit{part\text{-}red}\ (\oplus)\ id_\oplus\ n\ [x_1, x_2, \ldots, x_m] \overset{def}{=} \\
&\quad [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n)},\ \ldots,\ x_{\sigma(m-n+1)} \oplus \cdots \oplus x_{\sigma(m)}]
\end{aligned}
$$

The types of $(\oplus)$, $id_\oplus$, n, xs, *and part-red are as follows:*

$$\begin{aligned}(\oplus) &: ((\alpha, \alpha) \to \alpha),\\ id_\oplus &: \alpha,\\ n &: \mathrm{int},\\ xs &: [\alpha]_m,\\ \textit{part-red}\ (\oplus)\ id_\oplus\ xs &: [\alpha]_{\frac{m}{n}}.\end{aligned}$$

This partial reduction reduces an array of m elements to an array of m/n elements, without respecting the order of the elements of the input array.

The reduction can be expressed as a partial reduction combined with a full reduction as shown in Equation (5.5). This rule ensures that we end up with one unique element.

$$\textit{reduce}\ (\oplus)\ id_\oplus \quad \to \quad \textit{reduce}\ (\oplus)\ id_\oplus \circ \textit{part-red}\ (\oplus)\ id_\oplus \tag{5.5}$$

The rule can be proven using the definitions of *reduce* and *part-red* as well as exploiting the commutative and associative property of $\oplus$. The proof is shown in Appendix A.

PARTIAL REDUCTION Equation (5.6) shows the rewrite rules for the partial reduction.

$$\begin{aligned}&\textit{part-red}\ (\oplus)\ id_\oplus\ n\\ &\qquad\to \quad \textit{reduce}\ (\oplus)\ id_\oplus\\ &\qquad\mid \quad \textit{part-red}\ (\oplus)\ id_\oplus\ n \circ \textit{reorder}\\ &\qquad\mid \quad \textit{join} \circ \textit{map}\ (\textit{part-red}\ (\oplus)\ id_\oplus\ n) \circ \textit{split}\ m\\ &\qquad\mid \quad \textit{iterate}\ \log_m(n)\ (\textit{part-red}\ (\oplus)\ id_\oplus\ m)\end{aligned} \tag{5.6}$$

The first option for partial reduction leads to the full reduction. This rule is obviously only valid if the types on both sides match.

The next possible derivation expresses the fact that it is possible to reorder the elements to be reduced, expressing the commutativity property we demand in our definition of reduction (see Definition 5.2).

The third option is actually the only place where parallelism is expressed in the definition of our reduction pattern. This rule expressed the fact that it is valid to partition the input elements first and then reduce them independently. This exploits the associativity property we require from the reduction operator.

Finally, the last option expresses the fact that it is possible to reduce the input array in multiple steps, by performing an iterative process where in each step a partial reduction is performed. This concept is very important when considering how the reduction function is

typically implemented on GPUs, as we saw in our discussion of the parallel reduction implementations shown in Listing 5.1–Listing 5.7.

All options can be proven using the definitions of *reduce* and *part-red* as well as the commutativity and associativity of $\oplus$. The third rule is shown via induction. All proofs can be found in Appendix A.

SIMPLIFICATION RULES Equation (5.7) shows our simplification rules. These rules express the fact that consecutive *split-join* pairs and *asVector-asScalar* pairs can be eliminated.

$$\begin{aligned} &\mathit{join}_n \circ \mathit{split}\ n && \to && \mathit{id} \\ &\mathit{split}\ n \circ \mathit{join}_n && \to && \mathit{id} \\ &\mathit{asScalar}_n \circ \mathit{asVector}\ n && \to && \mathit{id} \\ &\mathit{asVector}\ n \circ \mathit{asScalar}_n && \to && \mathit{id} \end{aligned} \tag{5.7}$$

These rules follow directly from the definitions of the involved patterns. The proofs are shown in Appendix A.

FUSION RULES Finally, our rules for fusing consecutive patterns are shown in Equation (5.8).

$$\begin{aligned} &\mathit{map}\ f \circ \mathit{map}\ g && \to && \mathit{map}\ (f \circ g) \\ &\mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_\oplus \circ \mathit{map}\ f && \to && \\ &\qquad \mathit{reduce\text{-}seq}\ \big(\ \lambda\ (a, b)\ .\ a \oplus (f\ b)\ \big)\ \mathit{id}_\oplus \end{aligned} \tag{5.8}$$

The first rule fuses the functions applied by two consecutive maps. The second rule fuses the map-reduce pattern by creating a lambda function that is the result of merging functions f and $(\oplus)$ from the original reduction and map, respectively. This rule only applies to the sequential *reduce* pattern since this is the only implementation not requiring the associativity property required by the more generic *reduce* pattern. The functional programming community has studied more generic rules for fusion [42, 97]. However, as we currently focus on a restricted set of patterns, our simpler fusion rules have, so far, proven to be sufficient.

The proofs for both rules are straightforward. The proof for the first rule is shown here, the one for the second rule is shown in Appendix A.

Proof of Equation (5.8); rule 1. Let $xs = [x_1, \ldots, x_n]$.

$$
\begin{aligned}
(\mathit{map}\ f \circ \mathit{map}\ g)\ xs &= \mathit{map}\ f\ (\mathit{map}\ g\ xs) \\
&\quad \{\text{definition of } \mathit{map}\} \\
&= \mathit{map}\ f\ [g\ x_1, \ldots, g\ x_n] \\
&\quad \{\text{definition of } \mathit{map}\} \\
&= [f\ (g\ x_1), \ldots, f\ (g\ x_n)] \\
&\quad \{\text{definition of } \circ\} \qquad \{\text{definition of } \mathit{map}\} \\
&= [(f \circ g)\ x_1, \ldots, (f \circ g)\ x_n] = \mathit{map}(f \circ g)\ xs
\end{aligned}
$$

□

SUMMARY Figure 5.7 gives an overview of all algorithmic rules defined in this subsection. The rules allow us to formalize different algorithmic implementation strategies: the rewrite rules regarding the *reduce* pattern (Figure 5.7e), for example, specify that an iterative implementation of the reduction as well as a divide-and-conquer style implementation are possible.

The split-join rule (Figure 5.7d) allows a divide-and-conquer style implementation of the *map* pattern. This eventually enables different parallel implementations which we can express with OpenCL, as we will see in the next subsection.

The rules presented here are by no means complete and can easily be extended to express more possible implementations. When adding new patterns to the system, the rules have to be extended as well.

In the next subsection we will discuss OpenCL-specific rewrite rules which allow us to map patter implementations to the low-level OpenCL concepts.

5.4.2 *OpenCL-Specific Rules*

In this section, we discuss our OpenCL-specific rules that are used to apply OpenCL optimizations and to lower high-level algorithmic concepts down to OpenCL-specific ones. The code generation process is described separately in the next section.

MAPS The rule in Equation (5.9) is used to produce the OpenCL-specific map patterns that match the thread hierarchy of OpenCL.

$$
\begin{aligned}
\mathit{map} \rightarrow\ & \mathit{map\text{-}workgroup} \mid \mathit{map\text{-}local} \\
\mid\ & \mathit{map\text{-}global} \mid \mathit{map\text{-}warp} \\
\mid\ & \mathit{map\text{-}lane} \mid \mathit{map\text{-}seq}
\end{aligned}
\tag{5.9}
$$

When generating code the code generator has to ensure that the OpenCL thread hierarchy is respected. For instance, it is only legal

$$f \;\rightarrow\; f \circ \mathit{map}\ \mathit{id} \;\mid\; \mathit{map}\ \mathit{id} \circ f$$

(a) Identity

$$\begin{aligned} &\mathit{iterate}\ 1\ f &&\rightarrow\; f \\ &\mathit{iterate}\ (m+n)\ f &&\rightarrow\; \mathit{iterate}\ m\ f \circ \mathit{iterate}\ n\ f \end{aligned}$$

(b) Iterate decomposition

$$\begin{aligned} &\mathit{map}\ f \circ \mathit{reorder} &&\rightarrow\; \mathit{reorder} \circ \mathit{map}\ f \\ &\mathit{reorder} \circ \mathit{map}\ f &&\rightarrow\; \mathit{map}\ f \circ \mathit{reorder} \end{aligned}$$

(c) Reorder commutativity

$$\mathit{map}\ f \;\rightarrow\; \mathit{join} \circ \mathit{map}\ (\mathit{map}\ f) \circ \mathit{split}\ n$$

(d) Split-join

$$\begin{aligned} &\mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus &&\rightarrow\; \mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus \circ \mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n \\ &\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n &&\rightarrow\; \mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus \\ &&&\mid\; \mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n \circ \mathit{reorder} \\ &&&\mid\; \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n) \circ \mathit{split}\ m \\ &&&\mid\; \mathit{iterate}\ \log_m(n)\ (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ m) \end{aligned}$$

(e) Reduction

$$\begin{aligned} &\mathit{join}_n \circ \mathit{split}\ n \;\mid\; \mathit{split}\ n \circ \mathit{join}_n &&\rightarrow\; \mathit{id} \\ &\mathit{asScalar}_n \circ \mathit{asVector}\ n \;\mid\; \mathit{asVector}\ n \circ \mathit{asScalar}_n &&\rightarrow\; \mathit{id} \end{aligned}$$

(f) Simplification rules

$$\begin{aligned} &\mathit{map}\ f \circ \mathit{map}\ g &&\rightarrow\; \mathit{map}\ (f \circ g) \\ &\mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_\oplus \circ \mathit{map}\ f &&\rightarrow \\ &\qquad \mathit{reduce\text{-}seq}\ \big(\ \lambda\ (a, b)\ .\ a \oplus (f\ b)\ \big)\ \mathit{id}_\oplus \end{aligned}$$

(g) Fusion rules

Figure 5.7: Overview of our algorithmic rewrite rules.

to nest a *map-local* inside a *map-workgroup* and it is not valid to nest a *map-global* or another *map-workgroup* inside a *map-workgroup*. In the current implementation of the code generator context information is maintained which records the nesting of patterns. Therefore, it is easy to detect wrongly nested *map* patterns and reject the malformed code.

Proof of Equation (5.9). All of the options in this rule are correct by definition, as all map patterns share the same execution semantics. □

REDUCTION There is only one rule for lowering the *reduce* pattern to OpenCL (Equation (5.10)), which expresses the fact that the only implementation known to the code generator is a sequential reduction.

$$\textit{reduce}\ (\oplus)\ \textit{id}_{\oplus} \quad \rightarrow \quad \textit{reduce-seq}\ (\oplus)\ \textit{id}_{\oplus} \tag{5.10}$$

Parallel implementations of the reduction are defined at a higher level by composition of other algorithmic patterns, as seen in the previous subsection. Most existing compilers and libraries which parallelize the reduction treat it directly as a primitive operation which is not expressed in terms of other more basic operations. With our approach it is possible to explore different implementations for the reduction by applying different rules.

The proof is straightforward (see Appendix A) given the associativity and commutativity of the $\oplus$ operator.

REORDER Equation (5.11) presents the rule that reorders elements of an array. The current implementation of the code generator supports two types of reordering: no reordering, represented by the *id* function, and reordering with a certain stride s: *reorder-stride* s As described earlier, the major use case for the stride reorder is to enable coalesced memory accesses.

$$\textit{reorder} \quad \rightarrow \quad \textit{id} \quad | \quad \textit{reorder-stride}\ s \tag{5.11}$$

The implementation of the code generator could be easily extended to support other kinds of reordering functions, e. g., for reversing an array, or transposing a matrix.

The proof for the *id* function is trivial. For the other option we have to show, that the *reorder-stride* pattern is a valid reordering, i. e., that it mapping of indices defines a permutation. We do this by showing that the mapping of indices as defined in the *reorder-stride* definition is a bijective function and, therefore, defines a permutation. The details can be found in Appendix A.

LOCAL AND GLOBAL MEMORY Equation (5.12) contains two rules that enable GPU local memory usage.

$$\begin{aligned} \textit{map-local}\ f &\rightarrow \textit{toGlobal}\ (\textit{map-local}\ f) \\ \textit{map-local}\ f &\rightarrow \textit{toLocal}\ (\textit{map-local}\ f) \end{aligned} \tag{5.12}$$

They express the fact that the result of a *map-local* can always be stored either in local memory or back in global memory. This holds since a *map-local* always exists within a *map-workgroup* for which the local memory is defined in OpenCL. These rules allow us to describe how the data is mapped to the GPU memory hierarchy.

As discussed earlier, the implementation of the code generator ensures that the OpenCL hierarchy is respected by only allowing corresponding nestings of *map* patterns.

Proof of Equation (5.12). These rules follow directly from the definition of *toGlobal* and *toLocal*, as these have no effect on the computed value, i. e., they behave like the *id* function. □

VECTORIZATION Equation (5.13) expresses the vectorization rule.

$$\textit{map}\ f \rightarrow \textit{asScalar} \circ \textit{map}\ (\textit{vectorize}\ n\ f) \circ \textit{asVector}\ n \tag{5.13}$$

Vectorization is achieved by using the *asVector* and corresponding *asScalar* patterns which change the element type of an array and adjust the length accordingly. This rule is only allowed to be applied once to a given *map* f pattern. This constrain can easily be checked by looking at the function's f type, i. e., if it is a vector type, the rule cannot be applied. The *vectorize* pattern is used to produce a vectorized version of the customizing function f. Note that the vector width n of the *asVector* pattern has to match with the factor used for the vectorization of the function.

The proof is straightforward given the definitions of all involved patterns and can be found in Appendix A.

SUMMARY Figure 5.8 shows an overview of the OpenCL-specific rewrite rules. Each rule formalizes a different implementation or optimization strategy in OpenCL.

- The map rules (Figure 5.8a) describe the usage of the OpenCL thread hierarchy with work-items and work-groups.
- The reduce rule (Figure 5.8b) specifies the simple sequential implementation of reduction in OpenCL, the parallel reduction is expressed in terms of other patterns as we saw in Section 5.4.1.
- The stride access rule (Figure 5.8c) enables coalesced memory access, which is crucial for performance as we saw in Section 5.1.

$$
\begin{array}{lcl}
\textit{map} & \rightarrow & \textit{map-workgroup} \mid \textit{map-local} \\
\mid & & \textit{map-warp} \mid \textit{map-lane} \\
\mid & & \textit{map-global} \mid \textit{map-seq}
\end{array}
$$

(a) Map

$$
\textit{reduce}\ (\oplus)\ \textit{id}_{\oplus} \rightarrow \textit{reduce-seq}\ (\oplus)\ \textit{id}_{\oplus}
$$

(b) Reduction

$$
\textit{reorder} \rightarrow \textit{id} \mid \textit{reorder-stride}\ s
$$

(c) Stride accesses or normal accesses

$$
\begin{array}{lcl}
\textit{map-local}\ f & \rightarrow & \textit{toGlobal}\ (\textit{map-local}\ f) \\
\textit{map-local}\ f & \rightarrow & \textit{toLocal}\ (\textit{map-local}\ f)
\end{array}
$$

(d) Local/Global memory

$$
\textit{map}\ f \rightarrow \textit{asScalar} \circ \textit{map}\ (\textit{vectorize}\ n\ f) \circ \textit{asVector}\ n
$$

(e) Vectorization

Figure 5.8: Overview of the OpenCL-specific rewrite rules.

- The local memory rule (Figure 5.8d) allows the usage of the fast local memory. We saw the benefits of using the local memory when evaluating the matrix multiplication expressed using the *allpairs* pattern in Chapter 4.
- Finally, the vectorization rule (Figure 5.8e) enables vectorization, which is a key optimization for the Intel CPU architectures as we will see in Chapter 6.

As for the algorithmic rules, the OpenCL-specific rules presented here are not complete and do not cover all possible optimizations in OpenCL. Nevertheless, we will see in Chapter 6 that these rules are a good starting set for generating efficient OpenCL code.

5.4.3 *Applying the Rewrite Rules*

In this section, we will discuss some examples to show how the rewrite rules can be used to systematically rewrite applications expressed with the patterns introduced in Section 5.3. We will start by looking back at the introductory example from Section 5.2. Then we will look at the parallel reduction example and show that we can

systematically derive optimized implementations equivalent to the implementations discussed in Section 5.1.

5.4.3.1 *A First Example: Scaling a Vector*

Equation (5.14) shows the implementation of the vector scaling example we used in Section 5.2 as our motivation example. The expression shown here directly corresponds to Figure 5.4a.

$$\begin{aligned} \mathit{mul3}\ x &= x \times 3 \\ \mathit{vectorScal} &= \mathit{map}\ \mathit{mul3} \end{aligned} \tag{5.14}$$

The application developer uses the algorithmic pattern *map* together with the customizing function mul3 which multiplies every element with the number 3. The following Equation (5.15) shows how this implementation can be systematically rewritten using the rewrite rules introduced in this section. The numbers above the equal signs refer to Figure 5.7 and Figure 5.8 indicating which rule was used in the step.

$$\begin{aligned} &\mathit{vectorScal} = \mathit{map}\ \mathit{mul3} \\ &\overset{5.7\,d}{=} \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathit{mul3}) \circ \mathit{split}\ n_1 \\ &\overset{5.8\,e}{=} \mathit{join} \circ \mathit{map}\ (\\ &\qquad \mathit{asScalar} \circ \mathit{map}\ (\mathit{vectorize}\ n_2\ \mathit{mul3}) \circ \mathit{asVector}\ n_2 \\ &\quad) \circ \mathit{split}\ n_1 \\ &\overset{5.8\,a}{=} \mathit{join} \circ \mathit{map\text{-}workgroup}\ (\\ &\qquad \mathit{asScalar} \circ \mathit{map\text{-}local}\ (\\ &\qquad\quad \mathit{vectorize}\ n_2\ \mathit{mul3} \\ &\qquad) \circ \mathit{asVector}\ n_2 \\ &\quad) \circ \mathit{split}\ n_1 \end{aligned} \tag{5.15}$$

To obtain the expression shown in Figure 5.4b we select $n_1 = 1024$ and $n_2 = 4$. This expression can then be used to generate OpenCL code. We will discuss the process of OpenCL code generation in the next section. But first we will discuss possible derivations for the parallel reduction.

5.4.3.2 *Systematic Deriving Implementations of Parallel Reduction*

In Section 5.1 we looked at several implementations of the parallel reduction manually optimized for an Nvidia GPU. In this subsection we want to resemble these implementations with corresponding expressions comprised of the patterns presented in Section 5.3. The implementations presented earlier in Listing 5.1–Listing 5.7 are manual implementations where optimizations have been applied ad-hoc. The key difference to the implementations presented in this section is, that

```
1 vecSum = reduce ∘ join ∘ map-workgroup (
2       join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3       iterate 7 (join ∘ map-local (reduce-seq (+) 0) ∘ split 2) ∘
4       join ∘ toLocal (map-local (map-seq id)) ∘ split 1
5   ) ∘ split 128
```

Listing 5.8: Expression resembling the first two implementations of parallel reduction presented in Listing 5.1 and Listing 5.2.

these are systematically derived from a single high-level expression using the rewrite rules introduced in this section. Therefore, these implementations can be generated systematically by an optimizing compiler. The rules guarantee that all derived expressions are semantically equivalent.

Each OpenCL low-level expression presented in this subsection is derived from the high-level expression Equation (5.16) expressing parallel summation:

$$vecSum = reduce\ (+)\ 0 \tag{5.16}$$

The formal derivations defining which rules to apply to reach an expression from the high-level expression shown here are presented in Appendix B for all expressions in this subsection.

FIRST PATTERN-BASED EXPRESSION Listing 5.8 shows our first expression implementing parallel reduction. This expression closely resembles the structure of the first two implementations presented in Listing 5.1 and Listing 5.2. First the input array is split into chunks of size 128 (line 5) and each work-group processes such a chunk of data. 128 corresponds to the work-group size we assumed for our implementations in Section 5.1. Inside of a work-group in line 4 each work-item first copies a single data item (indicated by *split* 1) into the local memory using the *id* function nested inside the *toLocal* pattern to perform a copy. Afterwards, in line 3 the entire work-group performs an iterative reduction where in 7 steps (this equals $\log_2(128)$ following rule 5.7e) the data is further divided into chunks of two elements (using *split* 2) which are reduced sequentially by the work-items. This iterative process resembles the for-loops from Listing 5.1 and Listing 5.2 where in every iteration two elements are reduced. Finally, the computed result is copied back to the global memory (line 2).

The first two implementations discussed in Section 5.1 are very similar and the only difference is which work-item remains active in the parallel reduction tree. Currently, we do not model this subtle difference with our patterns, therefore, we cannot create an expression which distinguishes between these two implementations. This is not a major drawback, because none of the three investigated architec-

```
1 vecSum = reduce ∘ join ∘ map-workgroup (
2       join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3       iterate 7 ( λ xs .
4           join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘
5               reorder-stride ((size xs)/2) $ xs ) ∘
6       join ∘ toLocal (map-local (map-seq id)) ∘ split 1
7   ) ∘ split 128
```

Listing 5.9: Expression resembling the third implementation of parallel reduction presented in Listing 5.3.

```
1 vecSum = reduce ∘ join ∘ map-workgroup (
2       join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3       iterate 7 ( λ xs . join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘
4                          reorder-stride ((size xs)/2) $ xs ) ∘
5       join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘ split 2 ∘
6       reorder-stride 128
7   ) ∘ split (2 × 128)
```

Listing 5.10: Expression resembling the fourth implementation of parallel reduction presented in Listing 5.4.

tures favoured the first over the second implementation, as we saw in Section 5.1. Therefore, our code generator always generates code matching the second implementation, as we will discuss in more detail in the next section.

AVOIDING INTERLEAVED ADDRESSING Listing 5.9 shows our second expression implementing parallel reduction, which resembles the third implementation of parallel reduction shown in Listing 5.3. The *reorder-stride* pattern is used which makes local memory bank conflicts unlikely. Please note that the pattern is used inside the *iterate* pattern. Therefore, the stride changes in every iteration, which is expressed by referring to the size of the array in the current iteration. We use a lambda expression to name the input array (xs), use a size function to access its size, and use the $ operator, known from Haskell, to denote function application, i. e., $f \circ g \$ x = (f \circ g)\ x$.

INCREASE COMPUTATIONAL INTENSITY PER WORK-ITEM Listing 5.10 shows our third expression implementing parallel reduction. This expression resembles the fourth Nvidia implementation shown in Listing 5.4. By replacing the copy operation into the local memory with a reduction of two elements we increase the computational intensity per work-item. The first *reorder-stride* pattern is used to ensure the coalesced memory access when accessing the global memory. As now each work-group processes twice the amount of

```
1  vecSum = reduce ∘ join ∘ map-workgroup (
2        join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3        join ∘ map-warp (
4            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 1 ∘
5            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 2 ∘
6            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 4 ∘
7            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 8 ∘
8            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 16 ∘
9            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 32
10       ) ∘ split 64 ∘
11       iterate 1 ( λ xs . join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘
12                          reorder-stride ((size xs)/2) $ xs ) ∘
13       join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘ split 2 ∘
14       reorder-stride 128
15   ) ∘ split 256
```

Listing 5.11: Expression resembling the fifth implementation of parallel reduction presented in Listing 5.5.

elements, we split the input data in twice the size of a work-group: 2×128. By choosing the numerical parameters we can shift the granularity and amount of work between a single and multiple work items. Here we choose a larger parameter then previously for *split* to increase the amount of work a work-group processes.

AVOID SYNCHRONIZATION INSIDE A WARP Listing 5.11 shows our fourth expression implementing parallel reduction. This expression closely resembles the fifth implementation of the parallel reduction shown in Listing 5.5. The *iterate* pattern has been changed from performing seven iterations down to a single one. This reflects the OpenCL implementation, where the processing of the last 64 elements is performed by a single warp. We express this using the *map-warp* pattern, where inside the *map-lane* pattern is used together with the *split* and *join* patterns to express that each work-item inside the warp performs a reduction of two elements at a time. Instead of using the *iterate* pattern, the single iteration steps has been unrolled, as it was the case in Listing 5.5. The strides of the *reorder-stride* pattern are computed based on the size of the array in each iteration step.

COMPLETE LOOP UNROLLING Listing 5.12 shows our fifth expression implementing parallel reduction. This expression closely resembles the sixth implementation of the parallel reduction shown in Listing 5.6. The difference to the previous expression in Listing 5.11 is small: we replace the *iterate* pattern with the individual iteration steps. As we assume a fixed work-group size of 128 work-items, it is known at compile time that only a single iteration step is required. If a larger

```
1  vecSum = reduce ∘ join ∘ map-workgroup (
2        join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3        join ∘ map-warp (
4            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 1 ∘
5            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 2 ∘
6            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 4 ∘
7            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 8 ∘
8            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 16 ∘
9            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 32
10       ) ∘ split 64 ∘
11       join ∘ map-local (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 64 ∘
12       join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘ split 2 ∘
13       reorder-stride 128
14   ) ∘ split 256
```

Listing 5.12: Expression resembling the sixth implementation of parallel reduction presented in Listing 5.6.

work-group size would be chosen, then the expression shown in Listing 5.12 would reflect this by including additional iteration steps.

FULLY OPTIMIZED IMPLEMENTATION Listing 5.13 shows our final expression implementing parallel reduction. This expression resembles the seventh, i. e., ultimately optimized implementation of the parallel reduction shown in Listing 5.7. As in the original OpenCL implementation, we increase the computational intensity compared to the previous implementations by increasing the number of elements processed by a single work-group. We express this by choosing a larger `blockSize` when splitting the input array the first time. The first *reorder-stride* expression ensures that memory accesses to the global memory are coalesced.

CONCLUSIONS In this subsection, we presented implementations of the parallel reduction exclusively composed of our patterns. These implementations resemble the OpenCL implementations presented in Section 5.1. The presented expressions are all derivable from a simple high-level expression describing the parallel summation. The derivations are a bit lengthy and thus not shown here, but instead in Appendix B.

By expressing highly specialized and optimized implementations we show how flexible and versatile our patterns and rules are. In Chapter 6, we will come back to these expressions and evaluate the performance achieved by the OpenCL code generated from them. We will see in the next section how these expressions can be turned into OpenCL code.

```
1  vecSum = reduce ∘ join ∘ map-workgroup (
2        join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3        join ∘ map-warp (
4            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 1 ∘
5            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 2 ∘
6            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 4 ∘
7            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 8 ∘
8            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 16 ∘
9            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 32
10       ) ∘ split 64 ∘
11       join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 64 ∘
12       join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘
13       split (blockSize/128) ∘ reorder-stride 128
14   ) ∘ split blockSize
```

Listing 5.13: Expression resembling the seventh implementation of parallel reduction presented in Listing 5.7.

Before we look at how OpenCL code is generated, we discuss one additional optimization: fusion of patterns.

5.4.3.3 *Systematic Fusion of Patterns*

Back in Chapter 4 in Section 4.3 we discussed how the sum of absolute values (*asum*) can be implemented in SkelCL. Two algorithmic skeletons, *reduce* and *map*, were composed to express this application as shown in Equation (5.17).

$$\text{asum}\ \vec{x} = \textit{reduce}\ (+)\ 0\ (\ \textit{map}\ (|\,.\,|)\ \vec{x}\ ) \tag{5.17}$$

$$\text{where:} \quad |a| = \begin{cases} a & \text{if } a \geqslant 0 \\ -a & \text{if } a < 0 \end{cases}$$

When evaluating the performance of the SkelCL implementation, we identified a problem: SkelCL treats each algorithmic skeleton separately, thus, forcing the *map* skeleton to write a temporary array back to global memory and then read it again for the next computation, which greatly reduces performance. The temporary array could be avoided, but in the library approach followed by SkelCL it is difficult to implement a generic mechanism for fusing algorithmic skeletons.

By using our pattern-based code generation approach presented in this chapter together with the rewrite rules, we are now able to address this issue. Our fusion rule (shown in Figure 5.7g) allows to fuse two patterns into one, thus, avoiding intermediate results. Figure 5.9 shows how we can derive a fused version for calculating *asum* from the high-level expression written by the programmer.

$$
\begin{aligned}
&\mathit{asum} = \mathit{reduce}\ (+)\ 0 \circ \mathit{map}\ (|\,.\,|)\\
&\overset{5.7e}{=} \mathit{reduce}\ (+)\ 0 \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0) \circ \mathit{split}\ n \circ \mathit{map}\ (|\,.\,|)\\
&\overset{5.7d}{=} \mathit{reduce}\ (+)\ 0 \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0) \circ \mathit{split}\ n \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ (|\,.\,|)) \circ \mathit{split}\ n\\
&\overset{5.7f}{=} \mathit{reduce}\ (+)\ 0 \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0) \circ \mathit{map}\ (\mathit{map}\ (|\,.\,|)) \circ \mathit{split}\ n\\
&\overset{5.7g}{=} \mathit{reduce}\ (+)\ 0 \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0 \circ \mathit{map}\ (|\,.\,|)) \circ \mathit{split}\ n\\
&\overset{5.8a}{=} \mathit{reduce}\ (+)\ 0 \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0 \circ \mathit{map\text{-}seq}\ (|\,.\,|)) \circ \mathit{split}\ n\\
&\overset{5.7e\&5.8b}{=} \mathit{reduce}\ (+)\ 0 \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{reduce\text{-}seq}\ (+)\ 0 \circ \mathit{map\text{-}seq}\ (|\,.\,|)) \circ \mathit{split}\ n\\
&\overset{5.7g}{=} \mathit{reduce}\ (+)\ 0 \circ\\
&\qquad \mathit{join} \circ \mathit{map}\ \big(\mathit{reduce\text{-}seq}\ (\lambda\ a, b\ .\ a + |\,b\,|)\ 0\big) \circ \mathit{split}\ n
\end{aligned}
$$

Figure 5.9: Derivation for *asum* to a fused parallel version. The numbers above the equality sign refer to the rules from Figure 5.7.

We start by applying the reduction rule 5.7e twice: first to replace *reduce* with *reduce* ∘ *part-red* and then a second time to expand *part-red*. We expand *map*, which can be simplified by removing the two corresponding *join* and *split* patterns. Then two *map* patterns are fused and in the next step the nested *map* is lowered into the *map-seq* pattern. We then first transform *part-red* back into *reduce* (using rule 5.7e) and then apply the OpenCL rule 5.8b. Finally, we apply rule 5.7g to fuse the *map-seq* and *reduce-seq* into a single *reduce-seq*. This sequence of transformations results in an expression which allows for a better OpenCL implementation since just a single *map* pattern is used customized with a reduction. No temporary storage is required for the intermediate result in this expression.

One interesting observation is that in the final expression the two customizing functions $+$ and $|\,.\,|$ are merged and a lambda expression has been created which is defined in terms of these functions: $\lambda\ a, b\ .\ a + |\,b\,|$. The generated lambda expression is not associative, as $a + |\,b\,| \neq b + |\,a\,|$. Therefore, a sequential implementation of reduction is used. Nevertheless, the final expression implements a parallel reduction, as the *map* pattern is used with the *split* and *join* patterns

to split the array in a divide-and-conquer style and the last pattern executed is a *reduce* pattern customized with addition which can be implemented in parallel, as we know.

5.4.4 *Towards Automatically Applying our Rewrite Rules*

The rewrite rules presented in this section define a design space of possible implementations for a given program represented as an algorithmic expression. The rules can safely be applied automatically by a compiler as they – provably – do not change the semantics of the input program.

Using the parallel reduction example, we have seen that multiple implementations can be derived for the same high-level algorithmic expression when applying different rules. This leads to the obvious, but non-trivial, central question: which rules should be applied in which order to obtain the best possible implementation for a particular target hardware device.

The system presented in this chapter constitutes the foundational work necessary to be able to raise this question in the first place and is, therefore, just an initial step towards a fully automated compiler generating the most optimized parallel implementation possible for a given hardware device from a single high-level program.

In Section 6.2.1 we will present an prototype implementation of a search tool, which applies the rules completely automatically. Our early tests with the parallel reduction program as an example suggest, that it is possible to automatically find good implementations on three different hardware architectures.

We intend to study techniques for efficiently searching the implementation space in the future, as will discuss in more detail in Section 7.3.

5.4.5 *Conclusion*

In this section, we have introduced a set of rewrite rules which can systematically rewrite expressions written using the patterns introduced earlier in Section 5.3. The power of our approach lies in the composition of the rules that produce complex low-level OpenCL-specific expressions from simple high-level algorithmic expressions.

We have seen how the rules can be used to transform simple expressions written by an application developer into highly specialized and optimized low-level OpenCL expressions. These low-level expressions match hardware-specific concepts of OpenCL, such as mapping computation and data to the thread and memory hierarchy, exploiting memory coalescing, and vectorization. Each single rule encodes a simple, easy to understand, and provable fact. By composition of

the rules we systematically derive low-level expressions which are semantically equivalent to the high-level expressions by construction.

In the next section we will investigate how OpenCL code is generated from the low-level expressions.

5.5 CODE GENERATOR & IMPLEMENTATION DETAILS

In this section, we discuss how a low-level expression comprising patterns from Section 5.3 and possibly derived using the rewrite rules from Section 5.4 is turned into OpenCL code. We will see that this process is surprisingly simple and straightforward. This is due to the fact, that all complex decisions regarding optimizations are made at an earlier stage: when applying the rewrite rules. This design was chosen deliberately: it follows the principle of separation of concerns and keeps the implementation of the code generator simple. The expression used as input explicitly specifies every important detail of the OpenCL code to be generated, such that for every expression there is a one-to-one mapping to OpenCL code.

We will start our discussion by looking at the parallel reduction example and studying how OpenCL code is generated for the expressions discussed in the previous section. We will then show how OpenCL code is generated for each of the patterns defined in Section 5.3. We will see that there are patterns for which it is not possible to generate OpenCL code without making important implementation decisions. These expressions do not specify the OpenCL implementation detailed enough. We can use the rewrite rules presented in Section 5.4 to transform these expression further until, finally, the expression is precise enough for the code generator.

We will then shift our focus to the implementation of the type system and how this helps to implement the static memory allocator inside our code generator.

Finally, we will provide some details about the software infrastructure we used in our implementation.

5.5.1 *Generating OpenCL Code for Parallel Reduction*

In the previous section, we discussed how multiple low-level expressions can be derived from the simple high-level pattern $\mathit{reduce}\ (+)\ 0$. These derived expressions (shown in Listing 5.8–Listing 5.13) resemble the OpenCL implementations we discussed in Section 5.1 at the beginning of this chapter.

The following expression shows the first expression we derived. The same code is shown in Listing 5.8.

$$\begin{aligned}\mathrm{vecSum} = \mathit{reduce} \circ \mathit{join} \circ \mathit{map\text{-}workgroup}\ (\\ \quad \mathit{join} \circ \mathit{toGlobal}\ (\mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathit{id})) \circ \mathit{split}\ 1 \circ\end{aligned}$$

iterate 7 (*join* ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2) ∘
join ∘ *toLocal* (*map-local* (*map-seq id*)) ∘ *split* 1
) ∘ *split* 128

Listing 5.14 shows the OpenCL code generated by our code generator for this expression. The overall structure of the expression can also be found in the OpenCL code, as highlighted on the left-handed side. OpenCL code is generated by traversing the expression and following the data flow. For each pattern visited, OpenCL code is generated. For some patterns, no code is generated, instead they change the array type and, therefore, have an effect on how the code for following patterns is generated.

The outermost for-loop (line 9 in Listing 5.14) is generated for the *map-workgroup* pattern. The loop variable `wg_id` is based on the identifier of the work-group. No code is emitted for the *split* 128 pattern, but rather its type influences the boundaries of the loop generated for the following *map-workgroup* pattern. In each iteration of the loop, a work-group processes a chunk of 128 elements of the input array.

The first block nested inside the for-loop (line 11—line 15) corresponds to the subexpression: *join* ∘ *toLocal* (*map-local* (*map-seq id*)) ∘ *split* 1. As previously, we assume a work-group size of 128 work-items, therefore, we know that after splitting the 128 elements processed by the work-group with *split* 1 each work-item processes *exactly one* element. Based on this knowledge, we do not emit a for-loop for the *map-local* pattern, as we did for the *map-workgroup* pattern, instead we emit the single line 12, where the local identifier of the work-item is obtained. In the following line 13, the `id` function is invoked (defined in line 1) to copy data from the input array to a local array `sdata1`. The index used to read from the global memory is derived from `wg_id` and `l_id`, which ensures that each work-item in each work-group reads a separate element. In contrast, the index for writing only uses `l_id`, because the result is stored in the local memory and, therefore, each work-group operates on a separate copy of `sdata1`. This section of the code is finished by the `barrier` in line 15, which is emitted for the *map-local* pattern to ensure proper synchronization between the work-items in the work-group. There is no synchronization statement necessary for the *map-seq* pattern, as this pattern is only used sequentially in the context of a single work-item.

The second block (line 16—line 35) corresponds to the subexpression performing the iteration: *iterate* 7 (...). For the *iterate* pattern, a for-loop is emitted (line 22) performing the actual iteration. The loop is annotated with a `#pragma unroll 1` statement, which prevents the OpenCL compiler from unrolling the loop. The reason for this implementation is, that we want to control the unrolling of this loop explicitly with the iteration rewrite rule introduced in Section 5.4. Additional code is generated before and after the for-loop. Before, a

```
1  float id(float x) { return x; }
2  float sumUp(float x, float y) { return x+y; }
3  kernel
4    void vecSum(global float* g_idata, global float* g_odata,
5                unsigned int N, local float* sdata) {
6      local float* sdata1 = sdata;
7      local float* sdata2 = &sdata1[128];
8      local float* sdata3 = &sdata2[64];
9      for (int wg_id = get_group_id(0); wg_id < (N / (128));
10          wg_id += get_num_groups(0)) {
11       {                                                  // map-local
12         int l_id = get_local_id(0);
13         sdata1[l_id] = id(g_idata[(wg_id * 128) + l_id]);
14       }
15       barrier(CLK_LOCAL_MEM_FENCE);
16       {                                                  // iterate
17         int size = 128;
18         local float* sin = sdata1;
19         local float* sout =
20                 ((7 & 1) != 0) ? sdata2 : sdata3;
21         #pragma unroll 1
22         for (int j = 0; j < 7; j += 1) {
23           int l_id = get_local_id(0);                    // map-local
24           if (l_id < size / 2) {
25             float acc = 0.0f;
26             for(int i = 0; i < 2; ++i) {
27               acc = sumUp(acc, sin[(l_id * 2) + i]); }
28             sout[l_id] = acc;
29           }
30           barrier(CLK_LOCAL_MEM_FENCE);
31           size = (size / 2);
32           sin = ( sout==sdata3 ) ? sdata3:sdata2;
33           sout = ( sout==sdata3 ) ? sdata2:sdata3;
34         }
35       }
36       {                                                  // map-local
37         int l_id = get_local_id(0);
38         if (l_id < 1) {
39           g_odata[wg_id + l_id] = id(sdata2[l_id]);
40         }
41       }
42     }                                                    // map-workgroup
43 }
```

Listing 5.14: OpenCL code generated for the expression in Listing 5.5.1 implementing parallel reduction.

variable (`size`) capturing the current size of the input array is created (line 17). The size of this variable is updated after the loop, based on the effect of the nested pattern on the size of the array. In this case, a partial reduction is performed reducing the array by half. Therefore, the `size` variable is halved in line 31 after every iteration. A pair of pointers is created (line 18 and line 19) and swapped after each iteration (line 32 and line 33), so that the nested pattern has fixed input and output pointers to operate on.

The nested block ranging from line 23 to line 30 corresponds to the pattern composition nested inside the *iterate* pattern: $\mathit{join} \circ \mathit{map\text{-}local}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2$. For the *map-local* pattern, the local work-group identifier is obtained and an if statement is emitted (line 24). No for-loop is emitted, as it is clear that the size of the array is maximal 128 (from the definition of `size` in line 17), because the variable is halved after each iteration (line 31). The size information of the array is available in the array's type and, therefore, available when code is generated for the *map-local* pattern. Inside the if statement the code for the *reduce-seq* pattern is emitted. A for-loop is generated (line 26) which iterates over the input chunk, which is in this case of size 2, due, to the *split* 2 pattern. An accumulation variable (`acc`, line 25) is initialized with the neutral value of the reduction and then used to store the temporary reduction results. In line 28, the result is stored to the memory using the `out` pointer prepared by the *iterate* pattern. Finally, for the *map-local* pattern a `barrier` for synchronization is emitted.

The final block (line 36—line 41) corresponds to the last subexpression: $\mathit{join} \circ \mathit{toGlobal}\ (\mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathit{id})) \circ \mathit{split}\ 1$. Similar to before, an if-statement is emitted for the *map-local* pattern and the copy from the local to the global memory is performed in line 39 by the code emitted for the *map-seq id* pattern. No synchronization statement has to be emitted for the *map-local* pattern, as there is no following pattern to synchronize with.

The OpenCL codes for the other parallel reduction implementations are similar, as the code generation process remains the same: the expression is traversed following the data flow. For each visited pattern, OpenCL code is emitted, or they influence the code generation by changing the type and, thus, encode information for the following patterns.

In the following section, we study the OpenCL code generated for each pattern individually.

5.5.2 *Generating OpenCL Code for Patterns*

Table 5.3 shows all patterns introduced in Section 5.3. The code generator does not know how to generate OpenCL code for all patterns, for example there are are many different options for implementing the

Algorithmic Patterns		OpenCL Patterns	
map	***zip***	***map-workgroup***	***reduce-seq***
reduce	***split***	***map-local***	***reorder-stride***
reorder	***join***	***map-global***	***toLocal***
	iterate	***map-warp***	***toGlobal***
		map-lane	***asVector***
		map-seq	***asScalar***
			vectorize

Table 5.3: Overview of all algorithmic and OpenCL patterns. Code is generated only for the patterns highlighted in bold.

reduce pattern in OpenCL, as we discussed in Section 5.1. Therefore, the code generator would have to make a choice which of the possible implementations to pick. We want to avoid such situations, as this complicates the implementation of the code generator and limits both its flexibility and the performance of the generated code. In our approach all decisions about the implementation of *reduce* has to be made before the code generator is invoked by applying the rewrite rules presented in Section 5.4 which allow to derive specialized low-level implementations from high-level expressions.

The code generator generates code only for the highlighted patterns in Table 5.3 (all patterns in the last three columns). The three patterns in the first column: *map*, *reduce*, and *reorder*, have to be eliminated from an expression before the code generation process can begin.

We will now discuss in more detail the code generation process for all highlighted patterns from Table 5.3.

ZIP The code generator emits no OpenCL code for the *zip* pattern. Instead *zip*'s type has an effect on how code for following patterns is generated. Let us look at the following example, where the *zip* and *map-global* patterns are used together:

$$\textit{map-global}\ (+)\ (\textit{zip}\ \mathtt{xs}\ \mathtt{ys}) \tag{5.18}$$

When processing this expression, the *zip* pattern is visited first. The type of *zip* makes it explicit to the code generator that, when emitting code for the following *map-global*, two elements – one element from each array – have to be read together. In the implementation, the code generator will investigate the type of the input array before emitting code for the *map-global*.

SPLIT AND JOIN Similar to the *zip* pattern, no OpenCL code is emitted for neither *split* nor *join*. By encoding the size of arrays in

the type system, the information on how the data was shaped by the *split* and *join* patterns is passed to the following patterns. This information is used later when generating the OpenCL code for performing OpenCL memory accesses. We will discuss the type system implementation in more detail in Section 5.5.3.

ITERATE For the *iterate* f n pattern a for-loop is emitted by the code generator. As seen in Listing 5.14 in the previous section, two pointers are generated and swapped after each iteration to provide input and output pointers to the nested pattern. A variable storing the size of the input array is generated and updated after each iteration, based on the effect the function f has on the array size when processing the array.

PARALLEL OPENCL MAPS The OpenCL code generation for each of the *map* patterns (*map-workgroup*, *map-local*, *map-global*, *map-warp*, and *map-lane*) is rather straightforward.

We saw examples of the generated OpenCL code in Listing 5.14 in the previous section. In general, a for-loop is emitted, where the loop variable refers to the corresponding identifier, i. e., the work-group id, local work-item id, global work-item id, and so on. After the loop, an appropriate synchronization mechanism is emitted. As there is no synchronization between work-items of different work-groups, the *map-workgroup* and *map-global* patterns emit no synchronization statement directly, but after these patterns the OpenCL kernel is terminated and a new OpenCL kernel is created to continue the computation. For the *map-local* and *map-warp* patterns, a `barrier` is emitted to synchronize the work-items organized inside a work-group. For the work-items organized inside a warp no synchronization is required, therefore, the *map-lane* emits no synchronization statement.

When the code generator knows statically that a for-loop will be iterated at most once, an if statement is emitted instead. If it is furthermore clear that the loop will be iterated exactly once, the loop can be avoided completely. We saw both of these cases in Listing 5.14 in the previous section. The code generator uses the array size information from the type system for this.

SEQUENTIAL MAP AND REDUCTION The OpenCL implementations of the sequential *map-seq* and *reduce-seq* patterns are shown in Listing 5.15 and Listing 5.16. Both implementations are straightforward. The *map-seq* implementation applies its customizing function to each element of the input array and stores the outputs in an output array. The *reduce-seq* implementation uses an accumulation variable to accumulate the result of the reduction while it iterates through the input array. After the for-loop, the result is stored in the output array.

```
1 for (int i = 0; i < size; ++i) {
2   output[out_index] = f(input[in_index]);
3 }
```

Listing 5.15: Structure of the OpenCL code emitted for the *map-seq* pattern.

```
1 float acc = 0.0f;
2 for (int i = 0; i < size; ++i) {
3   acc = f(acc, input[in_index]);
4 }
5 output[out_index] = acc;
```

Listing 5.16: Structure of the OpenCL code emitted for the *reduce-seq* pattern.

The input and output indices: `in_index` and `out_index` are generated based on the pattern in which the *map-seq* or *reduce-seq* is nested in. We saw in Listing 5.14 that access to the global memory is based on the work-group and work-item identifier, while the local memory access is only based on the work-item identifier, because each work-item has its exclusive copy of local memory to operate on. The *reorder-stride* pattern influences the generation of the indices too, as we describe next.

REORDER-STRIDE No code is emitted for the *reorder-stride* pattern, rather it influences the generation of the next input index which is used to read from memory.

When visiting the *reorder-stride* pattern an information about the stride used (s) is stored on a stack which is consumed the next time an input index is generated. The index generation then emits an index which controls which element is read by which thread.

The formula for *reorder-stride* s (see Definition 5.9) is defined as: $(j-1)/n + s \times ((j-1) \bmod n) + 1$, where j is the 1-based index to be changed and $n = size/s$. In the 0-based indexing in OpenCL the formula simplifies to: $j/n + s \times (j \bmod n)$.

The second expression implementing parallel reduction shown in Listing 5.9 added a *reorder-stride* $size/2$ pattern as compared to the first expression for which we showed the compiled OpenCL code in Listing 5.14. Here $size$ is denoting the size of the input array. When looking at the OpenCL code generated and shown in Listing 5.14, we can see that the index generated for the for-loop in line 27 is: `l_id * 2 + i`. Together with the array length, this gives us:

```
j = l_id * 2 + i
s = size/2
n = 2
```

After applying the *reorder-stride* formula, we get:

```
(l_id * 2 + i) / 2 + (size/2) * ((l_id * 2 + i) % 2)
```

Which can be simplified by applying integer division and modulo rules. Finally, giving us:

```
l_id + (size/2) * i
```

Which is the index used in the OpenCL code after performing the reordering with the *reorder-stride* pattern.

Now we can also see why this reordering is called *reorder-stride*, as `i` is the innermost loop index starting at `0` and incrementing it will multiply the *stride* which is added as an offset to `l_id`. This index will ensure that each work-item reads a different element from the input array than before. In this case, the reordering is applied to avoid local memory bank conflicts.

toLOCAL AND toGLOBAL For both patterns, no OpenCL code is emitted. The patterns change the memory location of the nested pattern, i. e., where the nested pattern should write to. When traversing an expression, these patterns will be always visited first before the nested pattern. The information on where to write is passed along when visiting the nested pattern.

asVECTOR, asSCALAR, AND VECTORIZE The *asVector*, and *asScalar* patterns influence the type system and no OpenCL code is emitted when visiting them. Type casts are emitted by the following patterns to reflect the change of data type in OpenCL.

The *vectorize* pattern is used to vectorize functions. Our current strategy is quite simplistic. When emitting the OpenCL code for a vectorized function, the data types of the input and output arguments are changed appropriately and the body of the function is checked if it contains operations which are supported on vector data types in OpenCL, e. g., the usual arithmetic operators. If the body contains other operations, our process fails and no OpenCL code can be generated. In the future, we plan to incorporate a more advanced approach for vectorization of functions, e. g., like the one presented in [98].

5.5.3 *The Type System and Static Memory Allocation*

Data types are usually primarily used to prevent errors when composing functions. While we use data types for the same purpose in our system, we also use them for an additional purpose: to store information about the size of arrays processed by our expressions. This information is used by the code generator to perform static memory allocation, i. e., to reserve the correct amount of memory required by a pattern.

Our *split* n pattern splits an array in chunks of a fixed size n. As this size is stored in the type system, it is naturally available when processing the next patterns, by investigating their input type. Therefore, there is no need to pass this information explicitly inside the code generator, it is rather implicitly available through the types involved.

When computing the amount of memory required to store the result of a pattern application, we can look at the result type of the pattern and easily infer the amount of memory from it. Using this approach, no dynamic memory allocation is required so far.

We currently do not perform any analysis to reuse memory objects, i. e., each pattern is assigned a newly allocated output array, even though arrays could be reused. In Listing 5.14 three local memory arrays (`sdata1`, `sdata2`, and `sdata3`) are used, while only a single array was used in the original OpenCL implementation. In the future, we intend to develop strategies to reuse memory and, thus, reduce the overall memory footprint. We want to adopt well-understood algorithms which are used for register allocation in compilers, like graph coloring [41].

5.5.4 *Implementation Details*

Our system is implemented in C++, using the template system and support for lambda functions. When generating code for a given low-level expression, two basic steps are performed. First, we use the Clang/LLVM compiler infrastructure to parse the expression and produce an abstract syntax tree for it. Second, we traverse the tree and emit code for every function call representing one of our low-level hardware patterns. We have implemented the type system using template meta-programming techniques.

To simplify our implementation we leverage the SkelCL library infrastructure for eventually executing the generated OpenCL expression and to transfer data to and from the GPU.

5.6 CONCLUSION

In this chapter we introduced a novel code generation technique for pattern-based programs. We started the chapter with an investigation of the portability of optimizations in OpenCL. We concluded, that optimizations are not performance portable in OpenCL and argued for a more systematic approach. In the following we introduced a set of high-level algorithmic patterns – similar to the algorithmic skeletons defined in SkelCL in Part 2 – but also low-level OpenCL-specific patterns resembling the OpenCL programming model. We presented a formal system of rewrite rules which allow for systematically rewriting of pattern-based expressions, especially, for transforming programs expressed with the high-level algorithmic patterns into

OpenCL-specific programs expressed with the low-level patterns. We showed the soundness of our approach by proving that the rewrite rules do not change the program semantics. Finally, we presented an OpenCL code generator which generates OpenCL code for programs expressed with the low-level rules.

The rewrite rules encode different algorithmic as well as low-level optimization choices. By applying the rules we derived expressions resembling manually optimized OpenCL code written by Nvidia and performed an optimization to fuse two pattern, enabling the generation of an optimized implementation for the *asum* benchmark, which we could not generate with SkelCL.

In the next chapter, we will evaluate the performance of the systematically generated OpenCL code. We will also investigate and evaluate a prototype search tool which applies the rewrite rules automatically to find good implementations on three different hardware architectures for achieving performance portability.

6 APPLICATION STUDIES

In this chapter we present application studies evaluating the performance of the OpenCL code generated from pattern-based expressions using our systematic code generation approach presented in the previous chapter. We first discuss our experimental setup used for the runtime experiments. We will then start our evaluation by looking at the parallel reduction example which we used throughout the previous chapter and compare the performance of the manually optimized OpenCL implementations against the systematically generated code. We will discuss a brief case study showing how the rewrite rules can be applied automatically and how effective such an automatic search strategy is for a simple application example. We will then investigate application examples from linear algebra, physics, and mathematical finance.

For all applications we show performance results comparing the generated OpenCL code against vendor provided libraries and manually tuned OpenCL code.

6.1 EXPERIMENTAL SETUP

We used three hardware platforms: an Nvidia GeForce GTX 480 GPU, an AMD Radeon HD 7970 GPU, and a dual socket Intel Xeon E5530 server with 8 cores in total. We used the latest OpenCL runtime from Nvidia (CUDA-SDK 5.5), AMD (AMD-APP 2.8.1) and Intel (XE 2013 R3). The GPU drivers installed were 310.44 for Nvidia and 13.1 for AMD on our Linux system.

We use the profiling APIs from OpenCL and CUDA to measure kernel execution time and the *gettimeofday* function for the CPU implementation. We exclude the data transfer time to and from the GPU in all runtime numbers reported in this chapter, as we want to focus on the quality of the generated OpenCL kernel code. We repeat each experiment 100 times and report median runtimes.

6.2 PARALLEL REDUCTION

In this section we evaluate the performance of three of the low-level expressions presented in Section 5.4.3.2 performing a parallel summation of an array. These expressions resemble corresponding OpenCL code provided by Nvidia discussed at the beginning of Chapter 5.

(a)

```
1 vecSum = reduce ∘ join ∘ map-workgroup (
2       join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3       iterate 7 (join ∘ map-local (reduce-seq (+) 0) ∘ split 2) ∘
4       join ∘ toLocal (map-local (map-seq id)) ∘ split 1
5   ) ∘ split 128
```

(b)

```
1 vecSum = reduce ∘ join ∘ map-workgroup (
2       join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3       iterate 7 ( λ xs . join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘
4                         reorder-stride ((size xs)/2) $ xs ) ∘
5       join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘ split 2 ∘
6       reorder-stride 128
7   ) ∘ split (2 × 128)
```

(c)

```
1  vecSum = reduce ∘ join ∘ map-workgroup (
2        join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
3        join ∘ map-warp (
4            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 1 ∘
5            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 2 ∘
6            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 4 ∘
7            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 8 ∘
8            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 16 ∘
9            join ∘ map-lane (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 32
10       ) ∘ split 64 ∘
11       join ∘ map-local (reduce-seq (+) 0) ∘ split 2  ∘ reorder-stride 64 ∘
12       join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘
13       split (blockSize/128)  ∘ reorder-stride 128
14   ) ∘ split blockSize
```

Figure 6.1: Three low-level expressions implementing parallel reduction.

All three expressions have been systematically derived from the high-level expression for the parallel summation:

$$vecSum = reduce\ (+)\ 0$$

The formal derivations are shown in Appendix B.

Figure 6.1 shows the three expressions we will use for our performance comparison. The first expression (Figure 6.1a) corresponds to the first unoptimized Nvidia implementation (Listing 5.2), the second expression (Figure 6.1b) corresponds to the fourth implementation by Nvidia which has some optimizations applied (Listing 5.4), and the third expression (Figure 6.1c) corresponds to the fully optimized Nvidia implementation (Listing 5.7).

Figure 6.2 shows the performance of these three versions compared to the corresponding native OpenCL implementations by Nvidia and two additional libraries providing implementations of parallel reduction on GPUs. CUBLAS [43] represents the CUDA-specific implemen-

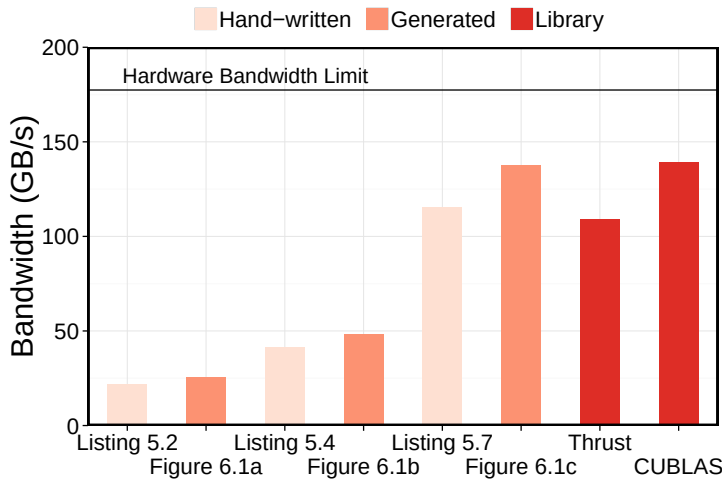


Figure 6.2: Performance comparisons for code generated for three low-level expressions against native OpenCL code.

tation of BLAS that only runs on Nvidia hardware. BLAS does not implement the parallel reduction but instead we used the *asum* benchmark for our comparison which performs a parallel reduction but in addition applies a function computing the absolute value on every element of the input vector. We also include a parallel reduction implemented with Thrust [17], a library for simplified GPU programming developed by Nvidia and based on CUDA.

The results are reported in GB/s, i. e., the achieved memory bandwidth which is computed by dividing the input data size in gigabytes by the absolute runtime in seconds. This metric allows to compare each version against the hardware bandwidth limit, which is marked by a horizontal line at the top of Figure 6.2. The performance of the generated OpenCL code from our three expressions matches, or even slightly outperforms, the corresponding native OpenCL implementations written by Nvidia. These results show that our systematically generated code can offer high performance, once the right set of optimizations – encoded in our rewrite rules – is applied. The performance of our generated code for the fully optimized low-level expression even matches the performance of the highly optimized CUBLAS library written by Nvidia and outperforms the Thrust library.

6.2.1 *Automatically Applying the Rewrite Rules*

For the parallel reduction benchmark we implemented a prototype search tool which automatically applies the rewrite rules for finding low-level expressions which offer high-performance. Our prototype tool starts with the high-level expression $\mathit{reduce}\ (+)\ 0$ and transforms the expression using the rewrite rules and performing runtime exper-

iments with the transformed expressions until a low-level OpenCL expression is found meeting our performance expectations. As discussed earlier in Chapter 5 multiple rewrite rules might be valid to be applied to a given expression, therefore, we implemented a simple strategy for deciding which rules to apply. This simple search strategy is loosely based on Bandit-based optimization [115].

The search is an iterative process. Given an expression we list all the rewrite rules which are valid to be applied. We use a Monte Carlo method for evaluating the potential impact of each rule by randomly walking down the search tree. We execute the code generated from the randomly chosen expressions on the parallel processor using OpenCL and measure its performance. The rule that promises the best performance following the Monte Carlo descent is chosen and the expression after the rule has been applied is used as the starting point for the next iteration. As the rules are chosen randomly the search process is not deterministic and different low-level expressions can be found when applying the search multiple times.

This process is repeated until we reach an expression where there are no rules to be applied to, or a certain depth of the search tree is reached. In addition to selecting the rules, we also search at the same time for parameters controlling our primitives such as the parameter for the *split* n pattern. We have limited the choices for these numerical parameters to a reasonable set of appropriate values for our test hardware.

We envision to replace this simplistic search strategy with more advanced techniques in the future.

FOUND EXPRESSIONS We performed the automatic search on all three of our test platforms for the parallel reduction benchmark.

The best performing low-level expression found by applying our automatic search technique are shown in Figure 6.3. The first expression (Figure 6.3a) was found on the Nvidia platform, the second expression (Figure 6.3b) on the AMD platform, and the third expression (Figure 6.3c) on the Intel platform. We can make several important observations. The first observation is, that none of the expressions make use of the local memory (although our systems fully support it). It is common wisdom that using local memory on the GPU enables high performance and in fact the tuned hand-written implementation by Nvidia uses local memory on the GPU. However, as we will see later in the results section, our automatically derived version is able to perform as well without using local memory. The second key observation is, that each work-item performs a large sequential reduction independent of all other threads, which does not require synchronization and, thus, avoids overheads.

While these observations are the same for all platforms, there are also crucial differences between the different low-level expressions.

(a) Nvidia

```
1 reduce ∘ join ∘ join ∘ map-workgroup (
2     toGlobal (map-local (reduce-seq (+) 0)) ∘ reorder-stride 2048
3   ) ∘ split 128 ∘ split 2048
```

(b) AMD

```
1 reduce ∘ join ∘ asScalar ∘ join ∘ map-workgroup (
2       map-local (map-seq (vectorize 2 id)∘
3         reduce-seq (vectorize 2 (+) 0)
4       ) ∘ reorder-stride 2048
5   ) ∘ split 128 ∘ asVector 2 ∘ split 4096
```

(c) Intel

```
1 reduce ∘ join ∘ map-workgroup (join ∘ asScalar ∘ map-local(
2       map-seq (vectorize 4 id) ∘ reduce-seq (vectorize 4 (+)) 0
3   ) ∘ asVector 4 ∘ split 32768) ∘ split 32768
```

Figure 6.3: Low-level expressions performing parallel reduction. These expressions are automatically derived by our prototype search tool from the high-level expression *reduce* (+) 0.

Both GPU expressions make use of the *reorder-stride* primitive, allowing for coalesced memory accesses. The AMD and Intel expressions are vectorized with a vector length of two and four respectively. The Nvidia version does not use vectorization since this platform does not benefit from vectorized code. On the CPU, the automatic search picked numbers for partitioning into work-groups and then into work-items in such a way that inside each work-group only a single work-item is active. This reflects the fact that there is less parallelism available on a CPU compared to GPUs.

Interestingly, the results of our search have some redundancies in the expressions. For example, we perform unnecessary copies on AMD and Intel by performing a *map-seq* with the identity nested inside. While this does not seem to affect performance much, a better search strategy could probably get rid of these artifacts and might achieve a slightly better performance.

PERFORMANCE OF FOUND EXPRESSIONS Figure 6.4 shows the performance of the code generated for the three expressions performing a parallel reduction shown in Figure 6.3. The three plots show the performance for the Nvidia platform (on the left), the AMD platform (in the middle), and the Intel platform (on the right). All plots are scaled according to the hardware bandwidth limit of the platform. On each platform we compare against a vendor provided, highly tuned implementation of the BLAS library, where we measured the bandwidth achieved for the *asum* application. While in the *asum* application an additional operation (applying the absolute value function) is performed for every data item, we showed in Section 4.3 that the performance difference for the parallel reduction benchmark and the

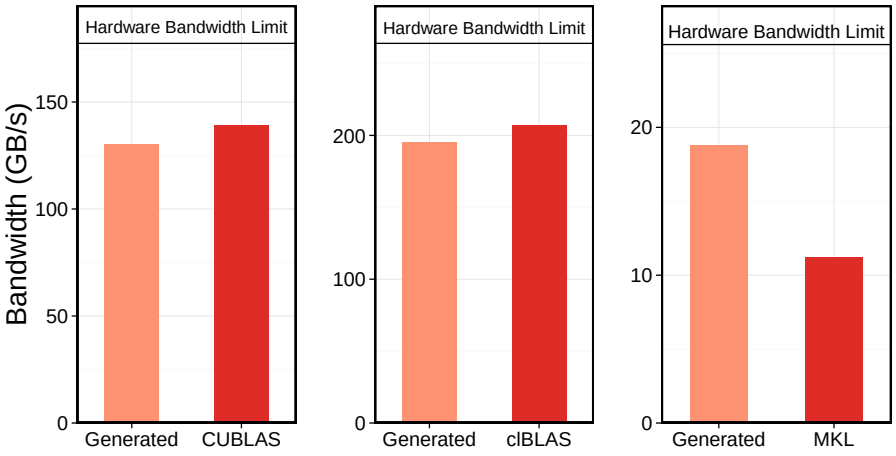


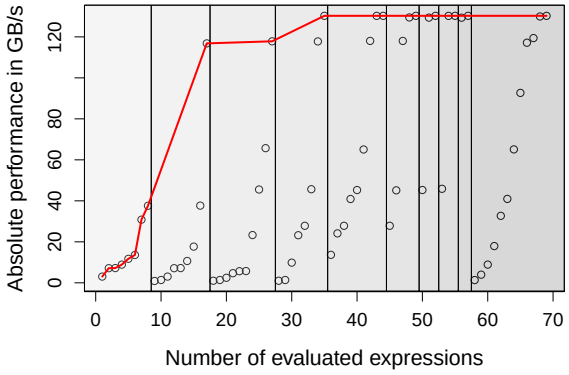
Figure 6.4: Performance comparisons for code generated for three automatically found low-level expressions against hardware-specific library code on three platforms.

asum benchmark is negligible when both are implemented properly. Therefore, we consider the BLAS implementations as valid contenders for a fair performance comparison.

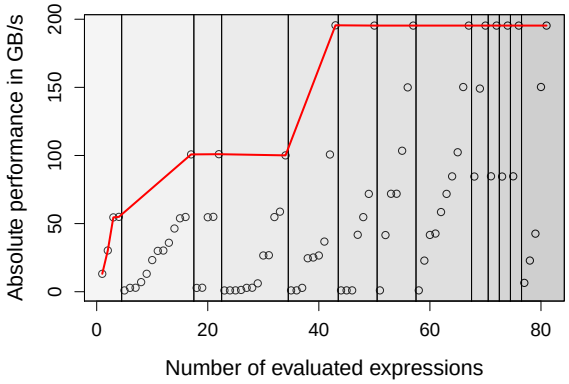
The results shows that the generated code for the automatically found expressions are on par with the CUBLAS implementation of Nvidia and the clBLAS implementation of AMD achieving about 95% of their performance. On the Intel platform our generated code actually outperforms the MKL implementation. This is due to the implementation of MKL and the particular multi-core CPU used in the experiments. For the *asum* benchmark the MKL implementation does not use thread level parallelism, presumably with the assumption that *asum* is a memory bound benchmark. The used multi-core CPU is actually a two socket machine where two chips are combined on a single motherboard. In this configuration there are two memory controllers available – one for each socket. Therefore, thread level parallelism is actual beneficial for the *asum* benchmark on this particular hardware giving our generated code a speedup of 1.67.

The three plots together also show that our approach offers true performance portability. While each individual BLAS implementation is not-portable and bound to a specific hardware architecture, our system automatically searched and found three expressions systematically derived from a single high-level representation which offer high performance on all three platforms. With our approach we achieve the same relative performance on all platforms which is within 75% of the corresponding hardware bandwidth limit.

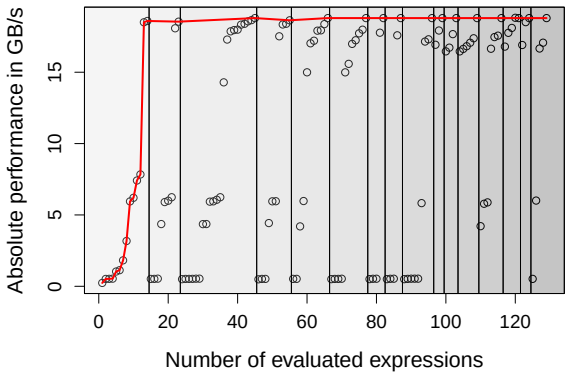
SEARCH EFFICIENCY We now investigate the efficiency of our simple search strategy. Figure 6.5 shows how many expressions were evaluated during the search. The evaluated expressions are grouped



(a) Nvidia GPU



(b) AMD GPU



(c) Intel CPU

Figure 6.5: Search efficiency. The vertical partitioning represents the number of fixed derivations in the search tree. The red line connects the fastest expressions found so far.

from left to right by the number of rules applied in the search tree. The red line connects the fastest expression found so far.

As can be seen the performance improves steadily for all three platforms before reaching a plateau. For both GPUs the best performance is reached after testing about 40 expressions. At this point we have fixed five derivations and found a subtree offering good performance for some expressions. Nevertheless, even in the later stages of the search many expressions offer poor performance, which is mainly due to the sensitivity of GPUs for selecting appropriate numerical parameters. On the CPU performance converges after testing about 20 expressions and more expressions offer good performance. This shows that for this particular benchmark the CPU is easier to optimize for and not as sensitive when selecting numerical parameters. Overall the search took less than one hours on each platform.

These results show, that applying our rules automatically is feasible and capable of finding high performing low-level expressions for high-level expressions written by the programmer. Our approach offers true performance portability where the portable high-level expression is systematically and automatically optimized for a particular hardware architecture delivering similar relative performance on all tested devices. Nevertheless, we only tested our simple automatic search strategy with the parallel reduction as a single benchmark. For more complex applications the search space becomes bigger and more advanced search techniques have to be applied. We intend to explore this topic in the future, as we will discuss in more detail in Section 7.3.

6.3 LINEAR ALGEBRA APPLICATIONS

In this section we evaluate four linear algebra applications: scaling a vector with a constant (*scal*), summing up the absolute values of a vector (*asum*), computing the dot product of two vectors (*dot*), and performing a matrix vector multiplication (*gemv*). We have performed experiments with two input sizes. For *scal*, *asum* and *dot*, the small input size corresponds to a vector size of 64MB. The large input size uses 512MB (the maximum OpenCL buffer size for our platforms). For *gemv*, we use an input matrix of 4096×4096 elements (64MB) and a vector size of 4096 elements (16KB) for the small input size. For the large input size, the matrix size is 8192×16384 elements (512MB) and the vector size 8192 elements (32KB).

We choose linear algebra kernels as benchmarks, because they are well known, easy to understand, and used as building blocks in many other applications. Listing 6.1 shows how the benchmarks are expressed using our high-level patterns. While the first three benchmarks perform computations on vectors, matrix vector multiplication illustrates a computation using 2D data structures.

```
1 scal α xs = map (λ x. α × x) xs
2 asum xs = reduce (+) 0 (map abs xs)
3 dot xs ys = reduce (+) 0 (map (×) (zip xs ys))
4 gemv mat xs ys α β =
    map (+) (zip (map (scal α ∘ dot xs) mat) (scal β ys))
```

Listing 6.1: Linear algebra kernels from the BLAS library expressed using our high-level algorithmic patterns.

For scaling (line 1), the *map* pattern is used for multiply each element with a constant α. The sum of absolute values (line 2) and the dot product (line 3) applications both perform a summation, which we express using the *reduce* pattern customized with the addition. For dot product, a pair-wise multiplication is performed before applying the reduction expressed using the *zip* and *map* patterns.

The line 4 shows matrix vector multiplication as defined in BLAS: $\vec{y} = \alpha A\vec{x} + \beta\vec{y}$. To multiply the matrix mat with the vector xs, the *map* pattern maps the computation of the dot-product with the input vector xs to each row of the matrix mat. Afterwards the resulting vector is added to the scaled vector ys using the *zip* and *map* patterns. Notice how we are reusing the expressions for dot-product and scaling as building blocks for the more complex matrix-vector multiplication. This shows one strength of our system: expressions describing algorithmic concepts can be reused, without committing to a particular low-level implementation. The dot-product from gemv may be implemented in a totally different way from the stand-alone dot-product.

6.3.1 *Comparison vs. Portable Implementation*

First, we show how our approach performs across our three test platforms. We use the BLAS OpenCL implementations written by AMD [150] as our baseline for this evaluation since it is implemented in OpenCL and functionally portable across all test platforms. Figure 6.6 shows the performance of our approach relative to clBLAS for our four benchmarks. As can be seen, we achieve better performance than clBLAS on most platforms and benchmarks. The speedups are the highest for the CPU, with up to 20×. The reason is that clBLAS was written and tuned specifically for an AMD GPU which usually exhibit a larger number of parallel processing units as CPUs. Our systematically derived expression for this benchmark is tuned for the CPU avoiding too much parallelism, which is what gives us such large speedup.

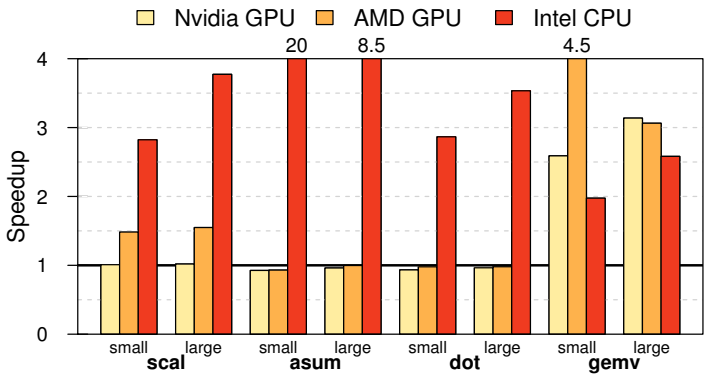


Figure 6.6: Performance of our approach relative to a portable OpenCL reference implementation (clBLAS).

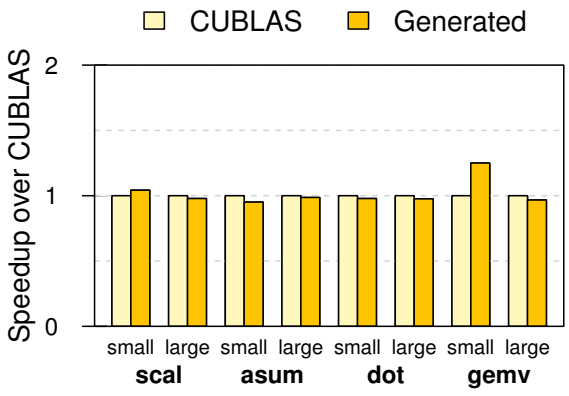
6.3.2 *Comparison vs. Highly-tuned Implementations*

We compare our approach with a state-of-the-art implementation for each platform. For Nvidia, we pick the highly tuned CUBLAS implementation of BLAS written by Nvidia [43]. For the AMD GPU, we use the same clBLAS implementation as before because it has been written and tuned specifically for AMD GPUs. Finally, for the CPU we use the Math Kernel Library (MKL) [93] implementation of BLAS written by Intel, which is known for its high performance.

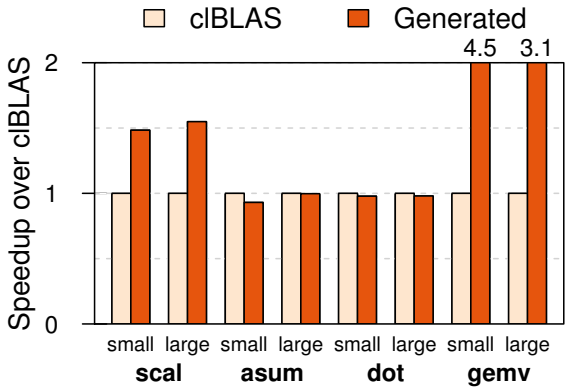
Figure 6.7a shows that the code generated by our approach matches the performance of CUBLAS for `scal`, `asum` and `dot` on the Nvidia GPU. For `gemv` we outperform CUBLAS on the small size by 20% while we are within 5% for the large input size. Given that CUBLAS is a proprietary library highly tuned for Nvidia GPUs, these results show that our technique is able to achieve high performance.

On the AMD GPU, we are surprisingly up to 4.5× faster than the clBLAS implementation on `gemv` small input size as shown in Figure 6.7b. The reason for this is the way how clBLAS is implemented: clBLAS generates the OpenCL code using fixed templates and in contrast to our approach, only one implementation is generated since they do not explore different pattern compositions.

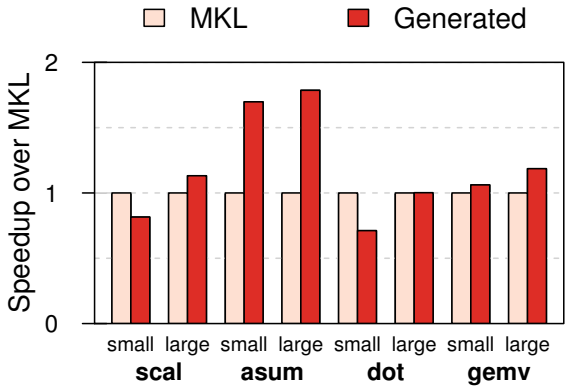
For the Intel CPU (Figure 6.7c), our approach beats MKL for one benchmark and matches the performance of MKL on most of the other three benchmarks. For the small input sizes on the `scal` and `dot` benchmarks we are within 13% and 30% respectively. For the larger input sizes, we are on par with MKL for both benchmarks. The `asum` implementation in the MKL does not use thread-level parallelism, while our implementation does and, thereby, achieves a speedup of up to 1.78 on the larger input size.



(a) Nvidia GPU



(b) AMD GPU



(c) Intel CPU

Figure 6.7: Performance comparison with state-of-the-art, platform-specific libraries: CUBLAS for Nvidia, clBLAS for AMD, MKL for Intel. Our approach matches the performance on all three platforms and outperforms clBLAS in some cases.

6.4 MOLECULAR DYNAMICS PHYSICS APPLICATION

The molecular dynamics (MD) application is a physics simulation taken from the SHOC [46] benchmark suite. It calculates the sum of all forces acting on a particle from its neighbors. Listing 6.2 shows the implementation using our high-level patterns.

The function updateF (line 1) updates the force f influencing particle p by computing and adding the force between particle p and one of its neighbors. Using the neighbor's index nId and the vector storing all particles ps, the neighboring particle is accessed (line 2) and the distance between the particle p and its neighbor n is computed (line 3). If the distance is below threshold t, the force between the two particles is calculated and added to the overall force f (line 4). If the distance is above the threshold, the force is not updated (line 5).

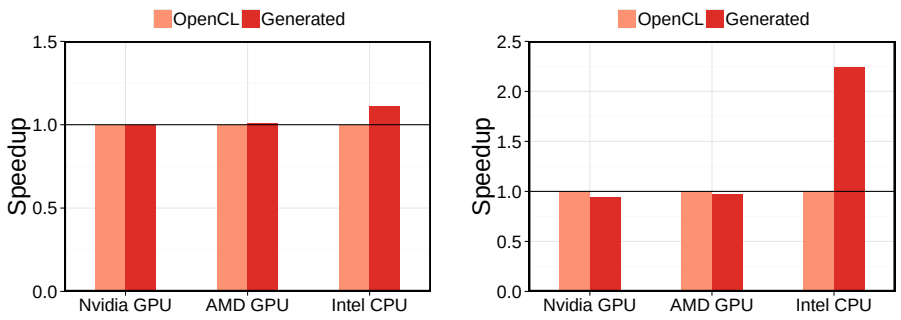
For computing the force for all particles ps, we use the *zip* pattern (line 9) to build a vector of pairs, where each pair combines a single particle with the indices of all of its neighboring particles (p and ns in line 7). The function which is applied to each pair by the *map* pattern in line 7 is expressed as a lambda expression. Computing the resulting force exerted by all the neighbors on one particle is done by applying the *reduce-seq* pattern on the vector ns storing the neighboring indices. We use function updateF inside the reduction to compute the force for each particle with index nId and add it to the overall force on p.

We use lambda expressions for binding of additional information as arguments to functions. This example shows that our patterns can be used to implement not only simple benchmark applications.

We performed measurements with an input size of 12288 particles for all three test platforms. The results are shown in Figure 6.8a. We can see that the automatically generated OpenCL code performs very close to the native OpenCL implementation and is even slightly faster on the Intel CPU.

```
1 updateF f nId p ps t =
2   let n = ps[nId]
3   let d = calculateDistance p n
4   if (d < t) f + (calculateForce d)
5   else f
6
7 md ps nbhs t = map (λ p,ns.
8     reduce-seq (λ f,nId. updateF f nId p ps t) 0 ns
9   ) (zip ps nbhs)
```

Listing 6.2: Molecular dynamics physics application expressed using our high-level algorithmic patterns.



(a) Molecular Dynamics (b) BlackScholes

Figure 6.8: Performance of our approach relative to portable OpenCL implementations of the MD and BlackScholes benchmarks.

6.5 MATHEMATICAL FINANCE APPLICATION

The *BlackScholes* application uses a Monte Carlo method for option pricing and computes for each stock price s a pair of call and put options. Listing 6.3 shows the BlackScholes implementation, where the function defined in line 1 computes the call and put option for a single stock price s. Two intermediate results d1 (line 2) and d2 (line 3) are computed and used to compute the two options, which are returned as a single pair (line 4). In line 6 the BSComputation function is applied to all stock prices (stored in the vector stockPrices) using the *map* pattern.

Figure 6.8b shows the result compared to an OpenCL implementation of the BlackScholes model provided by Nvidia as part of their software development kit [121]. We measured the runtime on all three test platforms for a problem size of 4 million stock prices. We see that our approach is on par with the performance of the Nvidia implementation on both GPUs. On the CPU, we actually achieve a 2.2× speedup due to the fact that the Nvidia implementation is tuned for GPUs while our implementation generates different code for the CPU.

```
1 BSComputation s =
2   let d1 = compD1 s
3   let d2 = compD2 d1 s
4   (compCallOption d1 d2 s, comPutOption d1 d2 s)
5
6 blackScholes stockPrices = map BSComputation stockPrices
```

Listing 6.3: BlackScholes mathematical finance application expressed using our high-level algorithmic patterns.

6.6 CONCLUSION

In this chapter we have evaluated the code generation approach introduced in Chapter 5. We have seen that our approach successfully addresses the performance portability challenge and generates code which achieves high performance on three distinct hardware platforms: a Nvidia GPU, an AMD GPU, and an Intel CPU. Furthermore, the comparison against the highly tuned BLAS libraries shows that our code generator can produce highly efficient code for high-level expressions by systematically transforming them into low-level expressions before compilation.

We have successfully addressed a drawback of SkelCL which we could not use for implementing an efficient implementation of the *asum* and the dot product benchmarks. The performance of the code generated by our novel code generation technique is comparable to the best implementations available, as our code generation approach fuses patterns and generates efficient kernels for these two benchmarks.

Finally, we presented a prototype tool which automatically applies the rewrite rules and was able to find implementations with high performance for the parallel reduction on all three tested platforms. The found expressions were significantly different from the implementations developed by Nvidia, but still achieved high performance exploiting 75% of the available hardware bandwidth on Nvidia's and AMD's GPUs as well as on Intel's CPU.

This concludes the two main technical parts of the thesis. In the next part we will summaries our work and contributions, discuss how the two presented projects relate to each other and how they can be combined and improved in the future. Finally, we will conclude the thesis with a comparison against related work.

Part IV

SUMMARY & CONCLUSION

7

TOWARDS A HOLISTIC SYSTEMATIC APPROACH FOR PROGRAMMING AND OPTIMIZING PROGRAMS

THE TWO PREVIOUS TECHNICAL PARTS have addresses the two main challenges we identified at the beginning: programmability and performance portability. In this chapter we will summarize SkelCL– the high-level programming model introduced in Part II which addresses the programmability challenge, and the novel pattern-based code generation technique introduced in Part III which addresses the performance portability challenge. We will especially refer back to the four main contributions of this thesis as stated in Chapter 1. Furthermore, we will describe how the two presented approaches relate to each other and how they can be combined in the future for creating a holistic systematic approach for programming and optimizing programs for many-core processors offering SkelCL's high-level abstractions and integration in C++ together with the portable and high performance provided by our code generator technique.

7.1 ADDRESSING THE PROGRAMMABILITY CHALLENGE

In Part II of this thesis, we introduced SkelCL which addresses the programmability challenge of modern parallel processors.

THE SKELCL PROGRAMMING MODEL In Chapter 3 we used a case study to show the drawbacks of programming with the state-of-the-art low-level programming approach OpenCL. The SkelCL programming model provides three high-level features which help to overcome these drawbacks, raise the level of abstraction for the programmer and, thus, simplify parallel programming:

- parallel container data types (explained in detail in Section 3.2.1) help to automate the low-level memory management as their data is transparently accessible to both CPU and GPUs;
- algorithmic skeletons (explained in detail in Section 3.2.2) are used for easily expressing parallel programs in a structured, high-level manner;
- data distribution and redistribution mechanisms (explained in detail in Section 3.2.3) greatly simplify programming of multi-

GPU systems by transparently performing all necessary data transfers.

The SkelCL programming model is implemented as a C++ library (explained in detail in Section 3.3), deeply integrated with features from the latest C++ standard.

In Chapter 4 we showed that the SkelCL programming model and its implementation as a C++ library are suitable for implementing real-world applications from a broad range of domains. For all investigated examples we showed that the programming is greatly simplified with shorter and easier to understand code. The SkelCL library offers competitive performance to manually written OpenCL code on single- and multi-GPU systems for all but two benchmarks. For these two benchmarks (*dot product* and *asum*) multiple OpenCL kernels are executed instead of a single fused one. The code generation technique presented in Part III overcomes this drawback of SkelCL.

The SkelCL programming model and its implementation is the first major contribution of this thesis.

ALGORITHMIC SKELETONS FOR STENCIL AND ALLPAIRS Alongside SkelCL we introduced two novel algorithmic skeletons. The *stencil* skeleton (explained in detail in Section 3.2.4.1) simplifies stencil computations common in domains like image processing. The *allpairs* skeleton (explained in detail in Section 3.2.4.3) allows programmers to easily express allpairs computations like matrix multiplication. We formally define both skeleton and provide efficient single- and multi-GPU implementations. For the allpairs skeleton we identified in Section 3.2.4.3 an optimization rule, which enables an optimized implementation especially beneficial on modern GPUs.

The evaluation for matrix multiplication (Section 4.4) shows the competitive performance of the provided implementation of the allpairs skeleton compared to highly tuned library code. We discussed performance results for the implementations of the stencil skeleton for image processing applications in Section 4.5 and a physics simulation in Section 4.7. These results show that similar performance is achieved as compared with manually tuned low-level OpenCL code. For both skeletons the evaluation shows that programming is greatly simplified and not only the boilerplate management code is avoided but GPU specific optimizations, like the usage of local memory, are performed hidden from the user.

The formal definitions and efficient GPU implementations of the stencil and allpairs skeletons is the second major contribution of this thesis.

7.2 ADDRESSING THE PERFORMANCE PORTABILITY CHALLENGE

In Part III of this thesis we introduced a novel code generation technique which addresses the performance portability challenge.

A FORMAL SYSTEM FOR REWRITING PATTERN-BASED PROGRAMS

We started Chapter 5 with an investigation into the portability of optimization using the low-level programming approach OpenCL and showed that optimizations in OpenCL are not performance portable. In the following we introduced a set of high-level and low-level patterns (explained in detail in Section 5.3) together with provably correct rewrite rules (explained in detail in Section 5.4). While the high-level patterns capture algorithmic concepts, very similar to the algorithmic skeletons used in SkelCL, the low-level patterns model specific features of the target low-level programming model OpenCL. The rewrite rules encode high-level algorithmic choices, as well as low-level optimizations which can be systematically applied to a pattern-based program. Especially, the rewrite rules explain how the high-level algorithmic concepts can be mapped to OpenCL, our target low-level programming model.

We show the soundness of the system by giving formal definitions of the semantics and types of each pattern and proving for each rewrite rule that it does not change the semantic of the rewritten expression. In Section 5.4.3 we showed how the rewrite rules can be systematically applied for deriving differently optimized, hardware-specific low-level expressions from a single high-level representation.

This formal foundation makes our rewrite approach suitable for automating the optimization of code in a compiler and is the third major contribution of this thesis.

A CODE GENERATOR OFFERING PERFORMANCE PORTABILITY

Based on the formal foundations, we developed and presented the design and implementation of a code generator (explained in detail in Section 5.5) which generates highly efficient, hardware-specific OpenCL code for different target platforms from a single pattern-based expression. The single high-level representation is transformed into a hardware-specific low-level representation using the rewrite rules, as shown in Section 5.4.3. The low-level representation is then compiled to efficient OpenCL code. Our implementation employs a powerful type system encoding information about the size of arrays used for static memory allocation. We use type inference for inferring most types automatically to free the programmer from specifying types explicitly.

Our performance evaluation in Chapter 6 shows that using this novel approach, OpenCL code is generated which performs compa-

rable to manually optimized library codes on three different parallel processors. This novel code generation approach offers true performance portability, since hardware-specific code is systematically generated from a single, portable high-level representation.

This code generator offering true performance portability is the fourth, and final, major contribution of this thesis.

7.3 FUTURE WORK: TOWARDS A HOLISTIC SYSTEMATIC APPROACH FOR PROGRAMMING AND OPTIMIZING PROGRAMS

The two separate approaches described in this thesis can naturally be combined in the future to obtain a single holistic approach which offers the advantages of both: the high-level abstractions from SkelCL which structure and simplify parallel programming as well as the highly optimized and portable performance delivered systematically by our novel code generation technique.

This combination makes sense as both approaches use structured parallel programming in the form of parallel patterns (or algorithmic skeletons as they are called in SkelCL) as their fundamental building block. In SkelCL the patterns are used by the programmer to describe the algorithmic structure of the program. In the code generator rewrite rules transform programs expressed with patterns into a low-level form from which efficient OpenCL code is generated.

As shown in Part II the SkelCL programming model provides a great programming interface successfully hiding complicated details of parallelism and the underlying hardware from the programmer. But the current implementation as a C++ library has some restrictions in certain areas, even somewhat limiting the expressiveness of the programming model. For example, the nesting of patterns is not supported in a library implementation. Furthermore, when evaluating SkelCL in Chapter 4 we identified a performance problem for two benchmarks because the current SkelCL library implementation does generate a separate OpenCL kernel for each pattern instead of generating a single fused kernel. The current library is optimized towards and tested on GPUs by Nvidia and does not necessary offer the same level of performance on other hardware platforms.

As shown in Part III, our code generator addresses these performance drawbacks of the SkelCL library implementation, systematically generating highly efficient code on three hardware platforms. However, currently the high-level and low-level patterns from Chapter 5 are not well integrated in a programming language like the SkelCL library is integrated in C++. This restricts the expressiveness and makes it difficult to implement complex real-world applications like the LM OSEM expressed in SkelCL as shown in Section 4.6.

A future holistic approach will avoid all of these drawbacks by using SkelCL as the *frontend* offering to the user its convenient programming interface integrated with the C++ programming language, combined with the code generator as the *backend* systematically compiling the pattern-based expressions into hardware-specific code.

In the following we will discuss possible future enhancements to SkelCL as well as the code generator.

7.3.1 *Enhancing the SkelCL Programming Model*

In this section we explore possible enhancements to the SkelCL programing model, as well as discuss current limitations of the C++ library implementation and how those could be liftet in the future. We start with the Stencil skeleton and how its implementation could be enhanced to improve its usability. We then discuss possible enhancements to lift the limitations regarding composing and nesting of SkelCL's skeletons and container data types. Next we discuss the possibility to extend SkelCL by supporting more algorithmic skeletons, especially, task-parallel skeletons for enhancing its expressiveness. Finally, we discuss how SkelCL could be extended for supporting truly heterogeneous execution where different types of parallel processors, e. g., CPU and GPU, are used efficiently together.

ENHANCING THE STENCIL SKELETON In SkelCL we introduced a new algorithmic skeleton for Stencil computations (Section 3.2.4.1). In Section 3.3.4.5 we discussed two implementations – `MapOverlap` and `Stencil`. Each implementation has its advantages and disadvantages regarding usability, expressiveness, and performance.

To improve the usability, the shape of the stencil could be inferred automatically from the customizing function instead of requiring the user to provide this information explicitly as it is currently the case.

To improve the expressiveness, more options for performing boundary handling could be added allowing more application to be easily expressed with the skeleton. Furthermore, many simulations could be expressed with the stencil skeleton if application-specific functions would be allowed for checking a terminating condition when performing stencil computations iteratively.

To improve the performance, the decision which implementation, `MapOverlap` or `Stencil`, to use in which situation could be taken automatically by the SkelCL implementation. A performance model predicting the runtime of each implementation in a certain situation could be built reflecting the performance characteristics of both implementations. Based on this model the runtime system could select the appropriate implementation for a given use case of the stencil skeleton.

OPTIMIZING DATA TRANSFER The memory management is completely hidden from the user in SkelCL. Data stored in SkelCL's container data types is made automatically accessible on CPU and GPUs. Currently SkelCL performs a single data transfer in OpenCL to copy a container's data to or from a GPU. When the data transfer time is large compared to the computational time it is often beneficial to upload the data in several chunks and start the computation on the first chunk while simultaneously uploading the next chunk. Using this strategy the data transfer can be overlapped with the computational time and the overall runtime is reduced. This optimization is not always valid as it must be possible to compute several chunks independent of each other. Fortunately, for the *map* and *zip* skeleton this is always true, therefore, this optimization could be added to SkelCL while preserving its completely transparent memory management.

ALLOW NESTING OF SKELETONS AND CONTAINER DATA TYPES Due to SkelCL's current implementation, the nesting of skeletons is not allowed in the library, therefore, it is not possible to express nested parallelism. Similarly, it is not possible to nest container data types in each other. SkelCL provides a special two-dimensional data type, but there is no generic mechanism for building higher-dimensional data types.

Both drawbacks are limitations given by the current library implementation. A skeleton is directly translated into an OpenCL kernel, preventing the nesting of skeletons. The container implementation assumes a flat representation of the data in memory, which is not necessarily the case when nesting containers.

By empowering SkelCL's implementation with our code generation technique we can overcome both drawbacks in the future. Nesting of computations expressed as skeletons as well as data in the form of arrays is fully supported by our current code generator. By integrating this implementation in SkelCL, nested parallelism can be exploited where OpenCL kernels and matching appropriate data representations are generated automatically.

ADDING SUPPORT FOR TASK-PARALLEL SKELETONS SkelCL currently focuses on data-parallel skeletons as they match the performance characteristics of modern GPU systems. Nevertheless, it could be useful to explore the introduction of task-parallel skeletons to SkelCL. The well-known pipeline skeleton could, for example, be useful for systematically optimizing the data transfer to and from GPUs by overlapping computation and communication. The pipeline skeleton could also be used in a multi-GPU setting where each GPU performs a different stage of the pipeline. Similarly, a task farm skeleton could be used to schedule different, possibly data-parallel, computations across multiple GPUs.

ADDING SUPPORT FOR TRULY HETEROGENEOUS EXECUTION
SkelCL uses OpenCL for its implementation, therefore, it is possible to use SkelCL not only for programming GPUs but also other types of accelerators and multi-core CPUs. As we showed in Part III, the performance of OpenCL is not portable across this broad range of parallel processors. Therefore, the current implementation of SkelCL would presumably not perform particular well on other types of parallel processors supported by OpenCL. Furthermore, in a system comprising multiple OpenCL devices SkelCL currently assumes that all device roughly process the data in an equal amount of time when distributing the data across the devices. This means that SkelCL's block distribution divides the input data into equally sized chunks, each processed by a different OpenCL device. This assumption works well for homogeneous multi-GPU systems as used in this thesis, but breaks down for truly heterogeneous execution where multiple parallel processors of different types are used together.

The code generation technique presented in Part III will address the first drawback and generate efficient and performance portable code for each single OpenCL device. More research has to be conducted for performing efficient execution on heterogeneous systems. The structured manner in which programs are expressed with SkelCL could help to address this issue, as researchers have already build performance models for structured programming [8, 19, 47, 84, 142] predicting runtime performance. Such models could be used for minimizing the overall runtime by most efficiently using all available hardware resources.

7.3.2 *Enhancing the Pattern-Based Code Generator*

This section discusses possible enhancements to our novel pattern-based code generator. Firstly, we discuss how the proposed rewrite rules can be efficiently applied automatically and how this will enable compilers to perform the advanced optimizations discussed throughout the thesis autonomously without any user interaction. We will then describe possible enhancements to the current OpenCL backend, before we explore the possibility to add additional backends producing efficient code in other low-level programming system, e. g., OpenMP or MPI. Finally, we will discuss how the current system can easily be extended with additional high-level patterns and rewrite rules.

AUTOMATICALLY APPLYING OF THE REWRITE RULES The formalism and implementation presented in this thesis constitute the foundational work necessary to systematically rewrite pattern-based expressions with the aim to apply optimizations beneficial for a given hardware platform. By applying the rewrite rules a design space of

possible implementations is built for a program represented with a corresponding high-level pattern-based expression. The formalism introduced in Chapter 5 ensures that the rewrite rules can safely be applied, as they do not change the programs semantics. This makes the rewrite rules suitable for automation inside a compiler.

In Section 6.2.1 we discussed a preliminary search tool used for determining which implementations in the large (and in fact unbound) design space offer good performance for the simple reduction benchmark. This tool builds and searches the tree of possible implementations by applying all possible rules at one level of the tree and then sampling their subtrees randomly applying rules until an expression is found for which OpenCL code can be executed. These expressions are then executed and their performance is measured. The performance results are used to decide which subtree should be further investigated by starting the same process again. We showed that even this rather simplistic and unguided technique was able to find highly efficient implementations on three different hardware architectures for the simple benchmark we investigated.

In future work a more sophisticated search technique should be developed to search the implementation space more efficiently and possibly even to give guarantees on the quality of the found expression. Two possible techniques can be used here which have already been applied in similar contexts: formal performance models and advanced machine learning techniques.

Formal performance models try to model the performance of programs without executing the program. Such models are usually built by experts encoding specific semantic knowledge and have shown promising results especially for structured programming approaches, like algorithmic skeletons [8, 47, 84], where precise observations can be made about the generic program structure instead of being limited to observations about concrete applications which cannot be easily generalized. In our setting, performance models could be created to estimate the performance of a given pattern-based expression and then used to guide the automatic application of rewrite rules, by estimating the benefit of applying the rule. If multiple rewrite rules are valid, the particular rule would be chosen for which the performance model predicts the larges performance benefit.

Machine learning techniques [20] aim to automatically derive models by learning from experience gathered during either a dedicated learning phase (offline learning) or by iteratively improving an existing model (online learning). Such a model could then be used to guide the rewrite process, similar to a manually constructed formal performance model. Machine learning is a broad field and many related techniques have been proposed to enable different forms of the generic idea. Furthermore, machine learning has been already successfully applied in the context of optimizing compilers [58] and au-

totuning [39]. In our context, a device-specific model could be trained using experimental evaluation on a set of training applications using a similar technique as our preliminary one presented in Section 6.2.1. Such a model would have to be built only once for each particular parallel processor and could afterwards be used to guide the rewrite processes for arbitrary applications being executed on the same processor. In our structured parallel programming approach, different applications are still expressed with the same foundational building blocks, such that performance characteristics learned from one application should be applicable for other applications.

This topic is surely one of the most interesting, but also most challenging areas of future work.

ENHANCE THE CURRENT OPENCL BACKEND As seen in Chapter 6, the current OpenCL backend implementation already offers portable high performance. Nevertheless, the backend could be further improved in the future. We discuss two enhancements to give an idea on possible future work in this area.

Adding support for more advanced vectorization techniques could be one future enhancement, as currently only simple arithmetic functions without any control flow can be automatically vectorized by our implementation. Dedicated tools vectorizing entire functions [98] could be combined and integrated into our backend to overcome this limitation.

The current memory allocation is rather basic and not optimized for space usage. Therefore, currently more intermediate buffers are allocated then necessary. We intend to use well-known compiler techniques, like graph coloring [118] usually used in register allocation, for improving the current allocation strategy. We also plan to add support for the private and constant memory regions defined in OpenCL, as we currently only support the global and local memory.

ADDING ADDITIONAL BACKENDS In this thesis, we based our formalism and implementation on the OpenCL standard. The same idea of high-level and low-level patterns bridged with rewrite rules could be applied to other low-level programming models as well. For example, it is possible following the same methodology to design a backend targeting MPI for distributed systems. While the high-level algorithmic patterns and the algorithmic rewrite rules are independent of the target programming model, the low-level hardware patterns and backend-specific rules are not. This design was chosen deliberately to achieve a clear separation of concern between the high-level programming interface as well as algorithmic optimizations and the low-level hardware-specific programming approach with device-specific optimizations.

An additional backend would, therefore, add low-level patterns describing the low-level target programming model in a structured and functional style using patterns. Furthermore, a set of rewrite rules has to be defined for lowering the high-level algorithmic concepts to appropriate low-level patterns and for expressing optimization choices in the low-level target programming approach. For MPI, patterns reflecting the distributed nature of the programming model would be added, e. g., a *mapNode* pattern for assigning work to a node in the distributed system. Similarly, novel rewrite rules could explain when it is possible to use collective operations, like *broadcast*, to optimize the data transfer.

One could even imagine, to combine multiple dedicated backends, e. g., the MPI-backend and the OpenCL-backend, to leverage the existing patterns and rules in a different setting, like a large scale heterogeneous cluster environment found in modern supercomputers.

ADDING ADDITIONAL HIGH-LEVEL PATTERNS AND ALGORITHMIC REWRITE RULES By design we started our code generation technique with a limited set of parallel patterns which cannot express all applications. By restricting ourself to this well understood set of pattern we are able to systematically generate portable and highly efficient code.

The current set of high-level algorithmic patterns and rewrite rules is powerful enough to express the benchmarks discussed in Chapter 6. Furthermore, the supported patterns allow already to express other higher level abstractions, like for example, the *allpairs* skeleton from SkelCL, which can be expressed by nesting two *map* pattern in each other. Nevertheless, by adding more high-level patterns and rules in the future, e. g., to support stencil applications, we will increase the expressiveness of our approach and make it more attractive to potential users.

8 COMPARISON WITH RELATED WORK

In this final chapter we compare the approaches presented in this thesis with related work.

8.1 RELATED WORK

Here we will discuss related projects which also aim to simplify parallel programming in general or programming of GPU systems in particular. We also include projects aiming at performance portability, as our approach does. We will start by looking at algorithmic skeleton libraries in general and then focus on more recent projects targeting GPU systems, like SkelCL does. Next, we will cover other structured parallel programming approaches, including the well-known *MapReduce* framework. We will then discuss the broad range of GPU programming approaches proposed in recent years, before looking at domain-specific approaches, including projects with particular focus on stencil computations. We will end with a discussion of related projects using rewrite rules for program optimizations.

For all projects we make clear how they relate to our work.

8.1.1 *Algorithmic Skeleton Libraries*

Numerous algorithmic skeleton libraries have been proposed since the introduction of algorithmic skeletons in the late 1980s [36]. A good and extensive overview reflecting the state of the art at the time when the work on this thesis was started in 2010 can be found in [72]. We will discuss here some representative examples of algorithmic skeleton libraries targeting different types of computer architectures.

Prominent algorithmic skeleton libraries targeting distributed systems are *Muesli* [103] and *eSkel* [37] which are both implemented using MPI [117]. There has also been work especially dedicated towards grids [8, 9] leading to the development of the *Higher Order Components (HOC)* [60, 61] which are implemented in Java.

Skeleton libraries for multi-core CPUs include *Skandium* [108] which uses Java threads, *FastFlow* [4, 5] which is implemented in C++ and has recently be extended towards distributed systems as well [3], and an extended version of *Muesli* [35] which uses OpenMP [127].

Of particular relevance for our comparison are the following recent skeleton libraries targeting GPU systems.

Muesli [65] and *FastFlow* [7, 25] have been extended for GPU systems using CUDA. Both libraries implemented support for execution of their data-parallel skeletons on GPU hardware, but not for their task-parallel skeletons. In Muesli data-parallel skeletons can be nested in task-parallel skeletons, but not the other way around. This type of nesting is also supported when the data-parallel skeleton is executed on a GPU. The data management between CPU and GPU is performed implicitly and automatically as it is the case for SkelCL, but different to our implementation data is transfered back to the CPU after each skeleton execution on the GPU. This makes the integration with the existing infrastructure in Muesli and FastFlow easier but obviously limits performance when multiple skeletons are executed on the GPU.

SkePU [48, 49, 64] is a skeleton library implemented in C++ and specifically targeted towards GPU systems, similar to SkelCL. Both approaches have been developed independently but implement very similar concepts and even a similar set of data-parallel algorithmic skeletons. Nevertheless, both projects have emphasis on different areas and are implemented in different ways. SkePU implements multiple backends for targeting different hardware devices. Currently, there exist an OpenMP backend for multi-core CPUs, OpenCL and CUDA backends for GPUs, and separate backends written in OpenCL and CUDA for multi-GPU execution.

The approach of developing multiple backends is contradictory to the idea of code and performance portability advocated in this thesis. SkelCL uses only a single OpenCL backend which could be combined in the future with our novel compiler technique to optimize code for different platforms.

Recently SkePU has implemented a similar scheme as SkelCL for managing data transfers [49], by using a lazy copying strategy which SkelCL does since its first implementation. SkePU now supports an automatic overlap of data transfer with computations, which is currently not supported in SkelCL.

A version of SkePU integrated with the StarPU runtime system [16] allows for hybrid CPU and GPU execution with a dynamic load balancing system provided by StarPU. Furthermore, SkePU allows to specify *execution plans* which determine the backend to be used for a particular data size of the problem, e. g., the OpenMP backend for small data size and the CUDA multi-GPU backend for larger data sizes. While SkelCL also fully support the execution on multi-core CPUs, single GPUs, and multi-GPU systems, we have currently no comparable mechanism to determine which hardware should be used for different data sizes.

SkelCL introduces data distributions to give users control over the execution in multi-GPU systems. SkePU does not offer such a feature and always splits the data across GPUs, therefore, complicated

multi-GPU applications like the LM OSEM presented and evaluated in Chapter 4 are not easy to implement in SkePU.

JPAI [69] is a recent skeleton library for seamless programming of GPU systems using Java. JPAI offers an object-oriented API which makes use of the new Java lambda expressions. At runtime before execution on the GPU the customizing functions of the skeletons are compiled to OpenCL using the Graal [59] compiler and virtual machine. There is currently no support for multi-GPU systems, as there is in SkelCL.

8.1.2 *Other Structured Parallel Programming Approaches*

There are other projects advocating structured parallel programming, even though they identify themselves not necessary as algorithmic skeleton libraries.

Delite [24, 29, 107] is a framework for building domain-specific languages which automatically exploit parallelism. To achieve this, Delite offers a set of basic parallel operators, very similar to algorithmic skeletons, which can be used as fundamental building blocks by the designer of a domain-specific language. The domain-specific language is compiled by Delite where C++ code is generated for multi-core CPUs and CUDA code for GPUs. For performing the compilation Delite uses *Lightweight Modular Staging (LMS)* [135] – a runtime code generation approach implemented as a library in the Scala programming language [122, 123]. LMS exploits the rich type system of Scala to give fine grained control over which expressions should be evaluated in Scala at compile time and for which expressions code should be generated. Neither Delite nor LMS address the performance portability issue we identified in this thesis and address with our novel compilation technique.

MapReduce [51] is a programming model advocated to simplify the programming of applications processing large amounts of data. Often these applications run inside of data centers, the cloud, or other possibly large scale distributed systems. Computations are divided into two steps called map and reduce. In the map step a user provided function is applied to each data item, usually represented as a key-value pair, in a possibly large collection of data. After the map step all matching values with the same key are grouped together and then in the reduce step all values for a given key can be aggregated by a second user provided function. These concepts are closely related to algorithmic skeletons, even though a slightly different terminology is used and only computations fitting this one computational pattern can be expressed. Lämmel discusses extensively MapReduce from a functional programming perspective and explores its foundations

in skeletal programming [104]. The generic algorithmic skeleton approach discussed in this thesis allows for implementing a broad range of applications, as shown in Chapter 4 and Chapter 6, and is not fixed to one particular application domain as MapReduce is.

Since its introduction MapReduce has found widespread use and several projects providing implementations on different architectures have been presented. The most prominent example is *Hadoop* [13] – an open source implementation in Java targeting cluster systems. Other work has targeted single- and multi-GPU systems [66, 144].

Threading Building Blocks (TBB) [134] is a software library developed by Intel. TBB offers parallel patterns, like `parallel_for` or `parallel_reduce`, as well as concurrent data structures, including `concurrent_queue`, `concurrent_vector`, and `concurrent_hash_map`. TBB can be used for programming Intel's multi-core CPUs and has recently been enabled for the Xeon Phi accelerators as well. In a separate book [113] three authors from Intel discuss how TBB, among other technologies advocated by Intel, can be used for structured parallel programming. The *C++ Extensions for Parallelism* [26] is a set of proposed extensions to the C++ standard adding parallel algorithms, similar to algorithmic skeletons, to C++. TBB as well as existing skeleton libraries, including SkelCL, could implement the proposed specification in the future and, thus, conform to a unified and standardized programming interface.

SkelCL currently is not optimized for multi-core CPUs as TBB is. When combined with our novel compilation approach, SkelCL will aim for generating highly efficient code for multi-core CPUs in the future as well.

8.1.3 *Related GPU Programming Approaches*

We already discussed some closely related approaches which can be used for program GPU systems, including *SkePU*, *Muesli*, *FastFlow*, and *JPAI*. Here we are going to discuss additional approaches which are especially targeted towards GPU programming.

Thrust [17] and *Bolt* [22] are C++ libraries developed by Nvidia and AMD respectively for simplify the programming of their GPUs. Both libraries offer a similar set of parallel patterns and interfaces similar to TBB and SkelCL. Thrust is implemented using CUDA and Bolt uses OpenCL, as SkelCL does. For data management both libraries offer separate data structures for the CPU and GPU. The programmer has explicit control, but also the additional burden, to move data between CPU and GPU. This is different to SkelCL's implementation where a unified data structure is provided which automatically and lazily manages the data transfers in the system.

Currently, neither Thrust nor Bolt support multi-GPU execution. In SkelCL multi-GPU support is a key feature built into the programming model with the support of data distributions giving programmers control over how the GPUs in the system should be used.

Obsidian [146, 147] is a project for performing GPU programming using the functional programming language Haskell [91]. GPU code is generated from expressions written in a domain specific language embedded in Haskell. Obsidian offers low-level primitives to the programmer for achieving competitive performance with manually written CUDA code.

Accelerate [30, 114] is also a domain specific languages for data-parallel programming embedded in Haskell. Accelerate makes use of Obsidian for generating GPU code and offers a higher level interface to the programmer than Obsidian does. Parallel combinators, like map, fold, or scan, are used to express programs at the algorithmic level. This is very similar to how functional programmers usually program in ordinary, sequential Haskell. This is also similar to our skeleton library SkelCL, but the use of Haskell makes Accelerate hard to use for programmers coming from more traditional imperative and object-oriented languages like C and C++. Furthermore, SkelCL has specific features, like the support of additional arguments, which enhance the flexibility in which it is used, allowing to efficiently implement real-world examples, e. g., the LM OSEM application discussed in Chapter 4.

Accelerate uses high-level optimizations [114] for fusing multiple skeleton executions into a single one – an optimization technique known as *deforestation* [154] in the functional community. This is different from SkelCL which always generates an OpenCL kernel for each skeleton. Therefore, Accelerate is able to generate efficient code for benchmarks where SkelCL performs poorly, including *asum* and the dot product. As discussed in Chapter 7, we intend to address this drawback of SkelCL in the future by incorporating our novel compilation technique which is able to perform this type of optimization as well. Furthermore, Accelerate generates CUDA code specialized for Nvidia GPUs and its implementation is not portable across architectures, as our compilation technique is.

Hijma et al. presents a stepwise-refinement methodology for programming GPUs and Intel Xeon Phi accelerators [86]. Programs are written using a `foreach` statement which allows parallel execution. By combining the program with a separate hardware description a compiler generates code for GPUs and accelerators. Using traditional compiler analysis the compiler provides feedback to the user on which part to focus the optimization efforts. The proposed system allows the user to stay at an abstract portable level or to decide to optimize

for a particular hardware architecture and lose portability. The presented results report substantial performance benefits comparing a high-level unoptimized implementation versus a non-portable implementation optimized for a particular hardware.

Using this approach the programmer is still performing the difficult optimizations manually, requiring detailed hardware knowledge. This is different using our compilation technique, where the compiler is empowered to perform advanced optimizations without user interaction. Furthermore, our approach does not force the user to choose between performance and portability, which is still required in this work.

Many projects following the tradition of *OpenMP* [127] working with annotations of sequential code have been proposed for GPU programming as well. Directives are used by the programmer to annotate sequential code, usually loops, which can be executed in parallel. A compiler supporting them reads the directives and generates parallel code automatically.

Early projects generating GPU code include *HMPP* [54], the *PGI Accelerator Compilers* (PGI has since been acquired by Nvidia), and *hiCUDA* [81]. All these projects contributed into a new unified standard called *OpenACC* [149].

OmpSs [62, 63] is a project using directives with a particular focus on task parallelism. Sequential code can be declared as a task via annotations and dependencies between tasks are specified by the programmer as well. A runtime system can exploit parallelism by executing independent tasks simultaneously. Data-parallel computations can also be executed on GPUs using OpenCL [63]. OmpSs influenced the development of the latest standard of *OpenMP* 4.0 [127] which now also includes directives for offloading computations to GPUs.

In OpenACC and OpenMP 4.0 directives for specifying the parallel computation in loops are provided as well as directives for explicitly specifying the data regions involved in the computation. After each computation data is copied back to the CPU and many GPU features can currently not be exploited with this approach, e. g., the usage of local memory. SkelCL as well as our code generation technique fully exploit all features provided by GPUs which are crucial for achieving high performance. All these approaches using directives promise minimal source code change for existing sequential code. SkelCL requires programmers to express their programs in a well structured way using algorithmic skeletons, which might force programmers to reimplement parts of their programs. We consider this actually a benefit of SkelCL, as restructuring the code using skeletons will likely increase its maintainability.

Many new programming languages have been proposed for GPU programming as well. Some existing programming languages have been extended to support GPU programming directly from the language without the need of specific libraries, including IBM's X10 language [148].

HiDP [161] is a language for expressing hierarchical data-parallel programs. Fundamental data-parallel building blocks like map are pre-defined in the language and used by programmers. Patterns can be nested, where this is not possible in SkelCL but fully supported in our novel compilation technique.

LiquidMetal [89] is a research project by IBM to support the programming of heterogeneous systems comprised of different hardware in a single programming language called *Lime* [15]. Support for programming GPU systems has recently been added [57]. Lime supports task-parallelism with the creation of tasks which can communicate with each other. Data-parallelism is also supported with skeleton-like patterns, including map and reduce.

Single Assignment C (SAC) [78] is a functional programming language supporting GPU programming by compiling SAC programs to CUDA [79]. The compiler automatically detects loops which can be offloaded to the GPU, generates the necessary CUDA code and performs the data management automatically. SAC has a special loop (`with`-loop) which is guaranteed to be free of dependencies, similar to SkelCL's map skeleton.

Copperhead [28] is a GPU language embedded in the Python programming language offering data-parallel primitives, including map, reduce, and scan. Nesting of patterns is supported and nested parallelism is statically mapped to the GPU thread hierarchy by the compiler and can be controlled via compiler switches. Currently, no systematic mechanism exists for exploring different mapping strategies. Our code generation approach allows to systematically explore different implementation strategies, including different mappings of nested data parallelism to the hardware.

NOVA [40] is a recent functional GPU programming language developed at Nvidia. It is intended to be used as an intermediate language produced by a domain specific programming interface. Programs are implemented in a Lisp-style notation where function application is written using prefix notation. Built-in parallel operations include map, reduce, and scan. Different to our approach fixed implementations of these patterns are used which are manually optimized for Nvidias GPUs.

Finally, *Petabricks* [12] allows the programmer to specify a range of valid implementations for a given problem. The programmer thereby specifies an implementation space the Petabricks compiler and runtime explores using autotuning strategies. The two main concepts are *transforms* which are functions which can be transformed by the Petrabricks compiler and *rules* which describe a possible way for performing the computation. By specifying multiple alternative rules the compiler is free to choose which one to apply at a given point in the program. Recent work has enabled the generation of GPU code as well [128].

Different to our code generation approach for tackling performance portability, Petabricks relies on static analysis for optimizing the generated code. Furthermore, the responsibility for specifying algorithmic choices is on the application programmer implementing a specific algorithm, whereas in our approach the algorithmic choices are defined and captured in the rewrite rules and then applied to the high-level code written by the programmer. Therefore, in our approach the algorithmic choices can be seen as implicit, or hidden from the user, while in Petabricks they are explicit to the programmer.

All these programming languages take a very different approach than SkelCL does by requiring programmers to learn a new language and reimplement their entire application in the new language. Interfacing code written in one of these new languages with legacy code is often complicated, whereas it is straightforward and easy when using SkelCL, because it is implemented as an ordinary C++ library.

8.1.4 *Related Domain Specific Approaches for Stencil Computations*

In this thesis we introduced two novel algorithmic skeletons, among them the *stencil* skeleton. Here we will discuss related work on domain specific approaches focused on stencil computations and the related domain of image processing. All these approaches are, by design, focused on one particular domain of applications, whereas SkelCL and our compilation technique are much more general.

PATUS [33] is a code generation and autotuning framework for stencil computations. The stencil computation is expressed in a domain specific language and separated from the execution strategy which is also written by the programmer in a second domain specific language but independent of the stencil executed. The PATUS compiler combines a stencil and a strategy to generate OpenCL code. PATUS focuses on single-GPU performance and does not support multi-GPU systems as SkelCL does. An autotuning component is used to experimentally determine and optimize implementation parameters and to choose the best execution strategy.

Different execution strategies can be used on different architectures to achieve some degree of performance portability, as the application-specific stencil code does not change. Our novel code generation approach goes further, where different implementation strategies are systematically derived from a single high-level expression and not picked from a limited set of manually written execution strategies.

PARTANS [110] is a software framework for stencil computations implemented in C++ with a focus on autotuning of stencil applications for multi-GPU systems. Stencils are expressed using a C++API in a style similar to our SkelCL C++ implementation of the stencil skeleton formally defined in this thesis. Autotuning is applied to determine the best parameters in the implementation. PARTANS pays particular attention to the communication in multi-GPU systems and detects different PCIe layouts which affect the data transfer rates between different GPUs in the system.

Overall PARTANS is much more focused on optimizing the communication in a multi-GPU system than SkelCL is and does not consider particular optimizations on single GPUs nor does it formally define skeleton computations as an algorithmic skeleton as this thesis does.

Impala [102] is a domain specific language developed at Saarland University for supporting stencil applications. Similar to the LMS project discussed earlier, code is generated at runtime and it can be precisely specified which part of the computation is executed at compile time and which part is executed in parallel by the GPU. This technique allows, for example, to generate GPU code where runtime checks for the boarder handling can be limited to the particular memory areas where boarder handling is actually required and avoided elsewhere.

Halide [131, 132] is a programming language for image processing applications. Halide is a purely functional language where a function producing an image is implemented by defining the computation performed to produce a single pixel. This function can be applied in different ways to produces the entire image, e. g., sequentially or in parallel. As applications are organized in a pipeline processing images in multiple stages, there is also the possibility to exploit pipeline parallelism. The decision how the functions are executed in Halide are defined in a *schedule* independent of the individual image processing operations performed.

Autotuning is used for automatically searching for good schedules [132]. Different from PATUS, Halide can automatically generate schedules, but they are restricted to image processing applications, e. g., assuming that pipelining is always possible. Our code generation approach is more general and can be applied to other types of application domains as well.

8.1.5 *Related Approaches using Rewrite Rules*

There has been related work on using rewrite rules for program optimizations, especially from a functional programming background.

The Bird-Meertens formalism [18] used as our notation in this thesis already defines equations which allow to transform programs at an algorithmic level. [73] shows a detailed case study of this technique. Targeting distributed systems, formal rules have been proposed for optimizing collective operations in message passing systems [74, 75, 77]. For the functional programming language Haskell, [97] discusses how rewriting can be used as an optimization technique and how the Glasgow Haskell Compiler (GHC) [90] supports the definition of rewrite rules.

Our rewrite rules can be seen in the same tradition as this work.

Spiral [124, 130, 140] is a project aiming at generating highly efficient code for signal processing applications and related domains. Rules are used to systematically describe domain specific transformations which are applied automatically by the compiler. Spiral specializes the generated code depending on the input problem size and achieves high performance using this technique. First focused on signal processing applications [130], Spiral has since been extended to generate code for linear algebra applications as well [140].

In contrast, our code generation technique systematically describes hardware optimizations of the OpenCL programming model and our approach is not domain specific as Spiral is, where the framework has to be re engineered for supporting a new application domain.

APPENDIX

A CORRECTNESS OF REWRITE RULES

This appendix present the proofs which show that the rewrite rules presented in Section 5.4 do not change the semantics of pattern-based programs.

A.1 ALGORITHMIC RULES

This section shows the proofs for the algorithmic rules defined in Section 5.4.1.

IDENTITY We repeat Equation (5.1) here:

$$f \quad \to \quad f \circ \mathit{map}\ \mathit{id} \quad | \quad \mathit{map}\ \mathit{id} \circ f$$

Proof of option 1. Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned}(f \circ \mathit{map}\ \mathit{id})\ xs &= f\ (\mathit{map}\ \mathit{id}\ xs) \\ &\quad \{\text{definition of } \mathit{map}\} \qquad\qquad \{\text{definition of } \mathit{id}\} \\ &= f\ ([\mathit{id}\ x_1, \mathit{id}\ x_2, \ldots, \mathit{id}\ x_n]) \quad = f\ xs\end{aligned}$$

□

Proof of option 2. Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned}(\mathit{map}\ \mathit{id} \circ\ f)\ xs &= \mathit{map}\ \mathit{id}\ (f\ xs) \\ &\quad \{\text{definition of } \mathit{map}\} \\ &= [\mathit{id}\ (f\ xs)_1, \mathit{id}\ (f\ xs)_2, \ldots, \mathit{id}\ (f\ xs)_n] \\ &\quad \{\text{definition of } \mathit{id}\} \\ &= f\ xs\end{aligned}$$

□

ITERATE DECOMPOSITION We repeat Equation (5.2) here:

$$\begin{aligned}\mathit{iterate}\ 1\ f &\to f \\ \mathit{iterate}\ (m+n)\ f &\to \mathit{iterate}\ m\ f \circ \mathit{iterate}\ n\ f\end{aligned}$$

Proof of option 1.

$$\begin{aligned}&\qquad\qquad \{\text{definition of } \mathit{iterate}\} \qquad \{\text{definition of } \mathit{iterate}\} \\ &\mathit{iterate}\ 1\ f\ xs = \mathit{iterate}\ (1-1)\ f\ (f\ xs) \quad = f\ xs\end{aligned}$$

□

Proof of option 2. Proof by induction. We start with the base case, let $n = 0$:

$$\begin{aligned}
&\{\text{definition of } \mathit{iterate}\}\\
\mathit{iterate}\ (m+0)\ f\ xs &= \mathit{iterate}\ m\ f\ (\mathit{iterate}\ 0\ f\ xs)\\
&\{\text{definition of } \mathit{iterate}\}\\
&= (\mathit{iterate}\ m\ f \circ \mathit{iterate}\ 0\ f)\ xs
\end{aligned}$$

We finish with the induction step $n-1 \to n$:

$$\begin{aligned}
&\{\text{definition of } \mathit{iterate}\}\\
\mathit{iterate}\ (m+n)\ f\ xs &= \mathit{iterate}\ (m+n-1)\ f\ (f\ xs)\\
&\{\text{induction hypothesis}\}\\
&= (\mathit{iterate}\ m\ f\ \circ \mathit{iterate}\ (n-1)\ f)(f\ xs)\\
&\{\text{definition of } \circ\}\\
&= \mathit{iterate}\ m\ f\ (\mathit{iterate}\ (n-1)\ f\ (f\ xs))\\
&\{\text{definition of } \mathit{iterate}\}\\
&= \mathit{iterate}\ m\ f\ (\mathit{iterate}\ n\ f\ xs)\\
&\{\text{definition of } \circ\}\\
&= (\mathit{iterate}\ m\ f \circ \mathit{iterate}\ n\ f)\ xs
\end{aligned}$$

□

REORDER COMMUTATIVITY We repeat Equation (5.3) here:

$$\begin{aligned}
\mathit{map}\ f \circ \mathit{reorder} &\to \mathit{reorder} \circ \mathit{map}\ f\\
\mathit{reorder} \circ \mathit{map}\ f &\to \mathit{map}\ f \circ \mathit{reorder}
\end{aligned}$$

Proof. We start with the expression $\mathit{map}\ f \circ \mathit{reorder}$.
Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned}
(\mathit{map}\ f \circ \mathit{reorder})\ xs &= \mathit{map}\ f\ (\mathit{reorder}\ xs)\\
&\{\text{definition of } \mathit{reorder}\}\\
&= \mathit{map}\ f\ [x_{\sigma(1)}, \ldots, x_{\sigma(n)}]\\
&\{\text{definition of } \mathit{map}\}\\
&= [f\ x_{\sigma(1)}, \ldots, f\ x_{\sigma(n)}]
\end{aligned}$$

Now we investigate the expression $reorder \circ map\ f$:

$$
\begin{aligned}
(reorder \circ map\ f)\ xs &= reorder\ (map\ f\ xs) \\
&\quad \{\text{definition of } map\} \\
&= reorder\ [f\ x_1, \ldots, f\ x_n] \\
&= reorder\ [y_1, \ldots, y_n] \qquad \text{with } y_i = f\ x_i \\
&\quad \{\text{definition of } reorder\} \\
&= [y_{\sigma(1)}, \ldots, y_{\sigma(n)}] \\
&\quad \{\text{definition of } y_i\} \\
&= [f\ x_{\sigma(1)}, \ldots, f\ x_{\sigma(n)}]
\end{aligned}
$$

As both expression we started with can be simplified to the same expression they have the same semantics, therefore, both options of the rule are correct. □

SPLIT-JOIN We repeat Equation (5.4) here:

$$map\ f \quad \to \quad join \circ map\ (map\ f) \circ split\ n$$

Proof. We start from the right-hand side and show the equality of both sides. Let $xs = [x_1, \ldots, x_m]$.

$$
\begin{aligned}
&(join \circ map\ (map\ f) \circ split\ n)\ xs = join\ (map\ (map\ f)\ (split\ n\ xs)) \\
&\quad \{\text{definition of } split\} \\
&\quad = join\ (map\ (map\ f)\ [[x_1, \ldots, x_n], \ldots, [x_{m-n+1}, \ldots, x_m]]) \\
&\quad \{\text{definition of } map\} \\
&\quad = join\ [map\ f\ [x_1, \ldots, x_n], \ldots, map\ f\ [x_{m-n+1}, \ldots, x_m]] \\
&\quad \{\text{definition of } map\} \\
&\quad = join\ [[f\ x_1, \ldots, f\ x_n], \ldots, [f\ x_{m-n+1}, \ldots, f\ x_m]] \\
&\quad \{\text{definition of } join\} \\
&\quad = [f\ x_1, \ldots, \ldots, f\ x_m] \\
&\quad \{\text{definition of } map\} \\
&\quad = map\ f\ xs
\end{aligned}
$$

□

REDUCTION We repeat Equation (5.5) here:

$$reduce\ (\oplus)\ id_\oplus \quad \to \quad reduce\ (\oplus)\ id_\oplus \circ part\text{-}red\ (\oplus)\ id_\oplus$$

Proof. We start from the right-hand side and show the equality of both sides. Let $xs = [x_1, \ldots, x_m]$.

$$
\begin{aligned}
&(\mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus \circ \mathit{part\text{-}red}\ (\oplus)\ \mathrm{id}_\oplus\ n)\ xs \\
&\quad = \mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus\ (\mathit{part\text{-}red}\ (\oplus)\ \mathrm{id}_\oplus\ n\ xs) \\
&\quad \{\text{definition of } \mathit{part\text{-}red}\} \\
&\quad = \mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus \\
&\qquad [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n)},\ \ldots,\ x_{\sigma(m-n+1)} \oplus \cdots \oplus x_{\sigma(m)}] \\
&\quad \{\text{definition of } \mathit{reduce}\} \\
&\quad = [(x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n)}) \oplus \cdots \oplus (x_{\sigma(m-n+1)} \oplus \cdots \oplus x_{\sigma(m)})] \\
&\quad \{\text{commutativity \& accociativity of } \oplus\} \\
&\quad = [x_1 \oplus \cdots \oplus x_m] \\
&\quad \{\text{definition of } \mathit{reduce}\} \\
&\quad = \mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus\ xs
\end{aligned}
$$

□

PARTIAL REDUCTION We repeat Equation (5.6) here:

$$
\begin{aligned}
&\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n \\
&\qquad\qquad \to \quad \mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus \\
&\qquad\qquad \mid \quad \mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n \circ \mathit{reorder} \\
&\qquad\qquad \mid \quad \mathit{join}_m \circ \mathit{map}\ (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n) \circ \mathit{split}\ m \\
&\qquad\qquad \mid \quad \mathit{iterate}\ \log_m(n)\ (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ m)
\end{aligned}
$$

Proof of option 1. Let $xs = [x_1, \ldots, x_m]$. Since the rules can only be valid if their types matches it must hold $n = m$.

$$
\begin{aligned}
&(\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ m)\ [x_1, \ldots, x_m] \\
&\quad \{\text{definition of } \mathit{part\text{-}red}\} \\
&\quad = [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(m)}] \\
&\quad \{\text{commutativity of } \oplus\} \quad \{\text{definition of } \mathit{reduce}\} \\
&\quad = [x_1 \oplus \cdots \oplus x_m] \qquad = \quad \mathit{reduce}\ (\oplus)\ \mathit{id}_\oplus\ xs
\end{aligned}
$$

□

Proof of option 2. Let $xs = [x_1, \ldots, x_m]$

$$
\begin{aligned}
&\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n\ xs \\
&\quad \{\text{definition of } \mathit{part\text{-}red}\} \\
&\quad = [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n)},\ \ldots,\ x_{\sigma(m-n+1)} \oplus \cdots \oplus x_{\sigma(m)}] \\
&\quad \{\text{represent permutation } \sigma \text{ with appropiate permutations } \sigma', \sigma''\} \\
&\quad = \big[x_{\sigma'(\sigma''(1))} \oplus \cdots \oplus x_{\sigma'(\sigma''(n))}, \ldots, \\
&\qquad x_{\sigma'(\sigma''(m-n+1))} \oplus \cdots \oplus x_{\sigma'(\sigma''(m))}\big] \\
&\quad \{\text{definition of } \mathit{part\text{-}red}\} \\
&\quad = \mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n\ [x_{\sigma''(1)}, \ldots, x_{\sigma''(m)}] \\
&\quad \{\text{definition of } \mathit{reorder}\} \\
&\quad = \mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n\ (\mathit{reorder}\ xs) \\
&\quad = (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n \circ \mathit{reorder})\ xs
\end{aligned}
$$

□

Proof of option 3. Let $xs = [x_1, \ldots, x_l]$.

$$
\begin{aligned}
&\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n\ xs \\
&\quad \{\text{definition of } \mathit{part\text{-}red}\} \\
&\quad = [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n)}, \ldots, x_{\sigma(l-n+1)} \oplus \cdots \oplus x_{\sigma(l)}] \\
&\quad \{\text{represent permutation } \sigma \text{ with appropiate permutations } \sigma_i\} \\
&\quad = \big[\, x_{\sigma_1(1)} \oplus \cdots \oplus x_{\sigma_1(n)},\ \ldots, \\
&\qquad\quad x_{\sigma_1(m-n+1)} \oplus \cdots \oplus x_{\sigma_1(m)}, \\
&\qquad\quad \ldots, \\
&\qquad\quad x_{\sigma_{l/m}(l-m+1)} \oplus \cdots \oplus x_{\sigma_{l/m}(l-m+n+1)},\ \ldots, \\
&\qquad\quad x_{\sigma_{l/m}(l-n+1)} \oplus \cdots \oplus x_{\sigma_{l/m}(l)} \,\big] \\
&\quad \{\text{definition of } \mathit{join}_m\} \\
&\quad = \mathit{join}_m \big[\quad [\, x_{\sigma_1(1)} \oplus \cdots \oplus x_{\sigma_1(n)}, \\
&\qquad\qquad\quad \ldots, \\
&\qquad\qquad\quad x_{\sigma_1(m-n+1)} \oplus \cdots \oplus x_{\sigma_1(m)}], \\
&\qquad\qquad \ldots, \\
&\qquad\qquad [\, x_{\sigma_{l/m}(l-m+1)} \oplus \cdots \oplus x_{\sigma_{l/m}(l-m+n+1)}, \\
&\qquad\qquad\quad \ldots, \\
&\qquad\qquad\quad x_{\sigma_{l/m}(l-n+1)} \oplus \cdots \oplus x_{\sigma_{l/m}(l)}] \quad\ \big] \\
&\quad \{\text{definition of } \mathit{part\text{-}red}\} \\
&\quad = \mathit{join}_m\ \big[\, \mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n\ [x_1, \ldots, x_m], \\
&\qquad\qquad \ldots, \\
&\qquad\qquad \mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n\ [x_{l-m+1+}, \ldots, x_l] \,\big] \\
&\quad \{\text{definition of } \mathit{map}\} \\
&\quad = \mathit{join}_m\ \big(\mathit{map}\ (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n) \\
&\qquad [[x_1, \ldots, x_m], \ldots, [x_{l-m+1+}, \ldots, x_l]]\big) \\
&\quad \{\text{definition of } \mathit{split}\} \\
&\quad = \mathit{join}_m\ \big(\mathit{map}\ (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n)\ (\mathit{split}\ m\ xs)\big) \\
&\quad = (\mathit{join}_m \circ \mathit{map}\ (\mathit{part\text{-}red}\ (\oplus)\ \mathit{id}_\oplus\ n) \circ \mathit{split}\ m)\ xs
\end{aligned}
$$

□

Proof of option 4. We will proof the following obvious equivalent reformulation of the rule:

$$\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n^m \to \textit{iterate}\ m\ (\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n)$$

Proof by induction. We start with the base case, let $m = 0$.

$$\begin{aligned}
\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n^0\ xs &= \textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ 1\ xs \\
&\quad \{\text{definition of } \textit{part-red}\} \\
&= xs \\
&\quad \{\text{definition of } \textit{iterate}\} \\
&= \textit{iterate}\ 0\ (\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n)\ xs
\end{aligned}$$

The induction step $(m-1) \to m$. Let $xs = [x_1, \ldots, x_l]$.

$$\begin{aligned}
&\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n^m\ xs \\
&\quad \{\text{definition of } \textit{part-red}\} \\
&= [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n^m)}, \ldots, x_{\sigma(l-n^m+1)} \oplus \cdots \oplus x_{\sigma(l)}] \\
&\quad \{\text{accociativity of } \oplus\} \\
&= [y_1 \oplus \cdots \oplus y_{(n^{m-1})}, \ldots, y_{(l/n-(n^{m-1}))} \oplus \cdots \oplus y_{(l/n)}] \\
&\qquad \text{where } y_i = (x_{\sigma((i-1)\times n+1)} \oplus \cdots \oplus x_{\sigma(i\times n)}) \\
&= \textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n^{(m-1)}\ [y_1, \ldots, y_{l/n}] \\
&\quad \{\text{definition of } y_i\} \\
&= \textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n^{(m-1)} \\
&\qquad [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n)}, \ldots, x_{\sigma(l-n+1)} \oplus \cdots \oplus x_{\sigma(l)}] \\
&\quad \{\text{induction hypothesis}\} \\
&= \textit{iterate}\ (m-1)\ (\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n) \\
&\qquad [x_{\sigma(1)} \oplus \cdots \oplus x_{\sigma(n)}, \ldots, x_{\sigma(l-n+1)} \oplus \cdots \oplus x_{\sigma(l)}] \\
&\quad \{\text{definition of } \textit{part-red}\} \\
&= \textit{iterate}\ (m-1)\ (\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n) \\
&\qquad (\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n\ xs) \\
&\quad \{\text{definition of } \textit{iterate}\} \\
&= \textit{iterate}\ m\ (\textit{part-red}\ (\oplus)\ \textit{id}_\oplus\ n)\ xs
\end{aligned}$$

□

SIMPLIFICATION RULES We repeat Equation (5.7) here:

$$
\begin{array}{lcl}
\mathit{join}_n \circ \mathit{split}\ n & \to & \mathit{id} \\
\mathit{split}\ n \circ \mathit{join}_n & \to & \mathit{id} \\
\mathit{asScalar}_n \circ \mathit{asVector}\ n & \to & \mathit{id} \\
\mathit{asVector}\ n \circ \mathit{asScalar}_n & \to & \mathit{id}
\end{array}
$$

Proof of option 1. Let $xs = [x_1, \ldots, x_m]$.

$$
\begin{aligned}
(\mathit{join}_n \circ \mathit{split}\ n)\ xs &= \mathit{join}_n\ (\mathit{split}\ n\ xs) \\
&\quad \{\text{definition of } \mathit{split}\} \\
&= \mathit{join}_n\ \big[[x_1, \ldots, x_n], \ldots, [x_{m-n+1}, \ldots, x_m]\big] \\
&\quad \{\text{definition of } \mathit{join}\} \\
&= xs
\end{aligned}
$$

□

Proof of option 2. Let $xs = \big[[x_1, \ldots, x_n], \ldots, [x_{m-n+1}, \ldots, x_m]\big]$.

$$
\begin{aligned}
(\mathit{split}\ n \circ \mathit{join}_n)\ xs &= \mathit{split}\ n\ (\mathit{join}_n\ xs) \\
&\quad \{\text{definition of } \mathit{join}\} \\
&= \mathit{split}\ n\ [x_1, \ldots, x_m] \\
&\quad \{\text{definition of } \mathit{split}\} \\
&= xs
\end{aligned}
$$

□

Proof of option 3. Let $xs = [x_1, \ldots, x_m]$.

$$
\begin{aligned}
(\mathit{asScalar}_n \circ \mathit{asVector}\ n)\ xs &= \mathit{asScalar}_n\ (\mathit{asVector}\ n\ xs) \\
&\quad \{\text{definition of } \mathit{asVector}\} \\
&= \mathit{asScalar}_n\ [\{x_1, \ldots, x_n\}, \ldots, \{x_{m-n+1}, \ldots, x_m\}] \\
&\quad \{\text{definition of } \mathit{asScalar}\} \\
&= xs
\end{aligned}
$$

□

Proof of option 4. Let $xs = [\{x_1, \ldots, x_n\}, \ldots, \{x_{m-n+1}, \ldots, x_m\}]$.

$$
\begin{aligned}
(\mathit{asVector}\ n \circ \mathit{asScalar}_n)\ xs &= \mathit{asVector}\ n\ (\mathit{asScalar}_n\ xs) \\
&\quad \{\text{definition of } \mathit{asScalar}\} \\
&= \mathit{asVector}\ n\ [x_1, \ldots, x_m] \\
&\quad \{\text{definition of } \mathit{asVector}\} \\
&= xs
\end{aligned}
$$

□

FUSION RULES We repeat Equation (5.8) here:

$$\begin{aligned} &\mathit{map}\ f \circ \mathit{map}\ g && \to && \mathit{map}\ (f \circ g) \\ &\mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_\oplus \circ \mathit{map}\ f && \to \\ &\qquad \mathit{reduce\text{-}seq}\ \big(\ \lambda\ (a, b)\ .\ a \oplus (f\ b)\ \big)\ \mathit{id}_\oplus \end{aligned}$$

Proof of option 1. Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned} (\mathit{map}\ f \circ \mathit{map}\ g)\ xs &= \mathit{map}\ f\ (\mathit{map}\ g\ xs) \\ &\quad \{\text{definition of } \mathit{map}\} \\ &= \mathit{map}\ f\ [g\ x_1, \ldots, g\ x_n] \\ &\quad \{\text{definition of } \mathit{map}\} \\ &= [f\ (g\ x_1), \ldots, f\ (g\ x_n)] \\ &\quad \{\text{definition of } \circ\} \\ &= [(f \circ g)\ x_1, \ldots, (f \circ g)\ x_n] \\ &\quad \{\text{definition of } \mathit{map}\} \\ &= \mathit{map}(f \circ g)\ xs \end{aligned}$$

□

Proof of option 2. Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned} &(\mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_\oplus \circ \mathit{map}\ f)\ xs \\ &\quad = \mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_\oplus\ (\mathit{map}\ f\ xs) \\ &\quad \{\text{definition of } \mathit{map}\} \\ &\quad = \mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_\oplus\ [f\ x_1, f\ x_2, \ldots, f\ x_n] \\ &\quad \{\text{definition of } \mathit{reduce\text{-}seq}\} \\ &\quad = [(\ldots((\mathit{id}_\oplus \oplus (f\ x_1)) \oplus (f\ x_2)) \cdots \oplus (f\ x_n))] \\ &\quad = [(\ldots((\mathit{id}_\oplus \odot x_1) \odot x_2) \cdots \odot x_n)] \\ &\qquad \text{where } (\odot) = \lambda\ (a, b)\ .\ a \oplus (f\ b) \\ &\quad \{\text{definition of } \mathit{reduce\text{-}seq}\} \\ &\quad = \mathit{reduce\text{-}seq}\ (\odot)\ \mathit{id}_\oplus\ xs \\ &\quad \{\text{definition of } \odot\} \\ &\quad = \mathit{reduce\text{-}seq}\ \big(\ \lambda\ (a, b)\ .\ a \oplus (f\ b)\ \big)\ \mathit{id}_\oplus\ xs \end{aligned}$$

□

A.2 OpenCL-specific rules

This section shows the proofs for the OpenCL-specific rules defined in Section 5.4.2.

maps We repeat Equation (5.9) here:

$$\begin{array}{lll} \mathit{map} \rightarrow \mathit{map\text{-}workgroup} & | & \mathit{map\text{-}local} \\ | \ \mathit{map\text{-}global} & | & \mathit{map\text{-}warp} \\ | \ \mathit{map\text{-}lane} & | & \mathit{map\text{-}seq} \end{array}$$

Proof. All of the options in this rule are correct by definition, as all map patterns share the same execution semantics. □

reduction We repeat Equation (5.10) here:

$$\mathit{reduce}\ (\oplus)\ \mathit{id}_{\oplus} \rightarrow \mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_{\oplus}$$

Proof. Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned} & \{\text{definition of } \mathit{reduce}\} \\ \mathit{reduce}\ (\oplus)\ \mathit{id}_{\oplus}\ xs &= x_1 \oplus \cdots \oplus x_n \\ & \{\text{associativity of } \oplus \text{ \& identity of } \mathit{id}_{\oplus}\} \\ &= (\ldots((\mathit{id}_{\oplus} \oplus x_1) \oplus x_2) \cdots \oplus x_n) \\ & \{\text{definition of } \mathit{reduce\text{-}seq}\} \\ &= \mathit{reduce\text{-}seq}\ (\oplus)\ \mathit{id}_{\oplus}\ xs \end{aligned}$$

□

reorder We repeat Equation (5.11) here:

$$\mathit{reorder} \rightarrow \mathit{id} \ | \ \mathit{reorder\text{-}stride}\ s$$

Proof of option 1. Let $xs = [x_1, \ldots, x_n]$.

$$\begin{aligned} \mathit{reorder}\ xs &= [x_{\sigma(1)}, \ldots, x_{\sigma(n)}] \\ & \{\text{choose } \sigma \text{ as the identity permutation}\} \Rightarrow \\ & [x_{\sigma(1)}, \ldots, x_{\sigma(n)}] = [x_1, \ldots, x_n] = \mathit{id}\ xs \end{aligned}$$

□

Proof of option 2. The definition of *reorder-stride* is as follows:

$$\begin{aligned} & \mathit{reorder\text{-}stride}\ s\ [x_1, \ldots, x_m] = [y_1, \ldots, y_m] \\ & \quad \text{where } y_i = x_{(i-1)\ \mathrm{div}\ n + s \times ((i-1)\ \mathrm{mod}\ n) + 1} \end{aligned}$$

We can express the relationship between ys and xs as a function σ with: $x_\sigma(i) = y_i$, i. e., $\sigma(i) = ((i-1) \text{ div } n + s \times ((i-1) \bmod n) + 1$. We want to show that σ is a permutation of the interval $[1, m]$, so that σ is a valid choice when rewriting *reduce*. We show the σ is a permutation by proving that σ is a bijective function mapping indices from the interval $[1, m]$ in the same interval.

First we show the injectivity, by showing:

$$\forall i, j \in [1, m] \text{ with } i \neq j \quad \Rightarrow \quad \sigma(i) \neq \sigma(j)$$

Let us assume without loss of generality that $i < j$.

As $i, j \in [1, m]$ and by definition of mod and div every summand in σ is positive. Therefore, for $\sigma(i) = \sigma(j)$ to be true all of their corresponding summands have to be equal. We will show, that this can never be the case. Let us write j as $j = i + k$ where $0 < k < m - 1$.

If we assume: $(i-1) \text{ div } n = (i+k-1) \text{ div } n$

$$\begin{aligned}
&\{\text{definition of div \& mod and } i, k > 0\} \\
&\Rightarrow (i-1) \bmod n \neq (i+k-1) \bmod n \\
&\{s > 0\} \\
&\Rightarrow s \times ((i-1) \bmod n) \neq s \times ((i+k-1) \bmod n) \\
&\Rightarrow ((i-1) \text{ div } n) + s \times ((i-1) \bmod n) + 1 \\
&\quad \neq ((j-1) \text{ div } n) + s \times ((j-1) \bmod n) + 1
\end{aligned}$$

If we assume the opposite $(i-1) \bmod n = (i+k-1) \bmod n$:

$$\begin{aligned}
&\{\text{definition of div \& mod and } i, k > 0\} \\
&\Rightarrow (i-1) \text{ div } n \neq (i+k-1) \text{ div } n \\
&\Rightarrow ((i-1) \text{ div } n) + s \times ((i-1) \bmod n) + 1 \\
&\quad \neq ((j-1) \text{ div } n) + s \times ((j-1) \bmod n) + 1
\end{aligned}$$

This shows the injectivity of σ.

Now we show the surjectivity, by showing:

$$\forall i \in [1, m] \quad \sigma(i) \in [1, m]$$

We know that $m = s \times n$

$$\Rightarrow (i-1) \text{ div } n \leqslant s \quad \forall i \in [1, m]$$

By definition of mod: $((i-1) \bmod n) \leqslant (n-1) \quad \forall i \in [1, m]$

$$\begin{aligned}
&\Rightarrow ((i-1) \text{ div } n) + s \times ((i-1) \bmod n) \leqslant s + s \times (n-1) \\
&\quad = s \times n = m
\end{aligned}$$

As already discussed is $\sigma(i) > 0\ \forall i \in [1, m]$, because of the definitions of mod, div, and σ.

Therefore, σ is injective and surjective, thus, bijective which makes it a well defined permutation of $[1, m]$.

□

LOCAL AND GLOBAL MEMORY We repeat Equation (5.12) here:

$$\begin{aligned} \textit{map-local}\ f &\to \textit{toGlobal}\ (\textit{map-local}\ f) \\ \textit{map-local}\ f &\to \textit{toLocal}\ (\textit{map-local}\ f) \end{aligned}$$

Proof. These rules follow directly from the definition of *toGlobal* and *toLocal*, as these have no effect on the computed value, i. e., they behave like the *id* function. □

VECTORIZATION We repeat Equation (5.13) here:

$$\textit{map}\ f \to \textit{asScalar} \circ \textit{map}\ (\textit{vectorize}\ n\ f) \circ \textit{asVector}\ n$$

Proof. Let $xs = [x_1, \ldots, x_m]$.

$$\begin{aligned} &\textit{map}\ f\ xs = [f\ x_1, \ldots, f\ x_m] \\ &\quad \{\text{definition of } \textit{asScalar}\} \\ &\quad = \textit{asScalar}\ [\{f\ x_1, \ldots, f\ x_n\}, \ldots, \{f\ x_{m-n+1}, \ldots, f\ x_m\}] \\ &\quad = \textit{asScalar}\ [f_n\ \{x_1, \ldots, x_n\}, \ldots, f_n\ \{x_{m-n+1}, \ldots, x_m\}] \\ &\quad \text{where } f_n\ \{x_1, \ldots, x_n\} = \{f\ x_1, \ldots, f\ x_n\} \\ &\quad \{\text{definition of } f_n \text{ and } \textit{vectorize}\} \\ &\quad = \textit{asScalar}\ [(\textit{vectorize}\ n\ f)\ \{x_1, \ldots, x_n\}, \ \ldots, \\ &\qquad\qquad (\textit{vectorize}\ n\ f)\ \{x_{m-n+1}, \ldots, x_m\}] \\ &\quad \{\text{definition of } \textit{map}\} \\ &\quad = \textit{asScalar}\ (\textit{map}\ (\textit{vectorize}\ n\ f) \\ &\qquad [\{x_1, \ldots, x_n\}, \ldots, \{x_{m-n+1}, \ldots, x_m\}] \\ &\quad \{\text{definition of } \textit{asVector}\} \\ &\quad = \textit{asScalar}\ (\textit{map}\ (\textit{vectorize}\ n\ f)\ (\textit{asVector}\ n\ xs)) \\ &\quad = (\textit{asScalar} \circ \textit{map}\ (\textit{vectorize}\ n\ f) \circ \textit{asVector}\ n)\ xs \end{aligned}$$

□

B

DERIVATIONS FOR PARALLEL REDUCTION

This appendix shows the derivations which transform the high-level expression *reduce* $(+)$ 0 into the low-level expressions shown in Section 5.4.3.2. The numbers above the equality sign refer to the rules from Figure 5.7 and Figure 5.8.

FIRST PATTERN-BASED EXPRESSION This is the derivation for the expression shown in Listing 5.8.

$$
\begin{aligned}
&\mathit{vecSum} = \mathit{reduce}\ (+)\ 0 \overset{5.7e}{=} \mathit{reduce} \circ \mathit{part\text{-}red}\ (+)\ 0\ 128 \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 128) \circ \mathit{split}\ 128 \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \big(\mathit{iterate}\ 7\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)\big) \circ \mathit{split}\ 128 \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \big(\\
&\qquad \mathit{iterate}\ 7\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2) \\
&\quad \big) \circ \mathit{split}\ 128 \\
&\overset{5.7a}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \big(\\
&\qquad \mathit{map}\ \mathtt{id} \circ \\
&\qquad \mathit{iterate}\ 7\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2) \circ \\
&\qquad \mathit{map}\ \mathtt{id} \\
&\quad \big) \circ \mathit{split}\ 128 \\
&\overset{5.7d}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \big(\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathtt{id}) \circ \mathit{split}\ 1 \circ \\
&\qquad \mathit{iterate}\ 7\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2) \circ \\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathtt{id}) \circ \mathit{split}\ 1 \\
&\quad \big) \circ \mathit{split}\ 128 \\
&\overset{5.8a}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map\text{-}workgroup}\ \big(\\
&\qquad \mathit{join} \circ \mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathtt{id}) \circ \mathit{split}\ 1 \circ \\
&\qquad \mathit{iterate}\ 7\ (\mathit{join} \circ \mathit{map\text{-}local}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2) \circ \\
&\qquad \mathit{join} \circ \mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathtt{id}) \circ \mathit{split}\ 1 \\
&\quad \big) \circ \mathit{split}\ 128
\end{aligned}
$$

$$
\begin{aligned}
&\overset{5.7e\&5.8b}{=} reduce \circ join \circ map\text{-}workgroup\ (\\
&\quad join \circ map\text{-}local\ (map\text{-}seq\ \mathtt{id}) \circ split\ 1 \circ\\
&\quad iterate\ 7\ (join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2) \circ\\
&\quad join \circ map\text{-}local\ (map\text{-}seq\ \mathtt{id}) \circ split\ 1\\
&)\ \circ split\ 128\\
&\overset{5.8d}{=} reduce \circ join \circ map\text{-}workgroup\ (\\
&\quad join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1 \circ\\
&\quad iterate\ 7\ (join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2) \circ\\
&\quad join \circ toLocal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1\\
&)\ \circ split\ 128
\end{aligned}
$$

AVOIDING INTERLEAVED ADDRESSING This is the derivation for the expression shown in Listing 5.9.

$$
\begin{aligned}
&\mathtt{vecSum} = reduce\ (+)\ 0 \overset{5.7e}{=} reduce \circ part\text{-}red\ (+)\ 0\ 128\\
&\overset{5.7e}{=} reduce \circ join \circ map\ (part\text{-}red\ (+)\ 0\ 128)\ \circ split\ 128\\
&\overset{5.7e}{=} reduce \circ join \circ map\ (\\
&\quad iterate\ 7\ (part\text{-}red\ (+)\ 0\ 2)\ )\ \circ split\ 128\\
&\overset{5.7e}{=} reduce \circ join \circ map\ (\\
&\quad iterate\ 7\ (part\text{-}red\ (+)\ 0\ 2\ \circ reorder)\ )\ \circ split\ 128\\
&\overset{5.7e}{=} reduce \circ join \circ map\ (\\
&\quad iterate\ 7\ (join \circ map\ (part\text{-}red\ (+)\ 0\ 2) \circ split\ 2 \circ reorder)\\
&)\ \circ split\ 128\\
&\overset{5.7a}{=} reduce \circ join \circ map\ (\\
&\quad map\ \mathtt{id} \circ\\
&\quad iterate\ 7\ (join \circ map\ (part\text{-}red\ (+)\ 0\ 2) \circ split\ 2 \circ reorder) \circ\\
&\quad map\ \mathtt{id}\\
&)\ \circ split\ 128\\
&\overset{5.7d}{=} reduce \circ join \circ map\ (\\
&\quad join \circ map\ (map\ \mathtt{id}) \circ split\ 1 \circ\\
&\quad iterate\ 7\ (join \circ map\ (part\text{-}red\ (+)\ 0\ 2) \circ split\ 2 \circ reorder) \circ\\
&\quad join \circ map\ (map\ \mathtt{id}) \circ split\ 1\\
&)\ \circ split\ 128
\end{aligned}
$$

$\overset{5.8a}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1 ∘
  iterate 7 (*join* ∘ *map-local* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*) ∘
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1
) ∘ *split* 128

$\overset{5.7e\&5.8b}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1 ∘
  iterate 7 (*join* ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘ *reorder*) ∘
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1
) ∘ *split* 128

$\overset{5.8c}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1 ∘
  iterate 7 (λ xs . *join* ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘
    reorder-stride ((size xs)/2) $ xs) ∘
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1
) ∘ *split* 128

$\overset{5.8d}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *toGlobal* (*map-local* (*map-seq id*)) ∘ *split* 1 ∘
  iterate 7 (λ xs . *join* ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘
    reorder-stride ((size xs)/2) $ xs) ∘
  ∘ *join* ∘ *toLocal* (*map-local* (*map-seq id*)) ∘ *split* 1
) ∘ *split* 128

INCREASE COMPUTATIONAL INTENSITY PER WORK-ITEM This is the derivation for the expression shown in Listing 5.10.

vecSum = *reduce* (+) 0 $\overset{5.7e}{=}$ *reduce* ∘ *part-red* (+) 0 256

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (*part-red* (+) 0 256) ∘ *split* 256

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
  iterate 8 (*part-red* (+) 0 2)) ∘ *split* 256

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
  iterate 8 (*part-red* (+) 0 2 ∘ *reorder*)) ∘ *split* 256

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
  iterate 8 (*join* ∘ *map* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*)
) ∘ *split* 256

$\overset{5.7a}{=}$ *reduce* ∘ *join* ∘ *map* (
  map id ∘
  iterate 8 (*join* ∘ *map* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*)
) ∘ *split* 256

$\overset{5.7d}{=}$ *reduce* ∘ *join* ∘ *map* (
  join ∘ *map* (*map* id) ∘ *split* 1 ∘
  iterate 8 (*join* ∘ *map* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*) ∘
) ∘ *split* 256

$\overset{5.7b}{=}$ *reduce* ∘ *join* ∘ *map* (
  join ∘ *map* (*map* id) ∘ *split* 1 ∘
  iterate 7 (*join* ∘ *map* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*)
  join ∘ *map* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*
) ∘ *split* 256

$\overset{5.8a}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1 ∘
  iterate 7 (*join* ∘ *map-local* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*)
  join ∘ *map-local* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder*
) ∘ *split* 256

$\overset{5.7e\&5.8b}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1 ∘
  iterate 7 (*join* ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘ *reorder*) ∘
  join ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘ *reorder*
) ∘ *split* 256

$\overset{5.8c}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *map-local* (*map-seq* id) ∘ *split* 1 ∘
  iterate 7 (λ xs . *join* ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘
    reorder-stride ((size xs)/2) $ xs) ∘
  join ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘
    reorder-stride 128
) ∘ *split* 256

$\overset{5.8d}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
  join ∘ *toGlobal* (*map-local* (*map-seq id*)) ∘ *split* 1 ∘
  iterate 7 (λ xs . *join* ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘
    reorder-stride ((size xs)/2) $ xs) ∘
  join ∘ *toLocal* (*map-local* (*reduce-seq* (+) 0)) ∘ *split* 2 ∘
    reorder-stride 128
) ∘ *split* 256

AVOID SYNCHRONIZATION INSIDE A WARP This is the derivation for the expression shown in Listing 5.11.

$$
\begin{aligned}
\mathit{vecSum} &= \mathit{reduce}\ (+)\ 0 \overset{5.7e}{=} \mathit{reduce} \circ \mathit{part\text{-}red}\ (+)\ 0\ 256 \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 256) \circ \mathit{split}\ 256 \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\ \mathit{iterate}\ 8\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)\ ) \circ \mathit{split}\ 256 \\
&\overset{5.7b}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\\
&\qquad \mathit{iterate}\ 6\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \\
&\qquad \mathit{iterate}\ 2\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)\ ) \circ \mathit{split}\ 256 \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\\
&\qquad \mathit{part\text{-}red}\ (+)\ 0\ 64 \circ \mathit{iterate}\ 2\ (\mathit{part\text{-}red}\ (+)\ 0\ 2 \circ \mathit{reorder})\\
&\quad)\ \circ \mathit{split}\ 256 \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\\
&\qquad \mathit{part\text{-}red}\ (+)\ 0\ 64 \circ \\
&\qquad \mathit{iterate}\ 2\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2 \circ \mathit{reorder})\\
&\quad)\ \circ \mathit{split}\ 256 \\
&\overset{5.7a}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\\
&\qquad \mathit{map}\ \mathtt{id} \circ \\
&\qquad \mathit{part\text{-}red}\ (+)\ 0\ 64 \circ \\
&\qquad \mathit{iterate}\ 2\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2 \circ \mathit{reorder})\\
&\quad)\ \circ \mathit{split}\ 256 \\
&\overset{5.7d}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathtt{id}) \circ \mathit{split}\ 1 \circ \\
&\qquad \mathit{part\text{-}red}\ (+)\ 0\ 64 \circ \\
&\qquad \mathit{iterate}\ 2\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2 \circ \mathit{reorder}) \circ \\
&\quad)\ \circ \mathit{split}\ 256 \\
&\overset{5.7b}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathtt{id}) \circ \mathit{split}\ 1 \circ \\
&\qquad \mathit{part\text{-}red}\ (+)\ 0\ 64 \circ \\
&\qquad \mathit{iterate}\ 1\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2 \circ \mathit{reorder})\\
&\qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \mathit{split}\ 2 \circ \mathit{reorder}\\
&\quad)\ \circ \mathit{split}\ 256
\end{aligned}
$$

$$
\begin{aligned}
\overset{5.7e}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\; \big(\\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{map}\; \mathtt{id}) \circ \mathit{split}\; 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 64)\; \mathit{split}\; 64 \circ \\
& \quad \mathit{iterate}\; 1\; (\mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder}) \\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder} \\
& \big) \circ \mathit{split}\; 256
\end{aligned}
$$

$$
\begin{aligned}
\overset{5.7e}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\; \big(\\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{map}\; \mathtt{id}) \circ \mathit{split}\; 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\; \big(\; \mathit{iterate}\; 6\; (\mathit{part\text{-}red}\; (+)\; 0\; 2)\; \big) \circ \mathit{split}\; 64 \circ \\
& \quad \mathit{iterate}\; 1\; (\mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder}) \\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder} \\
& \big) \circ \mathit{split}\; 256
\end{aligned}
$$

$$
\begin{aligned}
\overset{5.7e}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\; \big(\\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{map}\; \mathtt{id}) \circ \mathit{split}\; 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\; \big(\; \mathit{iterate}\; 6\; (\mathit{part\text{-}red}\; (+)\; 0\; 2\; \circ \mathit{reorder})\; \big) \circ \mathit{split}\; 64 \circ \\
& \quad \mathit{iterate}\; 1\; (\mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder}) \\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder} \\
& \big) \circ \mathit{split}\; 256
\end{aligned}
$$

$$
\begin{aligned}
\overset{5.7e}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\; \big(\\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{map}\; \mathtt{id}) \circ \mathit{split}\; 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\; \big(\; \mathit{iterate}\; 6\; (\mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \\
& \qquad \mathit{reorder})\; \big) \circ \mathit{split}\; 64 \circ \\
& \quad \mathit{iterate}\; 1\; (\mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder}) \\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder} \\
& \big) \circ \mathit{split}\; 256
\end{aligned}
$$

$$
\begin{aligned}
\overset{5.7b}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\; \big(\\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{map}\; \mathtt{id}) \circ \mathit{split}\; 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\; \big(\\
& \qquad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2)) \circ \mathit{split}\; 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2)) \circ \mathit{split}\; 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2)) \circ \mathit{split}\; 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2)) \circ \mathit{split}\; 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2)) \circ \mathit{split}\; 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2)) \circ \mathit{split}\; 2 \circ \mathit{reorder} \\
& \quad \big) \circ \mathit{split}\; 64 \circ \\
& \quad \mathit{iterate}\; 1\; (\mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder}) \circ \\
& \quad \mathit{join} \circ \mathit{map}\; (\mathit{part\text{-}red}\; (+)\; 0\; 2) \circ \mathit{split}\; 2 \circ \mathit{reorder} \\
& \big) \circ \mathit{split}\; 256
\end{aligned}
$$

5.8a & 5.7e $\underline{=}$ 5.8b & 5.8d

$$
\begin{aligned}
&\mathit{reduce} \circ \mathit{join} \circ \mathit{map\text{-}workgroup}\ (\\
&\quad \mathit{join} \circ \mathit{toGlobal}\ (\mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathit{id})) \circ \mathit{split}\ 1 \circ\\
&\quad \mathit{join} \circ \mathit{map\text{-}warp}\ (\\
&\qquad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 1 \circ\\
&\qquad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 2 \circ\\
&\qquad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 4 \circ\\
&\qquad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 8 \circ\\
&\qquad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 16 \circ\\
&\qquad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 32\\
&\quad) \circ \mathit{split}\ 64 \circ\\
&\quad \mathit{iterate}\ 1\ (\lambda\ \mathtt{xs}\ .\ \mathit{join} \circ \mathit{map\text{-}local}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2 \circ\\
&\qquad \mathit{reorder\text{-}stride}\ ((\mathtt{size}\ \mathtt{xs})/2)\ \$\ \mathtt{xs}\ ) \circ\\
&\quad \mathit{join} \circ \mathit{toLocal}\ (\mathit{map\text{-}local}\ (\mathit{reduce\text{-}seq}\ (+)\ 0)) \circ \mathit{split}\ 2 \circ\\
&\qquad \mathit{reorder\text{-}stride}\ 128\\
&)\ \circ \mathit{split}\ 256
\end{aligned}
$$

COMPLETE LOOP UNROLLING This is the derivation for the expression shown in Listing 5.12. We continue with the last expression from the previous derivation.

$$
\begin{aligned}
&\mathit{vecSum} = \mathit{reduce}\ (+)\ 0\\
&\quad \underline{=}\ \text{derivation above}\\
&\quad \mathit{reduce} \circ \mathit{join} \circ \mathit{map\text{-}workgroup}\ (\\
&\qquad \mathit{join} \circ \mathit{toGlobal}\ (\mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathit{id})) \circ \mathit{split}\ 1 \circ\\
&\qquad \mathit{join} \circ \mathit{map\text{-}warp}\ (\\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 1 \circ\\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 2 \circ\\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 4 \circ\\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 8 \circ\\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 16 \circ\\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 32\\
&\qquad) \circ \mathit{split}\ 64 \circ\\
&\qquad \mathit{iterate}\ 1\ (\lambda\ \mathtt{xs}\ .\ \mathit{join} \circ \mathit{map\text{-}local}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2 \circ\\
&\qquad\quad \mathit{reorder\text{-}stride}\ ((\mathtt{size}\ \mathtt{xs})/2)\ \$\ \mathtt{xs}\ ) \circ\\
&\qquad \mathit{join} \circ \mathit{toLocal}\ (\mathit{map\text{-}local}\ (\mathit{reduce\text{-}seq}\ (+)\ 0)) \circ \mathit{split}\ 2 \circ\\
&\qquad\quad \mathit{reorder\text{-}stride}\ 128\\
&\quad)\ \circ \mathit{split}\ 256
\end{aligned}
$$

$$
\begin{aligned}
&\overset{5.7b}{=} \\
&\quad \mathit{reduce} \circ \mathit{join} \circ \mathit{map\text{-}workgroup}\ \big(\\
&\qquad \mathit{join} \circ \mathit{toGlobal}\ (\mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathit{id})) \circ \mathit{split}\ 1 \circ \\
&\qquad \mathit{join} \circ \mathit{map\text{-}warp}\ \big(\\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 1 \circ \\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 2 \circ \\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 4 \circ \\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 8 \circ \\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 16 \circ \\
&\qquad\quad \mathit{join} \circ \mathit{map\text{-}lane}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 32 \\
&\qquad \big) \circ \mathit{split}\ 64 \circ \\
&\qquad \mathit{join} \circ \mathit{map\text{-}local}\ (\mathit{reduce\text{-}seq}\ (+)\ 0) \circ \mathit{split}\ 2\ \circ \mathit{reorder\text{-}stride}\ 64) \circ \\
&\qquad \mathit{join} \circ \mathit{toLocal}\ (\mathit{map\text{-}local}\ (\mathit{reduce\text{-}seq}\ (+)\ 0)) \circ \mathit{split}\ 2 \circ \\
&\qquad\quad \mathit{reorder\text{-}stride}\ 128 \\
&\quad \big) \circ \mathit{split}\ 256
\end{aligned}
$$

FULLY OPTIMIZED IMPLEMENTATION This is the derivation for the expression shown in Listing 5.13.

$$
\begin{aligned}
&\mathit{vecSum} = \mathit{reduce}\ (+)\ 0 \overset{5.7e}{=} \mathit{reduce} \circ \mathit{part\text{-}red}\ (+)\ 0\ \mathtt{blockSize} \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ \mathtt{blockSize}) \circ \mathit{split}\ \mathtt{blockSize} \\
&\overset{5.7e}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \big(\\
&\qquad \mathit{iterate}\ \log_2(\mathtt{blockSize})\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \\
&\quad \big) \circ \mathit{split}\ \mathtt{blockSize} \\
&\overset{5.7b}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \big(\\
&\qquad \mathit{iterate}\ 7\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \\
&\qquad \mathit{iterate}\ (\log_2(\mathtt{blockSize}/128))\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \\
&\quad \big) \circ \mathit{split}\ \mathtt{blockSize} \\
&\overset{5.7b}{=} \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \big(\\
&\qquad \mathit{iterate}\ 6\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \\
&\qquad \mathit{iterate}\ 1\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \circ \\
&\qquad \mathit{iterate}\ (\log_2(\mathtt{blockSize}/128))\ (\mathit{part\text{-}red}\ (+)\ 0\ 2) \\
&\quad \big) \circ \mathit{split}\ \mathtt{blockSize}
\end{aligned}
$$

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
  part-red (+) 0 64 ∘
  part-red (+) 0 2 ∘
  part-red (+) 0 (blockSize/128)
) ∘ *split* blockSize

$\overset{5.7a}{=}$ *reduce* ∘ *join* ∘ *map* (
  map id ∘
  part-red (+) 0 64 ∘
  part-red (+) 0 2 ∘
  part-red (+) 0 (blockSize/128)
) ∘ *split* blockSize

$\overset{5.7d}{=}$ *reduce* ∘ *join* ∘ *map* (
  join ∘ *map* (*map* id) ∘ *split* 1 ∘
  part-red (+) 0 64 ∘
  part-red (+) 0 2 ∘
  part-red (+) 0 (blockSize/128)
) ∘ *split* blockSize

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
  join ∘ *map* (*map* id) ∘ *split* 1 ∘
  join ∘ *map* (*part-red* (+) 0 64) *split* 64 ∘
  part-red (+) 0 2 ∘
  part-red (+) 0 (blockSize/128)
) ∘ *split* blockSize

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
  join ∘ *map* (*map* id) ∘ *split* 1 ∘
  join ∘ *map* (*iterate* 6 (*part-red* (+) 0 2)) ∘ *split* 64 ∘
  part-red (+) 0 2 ∘
  part-red (+) 0 (blockSize/128)
) ∘ *split* blockSize

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
  join ∘ *map* (*map* id) ∘ *split* 1 ∘
  join ∘ *map* (*iterate* 6 (*part-red* (+) 0 2 ∘ *reorder*)) ∘ *split* 64 ∘
  part-red (+) 0 2 ∘
  part-red (+) 0 (blockSize/128)
) ∘ *split* blockSize

$$
\begin{aligned}
\overset{5.7e}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \Big(\\
& \quad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathtt{id}) \circ \mathit{split}\ 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\ \Big(\ \mathit{iterate}\ 6\ (\mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)\ \circ \mathit{split}\ 2 \circ \\
& \qquad \mathit{reorder})\ \Big) \circ \mathit{split}\ 64 \circ \\
& \quad \mathit{part\text{-}red}\ (+)\ 0\ 2 \circ \\
& \quad \mathit{part\text{-}red}\ (+)\ 0\ (\mathtt{blockSize}/128) \\
& \Big)\ \circ \mathit{split}\ \mathtt{blockSize}
\end{aligned}
$$

$$
\begin{aligned}
\overset{5.7b}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \Big(\\
& \quad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathtt{id}) \circ \mathit{split}\ 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\ \Big(\\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \\
& \quad \Big) \circ \mathit{split}\ 64 \circ \\
& \quad \mathit{part\text{-}red}\ (+)\ 0\ 2 \circ \\
& \quad \mathit{part\text{-}red}\ (+)\ 0\ (\mathtt{blockSize}/128) \\
& \Big)\ \circ \mathit{split}\ \mathtt{blockSize}
\end{aligned}
$$

$$
\begin{aligned}
\overset{5.7e}{=}\; & \mathit{reduce} \circ \mathit{join} \circ \mathit{map}\ \Big(\\
& \quad \mathit{join} \circ \mathit{map}\ (\mathit{map}\ \mathtt{id}) \circ \mathit{split}\ 1 \circ \\
& \quad \mathit{join} \circ \mathit{map}\ \Big(\\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \circ \\
& \qquad \mathit{join} \circ \mathit{map}\ (\mathit{part\text{-}red}\ (+)\ 0\ 2)) \circ \mathit{split}\ 2 \circ \mathit{reorder} \\
& \quad \Big) \circ \mathit{split}\ 64 \circ \\
& \quad \mathit{part\text{-}red}\ (+)\ 0\ 2 \circ \mathit{reorder} \circ \\
& \quad \mathit{part\text{-}red}\ (+)\ 0\ (\mathtt{blockSize}/128) \circ \mathit{reorder} \\
& \Big)\ \circ \mathit{split}\ \mathtt{blockSize}
\end{aligned}
$$

$\overset{5.7e}{=}$ *reduce* ∘ *join* ∘ *map* (
join ∘ *map* (*map* id) ∘ *split* 1 ∘
join ∘ *map* (
join ∘ *map* (*part-red* (+) 0 2)) ∘ *split* 2 ∘ *reorder* ∘
join ∘ *map* (*part-red* (+) 0 2)) ∘ *split* 2 ∘ *reorder* ∘
join ∘ *map* (*part-red* (+) 0 2)) ∘ *split* 2 ∘ *reorder* ∘
join ∘ *map* (*part-red* (+) 0 2)) ∘ *split* 2 ∘ *reorder* ∘
join ∘ *map* (*part-red* (+) 0 2)) ∘ *split* 2 ∘ *reorder* ∘
join ∘ *map* (*part-red* (+) 0 2)) ∘ *split* 2 ∘ *reorder*
) ∘ *split* 64 ∘
join ∘ *map* (*part-red* (+) 0 2) ∘ *split* 2 ∘ *reorder* ∘
join ∘ *map* (*part-red* (+) 0 (blockSize/128))∘
split (blockSize/128) ∘ *reorder*
) ∘ *split* blockSize

$\overset{5.8a \;\&\; 5.7e \;\&\; 5.8b \;\&\; 5.8d}{=}$ *reduce* ∘ *join* ∘ *map-workgroup* (
join ∘ *toGlobal* (*map-local* (*map-seq* id)) ∘ *split* 1 ∘
join ∘ *map-warp* (
join ∘ *map-lane* (*reduce-seq* (+) 0)) ∘ *split* 2 ∘ *reorder-stride* 1 ∘
join ∘ *map-lane* (*reduce-seq* (+) 0)) ∘ *split* 2 ∘ *reorder-stride* 2 ∘
join ∘ *map-lane* (*reduce-seq* (+) 0)) ∘ *split* 2 ∘ *reorder-stride* 4 ∘
join ∘ *map-lane* (*reduce-seq* (+) 0)) ∘ *split* 2 ∘ *reorder-stride* 8 ∘
join ∘ *map-lane* (*reduce-seq* (+) 0)) ∘ *split* 2 ∘ *reorder-stride* 16 ∘
join ∘ *map-lane* (*reduce-seq* (+) 0)) ∘ *split* 2 ∘ *reorder-stride* 32
) ∘ *split* 64 ∘
join ∘ *map-local* (*reduce-seq* (+) 0) ∘ *split* 2 ∘ *reorder-stride* 64 ∘
join ∘ *toLocal* (*map-local* (*reduce-seq* (+) 0))∘
split (blockSize/128) ∘ *reorder-stride* 128
) ∘ *split* blockSize

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

BIBLIOGRAPHY

[1] *2014 OpenCL™ Optimization Guide*. Intel. 2014. URL: https://software.intel.com/en-us/iocl_2014_opg (cit. on pp. 118, 142).

[2] *Accelerated Parallel Processing Math Libraries (APPML)*. Version 1.10. AMD. Mar. 2013. URL: http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/ (cit. on pp. 87, 88).

[3] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, M. Torquati. "Targeting Distributed Systems in FastFlow." In: *Euro-Par 2012: Parallel Processing Workshops, August 27-31, 2012. Revised Selected Papers*. Edited by Ioannis Caragiannis et al. Vol. 7640. Lecture Notes in Computer Science. Springer, 2012, pp. 47–56 (cit. on pp. 23, 203).

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati. "Accelerating Code on Multi-cores with FastFlow." In: *Euro-Par 2011 Parallel Processing - 17th International Conference, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*. Edited by Emmanuel Jeannot, Raymond Namyst, and Jean Roman. Vol. 6853. Lecture Notes in Computer Science. Springer, 2011, pp. 170–181 (cit. on p. 203).

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati. "FastFlow: high-level and efficient streaming on multi-core." In: *Programming Multi-core and Many-core Computing Systems*. Edited by Sabri Pllana and Fatos Xhafa. Wiley Series on Parallel and Distributed Computing. Wiley-Blackwell, Oct. 2014 (cit. on pp. 23, 203).

[6] M. Aldinucci, M. Meneghin, M. Torquati. "Efficient Smith-Waterman on Multi-core with FastFlow." In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*. Edited by Marco Danelutto, Julien Bourgeois, and Tom Gross. IEEE Computer Society, 2010, pp. 195–199 (cit. on p. 24).

[7] M. Aldinucci, C. Spampinato, M. Drocco, M. Torquati, S. Palazzo. "A parallel edge preserving algorithm for salt and pepper image denoising." In: *3rd International Conference on Image Processing Theory Tools and Applications, IPTA 2012, 15-18 October 2012, Istanbul, Turkey*. Edited by Khalifa Djemal and Mohamed Deriche. IEEE, 2012, pp. 97–104 (cit. on p. 204).

[8] M. Alt. "Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance." PhD Thesis. 2007 (cit. on pp. 24, 199, 200, 203).

[9] M. Alt, S. Gorlatch. "Using Skeletons in a Java-Based Grid System." In: *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*. Edited by Harald Kosch, László Böszörményi, and Hermann Hellwagner. Vol. 2790. Lecture Notes in Computer Science. Springer, 2003, pp. 742–749 (cit. on p. 203).

[10] *AMD Accelerated Parallel Processing OpenCL User Guide*. rev 1.0. AMD. Dec. 2014. URL: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/documentation/ (cit. on pp. 118, 142).

[11] *AMD OpenCL Accelerated Parallel Processing Software Development Kit (APP SDK)*. Version 2.8. AMD. Dec. 2012 (cit. on p. 99).

[12] J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, S. P. Amarasinghe. "PetaBricks: a language and compiler for algorithmic choice." In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Edited by Michael Hind and Amer Diwan. ACM, 2009, pp. 38–49 (cit. on p. 210).

[13] *Apache Hadoop*. Apache Software Foundation. 2014. URL: http://hadoop.apache.org/ (cit. on p. 206).

[14] N. Arora, A. Shringarpure, R. W. Vuduc. "Direct N-body Kernels for Multicore Platforms." In: *Proceedings of the International Conference on Parallel Processing*. ICPP '09. Vienna, Austria: IEEE Computer Society, Sept. 2009, pp. 379–387 (cit. on p. 45).

[15] J. S. Auerbach, D. F. Bacon, P. Cheng, R. M. Rabbah. "Lime: a Java-compatible and synthesizable language for heterogeneous architectures." In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Edited by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 89–108 (cit. on p. 209).

[16] C. Augonnet, S. Thibault, R. Namyst, P. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures." In: *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*. Edited by Henk J. Sips, Dick H. J.

Epema, and Hai-Xiang Lin. Vol. 5704. Lecture Notes in Computer Science. Springer, 2009, pp. 863–874 (cit. on p. 204).

[17] N. Bell, J. Hoberock. "Thrust: A Productivity-Oriented Library for CUDA." In: *GPU Computing Gems*. Edited by Wen-mei W. Hwu. Morgan Kaufmann, 2011, pp. 359–371 (cit. on pp. 179, 206).

[18] R. S. Bird. "Lectures on Constructive Functional Programming." In: *Constructive Methods in Computer Science*. Edited by Manfred Broy. Springer-Verlag, 1988, pp. 151–218 (cit. on pp. 36, 137, 212).

[19] H. Bischof, S. Gorlatch, E. Kitzelmann. "Cost Optimality And Predictability Of Parallel Programming With Skeletons." In: *Parallel Processing Letters* 13.4 (2003), pp. 575–587 (cit. on pp. 24, 199).

[20] C. M. Bishop. *Pattern Recognition and Machine Learning*. New York, NY, USA: Springer, Oct. 1, 2007 (cit. on p. 200).

[21] G. E. Blelloch. "Prefix Sums and Their Applications." In: *Synthesis of Parallel Algorithms*. Edited by John H. Reif. San Mateo, CA, USA: Morgan Kaufmann, 1991, pp. 35–60 (cit. on p. 39).

[22] *Bolt C++ Template Library. C++ template library for heterogeneous compute*. AMD. 2014. URL: https://github.com/HSA-Libraries/Bolt (cit. on p. 206).

[23] S. Breuer. "Introducing a Skeleton for Stencil Computations to the SkelCL Library." Masterthesis. Jan. 2014 (cit. on p. 66).

[24] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, K. Olukotun. "A Heterogeneous Parallel Framework for Domain-Specific Languages." In: *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. Edited by Lawrence Rauchwerger and Vivek Sarkar. IEEE Computer Society, 2011, pp. 89–100 (cit. on p. 205).

[25] D. Buono, M. Danelutto, S. Lametti, M. Torquati. "Parallel Patterns for General Purpose Many-Core." In: *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, February 27 - March 1, 2013*. IEEE Computer Society, 2013, pp. 131–139 (cit. on p. 204).

[26] *C++ Extensions for Parallelism*. N4312. ISO/IEC – JTC1 – SC22 – WG21. Nov. 21, 2014 (cit. on p. 206).

[27] H. Cao, Y. Zhao, S. Ho, E. Seelig, Q. Wang, R. Chang. "Random Laser Action in Semiconductor Powder." In: *Physical Review Letters* 82.11 (1999), pp. 2278–2281 (cit. on p. 110).

[28] B. C. Catanzaro, M. Garland, K. Keutzer. "Copperhead: compiling an embedded data parallel language." In: *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*. Edited by Calin Cascaval and Pen-Chung Yew. ACM, 2011, pp. 47–56 (cit. on p. 209).

[29] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, K. Olukotun. "A domain-specific approach to heterogeneous parallelism." In: *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*. Edited by Calin Cascaval and Pen-Chung Yew. ACM, 2011, pp. 35–46 (cit. on p. 205).

[30] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, V. Grover. "Accelerating Haskell array codes with multicore GPUs." In: *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*. Edited by Manuel Carro and John H. Reppy. ACM, 2011, pp. 3–14 (cit. on p. 207).

[31] D. Chang, A. H. Desoky, M. Ouyang, E. C. Rouchka. "Compute Pairwise Manhattan Distance and Pearson Correlation Coefficient of Data Points with GPU." In: *Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*. Edited by Haeng-Kon Kim and Roger Y. Lee. SNPD '09. Daegu, Korea: IEEE Computer Society, May 2009, pp. 501–506 (cit. on pp. 45, 46, 49).

[32] S. Chennupaty, P. Hammarlund, S. Jourdan. *Intel 4th Generation Core Processor (Haswell)*. Hot Chips: A Symposium on High Performance Chips. Intel. Aug. 2013 (cit. on p. 11).

[33] M. Christen, O. Schenk, H. Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures." In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. IEEE, 2011, pp. 676–687 (cit. on p. 210).

[34] P. Ciechanowicz, P. Kegel, M. Schellmann, S. Gorlatch, H. Kuchen. "Parallelizing the LM OSEM Image Reconstruction on Multi-Core Clusters." In: *Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo 2009, 1-4 September 2009, Lyon, France*. Edited by Barbara M. Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans J. Peters, and Thierry Priol. Vol. 19. Advances in Parallel Computing. IOS Press, 2009, pp. 169–176 (cit. on p. 24).

[35] P. Ciechanowicz, H. Kuchen. "Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures." In: *12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010, 1-3 September 2010, Melbourne, Australia*. IEEE, 2010, pp. 108–113 (cit. on pp. 23, 203).

[36] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991 (cit. on pp. 21, 36, 203).

[37] M. Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming." In: *Parallel Computing* 30.3 (2004), pp. 389–406 (cit. on p. 203).

[38] R. Cole, U. Vishkin. "Faster optimal parallel prefix sums and list ranking." In: *Information and Computation* 81.3 (1989), pp. 334–352 (cit. on p. 39).

[39] A. Collins, C. Fensch, H. Leather, M. Cole. "MaSiF: Machine learning guided auto-tuning of parallel skeletons." In: *20th Annual International Conference on High Performance Computing, HiPC 2013, Bengaluru (Bangalore), Karnataka, India, December 18-21, 2013*. IEEE Computer Society, 2013, pp. 186–195 (cit. on p. 201).

[40] A. Collins, D. Grewe, V. Grover, S. Lee, A. Susnea. "NOVA: A Functional Language for Data Parallelism." In: *ARRAY'14: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, United Kingdom, June 12-13, 2014*. Edited by Laurie J. Hendren, Alex Rubinsteyn, Mary Sheeran, and Jan Vitek. ACM, 2014, p. 8 (cit. on p. 209).

[41] K. D. Cooper, L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004 (cit. on p. 175).

[42] D. Coutts, R. Leshchinskiy, D. Stewart. "Stream Fusion: from Lists to Streams to Nothing at All." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP. Freiburg, Germany: ACM, 2007, pp. 315–326 (cit. on p. 153).

[43] *CUDA Basic Linear Algebra Subroutines (cuBLAS)*. Version 6.5. Nvidia. 2014. URL: http://developer.nvidia.com/cublas (cit. on pp. 87, 88, 129, 178, 186).

[44] *CUDA C Programming Guide*. v6.5. Nvidia. Aug. 2014. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide (cit. on pp. 6, 17, 68, 118, 142).

[45] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare. *Structured Programming*. New York, NY, USA: Academic Press, Feb. 11, 1972 (cit. on p. 20).

[46] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, J. S. Vetter. "The Scalable Heterogeneous Computing (SHOC) benchmark suite." In: *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*. Edited by David R. Kaeli and Miriam Leeser. Vol. 425. ACM International Conference Proceeding Series. ACM, 2010, pp. 63–74 (cit. on p. 188).

[47] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu. "Parallel Programming Using Skeleton Functions." In: *PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, June 14-17, 1993, Proceedings*. Edited by Arndt Bode, Mike Reeve, and Gottfried Wolf. Vol. 694. Lecture Notes in Computer Science. Springer, 1993, pp. 146–160 (cit. on pp. 24, 199, 200).

[48] U. Dastgeer, J. Enmyren, C. W. Kessler. "Auto-tuning SkePU: A Multi-backend Skeleton Programming Framework for multi-GPU Systems." In: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. IWMSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 25–32 (cit. on p. 204).

[49] U. Dastgeer, C. Kessler. "Smart Containers and Skeleton Programming for GPU-based Systems." In: *Proceedings 7th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2014)*. Amsterdam, Netherlands, July 2014 (cit. on p. 204).

[50] C. O. Daub, R. Steuer, J. Selbig, S. Kloska. "Estimating mutual information using B-spline functions – an improved similarity measure for analysing gene expression data." In: *BMC Bioinformatics* 5.1 (Aug. 2004), p. 118 (cit. on p. 49).

[51] J. Dean, S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. Edited by Eric A. Brewer and Peter Chen. USENIX Association, 2004, pp. 137–150 (cit. on p. 205).

[52] R. Dennard, V. Rideout, E. Bassous, A. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *Solid-State Circuits, IEEE Journal of* 9.5 (Oct. 1974), pp. 256–268 (cit. on p. 4).

[53] E. W. Dijkstra. "Letters to the editor: go to statement considered harmful." In: *Commun. ACM* 11.3 (1968), pp. 147–148 (cit. on p. 20).

[54] R. Dolbeau, S. Bihan, F. Bodin. "HMPP: A hybrid multi-core parallel programming environment." In: *Proceedings of the first Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*. 2007 (cit. on p. 208).

[55] J. Dongarra. "Basic Linear Algebra Subprograms Technical (Blast) Forum Standard (1)." In: *International Journal of High Performance Computing Applications* 16.1 (Feb. 2002), pp. 1–111 (cit. on pp. 40, 79).

[56] J. Dongarra. "Basic Linear Algebra Subprograms Technical (Blast) Forum Standard (2)." In: *International Journal of High Performance Computing Applications* 16.2 (May 2002), pp. 115–199 (cit. on pp. 40, 79).

[57] C. Dubach, P. Cheng, R. M. Rabbah, D. F. Bacon, S. J. Fink. "Compiling a high-level language for GPUs: (via language support for architectures and compilers)." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Edited by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 1–12 (cit. on p. 209).

[58] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, M. F. P. O'Boyle. "Portable compiler optimisation across embedded programs and microarchitectures using machine learning." In: *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*. Edited by David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez. ACM, 2009, pp. 78–88 (cit. on p. 200).

[59] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, H. Mössenböck. "Graal IR: An Extensible Declarative Intermediate Representation." In: *Proceedings of the 2nd Asia-Pacific Programming Languages and Compilers Workshop*. Shenzhen, China, Feb. 2013 (cit. on p. 205).

[60] J. Dünnweber, S. Gorlatch. *Higher-Order Components for Grid Programming - Making Grids More Usable*. Springer, 2009 (cit. on p. 203).

[61] J. Dünnweber, S. Gorlatch. "HOC-SA: A Grid Service Architecture for Higher-Order Components." In: *2004 IEEE International Conference on Services Computing (SCC 2004), 15-18 September 2004, Shanghai, China*. IEEE Computer Society, 2004, pp. 288–294 (cit. on p. 203).

[62] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas. "Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures." In: *Parallel Processing Letters* 21.2 (2011), pp. 173–193 (cit. on p. 208).

[63] V. K. Elangovan, R. M. Badia, E. A. Parra. "OmpSs-OpenCL Programming Model for Heterogeneous Systems." In: *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers*. Edited by Hironori Kasahara and Keiji Kimura. Vol. 7760. Lecture Notes in Computer Science. Springer, 2012, pp. 96–111 (cit. on p. 208).

[64] J. Enmyren, C. Kessler. "SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems." In: *Proceedings 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010)*. Baltimore, MD, USA (cit. on p. 204).

[65] S. Ernsting, H. Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters." In: *IJHPCN* 7.2 (2012), pp. 129–138 (cit. on p. 204).

[66] W. Fang, B. He, Q. Luo, N. K. Govindaraju. "Mars: Accelerating MapReduce with Graphics Processors." In: *IEEE Trans. Parallel Distrib. Syst.* 22.4 (2011), pp. 608–620 (cit. on p. 206).

[67] D. P. Friedman, D. S. Wise. "Aspects of Applicative Programming for Parallel Processing." In: *IEEE Transaction on Computers* 27.4 (1978), pp. 289–296 (cit. on p. 80).

[68] M. Friese. "Extending the Skeleton Library SkelCL with a Skeleton for Allpairs Computations." Masterthesis. Mar. 2013 (cit. on p. 70).

[69] J. J. Fumero, M. Steuwer, C. Dubach. "A Composable Array Function Interface for Heterogeneous Computing in Java." In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, June 2014, pp. 44–49 (cit. on p. 205).

[70] M. Garland, D. B. Kirk. "Understanding throughput-oriented architectures." In: *Commun. ACM* 53.11 (2010), pp. 58–66 (cit. on pp. 3, 11, 13, 14).

[71] *Girl (Lena, or Lenna)*. University of Southern California SIPI Image Database. URL: `http://sipi.usc.edu/database/database.php?volume=misc` (cit. on pp. 98, 101).

[72] H. González-Vélez, M. Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers." In: *Softw., Pract. Exper.* 40.12 (2010), pp. 1135–1160 (cit. on pp. 22, 203).

[73] S. Gorlatch. "From transformations to methodology in parallel program development: A case study." In: *Microprocessing and Microprogramming* 41.8-9 (1996), pp. 571–588 (cit. on p. 212).

[74] S. Gorlatch. "Send-receive considered harmful: Myths and realities of message passing." In: *ACM Trans. Program. Lang. Syst.* 26.1 (2004), pp. 47–56 (cit. on pp. 21, 212).

[75] S. Gorlatch. "Toward Formally-Based Design of Message Passing Programs." In: *IEEE Trans. Software Eng.* 26.3 (2000), pp. 276–288 (cit. on pp. 24, 212).

[76] S. Gorlatch, M. Cole. "Parallel Skeletons." In: *Encyclopedia of Parallel Computing*. Edited by David Padua. Berlin: Springer, 2011, pp. 1417–1422 (cit. on pp. 34, 36).

[77] S. Gorlatch, C. Wedler, C. Lengauer. "Optimization Rules for Programming with Collective Operations." In: *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings*. IEEE Computer Society, 1999, pp. 492–499 (cit. on p. 212).

[78] C. Grelck, S. Scholz. "SAC - A Functional Array Language for Efficient Multi-threaded Execution." In: *International Journal of Parallel Programming* 34.4 (2006), pp. 383–427 (cit. on p. 209).

[79] J. Guo, J. Thiyagalingam, S. Scholz. "Breaking the GPU programming barrier with the auto-parallelising SAC compiler." In: *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*. Edited by Manuel Carro and John H. Reppy. ACM, 2011, pp. 15–24 (cit. on p. 209).

[80] M. Haidl. "Simulation of Random Lasing on Modern Parallel Computer Systems." (in German). Diploma Thesis. July 2011 (cit. on p. 111).

[81] T. D. Han, T. S. Abdelrahman. "hiCUDA: High-Level GPGPU Programming." In: *IEEE Trans. Parallel Distrib. Syst.* 22.1 (2011), pp. 78–90 (cit. on p. 208).

[82] M. Harris. *Optimizing Parallel Reduction in CUDA*. Nvidia. 2007 (cit. on pp. 118, 119, 129, 132).

[83] M. Harris, S. Sengupta, J. D. Owens. "Parallel Prefix Sum (Scan) with CUDA." In: *GPU Gems 3*. Edited by Hubert Nguyen. Boston, MA, USA: Addison-Wesley Professional, 2007, pp. 851–876 (cit. on pp. 39, 60).

[84] Y. Hayashi, M. Cole. "Static performance prediction of skeletal parallel programs." In: *Parallel Algorithms Appl.* 17.1 (2002), pp. 59–84 (cit. on pp. 24, 199, 200).

[85] C. Hewitt, H. G. Baker. "Laws for Communicating Parallel Processes." In: *IFIP Congress*. 1977, pp. 987–992 (cit. on p. 80).

[86] P. Hijma, R. V. Nieuwpoort, C. J. H. Jacobs, H. E. Bal. "Stepwise-refinement for performance: a methodology for many-core programming." In: *Concurrency and Computation: Practice and Experience* (2015), pp. 1–40 (cit. on p. 207).

[87] L. Hochstein, V. R. Basili, U. Vishkin, J. Gilbert. "A pilot study to compare programming effort for two parallel programming models." In: *Journal of Systems and Software* 81.11 (2008), pp. 1920–1930 (cit. on p. 75).

[88] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. R. Basili. "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers." In: *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*. 2005, p. 35 (cit. on p. 75).

[89] S. S. Huang, A. Hormati, D. F. Bacon, R. M. Rabbah. "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary." In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. Edited by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 76–103 (cit. on p. 209).

[90] P. Hudak, J. Hughes, S. L. P. Jones, P. Wadler. "A history of Haskell: being lazy with class." In: *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. Edited by Barbara G. Ryder and Brent Hailpern. ACM, 2007, pp. 1–55 (cit. on p. 212).

[91] P. Hudak. "Report on the Programming Language Haskell, A Non-strict, Purely Functional Language." In: *SIGPLAN Notices* 27.5 (1992), p. 1 (cit. on p. 207).

[92] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel. Sept. 2014. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html (cit. on p. 13).

[93] *Intel Math Kernel Library (MKL)*. Version 11.2. Intel. 2014. URL: https://software.intel.com/en-us/intel-mkl (cit. on pp. 129, 186).

[94] *International Standard: Programming Languages – C*. ISO/IEC 9889:2011. ISO/IEC – JTC1 – SC22 – WG14. Dec. 8, 2011 (cit. on p. 16).

[95] X. Jiang, C. Soukoulis. "Time dependent theory for random lasers." In: *Physical review letters* 85.1 (2000), pp. 70–3 (cit. on p. 110).

[96] J. P. Jones. "SPMD cluster-based parallel 3D OSEM." In: *Nuclear Science Symposium Conference Record, 2002 IEEE*. IEEE, Nov. 2002, 1495–1499 vol.3 (cit. on p. 104).

[97] S. P. Jones, A. Tolmach, T. Hoare. "Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC." In: *2001 Haskell Workshop*. 2001 (cit. on pp. 153, 212).

[98] R. Karrenberg, S. Hack. "Whole-function vectorization." In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO. ACM, 2011, pp. 141–150 (cit. on pp. 174, 201).

[99] D. B. Kirk, W. W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2010 (cit. on pp. 87, 88).

[100] S. Knitter, M. Kues, M. Haidl, C. Fallnich. "Linearly polarized emission from random lasers with anisotropically amplifying media." In: *Optics Express* 21.25 (2013), pp. 31591–31603 (cit. on p. 111).

[101] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.)* Boston, MA, USA: Addison-Wesley, 1998 (cit. on p. 39).

[102] M. Köster, R. Leißa, S. Hack, R. Membarth, P. Slusallek. "Code Refinement of Stencil Codes." In: *Parallel Processing Letters* 24.3 (2014) (cit. on p. 211).

[103] H. Kuchen. "A Skeleton Library." In: *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings*. Edited by Burkhard Monien and Rainer Feldmann. Vol. 2400. Lecture Notes in Computer Science. Springer, 2002, pp. 620–629 (cit. on pp. 23, 203).

[104] R. Lämmel. "Google's MapReduce Programming Model – Revisited." In: *Science of Computer Programming* 68.3 (Oct. 2007), pp. 208–237 (cit. on pp. 49, 206).

[105] C. Lattner, V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–86 (cit. on p. 51).

[106] E. A. Lee. "The Problem with Threads." In: *IEEE Computer* 39.5 (2006), pp. 33–42 (cit. on p. 4).

[107] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, K. Olukotun. "Implementing Domain-Specific Languages for Heterogeneous Parallel Computing." In: *IEEE Micro* 31.5 (2011), pp. 42–53 (cit. on p. 205).

[108] M. Leyton, J. M. Piquer. "Skandium: Multi-core Programming with Algorithmic Skeletons." In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*. Edited by Marco Danelutto, Julien Bourgeois, and Tom Gross. IEEE Computer Society, 2010, pp. 289–296 (cit. on pp. 23, 203).

[109] E. Lindholm, J. Nickolls, S. F. Oberman, J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture." In: *IEEE Micro* 28.2 (2008), pp. 39–55 (cit. on p. 119).

[110] T. Lutz, C. Fensch, M. Cole. "PARTANS: An autotuning framework for stencil computation on multi-GPU systems." In: *TACO* 9.4 (2013), p. 59 (cit. on p. 211).

[111] B. B. Mandelbrot. "Fractal Aspects of the Iteration of $z \mapsto \lambda z(1-z)$ for Complex λ and z." In: *Annals of the New York Academy of Sciences* 357.1 (1980), pp. 249–259 (cit. on p. 76).

[112] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, Y. Akashi. "A Fusion-Embedded Skeleton Library." In: *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*. Edited by Marco Danelutto, Marco Vanneschi, and Domenico Laforenza. Vol. 3149. Lecture Notes in Computer Science. Springer, 2004, pp. 644–653 (cit. on p. 23).

[113] M. McCool, A. D. Robinson, J. Reinders. *Structured Parallel Programming. Patterns for Efficient Computation*. Boston, MA, USA: Morgan Kaufmann, 2012 (cit. on pp. 21, 206).

[114] T. L. McDonell, M. M. T. Chakravarty, G. Keller, B. Lippmeier. "Optimising purely functional GPU programs." In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Edited by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 49–60 (cit. on p. 207).

[115] F. Mesmay, A. Rimmel, Y. Voronenko, M. Püschel. "Bandit-based optimization on graphs with application to library performance tuning." In: *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. Edited by Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman. Vol. 382. ACM International Conference Proceeding Series. ACM, 2009, pp. 729–736 (cit. on p. 180).

[116] G. E. Moore. "Cramming more components onto integrated circuits." In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85 (cit. on p. 4).

[117] *MPI: A Message Passing Interface Standard*. Version 3.0. Message Passing Interface Forum. Sept. 21, 2012 (cit. on p. 203).

[118] S. Muchnick. *Advanced Compiler Design and Implementation*. San Fransisco, CA, USA: Morgan Kaufmann, Aug. 15, 1997 (cit. on p. 201).

[119] Aaftab Munshi, ed. *The OpenCL Specification*. Version 1.2, Document Revision 19. Khronos OpenCL Working Group. Nov. 2012 (cit. on pp. 6, 17, 35).

[120] M. Nixon, A. S. Aguado. *Feature Extraction & Image Processing for Computer Vision*. 3rd. Academic Press, Aug. 2012 (cit. on p. 65).

[121] *Nvidia CUDA Toolkit*. Nvidia. Apr. 2015 (cit. on p. 189).

[122] M. Odersky. "The Scala experiment: can we provide better language support for component systems?" In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Edited by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 166–167 (cit. on p. 205).

[123] M. Odersky, T. Rompf. "Unifying functional and object-oriented programming with Scala." In: *Commun. ACM* 57.4 (2014), pp. 76–86 (cit. on p. 205).

[124] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, M. Püschel. "Spiral in scala: towards the systematic construction of generators for performance libraries." In: *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*. Edited by Jaakko Järvi and Christian Kästner. ACM, 2013, pp. 125–134 (cit. on p. 212).

[125] K. Olukotun, L. Hammond. "The future of microprocessors." In: *ACM Queue* 3.7 (2005), pp. 26–29 (cit. on p. 3).

[126] *OpenCL Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor*. Intel. Jan. 2014. URL: https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor (cit. on pp. 118, 142).

[127] *OpenMP Application Program Interface*. Version 4.0. OpenMP Architecture Review Board. July 2013 (cit. on pp. 16, 203, 208).

[128] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, S. P. Amarasinghe. "Portable performance on heterogeneous architectures." In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. Edited by Vivek Sarkar and Rastislav Bodík. ACM, 2013, pp. 431–444 (cit. on p. 210).

[129] *Pthreads: POSIX. 1c, Threads Extension*. IEEE Std 1003.1c-1995. IEEE. 1995 (cit. on p. 16).

[130] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. R. Johnson, D. A. Padua, M. M. Veloso, R. W. Johnson. "Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms." In: *IJHPCA* 18.1 (2004), pp. 21–45 (cit. on p. 212).

[131] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, F. Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." In: *ACM Trans. Graph.* 31.4 (2012), p. 32 (cit. on p. 211).

[132] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. P. Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Edited by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 519–530 (cit. on p. 211).

[133] A. J. Reader, K. Erlandsson, M. A. Flower, R. J. Ott. "Fast accurate iterative reconstruction for low-statistics positron volume imaging." In: *Physics in Medicine and Biology* 43.4 (1998), p. 835 (cit. on pp. 27, 28, 103).

[134] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007 (cit. on p. 206).

[135] T. Rompf, M. Odersky. "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs." In: *Commun. ACM* 55.6 (2012), pp. 121–130 (cit. on p. 205).

[136] A. Sarje, S. Aluru. "All-pairs Computations on Many-core Graphics Processors." In: *Parallel Computing* 39.2 (Feb. 2013), pp. 79–93 (cit. on pp. 69, 70).

[137] M. Schellmann, S. Gorlatch, D. Meiländer, T. Kösters, K. Schäfers, F. Wübbeling, M. Burger. "Parallel Medical Image Reconstruction: From Graphics Processors to Grids." In: *Proceedings of the 10th International Conference on Parallel Computing Technologies*. PaCT '09. Novosibirsk, Russia: Springer, 2009, pp. 457–473 (cit. on pp. 27, 31, 103, 106).

[138] P. Sebbah, C. Vanneste. "Random laser in the localized regime." In: *Physical Review B* 66.14 (2002), pp. 1–10 (cit. on p. 110).

[139] R. L. Siddon. "Fast calculation of the exact radiological path for a three-dimensional CT array." In: *Medical Physics* 12.2 (1985), pp. 252–255 (cit. on p. 28).

[140] D. G. Spampinato, M. Püschel. "A Basic Linear Algebra Compiler." In: *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, Orlando, FL, USA, February 15-19, 2014*. Edited by David R. Kaeli and Tipp Moseley. ACM, 2014, p. 23 (cit. on p. 212).

[141] *Standard for Programming Language C++*. ISO/IEC 14882:2014. ISO/IEC – JTC1 – SC22 – WG21. Dec. 15, 2014 (cit. on p. 16).

[142] A. Stegmeier, M. Frieb, R. Jahr, T. Ungerer. "Algorithmic Skeletons for Parallelization of Embedded Real-time Systems." In: *Proceedings of the 3rd Workshop on High-performance and Real-time Embedded Systems*. HiRES 2015. Amsterdam, Netherlands, Jan. 2015 (cit. on pp. 24, 199).

[143] M. Steuwer, C. Fensch, S. Lindley, C. Dubach. "Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Patterns to High-Performance OpenCL Code." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP. accepted for publication. Vancouver, Canada: ACM, 2015 (cit. on pp. 138, 142).

[144] J. A. Stuart, J. D. Owens. "Multi-GPU MapReduce on GPU Clusters." In: *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. IEEE, 2011, pp. 1068–1079 (cit. on p. 206).

[145] H. Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." In: *Dr. Dobb's Journal* 30.3 (Mar. 2005) (cit. on pp. 4, 5).

[146] J. Svensson, K. Claessen, M. Sheeran. "GPGPU kernel implementation and refinement using Obsidian." In: *Proceedings of the International Conference on Computational Science, ICCS 2010, University of Amsterdam, The Netherlands, May 31 - June 2, 2010*. Edited by Peter M. A. Sloot, G. Dick van Albada, and Jack Dongarra. Vol. 1. Procedia Computer Science 1. Elsevier, 2010, pp. 2065–2074 (cit. on p. 207).

[147] J. Svensson, M. Sheeran, K. Claessen. "Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors." In: *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*. Edited by Sven-Bodo Scholz and Olaf Chitil. Vol. 5836. Lecture Notes in Computer Science. Springer, 2008, pp. 156–173 (cit. on p. 207).

[148] M. Takeuchi, Y. Makino, K. Kawachiya, H. Horii, T. Suzumura, T. Suganuma, T. Onodera. "Compiling X10 to Java." In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. X10 '11. San Jose, California: ACM, 2011, 3:1–3:10 (cit. on p. 209).

[149] *The OpenACC™ Application Programming Interface*. Version 2.0. OpenACC. June 2013 (cit. on pp. 16, 208).

[150] *The OpenCL BLAS library (clBLAS)*. Version 2.2.0. AMD. 2014. URL: http://github.com/clMathLibraries/clBLAS (cit. on pp. 129, 185).

[151] *Tuning CUDA Applications for Kepler*. Nvidia (cit. on p. 16).

[152] S. E. Umbaugh. *Computer Vision and Image Processing*. New Jersey, USA: Prentice Hall PTR, 1997 (cit. on pp. 44, 94).

[153] S. P. Vanderwiel, D. Nathanson, D. J. Lilja. "Complexity and Performance in Parallel Programming Languages." In: *1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97), 1 April 1997, Geneva, Switzerland*. 1997, p. 3 (cit. on p. 75).

[154] P. Wadler. "Deforestation: Transforming Programs to Eliminate Trees." In: *Theor. Comput. Sci.* 73.2 (1990), pp. 231–248 (cit. on p. 207).

[155] *White Paper – AMD Graphics Cores Next (GCN) Architecture*. AMD. June 2012 (cit. on pp. 129, 131).

[156] *White Paper – First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*. Intel. 2008 (cit. on p. 129).

[157] *Whitepaper – NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. v1.1. Nvidia. 2009 (cit. on pp. 119, 129).

[158] *Whitepaper – NVIDIA's Next Generation CUDA Compute Architecture: Kepler*. v1.1. Nvidia. 2012 (cit. on p. 13).

[159] A. Yamilov, X. Wu, H. Cao, A. Burin. "Absorption-induced confinement of lasing modes in diffusive random media." In: *Optics Letters* 30.18 (2005), pp. 2430–2432 (cit. on p. 110).

[160] K. Yee. "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media." In: *IEEE Transactions on Antennas and Propagation* (1966) (cit. on p. 110).

[161] Y. Zhang, F. Mueller. "Hidp: A hierarchical data parallel language." In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 2013, pp. 1–11 (cit. on p. 209).

LEBENSLAUF

ZUR PERSON

geboren am 21.05.1985 in Duisburg	
Familienstand:	ledig
Nationalität:	deutsch
Name des Vaters:	Peter Steuwer, geb. Kremer
Name der Mutter:	Bärbel Steuwer

SCHULBILDUNG

09/1991 – 07/1995	Grundschule, Duisburg
08/1995 – 06/2004	Gesamtschule, Dinslaken
	Hochschulreife (Abitur) am 24.06.2004

STUDIUM

10/2005 – 09/2010	Diplomstudiengang Informatik mit Nebenfach Mathematik, Westfälische Wilhelms-Universität Münster
	Diplomprüfung Informatik am 13.09.2010
seit 10/2010	Promotionsstudiengang Informatik, Westfälische Wilhelms-Universität Münster

TÄTIGKEITEN

07/2004 – 04/2005	Zivildienst, Dinslaken
03/2008 – 02/2010	Studentische Hilfskraft, Westfälische Wilhelms-Universität Münster
10/2010 – 09/2014	Wissenschaftlicher Mitarbeiter, Westfälische Wilhelms-Universität Münster
seit 10/2014	Research Associate, The University of Edinburgh

BEGINN DER DISSERTATION

seit 10/2010	Institut für Informatik, Westfälische Wilhelms-Universität Münster
	betreut durch Prof. Dr. Sergei Gorlatch

COLOPHON

This document was typeset using the LaTeX 2_ε typesetting system originally developed by Leslie Lamport, based on TeX created by Donald Knuth. For the typographical look-and-feel the `classicthesis` style developed by André Miede was used. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". The Fonts used in this document are Palatino by Hermann Zapf, `DejaVu Sans Mono` by Štěpán Roh and others, and AMS Euler by Hermann Zapf and Donald Knuth.