THE UNIVERSITY *of* EDINBURGH

MAPPING PARALLEL PROGRAMS TO HETEROGENEOUS MULTI-CORE SYSTEMS

DOMINIK GREWE

Doctor of Philosophy
School of Informatics
University of Edinburgh
2013

ABSTRACT

Heterogeneous computer systems are ubiquitous in all areas of computing, from mobile to high-performance computing. They promise to deliver increased performance at lower energy cost than purely homogeneous, CPU-based systems. In recent years GPU-based heterogeneous systems have become increasingly popular. They combine a programmable GPU with a multi-core CPU. GPUs have become flexible enough to not only handle graphics workloads but also various kinds of general-purpose algorithms. They are thus used as a coprocessor or accelerator alongside the CPU.

Developing applications for GPU-based heterogeneous systems involves several challenges. Firstly, not all algorithms are equally suited for GPU computing. It is thus important to carefully map the tasks of an application to the most suitable processor in a system. Secondly, current frameworks for heterogeneous computing, such as OpenCL, are low-level, requiring a thorough understanding of the hardware by the programmer. This high barrier to entry could be lowered by automatically generating and tuning this code from a high-level and thus more user-friendly programming language. Both challenges are addressed in this thesis.

For the task mapping problem a machine learning-based approach is presented in this thesis. It combines static features of the program code with runtime information on input sizes to predict the optimal mapping of OpenCL kernels. This approach is further extended to also take contention on the GPU into account. Both methods are able to outperform competing mapping approaches by a significant margin.

Furthermore, this thesis develops a method for targeting GPU-based heterogeneous systems from OpenMP, a directive-based framework for parallel computing. OpenMP programs are translated to OpenCL and optimized for GPU performance. At runtime a predictive model decides whether to execute the original OpenMP code on the CPU or the generated OpenCL code on the GPU. This approach is shown to outperform both a competing approach as well as hand-tuned code.

ACKNOWLEDGEMENTS

First of all, I would like to thank my academic adviser, Professor Michael O'Boyle, for his support over the past years. His advice and his guidance have been invaluable.

I would further like to thank my friends and colleagues in the CArD research group for the inspiring discussions and generous support. In particular, I would like to thank Alberto, Chris, Christophe, Hugh and Zheng.

Thanks a lot to Anton, for a great time at ARM, and to Vinod and Sean, for three insightful months at NVIDIA.

Finally, I would like to thank my family for their unconditional support and, of course, Hannah who has always been there for me.

DECLARATION

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the material used in this thesis has been published in the following papers:

- Dominik Grewe, Zheng Wang and Michael F.P. O'Boyle. "A Workload-Aware Mapping Approach For Data-Parallel Programs". In *6th International Conference on High-Performance and Embedded Architectures and Compilers*, January 2011.
- Dominik Grewe and Anton Lokhmotov. "Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation". In *4th Workshop on General Purpose Processing on Graphics Processing Units*, March 2011.
- Dominik Grewe and Michael F.P. O'Boyle. "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL". In *20th International Conference on Compiler Construction*, March 2011.
- Dominik Grewe, Zheng Wang and Michael F.P. O'Boyle. "Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems". In *International Symposium on Code Generation and Optimization*, February 2013.
- Dominik Grewe, Zheng Wang and Michael F.P. O'Boyle. "OpenCL Task Partitioning in the Presence of GPU Contention". In *26th International Workshop on Languages and Compilers for Parallel Computing*, September 2013.

Dominik Grewe, December 15, 2013

CONTENTS

1 INTRODUCTION 1
1.1 Two Challenges of GPU-based Heterogeneous Computing 2
1.2 Contributions 5
1.3 Thesis Outline 6
2 BACKGROUND 9
2.1 Heterogeneous Systems 9
2.1.1 Graphics Processing Units 10
2.1.2 Comparison of CPU to GPU architectures 11
2.1.3 Discrete vs. Integrated GPUs 12
2.2 Parallel Programming Languages and Frameworks 13
2.2.1 OpenMP 13
2.2.2 OpenCL 14
2.3 Machine Learning 17
2.3.1 Terminology 18
2.3.2 Data Preprocessing 19
2.3.3 Classification Algorithms 21
2.4 Machine Learning in Program Optimization 24
2.4.1 Program Characterization 25
2.4.2 Putting it all together 27
2.5 Evaluation Methodology 27
2.5.1 Cross-Validation 28
2.5.2 Relative Performance 29
2.6 Summary 30
3 RELATED WORK 31
3.1 Task Scheduling on Heterogeneous Systems 31
3.1.1 Static Scheduling 32
3.1.2 Dynamic Scheduling 34
3.2 Task Mapping on GPU-based Heterogeneous Systems 35
3.2.1 Scheduling Tasks to Devices 36
3.2.2 Mapping Tasks Across Devices 40
3.3 Optimising GPGPU programs 43
3.3.1 Kernel Code Optimizations 44
3.3.2 Automatic Data Management 47
3.4 Mapping High-Level Languages to Heterogeneous Systems 48
3.4.1 C-Based Languages 48
3.4.2 Streaming Languages 49
3.4.3 Pattern-Based Languages 50

3.4.4 Directive-Based Languages for GPU Programming 51
3.5 Machine Learning in Program Optimization 52
3.5.1 Compiler Optimization 53
3.5.2 Mapping Parallelism to Multi-Core Systems 54
3.6 Summary 55
4 A STATIC TASK PARTITIONING APPROACH FOR HETEROGENEOUS SYSTEMS USING OPENCL 57
4.1 Introduction 57
4.2 Motivation 58
4.3 Partitioning Data-Parallel Tasks 59
4.3.1 Static Code Features 60
4.3.2 Building the Predictor 62
4.3.3 Deployment 64
4.4 Methodology 65
4.4.1 Experimental Setup 65
4.4.2 Evaluation Methodology 66
4.5 Results 66
4.5.1 Speedup over Single-core Performance 69
4.5.2 Prediction Accuracy 73
4.6 Summary 74
5 TASK PARTITIONING IN THE PRESENCE OF GPU CONTENTION 75
5.1 Introduction 75
5.2 Motivation 76
5.2.1 Program Behaviour in the Presence of GPU contention 76
5.2.2 Dynamic Partitioning Schemes 77
5.3 OpenCL Applications in the Presence of Contention 79
5.4 A Predictive Model for OpenCL Task Partitioning 82
5.4.1 The Features of the Model 82
5.4.2 Collecting Training Data 83
5.4.3 Building the Model 84
5.4.4 Deployment of the Model 84
5.5 Experimental Methodology 85
5.5.1 Experimental Setup 85
5.5.2 Evaluation Methodology 87
5.6 Results 88
5.6.1 Comparison to Oracle 88
5.6.2 Comparison to Dynamic Mapping Schemes 91
5.7 Summary 93
6 PORTABLE MAPPING OF DATA PARALLEL PROGRAMS TO OPENCL FOR HETEROGENEOUS SYSTEMS 97
6.1 Introduction 97

6.2 Motivation . . . 99
6.3 Overall Scheme . . . 99
6.4 Code Generation and Optimization . . . 102
6.4.1 OpenCL Code Optimization . . . 102
6.5 Predicting the Mapping . . . 104
6.5.1 Training the Predictor . . . 105
6.5.2 Code Features . . . 105
6.5.3 Runtime Deployment . . . 106
6.6 A Model for Dynamic Index Reordering . . . 106
6.7 Experimental Methodology . . . 107
6.7.1 Experimental Setup . . . 107
6.7.2 Evaluation Methodology . . . 108
6.8 Experimental Results . . . 108
6.8.1 Performance Evaluation . . . 109
6.8.2 Comparison to State-of-the-Art . . . 111
6.8.3 OpenCL Code Performance Comparison . . . 112
6.8.4 Analysis of Predictive Model . . . 113
6.8.5 Performance on Integrated Systems . . . 115
6.9 Evaluation of Dynamic Index Reordering . . . 116
6.9.1 Performance Impact . . . 116
6.9.2 Analysis of Predictive Model . . . 118
6.10 Summary . . . 119
7 CONCLUSION 121
7.1 Contributions . . . 121
7.1.1 Task Mapping . . . 121
7.1.2 Code Generation and Tuning . . . 122
7.2 Critical Analysis . . . 123
7.2.1 Machine Learning-Based Task Mapping . . . 123
7.2.2 Training Data . . . 124
7.2.3 Static Code Features . . . 124
7.2.4 Data Transformations . . . 125
7.2.5 Choice of High-Level Language . . . 125
7.3 Future Work . . . 126
7.3.1 Different Optimization Metrics . . . 126
7.3.2 Hybrid Task Mapping . . . 126
7.3.3 Mapping Programs with Multiple Kernels . . . 127
7.4 Summary . . . 127
A BENCHMARKS USED IN CHAPTER 4 129
BIBLIOGRAPHY 139

1 INTRODUCTION

The majority of computer systems are heterogeneous, i. e. they contain a number of processing units with different characteristics. Typically a *general-purpose* CPU is combined with a number of *special-purpose* hardware units. System-on-Chips (SoCs) for embedded systems such as mobile phones are heterogeneous systems, for example. They have a CPU for executing the operating system and user applications, but they also contain special-function hardware for common tasks such as audio and video processing.

Heterogeneous systems have also been used in high-performance computing (HPC). Many HPC workloads are highly parallel, e. g. physics simulation, as opposed to desktop software which is often sequential. CPUs are therefore combined with coprocessors (or accelerators) which have been specifically designed for such parallel workloads. Examples of these coprocessors are vector or SIMD processors designed by companies such as Fujitsu or ClearSpeed. Because these processors only serve a small market they are expensive, which explains their limited use in today's HPC systems.

Desktop and laptop computers also comprise different types of processors and are thus heterogeneous systems. In addition to a CPU, these systems contain, for example, hardware for graphics acceleration. These graphics accelerators, also known as graphics processing units or GPUs, have evolved from pure fixed-function hardware units to fully programmable processors in order to handle ever more complex graphics tasks, e. g. in computer games. This development initiated the era of GPGPU (general-purpose computing on GPUs) where GPUs are used as coprocessors for non-graphics workloads. As section 2.1.1 shows, GPUs are highly parallel processors which are a good match for many HPC workloads. Because GPUs were already ubiquitous, it was easy for programmers to see if their applications could benefit from GPU acceleration. Furthermore, the market for GPUs is immense[1], making them a much cheaper alternative to coprocessors specifically designed for the HPC market. For these reasons GPU-based, heterogeneous systems are now commonplace in high-performance computing. In addition, GPUs are also increasingly used as coprocessors for non-graphics applications in desktop and mobile computing.

Developing software for heterogeneous systems is challenging. Mainstream programming languages typically only target the CPU. Additional libraries or programming frameworks are required to utilize other types of processors. This constitutes a

1 More than 100 million GPUs have been shipped in the first quarter of 2013 alone (Jon Peddie Research, 2013)

barrier for many programmers. Furthermore, the programmer has to decide which parts of a program to execute on which processors.

1.1 TWO CHALLENGES OF GPU-BASED HETEROGENEOUS COMPUTING

Heterogeneous systems based on CPUs and GPUs are typically programmed in low-level languages such as OpenCL or CUDA. The programmer writes *kernel functions* representing the computational tasks of an application. When using CUDA, kernel functions are always executed on the GPU. In OpenCL, however, they can be executed on both the CPU and the GPU. It is up to the programmer to map the tasks to devices, i. e. deciding which task should be executed on which device. This constitutes the first challenge of heterogeneous computing: task mapping.

The second one is the challenge of code generation and tuning. Both CUDA and OpenCL require the programmer to re-write programs to make them fit with the corresponding model of computation. Furthermore, the languages are very low-level, requiring a thorough understanding of the targeted platforms in order to achieve good performance.

The two challenges are now discussed in more detail.

Challenge 1: Task Mapping

On heterogeneous systems, programs need to be mapped to the devices of the systems. In this thesis, programs are mapped with the goal of improving performance, i. e. minimizing running time, but other metrics, such as power consumption, can be used too. Mapping programs typically involves partitioning the program into tasks and then scheduling the tasks to the devices. In OpenCL, the programmer partitions the program by identifying individual tasks. Each task can then be scheduled to a device. However, tasks in OpenCL are data-parallel tasks with only limited dependencies (see section 2.2.2). This means that a task can be further partitioned into sub-tasks that are individually scheduled to the different devices. In this case, task mapping is about deciding how to partition the workload of a task between the available processors.

Many factors need to be considered when making mapping decisions. First, there is device suitability: which device can process a given task faster. Data locality can also be important because on many platforms the devices have separate memory spaces. If the data is not in the chosen device's memory it needs to be copied there, which can substantially add to the overall execution time. Another factor is the input size. GPUs typically require large amounts of work to fully utilize all their resources. So even though a task may be principally suited for GPU execution, if the input to the task is not big enough, the CPU may be faster. Other factors include the maturity of

the compiler (What is the quality of the generated code for a specific processor?) and the availability of resources (Does the program have to share resources with other applications?).

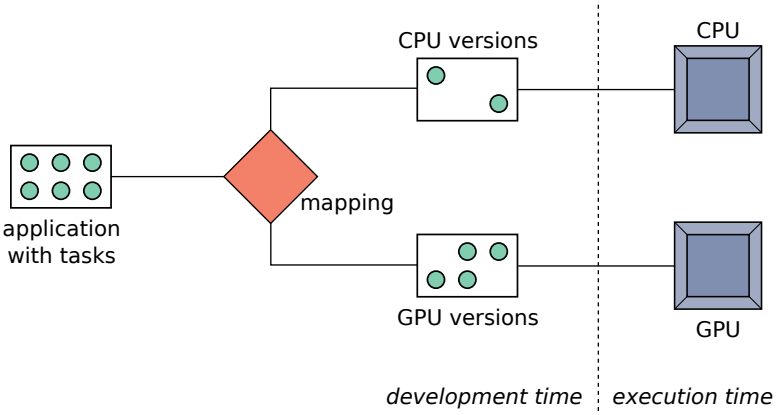
Task mapping typically happens at one of two stages of application development. A common approach is to schedule tasks as soon as they are identified. Each task's suitability for the different processors is considered using the programmer's knowledge of the hardware. Together with information about data locality, a decision is made whether this task should be executed on the CPU or the GPU. Now that the tasks are scheduled, one version of each task can be written and tuned for the chosen device. At run time the tasks are executed on their designated device. This is depicted in figure 1.1a. The main advantage of this approach is that it reduces development time, because only one version needs to be developed for each task. However, the mapping cannot be easily adapted, e. g. to different hardware or varying resource availability. Furthermore, there is no way to actually check that the chosen mapping is optimal.

A more flexible approach is to only make the scheduling decision at run time, as shown in figure 1.1b. This way, factors such as the input size or device suitability can also be taken into account. The main drawback of this approach is that it either requires multiple code versions for each tasks or, if a single version is used for all devices, to potentially accept suboptimal performance. Furthermore, it requires the programmer to implement some logic that makes the scheduling decisions. All of this increases the effort needed on the programmer's side.

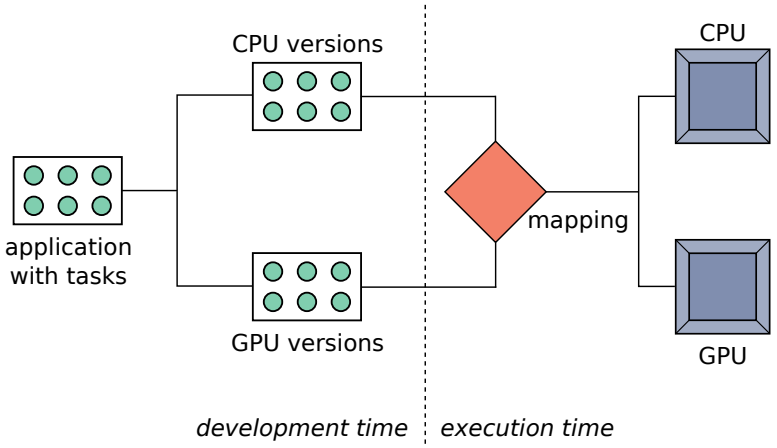
What is desirable is an approach that *automatically* makes mapping decisions for the user at run time, taking all the factors mentioned above into account. It should be *portable* across different systems so that no effort from the programmer is needed when running the program on a different system. Such an approach can either be based on OpenCL, so that the programmer only has to provide a single version of each task that is run on all devices, or it is implemented as part of a compiler for a higher-level language that automatically generates device-specific code. This way, the programmer does not have to use low-level languages such as OpenCL or CUDA at all. This approach is discussed in the next section.

Challenge 2: Code Generation and Tuning

Programming languages for heterogeneous computing, such as OpenCL or CUDA, are very low-level and require careful tuning of the code, especially when targeting the GPU. This constitutes a high barrier to entry for heterogeneous computing because programmers must not only learn a new language, but must also have a thorough understanding of the underlying hardware in order to reap the benefits of heterogeneous computing. Automatically generating and tuning this low-level code



(a) task mapping at development time



(b) task mapping at execution time

Figure 1.1: Task mapping at two different stages of development. If deciding the mapping at development time only one version of each task needs to be generated but the mapping is fixed (a). A more flexible mapping can be achieved by deferring the mapping decision to execution time (b).

from a *high-level* language is a way to lower this barrier. The programmer can use a (more familiar) high-level language and does not have to worry about the characteristics of the hardware. This burden now falls to the compiler and runtime system.

The choice of the high-level language is important. For the programmer, it has to be easy to use and flexible enough to express a wide range of problems. Ideally it is an extension of a popular programming language so that existing applications can be updated easily. For the compiler writer, it is important that the language provides enough high-level information to be able to generate efficient code. When generating code for GPUs, it has to be carefully tuned in order to get any benefits over CPU execution.

An example of a language with high programmer familiarity is OpenACC (OpenACC, 2013). It allows the programmer to annotate existing programs with regions that are to be accelerated on the GPU. While it significantly eases programming, OpenACC mainly does away with writing boilerplate code. It still requires the user to have an understanding of how GPUs work and carefully tune for performance. At the other end of the spectrum, more abstract languages, e. g. functional programming languages such as Haskell (Chakravarty et al., 2011), are used to target GPUs. Here, the user is completely unaware of the targeted platform and relies on the compiler and runtime for high performance. However, these languages are unfamiliar to many programmers and often restrict them to certain styles of programming. This can have a negative impact on performance if an algorithm cannot be expressed in the optimal way and the compiler is unable to automatically transform the code. A balance between the two should be found.

1.2 CONTRIBUTIONS

This thesis provides solutions to the two challenges described above. Two task mapping approaches are presented, both relying on machine learning methods. The approaches predict an optimal *static* mapping for OpenCL kernels in the sense that the work is split in exactly two pieces, one for the CPU and one for the GPU. The decision is only made at run time, however, taking dynamic features such as the input size and resource availability into account. They provide a flexible way of mapping OpenCL programs onto heterogeneous systems without any involvement by the programmer.

The solution for the code generation and tuning problem uses OpenMP as a high-level language, from which OpenCL code is generated. A task mapping method is also included to decide on the optimal device for an application.

The following list briefly summarizes the main contributions of this thesis:

- Firstly, a static task partitioning scheme for OpenCL kernels is proposed. Using machine learning techniques, a predictive model is developed that accurately determines how to partition the work between the CPU and the GPU. This decision is based on static code features extracted from the OpenCL kernel code. Furthermore, data transfer costs are taken into account. Across a number of benchmarks, this method outperforms competing approaches such as a dynamic task farm mapper. This work is presented in chapter 4.

- Secondly, the partitioning model is extended to explicitly take the impact of resource contention on the GPU into account. It is shown that current approaches fail to adapt to GPU contention due to the fact that kernels have exclusive access to GPUs and cannot be preempted. The proposed approach explicitly takes contention into account when making mapping decisions and is thus able to adapt

to it. It outperforms dynamic approaches when evaluated over a range of benchmarks and contention scenarios. Chapter 5 provides a detailed discussion.

- Thirdly, this thesis proposes an approach for targeting heterogeneous systems from OpenMP programs. OpenMP provides an easy path to upgrade existing sequential applications to make use of parallel hardware. The proposed approach translates OpenMP programs into OpenCL programs targeted at GPUs, applying a number of transformations to improve performance. Rather than always using the OpenCL version, it incorporates a model that predicts at run time whether the OpenMP version for CPUs or the OpenCL version for GPUs will be faster on a given system. Across a range of benchmarks and systems, it outperforms a competing approach and even hand-tuned versions of the benchmarks. Chapter 6 introduces this approach in more detail.

1.3 THESIS OUTLINE

The remainder of this thesis is structured as follows:

CHAPTER 2 provides information on heterogeneous systems and how to program them. It further introduces basic machine learning concepts and how machine learning techniques are used for program optimization. The chapter also discusses how these approaches are typically evaluated.

CHAPTER 3 presents related work. Firstly, it discusses work on task mapping on heterogeneous systems. This is followed by an overview of optimization techniques for GPUs and approaches for mapping high-level languages to heterogeneous systems. The chapter also discusses prior work on machine learning-based program optimization.

CHAPTER 4 develops a static task partitioning scheme for OpenCL. It shows how statically extracted code features can be used to determine the optimal mapping of OpenCL kernels between the CPU and the GPU. The approach is evaluated on a large set of benchmarks and compared with alternative mapping schemes. This chapter is based on the work published in (Grewe and O'Boyle, 2011).

CHAPTER 5 looks into task mapping in the presence of contention on the GPU. It presents a machine-learning based approach that explicitly uses information about GPU contention to adapt to co-running applications using the GPU. On a set of benchmarks and on varying contention scenarios, this method is shown to outperform other mapping approaches.

CHAPTER 6 proposes a portable, compiler-based approach for mapping OpenMP programs to heterogeneous systems. It describes several transformations, notably data layout transformations, that help improve performance when targeting the GPU. It further describes a machine-learning based model that predicts whether a program should be executed on the CPU or the GPU. On a full suite of benchmarks, this approach is shown to outperform a competing approach as well as hand-written code. This chapter is based on the work published in (Grewe et al., 2013).

CHAPTER 7 concludes this thesis with a summary of the main contributions of this work. It provides a critical analysis of some of the technical aspects and discusses ideas for potential future work.

2

BACKGROUND

This chapter provides the background knowledge necessary to understand the details of the contributions described in the remainder of this thesis. Key concepts and terminology are introduced for, among other things, heterogeneous computing and machine learning.

The chapter starts with an introduction to GPU-based heterogeneous systems in section 2.1. This is followed in section 2.2 by a description of parallel programming languages and frameworks with a focus on OpenCL, a standardized framework for heterogeneous programming. Section 2.3 provides a concise introduction to basic machine learning principles and the use of machine learning in program optimization is discussed in section 2.4. Section 2.5 describes common techniques used to evaluate the contributions of this thesis. The chapter concludes with a summary in section 2.6.

2.1 HETEROGENEOUS SYSTEMS

The term *heterogeneous systems* defines computer systems comprising multiple processors with different capabilities and architectures. A wide array of heterogeneous systems can be found in today's computers, varying in the number of processor types and the degree of heterogeneity.

At one end of the spectrum are single-ISA systems comprising multiple CPU cores. The cores may run at different frequencies and often implement different architectural choices, e.g. pipeline depth, in-order vs. out-of-order. These are general-purpose processors but they vary in both performance and power efficiency. A recent example of these types of heterogeneous systems is ARM's big.LITTLE architecture (Greenhalgh, 2011). It combines high-performance oriented processors, e. g. ARM Cortex-A15, with energy efficiency oriented processors, e. g. ARM Cortex-A7, on the same chip.

At the other end of the spectrum are systems with a mixture of both general-purpose CPUs as well as highly specialized processors. While the CPU executes the operating system and user applications, there are various processors for specific tasks such as audio and video processing or cryptography. System-on-Chips, as found in mobile phones, are typical instances of this type of heterogeneous system.

In this thesis, the focus is on CPU-GPU systems, i. e. systems comprising a general-purpose, multi-core CPU as well as a programmable GPU. GPUs have evolved from being a special-purpose processor purely for graphics tasks, to programmable accelerators that can run a wide range of applications. While it has become more flexible,

the GPU's architecture is still distinctly different from the CPU's, as the next sections will show.

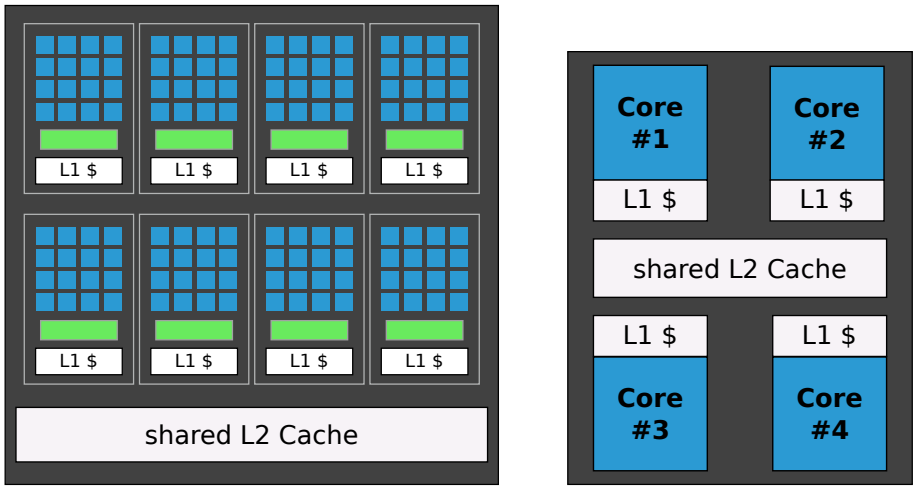
2.1.1 *Graphics Processing Units*

Graphics processing units or GPUs have evolved immensely since the first generation of graphics accelerators. Early graphics cards comprised fixed-function hardware units that implemented a fixed set of operations. Over the years, the hardware has become more flexible in order to allow programmers to implement increasingly complex graphical effects. The latest generations of GPUs have become so flexible that they can execute algorithms from a whole range of application areas, some of them in no way related to graphics. This style of computing has become known as *general-purpose computing on* GPUs or GPGPU.

Figure 2.1a shows a simplified diagram of a typical GPU, based on the NVIDIA Fermi architecture (NVIDIA, 2009). GPUs comprise a large number of simple *processing cores* divided into groups. In NVIDIA terminology these groups are called *Streaming Multiprocessors* or SMs. Processing cores within each SM work in lockstep, i. e. all cores are executing the same instruction at any point in time. This is similar to the SIMD (single instruction, multiple data) model of computation, where the same instruction is concurrently executed on multiple data items. In the case of GPUs, it is sometimes referred to as SPMD (single program, multi data) (Hwu et al., 2009). This style of computing allows processing cores to share logic between them, e. g. the instruction fetch unit. It also means, however, that when multiple threads assigned to the same SM execute different paths of a program, the execution of threads in different paths is serialized, losing the advantage of highly parallel execution. Each SM works as an independent unit, i. e. threads assigned to *different* SMs can follow different paths without a performance penalty. Furthermore, synchronization is restricted to threads within the same SM.

Each SM has its own *local memory* that is shared by the threads assigned to that SM. The local memory storage is small, usually around 64KB, but provides fast access. In older generations of GPUs, the local memory is managed by software, but in current architectures it is often divided into a software-managed scratchpad memory and a hardware-managed level-1 (L1) cache. The scratchpad memory can be directly accessed in programming languages such as OpenCL or CUDA (see section 2.2.2).

When threads co-running on an SM issue memory requests, these requests can be merged if certain constraints are met. This is called *memory coalescing* and is crucial to achieve the peak memory bandwidth on these GPUs. Specifically, when threads running on consecutive processing cores access data that is stored close to each other in memory, these accesses can often be coalesced. The exact conditions under which memory coalescing takes place vary across GPU architecures.



(a) General-purpose GPU (b) Multi-core CPU

Figure 2.1: Diagram of typical general-purpose GPU and multi-core CPU architecture. The GPU is modeled on the NVIDIA Fermi architecture.

Shared across all SMs is the L2 cache that caches accesses to global memory. Traditionally, GPUs have a separate global memory space from the system's main memory. In some recent systems, however, the GPU can also directly access the main memory. Section 2.1.3 provides a more detailed discussion on how GPUs are integrated within heterogeneous systems.

2.1.2 *Comparison of CPU to GPU architectures*

CPUs and GPUs are designed with very different goals in mind. The CPU is the primary processor which executes the operating system and the majority of user applications. It thus needs to be flexible and provide good performance on a wide range of applications. Since many applications are sequential rather than parallel, CPUs are designed to *minimize the latency* of single processing threads. GPUs, on the other hand, only execute programs that are specifically targeted towards them. Typical computer graphics tasks exhibit large amounts of parallelism. GPUs are thus designed to *maximize the throughput* of many processing threads (Garland and Kirk, 2010). These different performance goals are evident in their architectures as shown in figure 2.1.

Multi-core CPUs typically contain a small number of processing cores which each work independently on a single thread of execution at a time. The cores are very powerful in that they contain large amounts of logic to speed up the execution of single threads, such as branch prediction, instruction reordering and pipelining (Hennessy and Patterson, 2012). In contrast, GPUs contain a large number of processing elements (or PEs) that are organized into groups of PEs, called SMs in NVIDIA terminology.

While SMs work independently of each other, the PEs in one SM can only execute the same instruction at the same time. Each PE is much simpler than a CPU core and thus slower when considering only a single thread of execution. To compensate for that, GPUs have large register files that allow for fast *context switches*, i. e. changing execution from one thread to another. On the CPU, this is a costly operation because registers and other processor state need to be stored in memory. The large number of processing elements and the ability for fast context switches mean that GPUs excel at highly parallel applications. Furthermore, it serves to *hide* latency, as opposed to CPUs which require caches to be latency tolerant.

While CPU cores each have their own L1 (level-1) cache, on the GPU, all processing cores in an SM share the L1 cache. Good memory performance on the GPU is only achieved when accesses to global memory can be coalesced. It thus requires memory accesses to be very regular. Similar to CPUs, spatial and temporal locality is important.

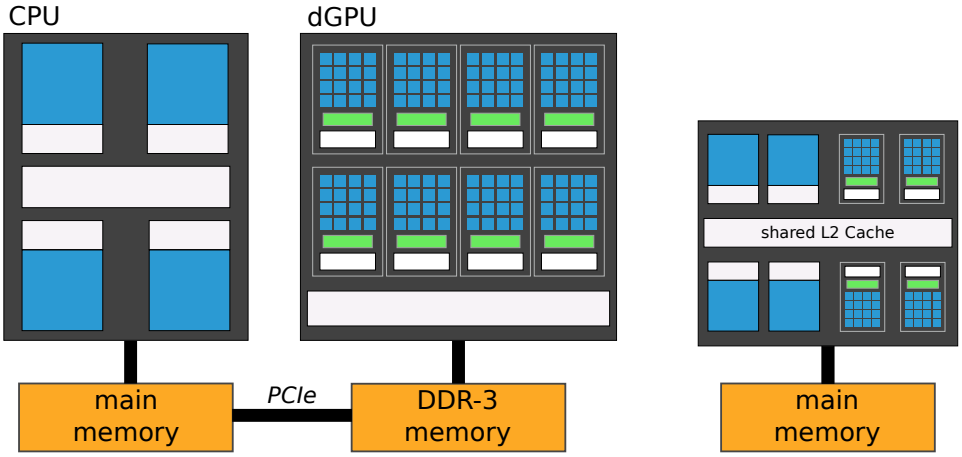
In summary, CPUs are more flexible than GPUs. However, they only achieve this by designating significant amounts of silicon to non-computational tasks. Hardware units such as the branch predictor do not perform any computation relevant for the currently executing application. GPU cores, on the other hand, are much leaner and even share resources between them. Tasks suited for GPU computing can thus be executed much more efficiently on the GPU than on the CPU, in terms of both performance and energy.

2.1.3 *Discrete vs. Integrated GPUs*

Heterogeneous CPU-GPU systems often come in one of two different flavours, as shown in figure 2.2. Traditionally, the GPU sits on an external card, the graphics card, that is connected to the computer's mainboard via PCI Express. These types of GPUs are called discrete GPUs or *d*GPU*s*. More recently, GPUs have also been directly integrated on the same chip as the CPU, e.g. on the Intel Ivy Bridge or AMD Fusion architectures. In this case, the GPU is referred to as an integrated GPU or *i*GPU.

On systems with discrete GPUs, the dGPU has its own physical memory rather than being directly connected to the main memory. Any data that is needed for computing on the GPU needs to be copied over to the GPU memory, and output data needs to be copied back to main memory before it can be used by the CPU. Due to bandwidth and latency limitations of the PCI Express bus, these data movements can lead to significant slow-downs.

In the case of integrated GPUs, the CPU and iGPU often share the same physical memory. The memory can either be logically divided, so that the CPU and iGPU have their own, separate memory spaces, or it can be truly shared by all processors. In the former case, data needs to be copied between the two memory spaces, just like in the



(a) Heterogeneous system with discrete GPU

(b) Heterogeneous system with integrated GPU

Figure 2.2: Diagram of two heterogeneous systems: one with a discrete GPU, the other with an integrated GPU

case of systems with dGPUs. However, these copies are less expensive than moving data over the PCI Express bus (Spafford et al., 2012).

2.2 PARALLEL PROGRAMMING LANGUAGES AND FRAMEWORKS

There are many parallel programming languages and frameworks supporting a number of parallel programming paradigms. This section focuses on task and data parallelism, which are the most popular parallel paradigms. In task parallelism, different execution threads perform different tasks on different data, whereas in data parallelism, the same task is performed on different data in parallel. Since data parallelism naturally maps to the SIMD style of GPUs as described in section 2.1, the focus in this thesis is on data parallelism.

2.2.1 *OpenMP*

OpenMP is a widely used parallel programming framework for C and Fortran targetting multi-core CPUs (Dagum and Menon, 1998). It supports both task and data parallelism, but is mainly used for the latter. OpenMP provides an easy way to parallelize existing applications by annotating with `#pragmas` sections of the program that can be run in parallel. A typical example is a loop where all loop iterations can be executed independently of all other iterations. This is also referred to as loop or data parallelism (Grama et al., 2003). An example code snippet of a parallel vector addition is shown below. The vector `C` is computed by adding the corresponding elements

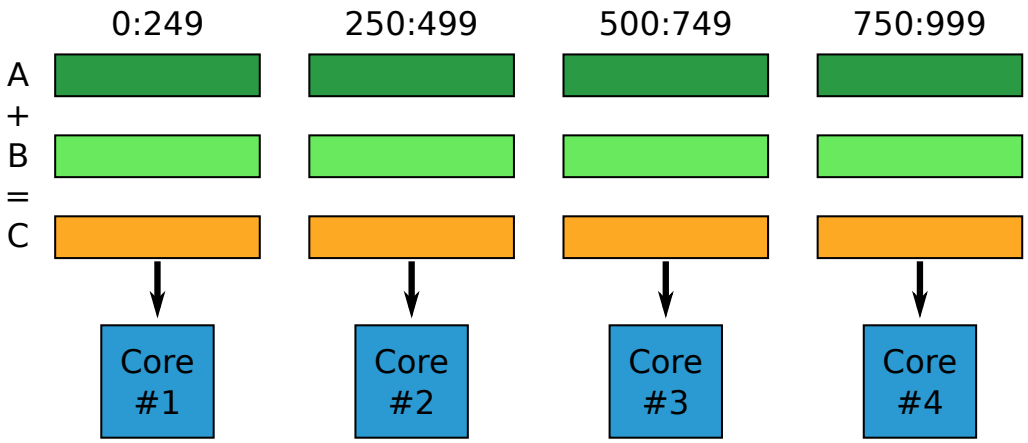


Figure 2.3: Static mapping of loop iterations to four processing cores for vector addition.

from vectors `A` and `B`. The order in which these additions take place does not matter. They can therefore be executed in parallel.

```
#pragma omp parallel for
for (int i=0; i<1000; i++)
  C[i] = A[i] + B[i];
```

Loops annotated as parallel are executed by dividing the loop iteration space into chunks and assigning those chunks to different processing cores. By default, the iteration space is statically divided into as many chunks as there are cores. Figure 2.3 illustrates this approach for the vector addition example on a four-core system. The iteration space is divided into four chunks so that the first 250 iterations are executed on the first core, the next 250 iterations are executed on the second core, and so on.

2.2.2 *OpenCL*

The Open Computing Language (OpenCL) is a language and API for parallel programming on heterogeneous systems (Khronos, 2013). It primarily targets GPUs and multi-core CPUs. While there is some rudimentary support for task parallelism, it was above all designed for data parallelism which can be exploited on the GPU.[1] Whereas OpenMP provides an easy path to upgrade existing applications, OpenCL requires a more significant rewrite of the application. Programs using OpenCL consist of two parts, the *host code* and the *device code*.

The host code contains much of the "usual" parts of a program, such as input and output handling and initialization of data structures. Furthermore, it is responsible

1 In the recently released OpenCL 2.0 specification additional features have been introduced to make OpenCL more widely applicable, such as support for shared virtual memory and dynamic parallelism. However, at the time of writing of this thesis, no implementation of this specification has been made available by a hardware vendor yet.

for setting up the computation on the processor devices, e.g. multi-core CPU or GPU, and handling data transfers between different memory spaces, e.g. main memory and GPU memory. This is done via calls to the OpenCL API. At the beginning of an OpenCL application, the host code sets up the OpenCL environment. This includes selecting the devices on which to run the device code. Communication with devices is handled via *command queues*, one (or more) for each device. To launch a task on a device, it is enqueued in the device's command queue. Command queues can be either in-order or out-of-order. In the latter case, *events* associated with tasks are used to express dependencies between tasks. The host code also needs to create *buffers* that contain the data the device code works on.

OpenCL device code is written in OpenCL C. This is a simplified version of C99 with some extensions, such as vector types and address space qualifiers (see below). There are also built-in functions for synchronization and mathematical functions. The device code describes the computation that is to be run on the processor devices. Data-parallel tasks are expressed as *kernels*, where the code describes the computation of a single *work-item*. When a kernel is launched by the host code, the number of work-items to be created is specified. The kernel code can be thought of as the body of the loop and the number of work-items is equivalent to the number of loop iterations. Work-items are arranged in a multi-dimensional grid (up to three dimensions), called *N-Dimensional Range* or *NDRange*. An NDRange is further divided into *work-groups*, i. e. subsets of work-items. This is illustrated in figure 2.4. Work-groups allow work-items to collaborate, e.g. by sharing memory, and to synchronize. Synchronization is only allowed between work-items in the same work-group, and not across work-groups. There are built-in functions to query the position of a work-item in the NDRange from the device code. The NDRange position can be thought of as the loop variable. The vector addition example from section 2.2.1 can be expressed in OpenCL as:

```
__kernel void vectoradd (__global float * A,
                         __global float * B,
                         __global float * C)
{
  int i = get_global_id(0);

  C[i] = A[i] + B[i];
}
```

The `__kernel` keyword indicates that this function is an OpenCL kernel function. The `__global` keyword in front of the parameters specifies the arrays' address spaces (see below). Each work-item queries its position in the NDRange using the built-in OpenCL function `get_global_id()`.

Separating work-items into work-groups leads to highly scalable code. Since synchronization can only be performed within work-groups, large numbers of work-

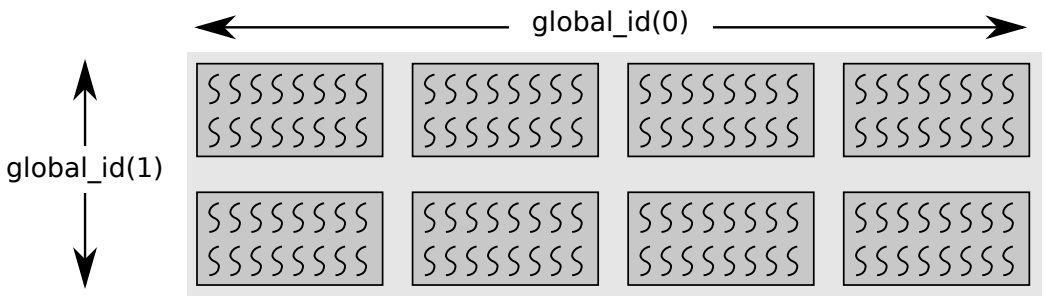


Figure 2.4: A two-dimensional NDRange of OpenCL work-items, arranged into work-groups of local size 8x2, the global size is 32x4.

groups can run in parallel without much overhead, only limited by hardware capabilities. It further allows kernel execution to be split across multiple devices. When splitting the NDRange at work-group boundaries, no synchronization is required across devices. This assumes, however, that atomic functions, as desribed below, are not used. Atomic operations across devices are generally not supported on current hardware.

OpenCL supports multiple address space qualifiers. By default, variables are *private*, i. e. there is a private copy of that variable for each work-item. Variables that are local to a work-group, and thus shared by all work-items in a work-group, are in the *local* address space. Data that is shared by all work-items is typically in the *global* address space, as in the example above. If data is read-only, however, it can also be declared as *constant*. This can be useful, for example, for small lookup tables where all work-items read the same entry at the same time, which leads to improved access times on some GPU architectures (Hwu et al., 2009).

As mentioned above, only work-items within a single work-group can synchronize with each other. OpenCL provides a *barrier* function which ensures that all work-items in a work-group execute that function before any work-item is allowed to proceed. The barrier can also act as a *memory fence* for both local and global data.

In addition to the primitive data types, OpenCL supports a range of *vector data types*. For every primitive data type, e. g. `float`, there are corresponding vector data types, e. g. `float2` or `float4`. When targeting hardware with vector units, e. g. Intel AVX or ARM NEON, operations on vector data types can be directly mapped to these units.

OpenCL further supports a range of atomic functions, including both arithmetical and logical operations. Atomic functions are the only means for work-items from different work-groups to collaborate. Kernels containing atomic functions cannot easily be partitioned and scheduled across multiple devices, because there is no guarantee of atomicity if atomic operations are performed on the same memory location from different devices, according to the OpenCL standard (Khronos, 2013).

OpenCL	CUDA
NDRange	grid
work-group	thread block
work-item	thread
global memory	global or device memory
constant memory	constant memory
local memory	shared memory
private memory	registers

Table 2.1: List of core OpenCL concepts and their equivalents in CUDA.

2.2.2.1 *CUDA*

The design of OpenCL is closely modeled on CUDA (Compute Unified Device Architecture), a proprietary framework for programming NVIDIA GPUs. Many of the concepts found in OpenCL have equivalent concepts in CUDA. Since CUDA was invented before OpenCL, much research has initially focused on CUDA.[2] Throughout this thesis, OpenCL terminology will be used. When discussing related work, however, CUDA terminology may be used in some cases. To understand how the two terminologies relate to each other, table 2.1 provides a list of core OpenCL concepts and their equivalents in CUDA. OpenCL terminology is generally more abstract whereas CUDA terminology is specific to NVIDIA GPUs.

2.3 MACHINE LEARNING

The goal of machine learning is to identify patterns in data. An example is the problem of recognizing handwritten digits, i. e. given an image of MxN pixels identify the digit $0, \ldots, 9$ represented in that image. Defining an algorithm to solve this problem is difficult, if not infeasible, to do for humans. There are too many ways how people write digits.

Machine learning uses a different approach to solving problems. Rather than hard-coding an algorithm, a program "learns" how to perform a task. It uses data observations to automatically tune a mathematical model for a given problem (Alpaydin, 2004; Bishop, 2006).

2 CUDA was first released in 2006 while the first OpenCL standard was only released at the end of 2008.

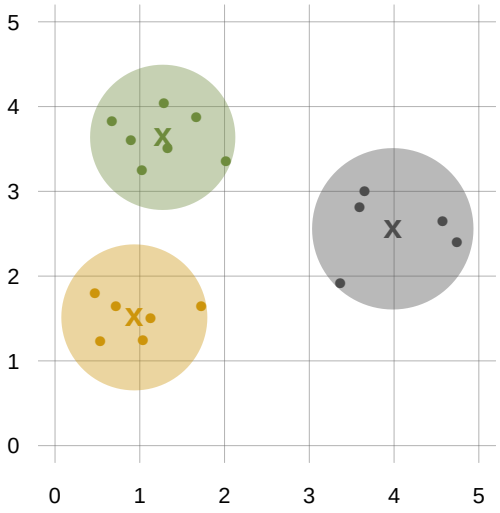


Figure 2.5: An example of clustering in a two-dimensional space. The cross marks the centre of each of the three clusters.

2.3.1 *Terminology*

Machine learning algorithms use *observations* to build a model of the problem at hand. An observation is a pair $< \vec{x}, t >$ where $\vec{x}$ represents the input to the problem, called *feature*, and t is the output, or *target*.

The *feature space* describes the space of all possible features. It is an n-dimensional space, where n is the number of features. Similarly, the *target space* describes the possible target values.

Two common classes of machine learning algorithms are *unsupervised learning* and *supervised learning*.

UNSUPERVISED LEARNING Unsupervised learning operates solely on the feature space. Given a set of data points in the feature space, the goal is to find regularities in the inputs. The only unsupervised learning technique used in this thesis is Principal Component Analysis (PCA). It is used as a pre-processing step on the training data before applying other methods. PCA is discussed in detail in section 2.3.2.2.

The canonical example of unsupervised learning is clustering, i. e. dividing a set of data points into clusters of points that are close to each other in the feature space. Figure 2.5 demonstrates this idea. The data points are divided into three clusters and each cluster is described by its centre point. When new data points come in, they can be associated with the cluster whose centre is closest to them. Clustering requires a distance metric to determine the proximity or similarity of two points to each other. In this example, euclidean distance is used, which is a common choice for clustering.

SUPERVISED LEARNING The goal of supervised learning is to find a relation between data points in the feature space and data points in the target space. In other words, it is generating a function

$$f : \vec{x} \rightarrow t.$$

This is done using a set of observations called the *training set*, where each observation represents a specific point in this relation between the two spaces. Building (or *training*) the model involves minimizing an *error function* on the training set. A commonly used error function is *mean squared error* defined as

$$\frac{1}{n}\sum_{i=1}^{n}(f(\vec{x_i}) - t_i)^2$$

where the $< \vec{x_i}, t_i >$ form the training set.

Checking the accuracy of a model is done using another, separate set of observations called *test set*. By comparing the output of the model given an observation ($f(\vec{x})$) to the actual target value t, the accuracy of the model can be computed.

The above mentioned problem of handwritten digit recognition is an example of supervised learning. The feature space is the space of all possible images of a certain size and the target space is the set of digits $0, \ldots, 9$. Problems of this kind, where the target space contains a fixed set of values, are called *classification problems*.

Another example of supervised learning is curve fitting. For example, in two-dimensional space the problem is to find a value y given another value x. Relations of this kind can be described by, for example, polynomials. Figure 2.6 shows an example of fitting a set of observations to a polynomial of degree 2. The polynomial can be described by three parameters a, b, c such that $y = ax^2 + bx + c$. These parameters are now optimized to minimize the error on the training set, using for example mean squared error as an error metric. Curve fitting is an example of *regression problems*, where the output values are contiguous.

2.3.2 *Data Preprocessing*

Solving problems using machine learning techniques generally involves two steps: preprocessing the data and constructing the model. Preprocessing is required to normalize the feature values and reduce the dimensionality of the input space.

2.3.2.1 *Normalization*

Normalization transforms the feature values so that for each feature the values lie in the same spectrum. Common transformations involve normalizing the data to the

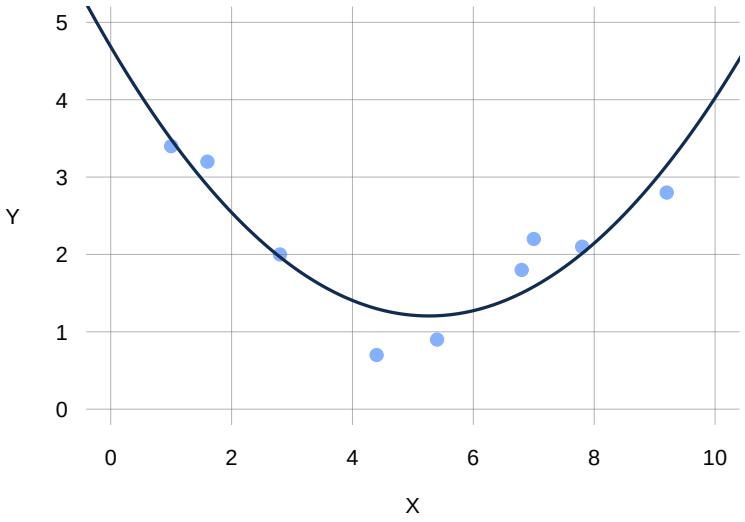


Figure 2.6: An example of curve-fitting in a two-dimensional space. The line represents a polynomial of degree 2 fitted to the observations.

range $[0, 1]$ or to "zero mean, unit variance", i. e. the values are normally distributed with a mean of 0 and a variance of 1.

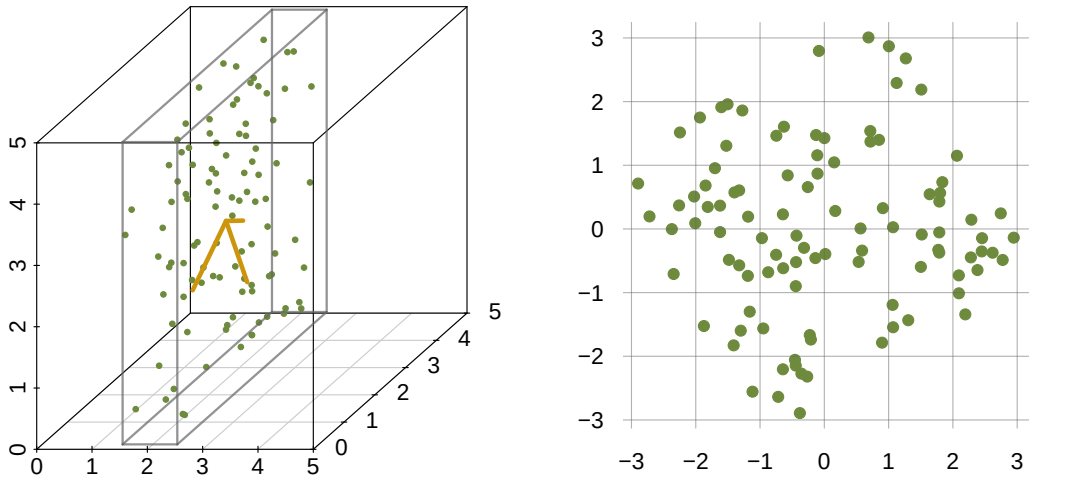
Normalization eliminates the problem of having different units for different features. After normalization, all values lie in the same range, regardless of the unit chosen. Furthermore, it ensures that all features are treated the same by the machine learning model. Without normalization, features whose values tend to be larger than those of other features may be treated as more important, because they have a bigger influence on the error function.

Another common normalization method is to apply a logarithmic function to the data. This method can be useful if values of a particular feature are exponential, i. e. there are lots of relatively small values and a small number of very large values. Applying a logarithmic function helps to highlight the differences between the small values.

2.3.2.2 *Principal Component Analysis*

Principal Component Analysis (PCA) is an unsupervised learning technique commonly used to reduce the dimensionality of the feature space, e. g. from an n-dimensional space to a k-dimensional space where $k < n$ (Pearson, 1901; Fodor, 2002). Reducing the dimensionality of the feature space reduces the complexity of the problem, which means a good model for it can be found more easily (Bishop, 2006). Feature sets often contain features which convey very little information, e. g. because they can be expressed as a linear combination of one or more other features. The dimensionality of the space can therefore be reduced without sacrificing accuracy.

PCA uses orthogonal linear transformations to find a set of *principal components*, i. e. linear combinations of the features in the original space. The principal compo-



(a) Data set in the original (3D) space.

(b) Data set after PCA in the reduced (2D) space.

Figure 2.7: Example of PCA on feature set in a three-dimensional space. In the original space (a) the variance along the X dimension is much smaller than along the Y and Z dimensions (as indicated by the light box). By computing the principal components, shown in (a), the data can be transformed into a two-dimensional space without losing much variance (b).

nents are chosen to maximize the variance, so that after projection into the new space the difference between the data points is greatest. They are further ordered by variance, so that the first component accounts for the greatest variability in the input data and so on. The first few components often account for most of the variance in the data, which means a small k can be chosen to significantly reduce the dimensionality.

Figure 2.7a shows an example of a data set in a three-dimensional space. The variance of the data along the X axis is significantly smaller than the variance along the Y and Z dimensions. This is indicated by the light box which represents the range of all values. The three principal components of the data are also shown in the graph. By the size of the components one can observe that only two of them account for most of the variance in the data. Therefore, the points can be transformed to a two-dimensional space using the two largest components without losing much variance in the data. The transformed data points are shown in figure 2.7b.

2.3.3 *Classification Algorithms*

The goal of classification algorithms is to assign a label to a feature vector. The target space thus contains a fixed number of possible values. In *binary* classification there are only two labels, e.g. `Yes` and `No`. Cases with more than two labels, e.g. the set of all digits, are referred to as *multi-class* classification.

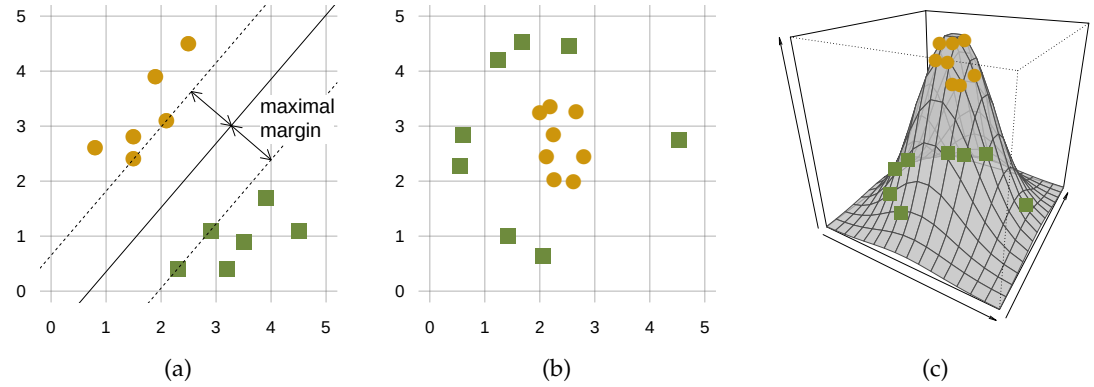


Figure 2.8: Example of a support vector machine model. In (a) the two-dimensional data can be separated by a hyperplane. The data in (b) can only be separated after it is mapped to a three-dimensional space (c).

Many different techniques exist to model classification problems (Bishop, 2006). In this thesis *support vector machines* and *decision trees*, two widely used modeling techniques, are used.

2.3.3.1 *Support Vector Machines*

Support vector machines (SVMs) are supervised learning models that can be used for both regression and classification problems (Boser et al., 1992; Cortes and Vapnik, 1995). When used for classification, SVMs first map the input feature space to a higher-dimensional space and then find a hyperplane that separates training points belonging to different classes. To assign a label to a new data point its features are mapped to the high-dimensional space and the label is assigned depending on which side of the hyperplane the point lies. Figure 2.8a shows an example of a hyperplane in a two-dimensional space separating points from two different classes. The hyperplane is chosen so as to maximize its distance from the data points. This ensures a clean separation between the different classes.

Mapping the feature space to a higher-dimensional space may seem counterintuitive, especially when considering that the dimensionality of the feature space is often reduced when preprocessing the data, e. g. using principal component analysis. However, in the higher-dimensional space, SVMs are more likely to find hyperplanes that accurately separate the data points into their corresponding classes. In the original space, it is often impossible to find such linear separations. Various functions, called *kernel functions*, can be used to map the features to a higher-dimensional space. Common choices include polynomial functions, the sigmoid function and radial basis functions (Olson and Delen, 2008). The latter is often used as a starting point and is depicted in figure 2.8c.

In principle SVMs only support binary classification (data points are either on one side of the hyperplane or on the other). There are, however, several methods to im-

plement multi-class classification on SVMs (see Hsu and Lin (2002) for a comparison). Two methods relying on multiple binary classifications are "one-against-all" (Bottou et al., 1994) and "one-against-one" (Knerr et al., 1990).

In the "one-against-all" method, k classifiers are used, where k is the number of classes. For the ith classifier, data points in the ith class are in one group and the remaining data points are in the other group. To assign a label to a new data point, each classifier is evaluated on the point's features and a confidence score is assigned. The classifier with the highest score assigns the label to the new data point.

The "one-against-one" method trains classifiers using only data points from two classes at a time. In total, $k(k-1)/2$ classifiers are trained, one for each pair of classes. Assigning a label to a new data point is done using a voting scheme. Each classifier is evaluated and a vote is given to the class chosen by the classifier. The class with the highest number of votes is chosen as the label for the new data point.

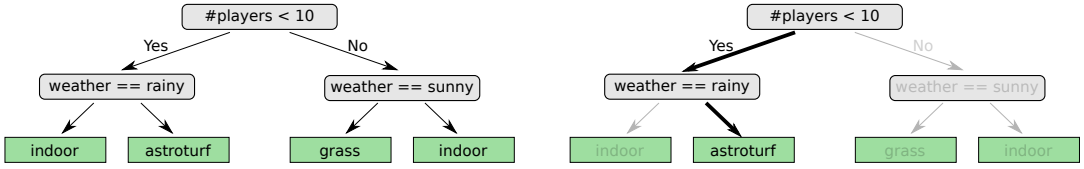
In this thesis the LIBSVM (Chang and Lin, 2011) implementation of support vector machines is used. It uses the "one-against-all" approach for multi-class classification problems, because it delivers comparable performance to the "one-against-one" method (Hsu and Lin, 2002) while having a lower complexity (the number of classifiers used is linear in the number of classes rather than quadratic).

2.3.3.2 *Decision Trees*

Decision trees are another supervised learning technique for classification problems (Quinlan, 1986). As the name suggests, it models problems as a binary decision tree. New data points are classified by traversing the tree from the root to a leaf. At each inner node of the tree one of the features is tested against a given value. In the case of numerical features, the value generally represents a threshold, i. e. the test compares whether the feature value is less than or greater than the threshold. In the case of categorical features, the test is whether or not the feature is in the specified category. Depending on the outcome of the test, either the left or the right subtree is then traversed. The leaves of the tree are labeled with classes. When a traversal reaches a leaf the new data point is classified according to the leaf's label.

An example decision tree is shown in figure 2.9. It models the problem of where to play football based on the number of players and the weather (either sunny, cloudy or rainy). Figure 2.9b shows the decision process for eight players on a cloudy day.

Decision trees are generally constructed in a top-down manner (Quinlan, 1986), although some bottom-up approaches exist as well (Barros et al., 2011). In the top-down approach each node of the tree is associated with a subset of the training data points; at the root this set is the entire training set. Given this subset, the feature and corresponding threshold value that most effectively split the data is chosen. Based on this test, the subset is further split into two sets that are assigned to the two children



(a) Decision tree.

(b) Traversing the tree.

Figure 2.9: Example decision tree for the problem of where to play football (a). On a cloudy day with 8 players the decision is to play on astroturf (b).

of the node. When a node's subset only contains data points from a single class, the set is not divided any further and the node is labeled with the data point's class.

While decision trees are often less accurate than support vector machines (Huang et al., 2003), they have the advantage of being easily interpretable. The decision process can be easily reproduced, which allows the user to gain an understanding of the underlying model.

In this thesis, the C4.5 decision tree algorithm by Quinlan (1993) is used. This algorithm uses information gain, or entropy, to split the data. At each node in the tree, the attribute with the highest information gain is chosen and the optimal split value is selected.

2.4 MACHINE LEARNING IN PROGRAM OPTIMIZATION

Optimizing programs for computer systems is a complex problem. The size of the optimization space of compiler flags, for example, can be on the order of 10^{17} (Dubach et al., 2009). Finding a good, not to mention optimal, setting is an infeasible task to do manually. Iterative compilation (Knijnenburg et al., 2002) has been used to automate the process of finding good compiler flags, but it is very time-consuming. Using machine learning, however, many of these complex problems can be solved much faster (see section 3.5 for examples).

In most cases, the goal of program optimization is to find good optimization parameters for a given program. Using the notation from section 2.3.1 this can be expressed as a function

$$f : \mathcal{P} \to \mathcal{O}$$

where $\mathcal{P}$ is the space of all programs and $\mathcal{O}$ is the space of all parameter settings controlling the optimization process.

The parameter space often contains one or more variables which can each take on a fixed number of values, e. g. whether or not to apply a certain transformation. When mapping parallel programs, for example, the space may simply contain the two options of CPU execution and GPU execution, i. e. $\mathcal{O} = \{\text{CPU}, \text{GPU}\}$. There are

two ways of building a model for problems like these. The direct method is to use a classification model where each parameter setting represents a class. A program belongs to a certain class if the corresponding parameter setting is optimal for this program. The indirect way is to build a regression model that predicts the running time of a program given a certain parameter setting. The best setting is chosen as the one for which the predicted running time is shortest. In this thesis, the direct way, using classification, is chosen. The indirect way of modeling optimization problems is generally more difficult to perform, because it solves a more complex problem than the direct way. However, if the optimization space is overly large (more than a few dozen points) the direct approach may become infeasible, because there are not enough data points for each class. In that case the regression model is the only feasible solution. This approach was used, for example, by Curtis-Maury et al. (2006) or Dubach et al. (2009). Since, in the case of mapping parallel programs, the optimization space is small enough (see section 2.4.2), classification models are used throughout this thesis.

The optimization space can usually be represented easily by a fixed number of classes. However, characterizing the programs, which form the input to the models, is more difficult. Machine learning models expect a vector of numbers (the features) as input, but how to meaningfully transform programs to features is non-trivial. The next section describes solutions to this problem.

2.4.1 *Program Characterization*

There are two commonly used techniques to characterize programs for machine learning models. The first one only uses static features extracted by analyzing the program code. If this information is not sufficient, dynamic features can be collected, e. g. using profiling.

2.4.1.1 *Static Feature Extraction*

Static feature extraction describes the process of statically analysing the source code of programs to extract information. Typical features include the number of memory accesses or floating point operations and information on memory access behaviour. Sometimes not all information can be extracted statically, e. g. loop iteration counts. In this case, either the user has to provided an estimate or the feature can be represented as a function that is dependent on some statically unknown value, such as the input size. In the latter case, the feature can be instantiated at run time when the value is known.

In this thesis, static feature extraction is performed on OpenCL kernel code. A tool based on clang (LLVM, 2013) has been developed that analyses the code. The OpenCL code is transformed into an abstract syntax tree (AST) by clang and the AST

Static Code Features	
# int operations	# global memory accesses
# int2 operations	# coalesced memory accesses
# int4 operations	# local memory accesses
# char operations	# constant memory accesses
# char2 operations	# statements
# char4 operations	# assignments
# float operations	# vector assignments
# float2 operations	# barriers
# float4 operations	# conditionals
# OpenCLmath operations	# loops
# atomic operations	size of static local memory

Table 2.2: Full list of features extracted by the static feature extraction tool.

is then traversed to extract various features. In addition to counting certain types of operations, a value analysis (Nielson et al., 2005) is performed to derive memory access patterns and loop counts. If a loop count depends on a kernel parameter, the affected features are represented as a function depending on that parameter. At runtime, when the parameter's value is known, the actual values of the features are computed.

The value analysis is also used to determine memory access patterns. By analyzing the indices of array accesses, the tool determines how many of the accesses to global memory are coalesced (see section 2.1.1).

A list of all features that are extracted by the tool is shown in table 2.2.

2.4.1.2 *Dynamic Features*

Data extracted using static analysis techniques can sometimes not be sufficient to accurately characterize programs. In that case, dynamic features can provide further information on the program. Dynamic features are typically collected using *profiling* tools. Programs can be *instrumented*, for example, to extract information on program behaviour or to read hardware instruction counters. The instrumented program is run once to collect the desired information, e.g. number of loads and stores, cache miss rate etc. Another approach is to run the program multiple times but with different parameter settings. By sampling the parameter space and measuring the running times of different settings, the model can extrapolate across the entire space.

The drawback of profiling is that it requires one or even many executions of the program, which can be prohibitive. Therefore, profiling is not used in this thesis.

Another class of dynamic features is, however, used in this thesis. Some information that is generally only known at runtime, such as a program's input size, can be crucial for making optimization decisions. The models are thus only evaluated at runtime, when this information becomes available. Unlike profiling, however, collecting this information does not require executing the code that is being optimized.

2.4.2 *Putting it all together*

This thesis considers the problem of mapping parallel programs to heterogeneous multi-core systems. It uses machine learning in two different ways. In the first method, only two mappings are possible. The program is either executed entirely on the CPU or on the GPU. This is modeled as a binary classification problem. In the second method, more fine-grained mappings, which (possibly) partition the work between the CPU and the GPU, are considered. This is modeled by eleven classes representing eleven ways to partition the work: $100/0, 90/10, 80/20, \ldots, 0/100$ where A/B means that $A\%$ of the work is mapped to the CPU and the remaining work ($B\%$) is mapped to the GPU.

Programs are characterized by performing static feature extraction on the OpenCL code. Where features cannot entirely be determined statically, dynamic information is used to instantiate them at run time. Rather than using the raw features extracted by the analysis tool, features are often combined to make them more meaningful. The exact combination of features used is described in the corresponding chapters.

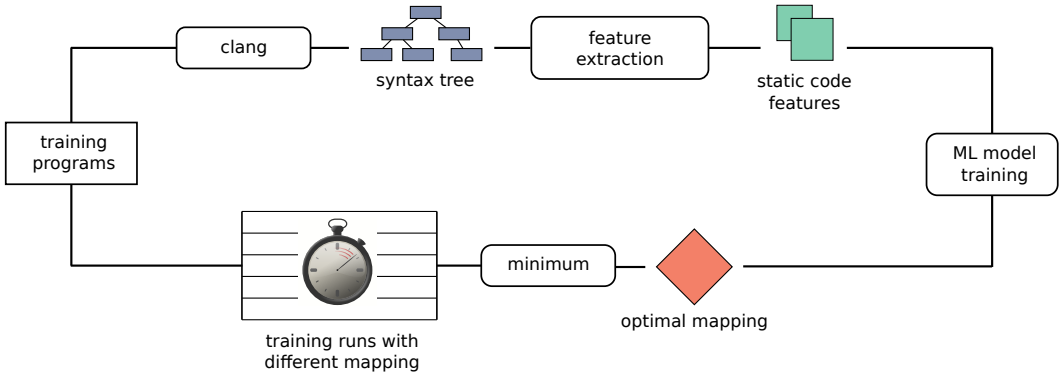
Training data is collected using a set of training programs. Training features are statically extracted from the programs' source code, as described above. Training targets are obtained by running each program with all possible mappings. The mapping which leads to the shortest running time is selected as the target for the particular training data point.

When evaluating the model on a new program, the program's features are extracted at compile time. At run time, they are instantiated with dynamic information and passed into the model. The program is mapped based on the model's prediction as to which mapping is optimal.

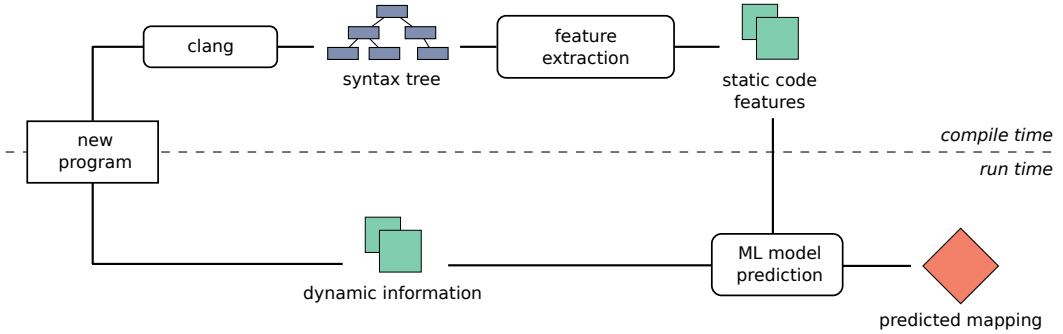
The entire process, from training to deployment, is depicted in figure 2.10.

2.5 EVALUATION METHODOLOGY

This section describes how the machine learning models built in this thesis are evaluated. While section 2.3.1 already introduced the concept of splitting the data into training and test sets a special validation method, called *cross-validation*, is used in this thesis. Furthermore, the models in this thesis are not evaluated on accuracy alone. In the setting of mapping parallel programs to heterogeneous systems, the performance



(a) Training: Features are extracted from each training program using a CLANG-based tool. Each program is run with all mappings with the one leading to the shortest time being picked as the optimal mapping.



(b) Deployment: At compile time, static code features are extracted from the new program. At run time, these features are instantiated with dynamic information to predict the optimal mapping for this program.

Figure 2.10: Process of using machine learning in program mapping, from training (a) to deployment (b). The deployment happens in two stages: feature extraction at compile time and prediction at run time.

penalty of a mis-prediction can vary wildly. The relative performance of the predicted mapping with respect to the optimal mapping is therefore an important evaluation metric.

2.5.1 *Cross-Validation*

Machine learning models are evaluated by splitting the available data into a training set and test set (see section 2.3.1). In some cases, there are not enough observations available to split the data into two sets that are each large enough to be representative of the overall problem. Cross-validation is a commonly used technique to overcome this issue.

Cross-validation performs several evaluations of the model, each time with a different training and test set. In an n-fold cross-validation, for example, the data is randomly split into n equally sized subsets. For the ith iteration, the ith subset is

used as the test set and the remaining $n-1$ subsets form the training set. It is important to stress that none of the data in the test set is used for training the model.

A special case of n-fold cross-validation is *leave-one-out cross-validation*, where n is equal to the number of data points. In other words, in each iteration only one of the data points is used for testing and all other points form the training set.

When using machine learning to optimize programs, it is common to have multiple data points from the same program, e. g. from multiple kernels or multiple input sizes. In this case, it is important to ensure that there is no data from the same program in both the training and test set. Even with leave-one-out cross-validation, all data points that originate from the same program as the one in the test set are removed from the training set.

CROSS-VALIDATION IN MODEL SELECTION AND TUNING Cross-validation can also be used to tune parameters of a model or even select the model itself. Many models have parameters that affect the shape of the model or the training process. Examples are the degree of the polynomial in curve fitting (see section 2.3.1) or the kernel function in a support vector machine (see section 2.3.3.1). For this purpose cross-validation is performed only on the training data, rather than the entire data set. The training data is divided into subsets that are used as training and *validation* sets in the cross-validation.

The model is tuned by searching the model parameter space. At each point in the space cross-validation is performed to evaluate the current setting. At the end of the search the parameter setting that produced the most accurate model is determined and the model is trained on the entire training data using this setting. This kind of performance tuning is valid because it only relies on the training data and does not use the test set.

2.5.2 *Relative Performance*

The goal of the work presented in this thesis is to improve the performance, i. e. reducing the execution time, of parallel programs by making efficient use of heterogeneous systems. All results are thus evaluated in terms of performance, even when it comes to evaluating the accuracy of machine learning models. Traditionally, these models are evaluated by their classification accuracy, i. e. how often is the right class predicted for a data point in the test set. While this is an important metric, what is more important when optimizing program performance is the resulting speed of the program. If the prediction is correct, the optimal performance is achieved. If it is incorrect, the performance may still be close to the optimum. It may, however, be much slower. Hence the importance of evaluating models based on performance rather than classification accuracy.

Two different performance metrics are used throughout this thesis: *speedup* and *oracle performance*. Speedup is the ratio of the execution time with respect to a certain *baseline* performance. In this thesis the baseline is either single-thread CPU execution or single-device execution. Which baseline is used is stated in the corresponding chapters. Speedup is computed as

$$\frac{t_b}{t}$$

where t_b is the execution time of the baseline and t is the execution time of the evaluated approach. Higher numbers are thus better. A speedup of greater than 1 corresponds to an increase in performance and a speedup of less than 1 corresponds to a decrease in performance.

Oracle performance is the performance relative to an "oracle", a (fictitious) approach that always performs optimally. This technique is used when evaluating approaches using machine learning models. It is computed as

$$\frac{t_p}{t_o}$$

where t_p is the execution time when using the prediction result and t_o is the execution time when using the oracle. If the model achieves a 100% accuracy the oracle performance is 1, otherwise it will be less than 1. Thus, the higher the number the better but it can never be greater than 1. When the optimality of a result is mentioned throughout this thesis, it always refers to the performance of the oracle.

Computing the oracle performance involves finding the optimal configuration for each data point in the test set. This is done by exhaustively evaluating all possible configurations and selecting the one which produces the shortest running time. This process can be very time consuming. However, it is only done for evaluation purposes and not needed in principle. If the configuration space is too large, sampling can be used to approximate the oracle performance, see for example Dubach et al. (2009).

2.6 SUMMARY

This chapter has described the key concepts of heterogeneous computer systems and how to program them, specifically using OpenCL. Furthermore, a basic introduction to machine learning methods was given and how these techniques are used for program optimization. Finally, common evaluation methodologies used to evaluate the contributions of this thesis were shown.

Before describing the contributions of this thesis in detail, starting in chapter 4, the following chapter provides a comprehensive discussion of related prior work relevant to this thesis.

3

RELATED WORK

This chapter discusses prior work related to the areas introduced in the previous chapter. A comprehensive review of publications in each area is provided.

The first two sections deal with mapping programs to heterogeneous systems. The mapping process can in general be divided into two phases. The program is first *partitioned* into tasks which are then *scheduled* to the different devices. Section 3.1 summarizes the considerable body of work on scheduling tasks to heterogeneous systems. It is assumed that programs have already been partitioned, e. g. by the programmer, or that the tasks originate from multiple programs. Section 3.2 discusses work on mapping programs to GPU-based systems.

Section 3.3 discusses automatic optimization techniques for GPU programming. It focuses on optimizations for both kernel code and data transfers. In section 3.4 prior work on mapping high-level languages to heterogeneous systems is introduced, including directive-based approaches such as OpenACC. This is followed by a discussion of machine learning-based techniques for program optimization in section 3.5. The chapter concludes with a brief summary of the discussed works in section 3.6.

3.1 TASK SCHEDULING ON HETEROGENEOUS SYSTEMS

Task scheduling on heterogeneous devices has been researched for more than 35 years. Usually, the heterogeneity stemmed from having different types of CPUs, either in the same computer or in a distributed system. The CPUs differed in terms of architectural features, such as cache sizes, and frequencies. Since each CPU only contained a single processing core, the individual tasks being scheduled were sequential.

Task scheduling for heterogeneous systems is known to be an NP-complete problem (Ibarra and Kim, 1977). Early work focused on finding heuristics with good, i. e. polynomial, running time that were able to generate schedules whose performance was within a small factor of the optimum. Rather than actually executing tasks on real machines results were usually obtained by simulation. The execution times of each task on each processor was assumed to be known and often these times were randomly generated (Braun et al., 2001). In case of dependencies between tasks they are expressed as a directed acyclic graph (DAG) where nodes represent tasks and edges represent the dependencies between tasks.

The following review of papers gives a brief overview of the large area of research being done on task scheduling for heterogeneous systems. It is by no means a com-

plete survey of the field. Section 3.2 provides a more in-depth review of research on task mapping for *GPU-based*, heterogeneous systems.

3.1.1 *Static Scheduling*

Static schedulers map a fixed set of tasks to processors in a heterogeneous system. All tasks are mapped up-front and the schedule is not altered during execution.

Scheduling Independent Tasks

Horowitz and Sahni (1976) propose some of the first techniques for scheduling a set of independent tasks to non-identical processors. Firstly, they propose dynamic programming solutions to find the optimal schedule for a given objective such as overall finishing time. Since these algorithms have a worst-case running time that is exponential in the number of tasks they also propose approximation algorithms which run in polynomial time.

Ibarra and Kim (1977) introduce multiple heuristics for scheduling a set of independent tasks to non-identical processors. The goal of the heuristics is to minimize the overall finishing time of all tasks. Five different greedy approaches are considered for the general case of $m \geqslant 2$ processors. All of them run in polynomial time and are within a factor m of the optimal schedule. Another heuristic is presented for the special case of $m = 2$ processors which also runs in polynomial time and is guaranteed to produce schedules within $(\sqrt{5}+1)/2$ of the optimal schedule. Two of the heuristics have later become known as Min-min and Max-min (Braun et al., 1999, 2001; Kim et al., 2003). The Min-min heuristic computes at each step the shortest finishing time (across processors) for all tasks that have not yet been scheduled. It then selects the one with the minimum finishing time. The Max-min heuristic also computes the shortest finishing times at each step but then selects the task with the maximum finishing time. The $O(m)$ bounds on optimality by Ibarra and Kim (1977) have been improved to $O(\sqrt{m})$ by Davis and Jaffe (1981) while still maintaining a polynomial running time.

Braun et al. (1999, 2001) compare a number of static scheduling heuristics for heterogeneous systems. The heuristics include the Min-min and Max-min heuristics by Ibarra and Kim (1977) as well as Simulated Annealing and Genetic Algorithm techniques. Execution times for each task on each processor are represented as a $\tau \times \mu$ matrix where τ is the number of tasks and μ is the number of machines. To cover a wide variety of scenarios several matrices are automatically generated based on particular objectives such as high or low machine heterogeneity. They conclude that the approach using Genetic Algorithms performs best but it is also one of the most computationally expensive heuristics. The significantly simpler Min-min heuristic delivers results that are almost as good.

Scheduling Tasks with Dependencies

So far all heuristics were based on the assumption that there are no dependencies between tasks. Topcuoglu et al. (1999, 2002) introduce two scheduling algorithms that allow for tasks to have dependencies. These are expressed in a directed acyclic graph (DAG), where nodes represent the tasks and an edge between nodes represents a dependence between the corresponding tasks. The cost of running each task on each processor is known as are the data transfer costs between dependent tasks. The first algorithm is the Heterogeneous Earliest-Finish-Time (HEFT) algorithm. At each step it selects the task with the highest *upward rank* and assign it to the processors that minimizes the overall finishing time. The upward rank is the length of the critical path from a node in the task graph to the exit node. The second algorithms is the Critical-Path-on-a-Processors (CPOP) algorithm which maps the critical path to the single processor that minimizes the critical path length. The remaining tasks are mapped according to the upward and downward rank. The main motivation behind this approach is to reduce the communication time between tasks on the critical path. On a set of randomly generated task graphs as well as graphs from selected applications the HEFT algorithms outperforms competing approaches, including CPOP.

Scheduling on Distributed Systems

Multiple scheduling algorithms have been proposed for *grid computing*, where a number of commodity machines are connected via a network, e. g. the internet. In these systems heterogeneity not only stems from having different processors but also different connections between the nodes. These systems are often modeled as graphs, with each node representing a machine and edges representing a connection between them. All tasks originate at the root node from where they are distributed across the system. Beaumont et al. (2002) introduce an algorithm for scheduling independent, equal-sized tasks to a grid modeled as an arbitrary tree. They propose a bandwidth-centric approach where all nodes are kept busy only if enough bandwidth is available. If bandwidth is limited some nodes may remain idle to reduce communication costs. Beaumont et al. (2003) also consider the problem of scheduling a large, arbitrarily divisible task onto a star-shaped network: a master node surrounded by worker nodes. They consider a *single-round* algorithm where the task is broken up into as many chunks as there are workers and then distributed by the master in a single round. They further consider a *multi-round* algorithm where the task is divided into a large number of small chunks that are distributed in multiple rounds. The multi-round algorithms is generally more efficient than the single-round one because workers can start processing sooner due to shorter communication times.

3.1.2 *Dynamic Scheduling*

Static scheduling assumes that the set of tasks to be scheduled is fixed and known before execution starts. It further relies on having accurate information on the running times of each task on all processors. Since these assumptions are not always realistic several dynamic scheduling algorithms have been proposed to overcome these limitations.

Maheswaran and Siegel (1998) propose a hybrid approach to tackle the issue of inaccurate execution time information. Initially, a static schedule is computed using estimates of execution times for each task on each processor. At runtime, when more accurate information on task completion times and machine availability become known the schedule is adapted. The tasks, whose dependencies are represented by a DAG, are divided into blocks of tasks that do not have any dependencies between them. The blocks are organized such that tasks in block i only depend on tasks in blocks 0 to $i-1$ and there is at least one task in block i being dependent on a task in block $i-1$. This ensures that no task in block i can be executed before task in block $i-1$ has finished. When the first task in block $i-1$ is being executed, the mapping for tasks in block i is being updated using the available runtime information. To evaluate their approach they randomly generate *actual* task completion times for each processor. They then introduce some random variation for the *estimated* times that are used for the initial static scheduling phase.

Maheswaran et al. (1999) also consider task scheduling in the case of randomly arriving tasks. Assuming all tasks are independent they propose two types of heuristics: immediate mode and batch mode. Immediate mode heuristics schedule a task to a machine as soon as it arrives. Batch mode heuristics, on the other hand, only make scheduling decisions at predetermined times called mapping events. At this time, all tasks that arrived since the last mapping event as well as tasks scheduled before but not yet executed are considered for mapping. Multiple heuristics for both types are considered and they conclude that the best heuristic depends on factors such as the arrival rate of tasks.

Scheduling on Single-ISA Systems

The previous approaches were theoretical in the sense that processors were only modeled by their completion times on tasks. The next two papers are closer to actual hardware by simulating a heterogeneous chip multiprocessor (CMP) based on existing processor cores. The cores differ in terms of architectural features, such as the cache size or superscalar width.

Kumar et al. (2005) consider task scheduling on heterogeneous single-ISA architectures. They show that, on average, these systems achieve a 29% performance improvement over an equivalent-area homogeneous system. Two heuristics are proposed that

each deal with two types of diversities of applications. The first one considers diversity between applications, the second one diversity over time within an application. To tackle diversity between applications they propose an algorithm that creates a static schedule whenever new tasks arrive. This schedule also considers tasks that are currently running and may re-schedule them to different processors. The second algorithm periodically enters a sampling phase where the scheduler permutes the mapping of tasks to processors. Using profiling information a new schedule is computed for the steady phase. To minimize the overhead of sampling, sampling phases are only entered when a change in application behaviour is detected. This is implemented by continuously monitoring the IPC of tasks.

A similar approach is proposed by Becchi and Crowley (2006). They use the ratio of IPCs between processors to schedule those tasks to the better performing core that benefit most from the additional speed. By periodically forcing task migration they account for changes in the workload and in phases in the applications themselves.

Scheduling on Distributed Systems

González-Vélez and Cole (2010) consider scheduling divisible tasks onto heterogeneous distributed systems. Their approach can be used for both single- and multi-round scheduling. A fitness index for the task to be scheduled is dynamically generated for each node in the system. The fitness index represents the suitability of a node for the task and is used to balance the work across the nodes. In an initial calibration phase a single work element is scheduled to each node. The execution times are measured and used to compute the initial fitness indices. For single-round scheduling the remaining work is now distributed across the nodes based on the fitness indices. In multi-round scheduling only a fraction of the work is distributed at first. After each round the fitness indices are updated according to the previous execution times in order to adapt to changes of a node's resource availability. Even though the multi-round approach incurs larger overheads it is favourable in non-dedicated systems where a node's resource availability can vary over time.

3.2 TASK MAPPING ON GPU-BASED HETEROGENEOUS SYSTEMS

With the emergence of programmable GPUs recent scheduling work has focused on systems where the heterogeneity comes from having two types of processors with very different characteristics, namely CPUs and GPUs (see section 2.1). Programming frameworks for these systems, such as OpenCL, are often targeted primarily at GPUs. However, performance can be significantly improved when using all processors available in the system (McIntosh-Smith, 2012). Since GPUs are highly parallel processors the tasks being mapped in this context are data-parallel tasks. This means that each

task can be further partitioned into sub-tasks which can be scheduled to multiple devices.

This section provides an in-depth review of research on task mapping for GPU-based heterogeneous systems. Firstly, it discusses approaches that schedule tasks to either the CPU or the GPU, i. e. no further partitioning takes place. Secondly, this section reviews work on partitioning tasks into subtasks which are then scheduled.

3.2.1 *Scheduling Tasks to Devices*

This section discusses work on scheduling tasks to GPU-based heterogeneous systems. The tasks are scheduled as a whole and not partitioned any further.

Scheduling Tasks with Dependencies

Current programming models for heterogeneous systems, such as OpenCL (Khronos, 2013), rely on the programmer to decide on which device to run a task. They are further required to explicitly copy the data needed by a task to the corresponding device. Augonnet et al. (2009b, 2011) introduce StarPU, a unified programming model for heterogeneous systems. It automatically schedules tasks to devices and transparently handles the necessary data transfers (Augonnet and Namyst, 2008). Its goal is to bridge the gap between the theoretical work done on scheduling and actual implementations for heterogeneous systems. In StarPU the user provides two versions of each task, one for the CPU and one for the GPU, and specifies inputs and outputs of each task. The runtime ensures that these data dependencies are adhered to. Furthermore, the user can specify explicit dependencies between tasks.

StarPU supports a small number of basic scheduling strategies (Augonnet et al., 2011). A simple greedy policy maps each task to whichever device is available which often leads to load imbalance. The weighted random strategy assigns tasks to devices with a probability proportional to the device's acceleration factor; a crude metric describing how much faster one device is over the other. This can either be set by the programmer or measured using reference benchmarks. The third strategy relies on user-specified performance models for each task. Based on these models the Heterogeneous Earliest-Finish-Time (HEFT) algorithm by Topcuoglu et al. (2002) is used. The scheduling policy based on HEFT outperforms the other two but it relies on the user to provide accurate performance models for each task. Augonnet et al. (2009a) try to overcome this issue by automatically building these models at execution time. When tasks are executed for the first time the measured run time on each device is stored in a hash table. When the task is executed again its previous run time is looked up and used for scheduling. The values in the hash table are continuously updated after each task execution. This approach has been extended for multi-GPU setups in another paper (Augonnet et al., 2010). On systems with multiple GPUs data

transfers can become a bottleneck. The authors thus extend their previous scheduling approaches with estimates of data transfer costs. These estimates are based on bandwidth and latency numbers as obtained using micro-benchmarks.

The setup of StarPU is similar to earlier work on scheduling where sets of inter-dependent tasks are scheduled (Topcuoglu et al., 1999, 2002; Maheswaran and Siegel, 1998). The main difference, however, is that in earlier work the individual tasks are *sequential* whereas in StarPU they are *data-parallel* due to the GPU's model of execution. The problem is that the number of applications containing many inter-dependent data-parallel tasks is small which limits the applicability of these approaches. This is confirmed by the fact that only three applications (blocked matrix multiplication, blocked Cholesky decomposition and blocked LU decomposition) are ever evaluated on StarPU (Augonnet et al., 2009b, 2011). Some of these applications, e. g. matrix multiplication, do not inherently contain multiple tasks and have to be divided by the user. This involves selecting a size for each sub-task which can have a significant impact on performance. This issue is not addressed in any of the papers.

Diamos and Yalamanchili (2008) developed another runtime systems for heterogeneous systems. Similar to StarPU applications are specified as sets of tasks with dependencies expressed implicitly by the inputs and outputs of tasks. The user provides separate implementations of each task - one for each target architecture - and the runtime schedules them on the devices. The scheduling is based on dynamically built performance models. The performance models use user-provided meta-data describing how a task's running time is dependent on input variables, such as input sizes. A multivariate regression model is then built for each task that models the task's runtime as a function of the input variables. The model is constructed at execution time using measured running times of previous task executions. No detailed information on how exactly the performance model is used for scheduling is given other than that the predicted times are used to minimize the overall execution times. Data transfer times, for example, do not seem to be considered. The evaluation of Harmony is limited to only three benchmarks. In none of them does the scheduler outperform a GPU-only approach by a significant margin.

Scheduling Individual Tasks

The approaches above - StarPU and Harmony - introduce their own runtime systems. They thus require substantial rewriting of applications to take advantage of the scheduling approaches. Furthermore, the user has to provide multiple versions of the tasks; one for each architecture. Becchi et al. (2010a) aim to provide scheduling across CPUs and GPUs without requiring the user to re-write any code. Their system intercepts predefined kernel functions and decides on which device to run the kernel. They do, however, still require two versions of each task. Furthermore, to ensure that data is kept consistent between the CPU and GPU the user has to insert synchroniza-

tion points, although this can be avoided by modifying the operating systems (Becchi et al., 2010b). Scheduling decisions are based on estimated task execution times and data transfer times. If, for example, the CPU version of a task is slightly slower than the GPU version the scheduler may decided to use the CPU if this avoids data transfers. Task execution times are measured during a profiling phase where each task is executed on both the CPU and GPU with different inputs. It is unclear, however, what happens if, at execution time, a task is executed with an input not used during profiling. Scheduling decisions are made locally whenever a kernel function is called. This may lead to suboptimal decisions due to a lack of knowledge about future tasks. Systems like StarPU or Harmony avoid this problem by making task executions asynchronous which allows them to schedule multiple tasks at once rather than one at a time.

Sun et al. (2012) built a runtime systems on top of OpenCL to provide a unified execution model for heterogeneous systems. They replace OpenCL's device-specific command queues with "work pools" that leave it up to a scheduler to select the most appropriate device for a task. The potential of this approach is demonstrated on the clSURF benchmark where they achieve significant performance improvements over only using the GPU. However, the mapping was done manually rather than by an automatic scheduling algorithm. It is thus not transferable to other benchmarks. Their system does, however, allow for implementing more sophisticated schedulers.

Scheduling in Multi-Programmed Systems

All approaches studied thus far are concerned with scheduling a single application. Jiménez et al. (2009), on the other hand, look into the problem of scheduling applications in a multi-programmed system. They randomly choose combinations of co-running applications whose tasks are scheduled by a common runtime. Similar to previous approaches the scheduler is based on past execution times of tasks on the different devices. On a system with n devices, the task is run on each device at the first n executions. Once the profiling data has been collected a list of "allowed" devices is built where a device is in the list if there is no other device that is significantly faster than it. Task are now scheduled only to devices on this list, assuming one of them is currently idle. If all devices are busy a simple round-robin strategy is used to assign the task to a device. This approach relies on online profiling of tasks to gather execution times on each device. In a multi-programmed environment, however, it may be difficult to obtain accurate profiling data, especially on the CPU where programs are sharing resources with co-running applications. This may lead to inaccurate data in practice and thus suboptimal scheduling decisions. As in all the approaches discussed above multiple versions of each task have to be provided by the user.

Gregg et al. (2010) also look at multi-programmed system. Whereas Jiménez et al. (2009) control all applications, they consider scheduling individual applications without knowledge of other applications. Their scheduler is based on information about the state of the system and historical performance data. If either both the CPU and GPU are idle or both are busy, the faster of them is selected. Otherwise a "contention factor" is considered, a constant describing the factor by which a task is slowed down if executed on a busy device. This factor is used to decide whether a task should be scheduled to the faster device even if it is busy and the slower device is idle. This heuristic is rather crude and does not necessarily reflect the trade-offs of that decision. It would also be difficult to implement this approach in practice because there is no easy way to determine if a GPU is busy or not. The authors currently assume on having this information available. Unlike previous approaches, Gregg et al. (2010) take advantage of OpenCL's capability to run code unmodified on different devices, thus requiring only a single code version of each task.

Pai et al. (2013) consider scheduling multiple concurrent kernels to the GPU. They observe that simply giving each kernel the resources it wants leads to no better performance than running the kernels one after another. They thus propose "elastic kernels" which allow for controlling the resource usage of individual kernels, such as the number of CUDA threads and thread blocks. By carefully selecting the resources given to concurrently running kernels performance can be improved over the default mapping. This work only considers scheduling kernels to the GPU and not to the CPU.

Improving Energy Efficiency

The goal of the large majority of task scheduling techniques is to reduce the *running time* of applications. Heterogeneous systems are attractive for this as they potentially deliver more performance than a homogeneous setup. Yet another advantage of heterogeneous systems, however, is their potential for being more *power efficient* than homogeneous systems (Kumar et al., 2003).

Liu et al. (2012) consider scheduling a set of tasks with individual deadlines to CPUs and GPUs. Their goal is to minimize power consumption while still meeting all of the application's deadlines. They apply dynamic voltage and frequency scaling (DVFS) to reduce a processor's voltage and frequency, thus saving energy but also slowing it down. Two mapping strategies were proposed: a static and a dynamic one. The authors assume that each task's worst case execution time is known, either through profiling or using worst-case execution time analysis (Wilhelm et al., 2008). In the static mapping scheme tasks are divided into "heavy" and "non-heavy" tasks, where a task is considered heavy if the load on its slower processor is greater than 1/2. The load of a task on a device is defined as the ratio of the task's running time on that device and the time available to complete the task, i. e. the difference between

its deadline and its arrival time. In each of the two sets the tasks are ordered by the "heterogeneity ratio". That is the ratio between the running time on the slower device and the running time on the faster device. Now the heavy tasks are assigned to their faster processor if possible, followed by the non-heavy tasks. If a task cannot be assigned to its faster processor it is assigned to the slower one. On each processor tasks are ordered using earliest-deadline first scheduling. After this initial assignment load balancing is performed to ensure that none of the processors process more work than the other. Finally, DVFS is applied by minimizing the voltage for each processor while still meeting all deadlines of tasks assigned to that processor. The dynamic mapping scheme augments the static one in two ways. Firstly it adapts processor voltage to actual running times of tasks, e. g.further reducing the voltage if applications finish earlier than expected. Secondly it deals with newly arriving tasks. If possible tasks are assigned to their faster processor. If, however, this would cause significant load imbalance and the new task has a high heterogeneity ratio, tasks with low heterogeneity ratios are swapped to make space for the new task. After each new assignment the processor's voltage is adjusted to ensure that all tasks finish in time. This paper considers a factor that is often neglected in scheduling, namely energy. In a scenario with task deadlines DVFS is very useful to reduce power consumption without sacrificing quality of service. However, the metrics used in this paper, such as heavy and non-heavy tasks, seem somewhat arbitrarily chosen. While the presented results are good no study into how they compare to an optimal schedule is performed.

3.2.2 *Mapping Tasks Across Devices*

The tasks being scheduled on GPU-based heterogeneous systems are data-parallel so as to fit the GPUs SIMD-like model of execution. It it thus possible to partition each task into several sub-tasks that can each be mapped to different devices. This way individual tasks can be distributed across devices rather than running on a single device only. Both static and dynamic approaches have been proposed. Dynamic approaches often rely on the *task farm* or *master-slave* method where the task is split into a number of chunks that are dynamically scheduled across the processors. The static schemes, on the other hand, split the work into exactly two chunks, one for the CPU and one for the GPU. This reduces the scheduling overhead but requires a model to determine the best work partitioning.

Static Mapping

Luk et al. (2009) introduce Qilin, a heterogeneous programming system containing a static scheduler for CPU-GPU systems. It is based on linear performance models that are built at execution time. The first time a task is being executed Qilin divides the input into two sub-tasks, one for the CPU and one for the GPU. Each sub-task is

further divided into chunks of different sizes that are executed on the corresponding device and their running times are measured. Given this information a linear model is fitted to the data on each device. The models are of the form $T = a + b \times N$ where T is the predicted running time and N is the input size. The next time the same task is being executed the models are used to compute the partitioning that minimizes the overall running time. The authors do not specify how to divide the inital task into sub-tasks. This could have an important impact on the accuracy of the model, especially for small input sizes. GPU execution involves a certain overhead which means that running times of small tasks are not necessarily indicative of running times of large tasks.

Jiang and Agrawal (2012) use linear performance models to partition map-reduce tasks on nodes in a distributed system. Initially, the task is distributed across nodes and in each node the assigned sub-task is partitioned between the CPU and GPU. Similar to Qilin (Luk et al., 2009) the first few executions of a task are used for profiling. Specifically, during the first execution the task is completely run on the CPU and during the second execution it is run on the GPU. Each time the running time is measured and this information is used to build a linear performance model for each device. Whereas Luk et al. (2009) use multiple executions with different input sizes for each device, Jiang and Agrawal (2012) only use a single run per device. They thus estimate constant overheads of CPU and GPU execution without further specifying how this information is obtained. Once a partitioning between the CPU and GPU has been computed, each sub-task is further split into chunks. On the CPU this serves to improve load balancing and on the GPU it helps to get around memory limitations. A similar mapping approach for map-reduce applications is proposed by Shirahata et al. (2010). Initially, each task is run on both the CPU and GPU and running times are measured. Given this data, an "acceleration factor" is computed that is used to determine the partitioning between the CPU and GPU.

Dynamic Mapping

Linderman et al. (2008) propose another framework, called Merge, for map-reduce computation on heterogeneous systems. As opposed to the previous approaches, however, it is distributing work across the processors dynamically rather than statically. In Merge, the user provides multiple implementations of the same map-reduce function. Each implementation is annotated with predicates describing constraints on when an implementation can be used. The predicates relate to input sizes, e. g. a sequential implementation should be used only for small input sizes, and architectures, e. g. to indicate GPU-specific implementations. During execution a task is recursively split into sub-tasks that are pushed to a task queue. If an implementation of that task exists and the corresponding device is available the task gets executed on the device. If multiple implementations can be used, the *most specific* one (with respect to

the predicates) is chosen. If no implementation is available the task is split and the sub-tasks are pushed to the queue. This mechanism ensures that the computation is distributed across all processors in a system with tasks being assigned to their most appropriate device. However, the runtime does not take the performance of the individual implementations into account. It schedules tasks to devices as soon as they are available even though it may be more beneficial to wait for a faster device to become availabe.

A different approach for dynamically mapping map-reduce applications is presented by Ravi et al. (2010). Their approach is based on a task-farm (or master-slave) model. A task is split into a number of chunks and initially one chunk is sent to each device. Once a device has finished processing its chunk it request another one until the entire task has been processed. The benefits of this approach are that it is simple to implement and does not require any off-line training or online profiling. There are, however, some pitfalls that have to be considered, such as choosing the right chunk size. As the authors demonstrate, the chunk size is crucial for performance. A small chunk size provides good load balancing but introduces lots of overhead. A large chunk size on the other hand reduces overheads but may lead to poor load balancing. Furthermore, good GPU performance can only be achieved with large chunks of work due to their highly-parallel architecture. The authors address this problems by assigning larger chunks of work to the GPU than to the CPU. In a follow-up paper Ravi and Agrawal (2011) also address the problem of choosing a good chunk size. They propose the use of a cost model to predict the optimal chunk size for an application. Their cost model, however, relies on knowing a task's running times on both the CPU and GPU which are being experimentally determined for each application. This defeats the advantage of having a dynamic model which in principle does not require off-line profiling.

Belviranli et al. (2013) propose another dynamic scheme for mapping tasks across processors in GPU-based systems. For each task their method executes in two phases. In the adaptive phase, online training is performed to compute the "computational weights", i. e. the performance ratio between the CPU and GPU. Small sub-tasks are executed on each device with their size gradually increasing. Once the computational weights have been accurately determined the completion phase is entered. In this phase the remaining work is divided into chunks whose sizes are determined using the computational weights. At first, large chunks are created to reduce scheduling overhead. Any left-over work is divided into smaller chunks to account for any load imbalance due to inaccurate computational weights. The authors clearly target only very large tasks where the overhead of the adaptive phase is negligible. For small and medium-sized tasks this approach does not seem suitable.

A very similar approach is proposed by Boyer et al. (2013). When a new task is executed, small chunks of increasing size are scheduled to each device. Using the running times on each device they work out how to optimally partition the remain-

ing work. The authors show how this scheme can be used to adapt to performance variability. This is demonstrated by changing the frequency of the GPU for different executions. They do not, however, show how their approach would perform if the frequency was varied while the task is being executed.

Mapping Accelerated OpenMP Programs

Scogland et al. (2012) introduce a new clause for executing accelerated OpenMP programs across CPUs and GPUs (see section 3.4.4 for a discussion on directive-based techniques for GPU programming). The `hetero` clause allows a user to specify how an accelerated region should be distributed across the CPU and GPU by choosing one of four scheduling strategies. The "static" strategy divides the work based on a ratio which is either user-defined or determined using estimations of the CPU and GPU peak performances. The "dynamic" scheduler initially partitions the work just like the static one. In subsequent executions of the same task, however, it adjusts the partitions based on the running times of previous executions. This scheduler is useful if a task is executed multiple times. If this is not the case the "split" scheduler can be used. It divides the tasks into equal-sized chunks and schedules each of them using the dynamic scheduler. This strategy works well if a task is only executed a small number of times but it is unsuitable if the task size is small. The "quick" scheduler is a hybrid of the split and dynamic schedulers. During the first execution of a task it runs a small chunk of the task using the split scheduler and the remaining chunk is partitioned according to those running times. For subsequent executions the dynamic strategy is used. This is very similar to previously discussed approaches (Luk et al., 2009; Belviranli et al., 2013; Boyer et al., 2013). The ideas behind these schedulers are not very novel. However, integrating them into a system such as accelerated OpenMP would provide a large group of users with helpful tools to efficiently execute code across processors in a heterogeneous system.

3.3 OPTIMISING GPGPU PROGRAMS

Writing GPGPU programs is a challenging task. The standard frameworks provided, such as CUDA and OpenCL, require a thorough understanding of heterogeneous architectures to achieve good performance. The two main issues to address are the tuning of kernel code for specific GPUs (Ryoo et al., 2008a; Hwu et al., 2009) and the management of data transfers between the CPU and GPU (Jablin et al., 2011). This section summaries the work on addressing these issues.

3.3.1 *Kernel Code Optimizations*

Both Ryoo et al. (2008a) and Hwu et al. (2009) provide an in-depth introduction to (NVIDIA) GPU architectures and a guide on how to optimize code for these processors. They study both the performance benefits of generic transformations, e. g. loop unrolling, loop tiling or data prefetching, and of GPU-specific optimizations, such as using local memory to reduce pressure on global memory bandwidth. While these studies help programmers understand the challenges of GPU computing it still requires a lot of effort to implement these transformation. The following approaches try to automate some of these transformation thus relieving the programmer of this burden.

Tuning Unoptimized GPU Kernels

CUDA-lite, proposed by Ueng et al. (2008), is a framework that automatically improves the performance of GPU kernels. The user writes a simple CUDA kernel without worrying about optimising the code. The tool then automatically transforms the code to improve global memory performance using the GPU's local memory. CUDA-lite requires the user to annotate which arrays to optimize. If the tool detects that memory coalescing of the annotated arrays can be achieved it generates a coalesced version of the code. Local memory is only used, however, to improve memory coalescing and other potential benefits, such as data reuse, are not considered.

Ryoo et al. (2008b,c) propose a guided search technique for finding good transformations for GPU kernels. The transformations they consider range from using local memory for data reuse and to improve global memory coalescing to traditional techniques such as loop unrolling and tiling. Instead of searching the entire optimization space they propose two techniques to prune unpromising configurations. "Threshold carving" eliminates all configurations that do not meet a certain criterion, e. g. that all memory accesses are coalesced. The remaining configurations are evaluated by executing them on the target hardware. This is useful if one particular performance factor, such as memory performance, is to be optimized. For cases where it is not obvious which factor should be optimized, the authors propose "tradeoff carving". Tradeoff carving takes multiple metrics into account and only retains the ones that lie on the Pareto-optimal curve. The authors demonstrate this technique by means of two metrics: "efficiency" and "utilization". Efficiency is the inverse of the total number of instructions being executed by a kernel. It is computed by static analysis of the code to estimate the number of instructions per work-item which is then multiplied by the number of work-items. Utilization refers to the utilization of compute resources on the GPU. It is an estimate of how many work-items will be able to run concurrently on the GPU. Using tradeoff carving the authors were able to prune 84%

of the configuration space on average while still achieving performance within 1% of the optimal configuration.

Baskaran et al. (2008) consider affine loop transformations to optimize GPU code. A large range of transformations are considered including the use of local memory to enable global memory coalescing and for data reuse, padding local arrays to avoid bank conflicts and loop tiling and unrolling. Optimal padding factors are determined by analysing the access patterns of local arrays. For each array all possible padding factors are evaluated and the one minimizing bank conflicts is chosen. The remaining parameters are determined by a guided search similar to Ryoo et al. (2008b,c). Using memory traffic estimation, unpromising configurations are pruned. All remaining configurations are executed and the fastest one is chosen. Affine loop analysis is a very powerful tool for automatic program optimization but its applicability is limited to loops with affine loop bounds and array indices. All benchmarks used in this study are from the linear algebra domain (matrix-vector product and matrix-matrix product). Performance results of benchmarks outside this domain are not presented.

Yang et al. (2010, 2012) present a compiler for GPU programs incorporating many of the optimizations mentioned above, including the use of local memory for data reuse and to achieve global memory coalescing. In addition, their compiler performs vectorization and considers merging work-items and work-groups that share data. The extent of merging work-items and the number of work-items per work-group is determined empirically by performing a search across a range of possible values. The authors also propose the use of machine description files to guide compiler optimizations such as vectorization.

Optimizing Memory Layouts

In the previously discussed approaches, memory coalescing was achieved through use of local memory. There is no other way to achieve this when only considering the kernel code. However, when also considering the host code of the application a much simpler method can be used to achieve memory coalescing, namely changing the data layout of arrays.

Che et al. (2011) use this idea to implement the Dymaxion API. This API allows the user to change the data layout of arrays while they are being copied to the GPU memory without having to substantially modify the code. To minimize the overhead of data remapping the data is divided into chunks and a chunk is sent to the GPU as soon as it has been remapped, even if other chunks still need to be processed. This effectively hides the overheads of remapping by overlapping it with the data transfer. As opposed to fully automatic approaches, this method relies on the user to understand if and when data remapping is beneficial for performance.

Dealing With Dynamic Irregularities

All approaches presented thus far tried to improve memory coalescing, a critical factor to achieve good performance on GPUs. They relied, however, on accesses being statically determinable. The following papers address the problem of memory coalescing for dynamic memory accesses, e. g. due to indirection. Furthermore, they tackle the issue of control flow divergence, where work-items in the warp (see section 2.1.1) take different paths through the kernel code.

Zhang et al. (2010) initially focussed on removing control flow divergence in GPU kernels. Control flow divergence commonly occurs when a conditional or a loop bound depends on data specific to a work-item and some of the values belonging to work-items in the same warp fulfil the condition and others do not. The basic idea to remove control flow divergence is to change the mapping of data to work-items so that all work-items in a warp take the same path through the code. Two methods to achieve this are proposed: reference indirection and data layout transformation. In the first method a new index array is created that contains the indices of the data elements to be accessed by the work-items. The problem with this approach is that data accesses to the original array are no longer necessarily coalesced. This problem can be overcome, however, by transforming the data layout of the array. The remapping of data can only be done at runtime when the data values are known. To minimize the overheads of remapping the authors overlap the remapping step with the execution of the GPU kernels.

In a follow-up paper, Zhang et al. (2011) extend this approach by also taking indirect memory accesses into account which typically lead to uncoalesced accesses. Similar techniques can be applied to improve the performance of memory accesses. A deeper analysis of the complexities and limitations of this approach is provided by Wu et al. (2013). They also present a proof showing that finding an optimal remapping of work-items to data is NP-complete.

Input-Aware Optimization

Typically static optimization approaches either ignore the impact of input sizes or they assume that it is static for a given application. Liu et al. (2009) show that the input size can have a significant impact on optimising GPU applications. They build a framework, called G-Adapt, that automatically finds good optimizations for different input sizes of an application. The approach is based on off-line training where a hill climbing search is performed to find good configurations for a range of training input sizes. This data is used to build an application-specific regression tree that predicts good optimizations given characteristics of the input, i. e.the size of the input. While input-sensitive optimization can be beneficial for performance, the presented approach requires an expensive training phase for each application without actually knowing whether the application requires input-sensitive tuning. The authors do

not discuss how this can be detected without having to first perform the expensive training step.

3.3.2 *Automatic Data Management*

In heterogeneous computing frameworks, such as OpenCL, it is left to the programmer to handle communication between devices. For complex programs it requires careful planning to ensure that data is kept coherent. Since data transfers can add significantly to the overheads of heterogeneous programming it is crucial to optimize communication in order to avoid unnecessary transfers. The following approaches aim to relieve the burden of data management from the user and provide fully automatic methods for efficiently handling data transfers.

Static Data Management

A fully static approach for data management is proposed by Amini et al. (2013). Their approach solely relies on static compiler analysis to determine when to transfer data between the CPU and GPU memories with the aim of transferring data to the GPU memory as *early* as possible while moving data back to the CPU memory as *late* as possible. They implemented this approach on top of a polyhedral compiler, thus limiting its applicability to affine programs.

Jablin et al. (2011) introduce CGCM, a CPU-GPU Communication Manager that works in conjunction with an automatic parallelising compiler. It is based on both static compiler passes and a runtime system to manage and optimize communication between the CPU memory and GPU memory. The compiler performs a liveness analysis for each GPU kernel and inserts calls to the runtime system's map and unmap functions before and after the kernel call, respectively. The runtime system keeps track of GPU memory allocations and transfers data between the CPU memory and GPU memory as necessary. While this ensures correctness it is often not optimal, e. g. when one kernel uses the output of another kernel the data does not have to be copied to the CPU memory between kernel calls. The authors address this issue by performing a "map promotion" pass which hoists mapping operations outside of loops or functions if it can prove that the corresponding data is not used by the CPU between kernel calls. To improve the applicability of map promotion two other compiler passes are introduced. "Alloca promotion" hoists local allocations up the call graph and "glue kernels" transforms short pieces of sequential code that prevent map promotion to single-threaded GPU functions.

Dynamic Data Management

CGCM uses type inference to classify pointers as either arrays of data values or arrays of pointers. It therefore cannot handle more complex data structures, e. g. recursive

data structures. This is addressed by DyManD, a dynamically managed data system presented by Jablin et al. (2012). The key idea of DyManD is to keep CPU and GPU versions of arrays at numerically equivalent addresses in the CPU and GPU memories. This is achieved by replacing the original versions of memory allocation functions such as `malloc` with DyManD-specific ones and turn global variables into dynamically allocated ones. Having both versions of the data at numerically equivalent addresses allows the runtime system to recursively copy data structures between memories without relying on static analysis.

Pai et al. (2012) propose an automatic data management scheme for X10 (Charles et al., 2005). It works on the granularity of "rails", X10's equivalent of arrays. For each rail the state (stale or not stale) of the copies in the CPU and GPU memories is tracked. When the rail is written by either the CPU or the GPU, the other device's copy is marked as stale. Before a device reads from a rail its copy is updated if it is stale. On the GPU these checks are performed before each kernel call, similar to CGCM (Jablin et al., 2011). The approach is different to CGCM, however, because it also inserts checks for data accesses by the CPU. Since performing these checks at every single read and write is very inefficient, they identify sets of rail accesses that only require a single coherence check for the entire set. Thus where CGCM copies data back to the CPU memory after one or several GPU kernel executions, this approach only triggers a copy just before the data is actually accessed on the CPU.

3.4 MAPPING HIGH-LEVEL LANGUAGES TO HETEROGENEOUS SYSTEMS

The previous section has discussed approaches that aim to simplify various aspects of GPU programming, e. g. kernel tuning or data management. They do, however, still require the user to write in a low-level programming language, such as OpenCL or CUDA. This section introduces a number of approaches that automatically generate low-level code from a high-level programming language, allowing the user to largely ignore GPU-specific tuning issues.

3.4.1 *C-Based Languages*

Lee et al. (2009) present a method for translating OpenMP programs (see section 2.2.1) to CUDA. Each parallel loop nest is translated to a CUDA kernel for execution on the GPU. A straightforward translation between the two languages does not always lead to good performance on the GPU because OpenMP programs are typically written with multi-core CPU execution in mind. The authors thus apply two optimizations to the original code before translating it to CUDA. The "parallel loop-swap" optimization performs loop interchange on nested loops to achieve memory coalescing on the GPU. The "loop-collapsing" technique merges a double-nested loop into a sin-

gle loop. This can enable memory coalescing and can avoid control flow divergence for irregular applications. In addition to these source-level optimizations some GPU-specific optimizations are performed during translation of the code to CUDA. These include the use of registers and local memory as a cache for frequently used data. They also perform data flow analysis to minimize data transfers. Apart from loop interchange no other transformations are applied to ensure memory coalescing, neither data layout transformations (Che et al., 2011) nor the use of local memory (Ueng et al., 2008; Ryoo et al., 2008c; Baskaran et al., 2008; Yang et al., 2012). As will be shown in chapter 6 these transformations are often crucial to achieve good performance when executing OpenMP code on GPUs. In the benchmarks chosen by the authors this problem does not arise however.

Baskaran et al. (2010) propose to generate CUDA code directly from C, without the need of user annotations as in the previous approach. However, their technique is restricted to affine loops because they rely on polyhedral analysis to discover parallelism. A range of optimizations are applied taken from the authors' previous paper on optimization techniques for GPUs (Baskaran et al., 2008). While the idea of this approach, going from sequential C code straight to the GPU, sounds desirable it is difficult to achieve in practice because many programs do not fit the affine computing model. The applicability of this technique is thus limited.

3.4.2 *Streaming Languages*

The next two papers use the StreamIt language (Thies et al., 2002) as a source language for GPU programming. A StreamIt program consists of a number of filters and communication channels between them. The communication channels describe how the output of one filter is consumed by another filter. In streaming applications a sequence (or stream) of data is passed through the filters thus creating a pipelined execution of the program.

Udupa et al. (2009) propose a software pipelining framework for mapping StreamIt programs to the GPU based on Integer Linear Programming (ILP). Each filter is mapped to a group of cores in the GPU so as to minimize the overall execution time. Profiling runs are used to determine the execution times of each individual filter and to find the optimal execution configuration, i. e. the number of work-items. Work-items running on the same group of cores execute the same filter for successive data items. For filters using more than one data item as input this leads to uncoalesced memory accesses if each work-item directly reads its data items from global memory. This issue is addressed by using local memory to load the entire data set required by all work-items into the local memory in a coalesced manner. This approach utilizes both the task-parallelism and the data-parallelism available in StreamIt programs by mapping it to groups of cores and cores within a group, respectively.

Hormati et al. (2011) use a slightly different approach in their "Sponge" compiler. Each filter in a StreamIt program is translated to a separate kernel so that data is processed in waves rather than as a pipeline. Filters are classified as either "High-Traffic" (HiT) or "Low-Traffic" (LoT). LoT filters do not access large amounts of data and can thus store their data in local memory to achieve coalesced accesses similar to the previous method. HiT filters, on the other hand, access too much data to store in local memory and thus access data directly in global memory. While this leads to uncoalesced accesses the authors say that by overlapping many accesses this cost can be amortized. If there are not enough work-items to fully utilize the GPU, Sponge creates so-called "helper threads" that do not perform any computation but help loading data into shared memory. Various other optimizations, such as prefetching and loop unrolling, are also applied.

StreamIt provides a high-level abstraction of programs that is inherently parallel. It is thus a good source language for targeting any kind of parallel hardware if the program fits this model. Many programs, however, cannot be naturally expressed as StreamIt programs, if at all, limiting the applicability of this approach to a small subset of problems.

3.4.3 *Pattern-Based Languages*

Dubach et al. (2012) introduce a compiler for "Lime", a Java-compatible language for heterogeneous computing. Lime programs consist of tasks and connections describing data dependencies between tasks. Unlike filters in StreamIt which are sequential, tasks in Lime can be parallel. Parallelism within a task is expressed as either a map or a reduce operation. The compiler maps parallel tasks to the GPU and sequential tasks to the CPU. A range of optimizations to exploit the GPU's memory hierarchy are applied. Furthermore the compiler is able to generate code using OpenCL's vector data types which leads to improved performance on certain GPU architectures. The Lime programming language allows users to express both task- and data-parallelism at a high level but they are restricted to data-parallel tasks that can be expressed as map or reduce operations. While this is arguably more convenient than the StreamIt programming model, further abstractions, e. g. stencil or scan operations, are required to support a wider range of applications.

A similar approach is presented by Sato and Iwasaki (2009). They use computational patterns, or skeletons (Cole, 1989), to express parallel operations in C programs. In contrast to Lime, communication between tasks is implicit in the inputs and outputs of skeleton instances. While they do not perform as many GPU-specific optimizations as Dubach et al. (2012) they *fuse* multiple tasks together if possible. Fusion is an important optimization that reduces temporary storage and kernel launch overheads. The only types of skeletons supported are map, reduce and zipwith (es-

sentially a map over two arrays instead of one). They thus suffer from the same limitations as Lime unless more types of skeletons will be supported.

3.4.4 *Directive-Based Languages for GPU Programming*

The goal of the approaches discussed thus far is to target GPUs from "generic" high-level, parallel programs that were not specifically written for GPUs. The programmer is not aware of the architecture being targeted and the burden of optimization falls entirely to the compiler and runtime system. Directive-based languages (Lee and Vetter, 2012) for GPU computing lie between these generic, high-level languages and low-level languages such as OpenCL or CUDA. While the programmer is aware of the GPU target and in charge of some optimizations, directive-based approaches use pragmas to annotate existing programs rather than requiring a rewrite in a low-level language.

Han and Abdelrahman (2009, 2011) present "hiCUDA", a directive-based language for GPU computing targeting NVIDIA GPUs. hiCUDA provides two sets of directives, one for computation and one for data. The directives for computation allow the user to specify kernels by annotating loop nests in a C program. Various clauses are provided to define the mapping of loop iterations to CUDA threads. The data directives specify the transfer of arrays between the CPU and GPU memories. Furthermore, there are directives for using local memory inside kernels. The authors have developed a compiler translating C programs with hiCUDA annotations to CUDA code. The generated code achieves good performance compared to hand-tuned code but it is still the user's responsibility to perform optimizations, e. g. using local memory and making sure memory accesses are coalesced.

A similar effort is presented by Lee and Eigenmann (2010, 2013). Their OpenMPC compiler translates OpenMP code with GPU-specific annotations to CUDA and is based on the author's prior work of generating CUDA code from OpenMP (Lee et al., 2009). Rather than relying on the compiler for optimizations, however, they provide a range of directives that allow the user to fine-tune the code. These include options for choosing the number of threads, caching variables in registers or local memory, loop interchange etc. Furthermore they provide tools for auto-tuning that allow for quickly searching the optimization space by pruning unpromising settings using static anaysis tools. Compared to hiCUDA, OpenMPC provides a much larger set of optimizations. The auto-tuning tools help inexperienced users to quickly navigate the optimization space and find good configurations. However, some critical optimizations, e. g. data layout transformations, are still left to the user.

OmpSs is an effort by Duran et al. (2011) to extend OpenMP with directives for supporting heterogeneous programming. They incorporate ideas from StarPU (Augonnet et al., 2009b, 2011) to map parallel tasks to the devices in a system (see section

3.2). OmpSs provides the user with clauses to specify the inputs and outputs of tasks. This allows the runtime to compute the tasks' inter-dependencies and automatically schedule them.

There are two commercial products using a directive-based approach for GPU computing: PGI by the Portland Group (Wolfe, 2010) and HMPP by CAPS Entreprise (Dolbeau et al., 2007). They both provide similar directives to hiCUDA and OpenMPC. The main difference is that they do not rely on the user to fully specify every aspect of the mapping to GPUs, such as data transfers and the mapping of loop iterations to threads. They do, however, allow the user to overwrite the compiler defaults for these options. This way they cater to both novices who want to quickly explore the potential of GPU computing for their applications as well as experts who want to fine-tune the performance of their programs.

The success of directive-based languages for GPU programming has let to the definition of the OpenACC standard (OpenACC, 2013) which is supported by a range of companies including Portland Group and CAPS Entreprise. Similar ideas will be integrated into the forthcoming release of OpenMP 4.0 (Stotzer et al., 2012) in order to provide a single framework for targeting both CPUs and GPUs.

Directive-based approaches for GPU programming present an easy upgrade path for legacy applications. They are more user-friendly than low-level languages such as OpenCL and CUDA and do not require the programmer to learn a new language. However, the user still needs to have an understanding of how GPUs work to achieve good performance. Furthermore, some high-level optimizations, e. g. concerning data layouts, cannot be performed by directive-based approaches.

Directive-based languages mainly support parallel-for operations, potentially with a reduction at the end. While similar to the map operations of pattern-based languages, parallel-for operations are more flexible. They do not, for example, dictate a relation between the iteration space and the data. On the other hand, pattern-based languages often provide more complex operations, such as filters or scans, which can be difficult to express using only parallel-for operations.

3.5 MACHINE LEARNING IN PROGRAM OPTIMIZATION

When optimising programs for a specific computer system there are many parameters that can be tuned, e. g. deciding which compiler transformations to apply and how to apply them. Typically, a compiler uses heuristics to make these decisions but these have to be hand-crafted by experts which is a time-consuming process. Furthermore heuristics can fail to capture the complex dependencies between transformations. An alternative is to use iterative compilation (Knijnenburg et al., 2002), where the optimization space is searched for an optimal configuration. Due to the large size of the

space, however, this approach is very time-consuming as it has to be repeated for each program.

Machine learning-based approaches have been proposed to overcome this issue. They rely on predictive models that, given a characterization of the program, predict the optimal configuration or at least predict a small set of candidate configurations that can be quickly evaluated. This section provides a brief overview of some of these approaches.

3.5.1 *Compiler Optimization*

One of the first applications of machine learning-based techniques in compilers was in the area of instruction scheduling. Moss et al. (1997) consider the problem of scheduling straight-line code using a greedy scheduler. At each step a new instruction is selected from the set of available instructions based on the partial schedule created thus far. The model built in this approach is based on triples (P, I_i, I_j) where P is the partial schedule and I_i and I_j are instructions. A triple is a positive example if selecting I_i over I_j given P leads to a better schedule and a negative example otherwise. Given a set of correctly labeled triples a binary classification model is built. Several models are explored, including decision trees and artificial neural networks. They all lead to similar results. The approach is able to match the performance of hand-crafted heuristics but rather than having to spend many hours crafting these heuristics they are built fully automatically.

Machine learning techniques have also been used for the loop unrolling transformation. Loop unrolling is a common transformation in compilers but determining whether or not it is benficial for performance is difficult. Furthermore the transformation is very sensitive to the unroll factor. Monsifrot et al. (2002) use decision trees to decide if loop unrolling should be applied to a particular loop which is characterized by static code features such as the number of memory accesses and the number of arithmetic expressions. On two different architectures their model achieves similar performance to hand-crafted heuristics. However, the authors do not address the problem of choosing unroll factors. This issue is addressed by Stephenson and Amarasinghe (2005) who model it as a multi-class classification problem where each class represents an unroll factor. Two models are used, namely nearest neighbour and Support Vector Machines (SVMs). Similar to the previous approach, loops are characterized by static code features. The authors are able to outperform the default compiler heuristic for loop unrolling using either of the models but the SVM achieves overall better results than the simpler nearest neighbour approach.

Instead of focusing on a single aspect of compiler optimizations, Agakov et al. (2006) consider selecting multiple transformations out of a set of available transformations. Iterative compilation has been used to tackle similar problems but its long

running time often makes it infeasible in practice. Agakov et al. tackle this issue by focusing the search of iterative compilation using machine learning methods. The programs are again characterized by static code features. To identify promising areas of the search space, programs are related to the training data using a nearest neighbour approach. Then either a random search is performed biased to the identified region or a genetic algorithm is used with the initial population being based on the region. This approach speeds up the search by an order of magnitude.

Dubach et al. (2009) completely do away with search and find good compiler settings immediately. Only one run of the program is needed to extract hardware performance counter data. Furthermore, their approach also takes hardware characteristics into account thus being able to predict across different systems without the need for re-training the model. Their approach achieves a performance of 67% of the performance of an iterative compilation approach without requiring multiple runs of the program.

3.5.2 *Mapping Parallelism to Multi-Core Systems*

The problem of mapping parallel programs to multi-core systems has been another area where machine learning techniques have been used. Deciding if and how to utilize the available parallelism in a program is a non-trivial task and the optimal mapping may change from system to system.

Curtis-Maury et al. (2006) consider the problem of selecting the optimal number of processors and the number of threads per processors in a multi-processor system with simultaneous multi-threading (SMT). During two test executions, with and without utilising SMT, the instructions-per-cycle (IPC) are measured and hardware performance counters are read. This information is passed to a regression-based model which predicts the thread mapping that achieves the best power-performance trade-off. Their approach has a high accuracy in finding the optimal configuration and reduces the number of profiling runs required over other, search-based techniques.

A similar approach is presented by Wang and O'Boyle (2009) whose goal is to select the optimal number of threads and OpenMP scheduling policy for parallel loops. Two models are used to solve the two problems individually. The number of threads is selected by evaluating an artificial neural network which predicts the speed-ups of different thread configurations and selecting the one predicted to achieve the highest speed-up. A Support Vector Machine is used to select the optimal scheduling policy. The input features for the models contain both static program features, e. g. the number of instructions, and dynamic features, e. g. cache miss rates. On both a multi-core Intel and an IBM Cell platform they achieve near-optimal performance. A similar technique is used by Tournavitis et al. (2009) in their auto-parallelising compiler to decide whether or not a parallel loop candidate is profitable to parallelize.

Wang and O'Boyle (2010) also use machine learning techniques to map StreamIt programs to multi-core processors. Their goal is to find the optimal program partitioning given features of the original partitioning. The nearest-neighbour technique is used to find the program in the training set that is most similar to the new program. This program's partitioning is then used as the partitioning for the new program. With this technique they are able to significantly outperform a comparable analytical scheme.

3.6 SUMMARY

This chapter has provided, to the best of the author's knowledge, a comprehensive review of prior related work in the various areas touched upon in this thesis. It has covered task mapping for heterogeneous systems, from early theoretical work to more recent approaches for GPU-based systems. Various methods for optimizing GPU programs have been discussed, including methods for tuning GPU kernel code and automatic data management. Furthermore, techniques for targeting heterogeneous systems from high-level languages have been presented. Finally, the chapter reviewed applications of machine learning methods to program optimization.

Starting with the following chapter the contributions of this thesis are discussed in more detail.

4

A STATIC TASK PARTITIONING APPROACH FOR HETEROGENEOUS SYSTEMS USING OPENCL

This chapter presents a machine learning-based method for statically partitioning OpenCL kernels across CPUs and GPUs. A predictive model is created which determines the optimal partitioning using static code features extracted from the OpenCL kernel code.

Section 4.2 showcases the potential benefits of partitioning the execution of OpenCL kernels across CPUs and GPUs. Section 4.3 describes how a machine-learning based model can be built for predicting the optimal partitioning. The evaluation methodology is explained in section 4.4 followed by an analysis of the results in section 4.5. The chapter concludes with some final remarks in section 4.6.

4.1 INTRODUCTION

Heterogeneous computing systems have become increasingly popular due to their potential of delivering high performance at relatively low energy costs (Khokhar et al., 1993; Kumar et al., 2005). However, these benefits come at the cost of increased complexity, e. g. determining the most suitable device for a given task. On GPU-based systems tasks are typically data-parallel because of the highly parallel architecture of GPUs as described in section 2.1.2. Data-parallel tasks can often be easily split into smaller sub-tasks and distributed across multiple devices. If the work is efficiently balanced across the available devices, performance can be significantly improved. Finding the optimal partitioning, however, is non-trivial.

Current frameworks for heterogeneous computing, such as OpenCL, rely on the programmer to map tasks to devices (see section 2.2.2). The optimal mapping is hardware-dependent, however, and changes from system to system. An automatic approach for partitioning work across devices is thus desirable. Several efforts have been made in this direction: Qilin (Luk et al., 2009) relies on extensive off-line profiling to create a performance model for each task on each device. This information is used to calculate a good partitioning across the devices. However, the initial profiling phase can be prohibitive in many situations. Ravi et al. (2010) develop a purely dynamic approach that divides a task into chunks that are distributed across processors in a task-farm manner. While this eliminates profiling, it incurs communication and other overheads.

The approach to partitioning data-parallel OpenCL tasks described in this chapter is a *purely static* approach. There is no profiling of the target program and the run-

time overhead of dynamic schemes is avoided. In this method static analysis is used to extract code features from OpenCL programs. Given this information, the system determines the best partitioning across the CPU and GPU in a system and divides the task into two chunks; one for the CPU and one for the GPU. Deriving the optimal partitioning from a program's features is a difficult problem and depends heavily on the characteristics of the system. The approach presented here therefore relies on machine-learning techniques to *automatically* build a model that maps code features to partitions. Because the process is entirely automatic, it is easily *portable* across different systems and implementations. When either change, the learning procedure is simply re-run without human intervention in building the model.

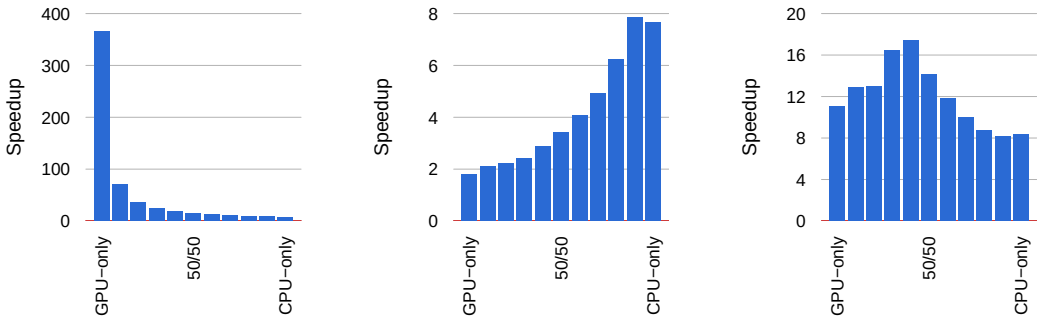
4.2 MOTIVATION

Determining the right mapping for a task is crucial to achieve good performance on heterogeneous architectures. This section illustrates this point by examining the performance of three OpenCL programs, each of which needs a different partitioning to achieve its best performance.

Figure 4.1 shows the speedup of three OpenCL programs with different mappings over single-core execution on a CPU. These programs were chosen from a large set of benchmarks studied in this chapter, as described in section 4.4.1, to represent classes of programs with very different performance characteristics. The specification of our system is also provided in section 4.4.1. The x-axis shows how much of the program's workload is executed on each device, i.e. the leftmost bar shows the speedup of GPU-only execution, one bar to the right shows the execution with 90% of work on the GPU and 10% on the CPUs and so on.

For the `coulombic potential` program (figure 4.1a), a GPU-only execution achieves by far the best performance. Scheduling an amount of work as small as 10% to the CPUs leads to a slowdown of more than 5 times and this value increases if more work is mapped to the CPUs. For these types of programs it is absolutely vital to know ahead of time what the optimal mapping is, because a small mistake is going to be very costly.

The `matrix-vector multiplication` program exhibits an entirely different behaviour (see figure 4.1b). The highest speedup is observed when 90% of the work is scheduled to the CPUs and only 10% to the GPU. The amount of computation per data item is fairly small and therefore the overhead of transferring data between main memory and the GPU's memory is not worthwhile for large parts of the computation. The `convolution` program in figure 4.1c shows yet another different behaviour: A roughly even partitioning of work between CPUs and GPU leads to the best speedup. Unlike the other cases, neither a GPU-only nor a CPU-only execution would achieve good performance.



(a) `coulombic potential` (b) `matrix-vector mult.` (c) `convolution`

Figure 4.1: The speedup over single-core performance of three representative OpenCL programs with different partitions. The significant variations demonstrate the need for program-specific mappings.

As these programs have shown, a partitioning scheme that takes program characteristics into account is necessary to achieve good performance on heterogeneous systems. Different programs need different mappings and for some of them making a small mistake means that large potential speedups are missed. As OpenCL is a fairly new framework, compilers are likely to improve in the near future. The CPU implementation, in particular, seems to have much room for improvement.

The next section describes a static partitioning approach based on static program code structure and machine learning. By using machine learning methods, this approach is *portable* across systems as well as implementations; a highly desirable property as program performance is likely to change across heterogeneous architectures and as OpenCL tools mature.

4.3 PARTITIONING DATA-PARALLEL TASKS

The approach presented in this chapter uses machine learning to predict the optimal partitioning for an OpenCL program. An overview of using machine learning models for program optimization has been given in section 2.4. The model in this chapter is solely based on compiler analysis of the program structure. Using the static analysis tool presented in section 2.4.1.1 a program is characterized as a fixed vector of real values. These form the input features of the model. The model's target is the optimal partitioning for a program across CPUs and the GPU.

This section describes which code features were used and how the machine learning-based model was built and then used to predict the optimal partitioning for any OpenCL program. Instead of relying on a single prediction model, hierarchical classification (Bishop, 2006) is used where a hierarchy of models is evaluated to find the answer. The individual models are instances of Support Vector Machines (SVMs) as introduced in section 2.3.3.1.

4.3.1 *Static Code Features*

The partitioning method introduced in this chapter is entirely based on static code features eliminating the need for expensive off-line profiling (Luk et al., 2009) and also avoiding the pitfalls of dynamic techniques (Ravi et al., 2010). However, the features need to carry enough information to characterize the behaviour of OpenCL programs. The feature extraction tool was introduced in section 2.4.1.1 which also provides a list of all code features. Instead of using all features as they are extracted by the tool, some of them are combined to provide more meaningful information.

Memory accesses to global memory can be coalesced, for example when adjacent work-items access adjacent memory locations. In this case multiple memory transfers can be coalesced into a single access increasing the overall memory bandwidth (see section 2.1.2). The feature extraction tool outputs the total number of coalesced memory accesses. Rather than using this number directly the *percentage* of global memory accesses that are coalesced is computed and used as a feature. This combined feature provides useful information as it immediately describes whether the majority of memory accesses are coalesced or not. Since GPU performance is very sensitive to memory coalescing this feature is crucial in determining the best partitioning between CPUs and the GPU.

The full list of static code features is shown in table 4.1. As indicated in the table, features describing absolute values are normalized. By multiplying the value by the number of work-items the *total* number of operations is computed for this program execution. Since a machine-learning model will not be able to relate two similar programs that have been executed with different input sizes, the total number of operations is divided by the data transfer size, to represent the number of operations *per data item*. In other words the normalized features are computed as

$$\text{operations in program code} \times \frac{\text{number of work-items}}{\text{data transfer size}}$$

Before the features are passed to the model, principal component analysis (PCA, see section 2.3.2.2) is applied to reduce the dimensionality of the feature space and normalize the data. Section 4.3.2.2 provides more details on this process.

The features describe both the computation and the memory operations of the kernel. First, it is important to describe the type and amount of computations (features 1-5). Some math operations, such as sine and cosine, can be mapped to special function units on some GPUs but may need to be emulated on CPUs, for example. Barriers (feature 6) may also cause different costs on different architectures.

Second, memory operations (features 7-10) are important to consider. Depending on the architecture and type of memory the cost of memory accesses may vary. Accessing local memory on GPUs, for example, is cheap because it is mapped to small

	Static Code Feature	cp	mvm	conv
1	int operations (norm.)	31.6	0.6	28.8
2	int4 operations (norm.)	0	0	0
3	float operations (norm.)	1,593.5	0.5	4.25
4	float4 operations (norm.)	0	0	0
5	intrinsic math operations (norm.)	249.9	0	0
6	barriers (norm.)	0	0.012	0.031
7	memory accesses (norm.)	0.125	0.5	2.6
8	% local memory accesses	0	0.04	0.88
9	% coalesced memory accesses	1	0.996	0
10	compute-memory ratio	15004	2.1	105.9
11	data transfer size (in bytes)	134,249,726	67,141,632	134,217,796
12	computation per data transfer	1,875	1.6	35.7
13	number of work-items	2,097,152	4,096	4,194,304

Table 4.1: List of static code features used to characterize OpenCL programs and the corresponding values of the three example programs.

on-chip memory. On CPUs, however, local and global memory both get mapped to the same memory space.

Discrete GPUs have a physically separate memory space and any data used during program execution needs to be copied to the GPU. The cost of data transfers between the memories is thus important. Features 11 and 12 capture the amount of memory to be transferred and how it compares to the amount of computation performed on the data. Lastly, feature 13 captures the overall size of the problem.

4.3.1.1 *Examples*

Table 4.1 shows the feature vectors for the example benchmarks introduced in section 4.2. The "computation to data transfer" ratio (feature 12), for example, is 1875 for the `coulombic potential` program, which is significantly higher than the value for `convolution` (35.7) and `matrix-vector multi- plication` (1.6). The number of compute operations (features 1-5) and the ratio between compute- and memory-operations (feature 10) is also higher for `coulombic potential` compared to the others.

These differences in input features reflect the different behaviours shown in figure 4.1. The optimal performance for the `coulombic potential` benchmark is achieved with GPU-only execution, because of the large number of compute-operations and

the relatively small data transfer overhead. For the `matrix-vector multiplication`, on the other hand, very few operations are performed for each data item and the data transfer costs undo any potential speedups the GPU may offer. The feature values of the `convolution` benchmark are in-between the values of the other two programs. This explains why a more balanced work distribution is beneficial.

4.3.2 *Building the Predictor*

Building a machine learning-based model involves the collection of training data which is used to fit the model to the problem at hand. In this chapter, the training data consists of static code features of other OpenCL programs and the optimal partitioning of the corresponding program. This allows for creating a model that maps program code features to the program's optimal partitioning. Rather than relying on an individual predictor, several models are combined to form a hierarchical classification model (Bishop, 2006).

4.3.2.1 *Collecting Training Data*

The training data for the model is divided into static code features (as described in section 4.3.1) and the optimal partitioning for the corresponding OpenCL program. The former will be the input (or features) for the model, whereas the latter is the output (or target) of the model.

Each program is run with varying partitionings, namely all work on the CPU, 90% of work on the CPU and the remaining 10% on the GPU, 80% on the CPU and 20% on the GPU and so on. The partitioning with the shortest run-time is selected as an estimate of the optimal work partitioning for the program.

4.3.2.2 *Two-Level Predictor*

According to the observations made in section 4.2 OpenCL programs can be loosely divided into three categories, namely programs that achieve the best performance when

(1) executed on GPU only

(2) executed on CPUs only

(3) partitioned and distributed over GPU and CPUs

Getting the partitioning right is especially vital for programs in category 1. Not mapping all the work on the GPU leads to significant slowdowns (see Fig. 4.1a). Similarly (though less drastic) for programs in category 2: If it is not worth copying the data back and forth to the GPU, one needs to make sure that the work is mapped to the CPU and any data transfer overhead is avoided.

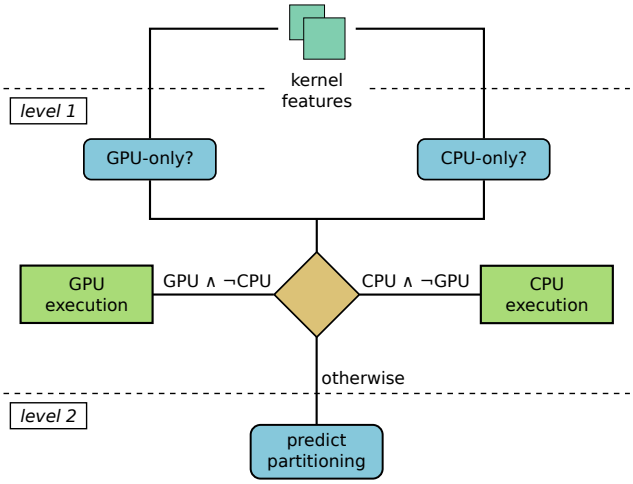


Figure 4.2: Overview of our prediction approach. Programs that should be entirely mapped to the GPU or to the CPUs are filtered out in the first level, while the second level handles programs that cannot be classified in level 1.

Therefore a *hierarchy* of predictors is used to model this problem. This approach is known as hierarchical classification (Bishop, 2006). In the first level, programs from categories 1 and 2 are filtered out and mapped to the GPU or the CPUs, respectively. The remaining programs are mapped according to a third predictor in level 2 (see Fig. 4.2). The kernel features are reduced to two and eleven principal components using PCA for the first- and second-level predictors, respectively.

Formally, input features are mapped to one out of 11 classes, where class 0 represents GPU-only execution and class 10 CPU-only execution. Let gpu and cpu be the first-level predictors and mix the second-level predictor. The hierarchical model can be described as

$$\text{prediction}(x) = \begin{cases} 0 & \text{if } gpu(x) \text{ and } \neg cpu(x) \\ 10 & \text{if } cpu(x) \text{ and } \neg gpu(x) \\ mix(x) & \text{otherwise} \end{cases}$$

LEVEL 1 PREDICTORS Focusing only on the extremes of the partitioning spectrum, the models in the first stage of the prediction are simple, but highly accurate (see section 4.5.2). One model predicts whether or not a program should be executed on the GPU only (category 1), while the other determines if a task is entirely mapped to CPUs (category 2). These "one-vs-all" classifiers (Rifkin and Klautau, 2004) are implemented as binary classifiers based on a support vector machine (SVM) with a linear kernel (see section 2.3.3.1).

LEVEL 2 PREDICTOR If the first level of predictors does not lead to a conclusion, the program is passed on to another predictor. This one is more complex, because it needs to map a program to one out of the 11 classes determined during training. Again, an SVM-based model is used but this time a radial basis function kernel is deployed to account for the increased complexity of this problem.

Whereas for the stage-1 models all of the available training data were used, only data from category 3 programs are used to train the predictor in the second level. This allows for the predictor to focus on programs whose optimal partitioning is likely to be neither CPU- nor GPU-only.

4.3.3 *Deployment*

At compile time, the program code is analyzed and code features are extracted. Because the model's input features incorporate the task's input size, the prediction cannot be made just yet. However, at run-time the input size is known and together with the previously extracted code features is passed to the model. The model's output is the optimal partitioning for this program and input size. This is used to partition the program between multiple devices. In OpenCL this can be easily done by launching a subset of the work-groups on one device and launching the remaining work-groups on the other. Due to OpenCL's model of computation no dependencies between work-groups are allowed (see section 2.2.2).

Due to the separate memory spaces of the devices, partitioning the computation also requires partitioning the data that is being read and written by the kernel. Working out which data needs to be mapped to which device is non-trivial. In this chapter it is assumed that this information has been provided externally (either by a static analysis tool or by the programmer).

Although the prediction is done at run-time, the overhead is negligible as it only takes in the order of microseconds to evaluate the models, the cost of which is included in our later results.

EXAMPLES When passing the features for the `coulombic potential` program as shown in figure 4.1a into the first-level predictors, there is a positive classification from the "GPU-only" predictor and a negative one from the "CPU-only" model. All of the computation is therefore immediately mapped to the GPU without evaluating the second level predictor. This leads to the optimal performance for this program with respect to the oracle.

For the `matrix-vector multiplication` program it is the other way around, i. e. the computation is entirely mapped to the CPU. Figure 4.1b shows that while this is not the optimal partitioning it still achieves 98% of the optimal performance.

	CPU	GPU
Architecture	2x Intel Xeon E5530	ATI Radeon HD 5970
Core Clock	2.4 GHz	725 MHz
Core Count	8 (16 w/ HyperThreading)	1600
Memory Size	24 GB	1 GB
Peak Performance	76.8 GFLOPS	928 GFLOPS
Compiler	GCC 4.4.1 w/ "-O3"	
OS	Ubuntu 9.10 64-bit	
OpenCL	ATI Stream SDK v2.01	

Table 4.2: Experimental Setup

With the `convolution` program, there is a negative classification from both first-level predictors. The program is thus passed on to the second-level predictor which predicts that 60% of the work should be mapped to the GPU and the remaining 40% to the CPUs. According to figure 4.1c this is the optimal mapping.

4.4 METHODOLOGY

This section describes the setup of the evaluation, including the platform and benchmarks used. It further describes other mapping approaches against which the method is compared.

4.4.1 *Experimental Setup*

PLATFORM All experiments were carried out on a heterogeneous computer comprising two quad-core CPUs with Intel HyperThreading and an ATI Radeon HD 5970 GPU. Table 4.2 shows more detailed information on the system as well as the software used. When the GPU was used, one CPU core was dedicated to managing the GPU. This has shown to be beneficial and is in line with observations made by Luk et al. (2009). Each experiment was repeated 20 times and the average execution time was recorded.

BENCHMARKS In total 47 different OpenCL kernels were used, collected from various benchmark suites: SHOC (Danalis et al., 2010), Parboil[1] (of Illinois at Urbana-

1 The Parboil benchmark suite only contains CUDA programs. We therefore translated the benchmarks from CUDA to OpenCL. The OpenCL source code can be found at `http://homepages.inf.ed.ac.uk/s0898672/opencl/benchmarks.html`

Champaign, 2013), NVIDIA CUDA SDK (NVIDIA Corp., 2013) and ATI Stream SDK (AMD/ATI, 2013). The full list of kernels is shown in tables 4.3 and 4.4, divided into the three categories mentioned above. When presenting the results in section 4.5 the benchmarks will be similarly divided.

By varying the programs' input sizes, a total of 220 experiments were conducted. The complete list of kernels and inputs is shown in appendix A. To evaluate the model the standard approach of cross-validation (see section 2.5) was used, which has the critical property that when evaluating the model on a certain program, no training data from this program was used to build the model.

4.4.2 *Evaluation Methodology*

We compare our approach to an "oracle", which provides an estimate of the upper bound performance (see section 2.5). To determine the oracle performance we tried all 11 partitions on the target program and selected the one with the lowest execution time. It may be possible that a partition that is not a multiple of 10% gives an even better speedup and thus our oracle is only an approximation.

We further evaluate two default strategies: "CPU-only" and "GPU-only" simply map the entire work to the CPUs or to the GPU, respectively. These are two very primitive methods that serve as a lower bound in the sense that any partitioning scheme should (on average) beat them to prove itself useful.

The fourth method we compare our approach against is a dynamic mapping scheme similar to what is presented by Ravi et al. (2010). They break up the work into a certain number of chunks. Initially, one chunk is sent to the GPU and one to the CPU. When either of them finishes, a new chunk is requested until the work is complete. We searched for the best number of chunks to divide the work into and found that using 8 chunks leads to the best overall performance on the set of kernels used in this chapter, providing the right balance between scheduling overhead and flexibility.

The performance of each mapping technique is evaluated by computing the speedup over single-core execution. To collect the single-core run-times we used the same OpenCL code, but instructed the OpenCL run-time to only use one CPU core. This may underestimate the performance of a sequential version as it is likely to be faster than the parallel OpenCL code on a single core. However, as this value is used solely as a baseline to compare speedups of the competing techniques, it is appropriate for our purposes.

4.5 RESULTS

This section presents the performance results of various mapping techniques. The predictive modeling approach introduced in this chapter is compared against two

Suite	Benchmark	Kernel
ATI Stream SDK	AES	encrypt
ATI Stream SDK	AES	decrypt
ATI Stream SDK	BoxFilter	horizontalLocal
ATI Stream SDK	MonteCarloAsian	calPriceVega
ATI Stream SDK	NBody	nbody_sim
SHOC	FFT	ifft1D
SHOC	MD	accel
SHOC	scan	scan
SHOC	sgemm	sgemmNT
NVIDIA CUDA SDK	MatrixMul	matrixMul
Parboil	cp	cuenergy
Parboil	mri-fhd	computeFH
Parboil	mri-q	computeQ
ATI Stream SDK	Histogram	histogram256
ATI Stream SDK	HistogramAtomics	histogramGlobal
NVIDIA CUDA SDK	Blackscholes	blackscholes
NVIDIA CUDA SDK	MatVecMul	matVecMulUncoalescedo
NVIDIA CUDA SDK	MatVecMul	matVecMulUncoalesced1
NVIDIA CUDA SDK	MatVecMul	matVecMulCoalescedo
NVIDIA CUDA SDK	MatVecMul	matVecMulCoalesced1
NVIDIA CUDA SDK	MatVecMul	matVecMulCoalesced2

Table 4.3: First half of the 47 OpenCL kernels used in this chapter divided by their relative performance on the CPUs and the GPU. For the first set of kernels the GPU is significantly faster than the CPU while it is the other way around for the second set of kernels.

Suite	Benchmark	Kernel
ATI Stream SDK	Blackscholes	blackscholes
ATI Stream SDK	BoxFilter	horizontal
ATI Stream SDK	BoxFilter	vertical
ATI Stream SDK	DCT	DCT
ATI Stream SDK	mandelbrot	mandelbrot
ATI Stream SDK	MatrixMul	matrixMul
ATI Stream SDK	MatrixMul	matrixMul_local
ATI Stream SDK	MatrixMul	matrixMul_local2
SHOC	FFT	fft1D
SHOC	scan	uniform
SHOC	sgemm	sgemmNN
NVIDIA CUDA SDK	ConvolutionSeparable	convolutionColumns
NVIDIA CUDA SDK	ConvolutionSeparable	convolutionRows
ATI Stream SDK	BinarySearch	binarySearch
ATI Stream SDK	BoxFilter	boxFilter
ATI Stream SDK	BoxFilter	horizontalSAT0
ATI Stream SDK	BoxFilter	horizontalSAT
ATI Stream SDK	BoxFilter	verticalSAT
ATI Stream SDK	HistogramAtomics	histogramLocal
ATI Stream SDK	MersenneTwister	gaussianRand
SHOC	MD	applyBoundary
SHOC	MD	updateVelocities
SHOC	MD	updateCoordinates
NVIDIA CUDA SDK	DotProduct	dotProduct
Parboil	mri-fhd	computeRhoPhi
Parboil	mri-q	computePhiMag

Table 4.4: Second half of the 47 OpenCL kernels used in this chapter. For these kernels the optimal performance is achieved with a fairly balanced partitioning between the devices.

static default strategies, namely "CPU-only" and "GPU-only", as well as with the dynamic scheduling method described in section 4.4.2. The performance achievable with an optimal partitioning is also presented to compare the approaches to the maximal available speed-ups. This is followed by an evaluation of the accuracy of the various predictors in the hierarchical model.

4.5.1 *Speedup over Single-core Performance*

The performance of the different partitioning schemes is evaluated with respect to the running time on a single CPU core. Because of the large number of experiments the programs are divided according to the three categories described in section 4.3.2.2: Figure 4.3 shows programs where a "GPU-only" mapping achieves more than 90% of the optimal performance, whereas figure 4.4 shows programs where "CPU-only" achieves more than 80% of the optimum. The remaining programs are shown in figure 4.5. This not only helps in understanding the results but also improves readability, as programs from different categories often have large differences in speed-up.

GPU-FRIENDLY BENCHMARKS Figure 4.3 shows the performance of the various techniques on OpenCL programs of category 1, i.e. programs that achieve the best speed-ups when executed on the GPU only. Unsurprisingly, the static "GPU-only" policy achieves near-optimal performance (compare to the right-most "oracle" bars). On these benchmarks this leads to an average speed-up of 112 over single-core performance. Similarly obvious is that the "CPU-only" method clearly loses on these benchmarks, only achieving an average speed-up of 8. The results for the dynamic approach are slightly better: because the GPU is much faster than the CPU on these programs, the majority of work will be scheduled to the GPU. However, some of the work is always scheduled to the CPUs which leads to significant slowdowns for most of the benchmarks when compared to the maximum performance. Overall, the dynamic scheduler achieves a speed-up of 49. Our prediction approach, on the other hand, classifies almost all programs correctly and therefore achieves almost the same performance as the "oracle" scheduler with 113 times of the single-core performance.

In the predictive model the majority of programs are filtered out by the first-level "GPU-only" predictor. Only very few are passed through to the next level. For those cases the second-level predictor makes the right decision to schedule the work to the GPU. More detailed information on the accuracy of the model's predictors will be shown in section 4.5.2.

CPU-FRIENDLY BENCHMARKS The performance of the different partitioning schemes on category 2 programs is shown in figure 4.4. This time around the static "CPU-only" policy achieves near-optimal performance. The average speed-up equates to

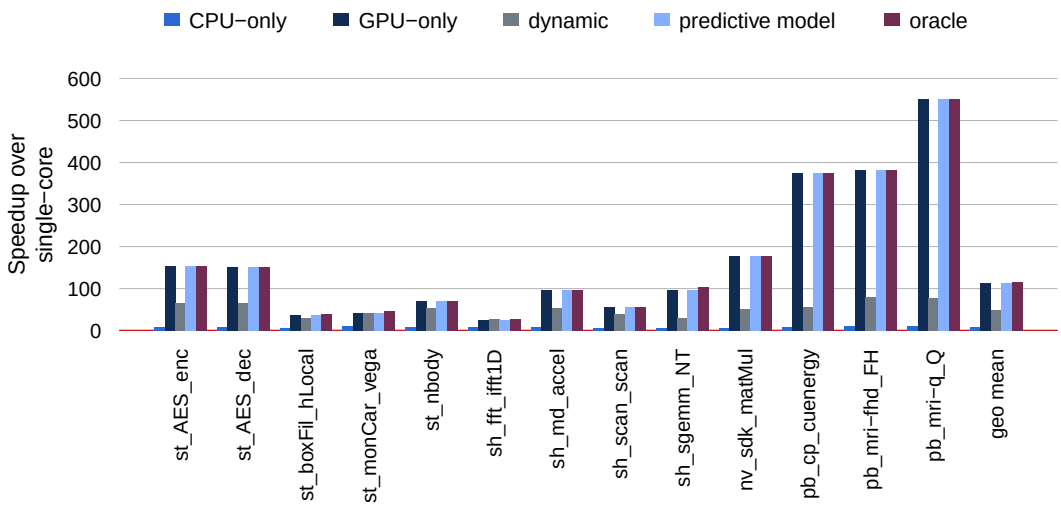


Figure 4.3: Performance of those applications where the best speed-up is achieved using only the GPU. The predictive model leads to near-optimal performance compared to the oracle and a speed-up of 2.3 over the dynamic scheme.

6.12, only marginally below the oracle performance of 6.36. Unsurprisingly the "GPU-only" method is worst for all programs, most of the time because shipping the data between main memory and GPU memory is not feasible. The average speed-up of "GPU-only" is 1.05, i. e. no significant improvement over single-core execution is observed. Again, the dynamic mapping method comes second to last. The overhead of having many chunks and sending data to the GPU is simply too big to achieve good performance on these programs and leads to a speed-up of only 2.15. The predictive model comes close to the optimal performance. With an average speed-up of 4.81 it is slower than the "CPU-only" policy on the CPU-friendly benchmarks, but significantly better than the dynamic scheme. This again shows the model's high accuracy for partitioning OpenCL programs. Just as with the GPU-friendly programs, most of the CPU-friendly programs are filtered out in stage 1 of the prediction process. The few remaining programs are accurately mapped by the second-level predictor. For more detailed information on the predictors' accuracies see section 4.5.2.

REMAINING BENCHMARKS Performance results for all the remaining benchmarks are shown in figure 4.5. The programs are grouped according to the maximum available speed-up, i. e. programs in figure 4.5a achieve a larger speed-up than programs in figure 4.5b, to improve readability.

Both non-adaptive policies, "CPU-only" and "GPU-only", do not do very well on most of these benchmarks, because the optimal performance is achieved when the work is distributed across both devices. On average, the "GPU-only" mapping achieves higher speed-ups (6.26 compared to 4.59), because the "CPU-only" policy misses out on potentially high speed-ups as shown in figure 4.5a. The dynamic

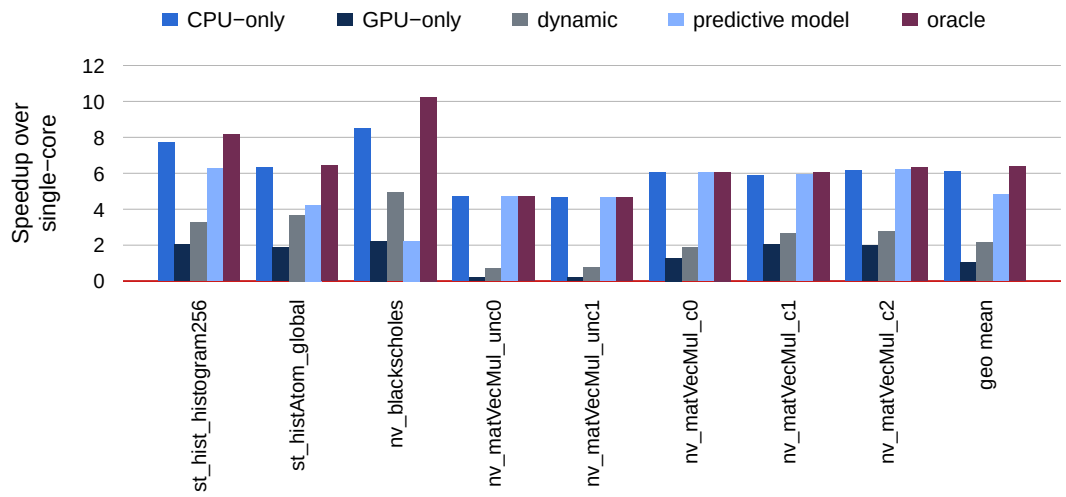


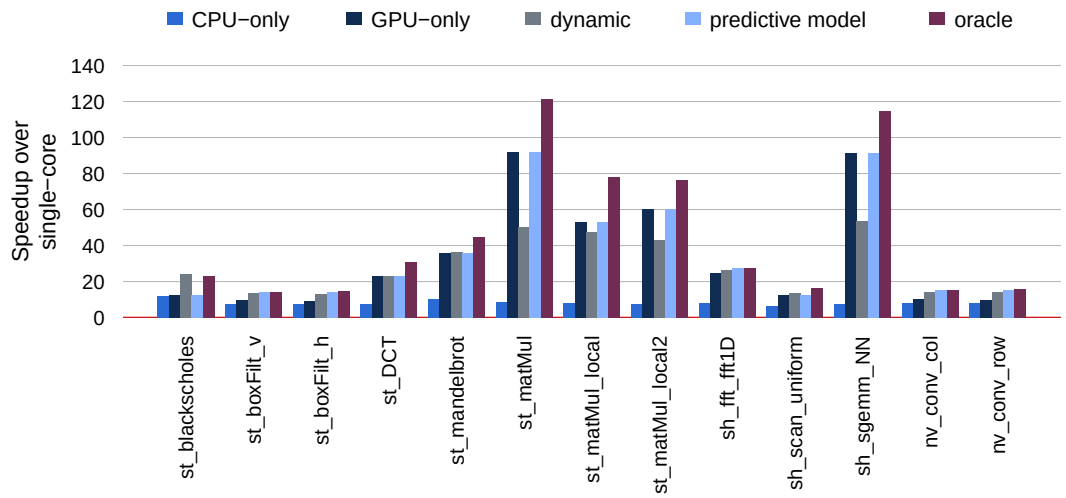
Figure 4.4: Performance of those applications where using only the CPU leads to almost optimal speed-up. The predictive model achieves a speed-up of 2.2 over the dynamic scheme.

scheme does reasonably well and achieves near-optimal performance for some benchmarks and an average speed-up of 8.00. However, all schemes are outperformed by our prediction approach which achieves a speed-up of 9.31 on average. Hence, even though the dynamic scheme shows its potential on these kind of programs, it is still outperformed by the predictive model due to reduced scheduling overhead and more accurate work distribution.

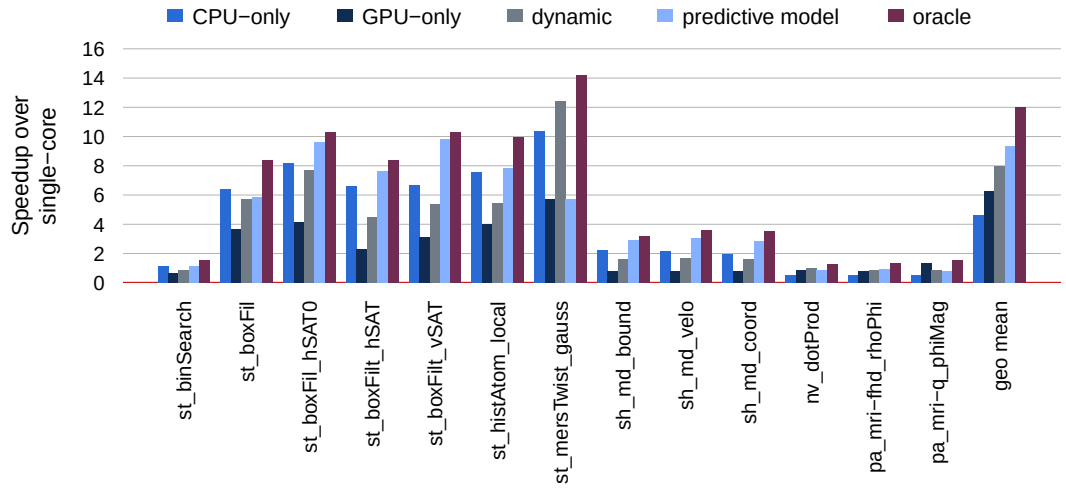
SUMMARY As was shown in this section, a fixed partitioning that is agnostic to program characteristics is unsuitable for heterogeneous environments. The "CPU-only" and "GPU-only" methods only apply to a limited set of benchmarks and cannot adapt to different programs. Different programs need different mappings, highlighting the need for adaptive techniques.

For the majority of programs, a partitioning of the work between CPUs and GPU is beneficial. While the dynamic partitioning method described in section 4.4.2 is designed to handle these cases, it is often unable to achieve good performance due to scheduling overhead and the inability to account for cases where a mixed execution is harmful. The predictive modeling approach, in contrast, explicitly handles all cases and minimizes overhead by making a static decision based on program code features.

Figure 4.6 shows the geometric mean over all benchmarks for the techniques presented in this chapter. The "CPU-only" scheme is by far the worst technique because it does not adapt to programs and misses out on large speed-ups with GPU-friendly programs. Although the "GPU-only" mapping does not adapt to programs either, it achieves a better overall speed-up because it correctly maps the programs that show high performance improvements over single-core execution. With an average speed-



(a)



(b)

Figure 4.5: Performance of those applications where a mix of both CPU- and GPU-execution leads to the best performance. The predictive model achieves a speed-up of 1.2 over the dynamic scheme and clearly outperforms both the CPU-only and GPU-only mapping.

up of 9.21 it is even marginally better than the dynamic method which achieves only 9.13 times the performance of single-core execution. This again is because the potential of GPU-friendly programs is not realized by the dynamic mapping approach. The approach presented in this chapter, on the other hand, leads to significantly higher speed-ups. With an average speed-up of 14.30, we achieve more than 80% of the upper bound which equates to a speed-up of 1.6 over the dynamic scheduler.

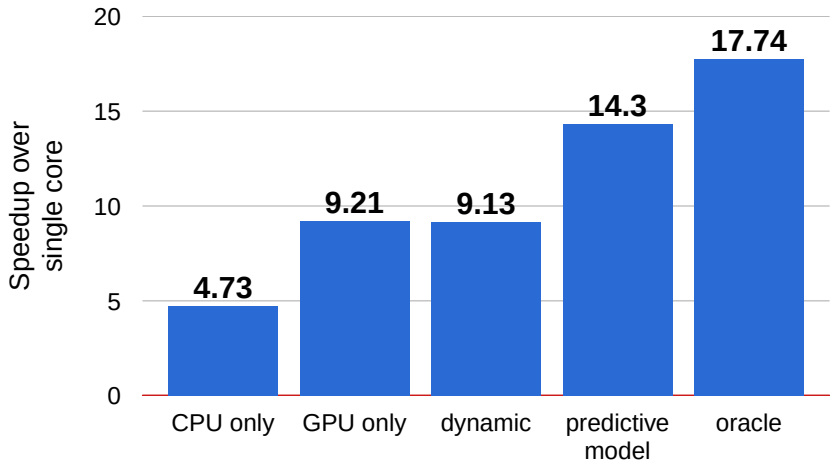


Figure 4.6: The average speed-up (geometric mean) over all benchmarks. The predictive modelling approach clearly outperforms all other competing techniques.

4.5.2 *Prediction Accuracy*

This section takes a closer look at the individual predictors of the hierarchical model. Table 4.5 shows the number of applications that achieve the best performance with GPU-only and CPU-only execution respectively. The job of the two first-level predictors is to filter out these applications and to pass on the rest. Therefore, the table shows the numbers broken down according to the predictions in the first level of our model.

Out of the 220 examined program-input pairs, 61 should be entirely mapped to the GPU. 52 of those are identified by the first-level predictor and thus mapped to the GPU straightaway. The remaining 9 program-input pairs are misclassified by the first-level predictor. However, they are passed on to the second-level predictor, which correctly maps them entirely to the GPU. 10 program-input pairs are incorrectly classified as GPU-only and therefore mapped to the GPU although it is not optimal. However, in half of the cases this still leads to more than 90% of the optimal performance. Overall, the GPU-only classifier has an accuracy of 91%.

19 of the 23 program-input pairs that should be entirely mapped to CPUs are correctly classified by the second predictor in the first level of the model. In the 10 cases of mis-classification an average performance of 78% of the optimum is still achieved. Overall, the CPU-only classifier achieves an accuracy of 95%.

As expected, the second-level predictor has a lower accuracy than its counterparts in level 1. This is because it is solving a much harder problem: instead of making a binary decision, one out of 11 classes need to be predicted. However, being off by just one or two classes often still leads to good performance, on average a performance of 80% of the optimum is still achieved. The level 2 predictor is within these bounds for 65% of all programs.

	GPU	¬ GPU
GPU predicted	52	10
¬ GPU predicted	9	149

	CPU	¬ CPU
CPU predicted	19	6
¬ CPU predicted	4	191

Table 4.5: The accuracy of the binary classifiers in the first level our predictor. The "GPU-only" model achieves an accuracy of 91% while the "CPU-only" model even achieves a 95% accuracy.

Looking at the model as a whole, it achieves an accuracy of 52%, i.e. in 52% of all program-input pairs it *exactly* predicts the optimal mapping. This leads to an average 85% of the optimal performance, compared to only 58% for the dynamic partitioning method.

4.6 SUMMARY

This chapter has developed a new approach for partitioning and mapping OpenCL programs on heterogeneous CPU-GPU systems. Given a data-parallel task, the technique predicts the optimal partitioning based on the task's code features. The model relies on machine-learning techniques, which makes it easily portable across architectures and OpenCL implementations. This is a desirable property as both hardware and software implementations are going to evolve.

When evaluated over 47 benchmark kernels, each with multiple input sizes, the model achieves an average speedup of 14.3 over single-core execution. Compared to a state-of-the-art dynamic partitioning approach this equates to a performance boost of 1.57 times. The approach also clearly outperforms the default strategies of using only the multi-core CPUs or only the GPU, which lead to a speedup of 4.73 and 9.21 over single-core execution, respectively.

The next chapter extends this approach by taking contention on the GPU into account. The technique developed in this chapter assumes that no other application is currently running on the system. With GPU programming becoming more commonplace in desktop and even mobile systems this is no longer realistic. While dynamic approaches such as the one used in this chapter may seem to be able to adapt to contention on the GPU, the next chapter shows why this is often not the case.

5

TASK PARTITIONING IN THE PRESENCE OF GPU CONTENTION

The previous chapter presented a technique for statically partitioning OpenCL kernels. An assumption made, however, was that the system the program is executed on is idle, i. e. there is no other program competing for resources. This chapter investigates how to optimally partition OpenCL programs when other programs are competing for access to the GPU. It shows that traditional mapping approaches fail to adapt to GPU contention and proposes an extension of the predictive model built in the previous chapter that explicitly takes GPU contention into account.

The need for such an approach is motivated in section 5.2, followed by a study of the behaviour of OpenCL programs in the presence of GPU contention in section 5.3. Section 5.4 describes the details of the predictive model. The evaluation methodology is presented in section 5.5 followed by the experimental results in section 5.6. The chapter concludes with some final remarks in section 5.7.

5.1 INTRODUCTION

The previous chapter has demonstrated the benefits of partitioning OpenCL tasks across processors in a heterogeneous system. Similar to many other static approaches (Luk et al., 2009; Jiang and Agrawal, 2012) it assumes that the program is run in isolation, i. e. no other program is competing for resources on the same machine. As heterogeneous computing becomes more popular in desktop and mobile computing, this assumption is no longer valid. Multiple programs may be competing for GPU resources at the same time.

Static partitioning schemes are unable to adapt to GPU contention unless it is explicitly taken into account. Dynamic schemes, on the other hand, intuitively seem to be able to adapt without modification. The task farm mapper presented in section 4.4.2 should simply assign fewer chunks of work to the GPU if there is heavy contention, for example. However, section 5.2 will show why dynamic schemes may struggle to achieve good performance in the presence of GPU contention.

In this chapter a new static mapping approach is introduced that explicitly takes GPU contention into account. It relies on a machine learning-based predictive model for determining a static partitioning given code features of the program and information on GPU contention. GPU contention is characterized by the median waiting time of the program on the GPU in the given system workload. This information can be easily extracted from the OpenCL API while the program is running.

The heterogeneous system considered in this chapter is an Intel IvyBridge chip. It contains both a quad-core CPU and an integrated GPU. These systems are becoming the norm on both desktop (e. g. Intel IvyBridge, AMD Trinity) and mobile systems (e. g. Qualcomm Snapdragon, NVIDIA Tegra). As opposed to systems with a discrete GPU, on these systems the CPU and GPU share the same physical memory, eliminating the need for costly data transfers between the two devices. This allows for more fine-grained collaboration between the two processors than previously possible.

Across a set of 22 benchmarks and 10 different contention scenarios, the predictive model achieves a speed-up of 1.92 over using only the GPU and 1.23 over using only the CPU. Two dynamic mapping approaches, are also evaluated which perform well on idle systems but do poorly in the presence of GPU contention. Compared to these dynamic approaches the predictive model achieves speed-ups of 2.56 and 1.54 respectively.

5.2 MOTIVATION

This section demonstrates the importance of explicitly taking GPU contention into account when mapping programs to heterogeneous system. Dynamic mapping schemes may intuitively seem to be able to adapt to contention but due to the non-preemptiveness of GPU execution this is not the case as will be shown.

5.2.1 *Program Behaviour in the Presence of GPU contention*

Figure 5.1 shows the running time of the `nbody` benchmark in multiple GPU contention scenarios. The medium and heavy contention scenarios are created by running a separate application which uses the GPU alongside the program to be optimized. More details on which applications were used is given in section 5.5.1. Along the x axis different static partitionings (represented as the percentage of work mapped to the CPU) are explored. On an idle system (no contention) the running time of using only the GPU ($x = 0$) is shorter than the running time of using only the CPU ($x = 100$). This changes, however, when GPU contention is introduced.

Similarly, the optimal partitioning between the devices changes in different contention scenarios. When the system is idle, the best performance is achieved by a $30 - 70$ split but already in medium contention more work should be assigned to the CPU, namely 50%. In a heavy contention scenario the runtime of the application increases significantly when some of the work is mapped to the GPU. The best performance is thus achieved when only the CPU is used. The partitioning that is optimal on an idle system ($x = 30$) leads to a 3x *slow-down* over the optimal partitioning.

This example demonstrates the need for partitioning techniques that can adapt to contention on the GPU. Static schemes that do not take this into account will fail to

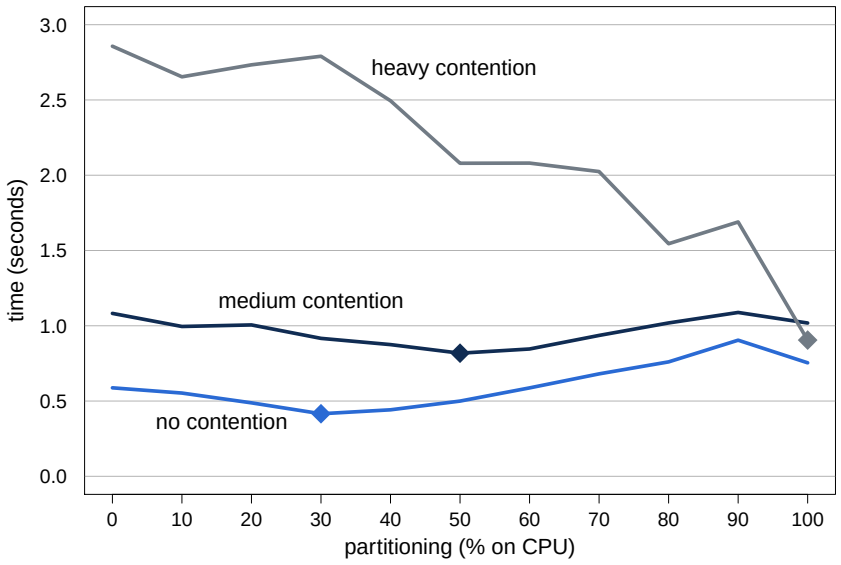


Figure 5.1: Running time of the `nbody` benchmark in multiple GPU contention scenarios for different partitionings. The diamonds indicate the optimal partitioning for the corresponding contention.

achieve good performance on systems being shared by multiple co-running applications. Dynamic schemes may intuitively seem to be more robust to GPU contention but this is not necessarily the case as the next section will show.

5.2.2 *Dynamic Partitioning Schemes*

Two dynamic partitioning schemes and their performance on both an idle system and in the presence of GPU contention are investigated. While they are able to achieve good performance on idle systems, they both fail to deliver performance when other programs are competing for the GPU.

The first scheme is the task farm policy introduced in section 4.4.2. The workspace of a single kernel launch is broken into a fixed number of chunks. Initially, one chunk is sent to each processor, i. e. the CPU and the GPU. When a processor has finished processing its chunk it gets assigned a new one until all chunks have been processed. Schemes like this have been proposed many times for mapping programs to heterogeneous systems (see section 3.2 for examples).

The second scheme finds a good partitioning using *online search*. Raman et al. (2012) used a similar online search based technique to determine the optimal number of CPU threads in the presence of co-running applications. The online search method partitions each kernel 50-50 between the CPU and GPU at the first execution. For future executions this split is adjusted dynamically based on observed running times of the kernel. A more detailed description is given in section 5.5.2. A limitation of

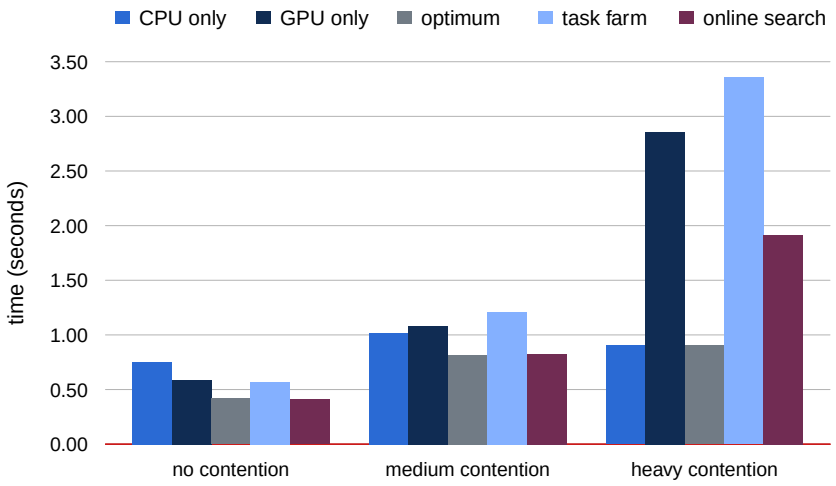


Figure 5.2: Running time of the `convolution` benchmark in multiple GPU contention scenarios with different mapping schemes.

this approach is that it assumes that a kernel is executed repeatedly. If this is not the case, however, it is unlikely to find a good partitioning.

5.2.2.1 *Performance in Different Contention Scenarios*

Figure 5.2 shows the performance of the dynamic mapping schemes compared to using the CPU or GPU only as well as the optimal static partitioning.

On an idle system the GPU outperforms the CPU by a factor of 1.3. Optimally partitioning the work between the processors leads to a 1.8x speed-up over CPU-only execution. While the online search method achieves similar performance, the task-farm mapper is only marginally faster than the GPU-only approach.

When introducing medium contention the order of the single-device approaches reverses; the GPU now only achieves 0.9x the performance of the CPU. The optimal partitioning gives a speed-up of 1.5 over CPU-only execution. While the task-farm mapper is not able to improve over CPU-only execution, the online search method almost matches the performance of the optimal partitioning.

So far, online search has been able to deliver good performance, but this changes in a more heavy contention scenario. In heavy contention the gap between CPU and GPU performance spreads significantly. In fact, the best performance is achieved by not using the GPU at all, thus avoiding the contention on the device. Both dynamic mapping strategies fail to deliver good performance in this scenario; the task farm mapper achieves 0.3x and the online search 0.5x the performance of the CPU.

In summary, the task farm approach is not able to match the performance of the best static partitioning. This is in line with the observations made in the previous chapter. When introducing contention to the GPU the performance gets even worse. The online search method achieves similar performance to the optimal partitioning

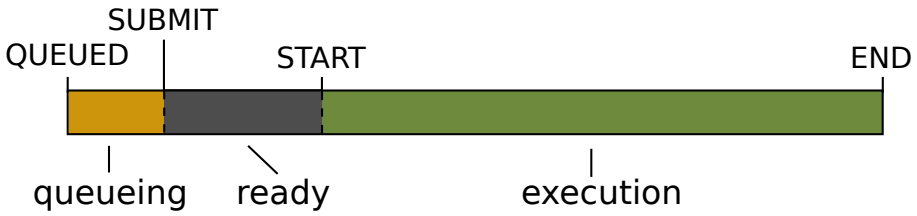


Figure 5.3: The three phases of launching and executing an OpenCL kernel. The labels on top correspond to the `CL_PROFILING_COMMAND_*` parameters passed to the `clGetEventProfilingInfo` function of the OpenCL API.

in medium contention scenarios. When a kernel is being executed a number of times it can quickly find a good partitioning between the devices. However, in a heavy contention scenario it is unable to find a good partitioning. The next section provides some insights as to why this is happening.

5.3 OpenCL applications in the presence of contention

OpenCL provides command queues to launch kernels on devices (see section 2.2.2). Each time a kernel is being executed it passes through multiple phases as depicted in figure 5.3. When a kernel is enqueued to the command queue on the host it enters the *queueing phase*. As soon as the kernel is ready for execution, e. g. all its dependencies are fulfilled, the OpenCL runtime submits the kernel to the device and the kernel enters the *ready phase*. When the device is free to execute the kernel the *execution phase* begins.

On an idle system the time spent in the queueing and ready phase is typically small compared to the execution phase for a kernel. When introducing device contention on the CPU this does not change because the kernel can run alongside other applications on the CPU. There is thus no need to delay kernel execution. The time spent in the execution phase does vary, however, depending on the contention. The kernel has to share resources with other applications which leads to a slow-down in its execution. On most GPUs, on the other hand, only one kernel is allowed to execute at a time.[1] A kernel may therefore spend an increased amount of time in the ready phase waiting for the GPU to become available. Time spent in the execution phase, however, is always the same because the kernel will always be executed in isolation, independent of any device contention. Furthermore, GPU tasks are non-preemptive, i. e. once a task gains access to the GPU no other task can use the GPU until the current task has finished execution. This means that a kernel scheduled to the GPU may have spent

1 NVIDIA GPUs allow concurrent executions of kernels from the same application but not from different applications.

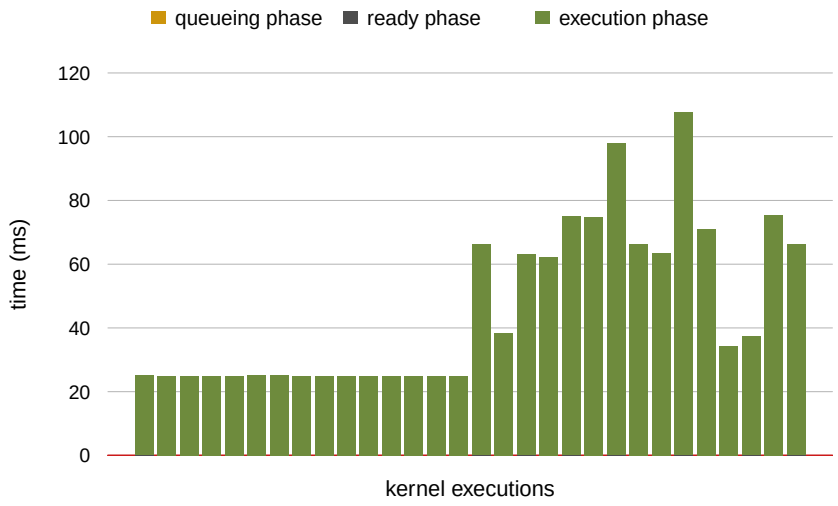
a long time in the ready phase if a different applications has previously launched a long-running kernel.

The OpenCL API provides an interface for querying the starting and finishing time of each of the phases as indicated by the labels in figure 5.3. This allows one to get insights into the behaviour of OpenCL applications in the presence of GPU contention. Figure 5.4 shows the behaviour of the `nbody` benchmark in the presence of heavy contention. In figure 5.4a the program is executed on the CPU in the presence of CPU contention, whereas in figure 5.4b the program is executed on the GPU in the presence of GPU contention. Each bar represents the running time of a kernel launch, divided into the times spent in each of the three phases. During the first few kernel executions the system is idle. Only halfway through the program execution, the contention is introduced on the respective device. This highlights the difference in behaviour on an idle system and one with resource contention.

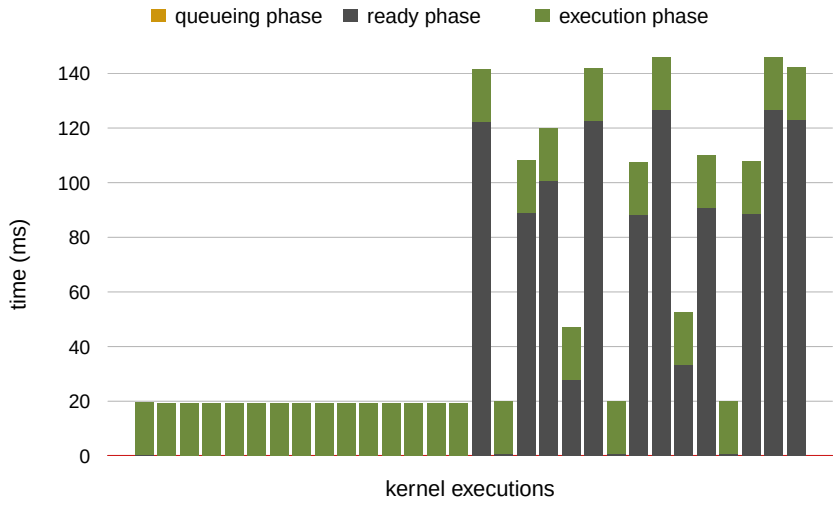
When the system is idle the overall running time of each kernel is dominated by the execution phase which is stable across these runs. Time spent in the queueing and ready phases is minimal (they are not even visible in the graphs) because no other applications compete for resources. The behaviour of the application is the same whether running on the CPU or the GPU. On the CPU each kernel execution takes around 25ms while it takes around 19ms on the GPU.

When a competing application is launched, however, the behaviour is different on the two devices. On the CPU the queueing and ready phase is still the same but the time spent in the execution phase has increased significantly. While there is some variation in the execution times most kernel executions now take around 60-80ms. On the GPU, on the other hand, time spent in the execution phase remains the same at 19ms because the kernel will always have exclusive access to all GPU resources. However, time spent waiting for the device (the ready phase) shows dramatic variation. Sometimes it is as low as on the idle system but it can be as high as 120ms (more than 6 times the time spent in the execution phase). The variation is due to other applications blocking the GPU. If a long-running kernel has been launched before a kernel is submitted it has to wait for the long-running kernel to finish. If, however, the long-running kernel is just about to finish when the kernel is launched the wait time is small. The total execution time of a kernel launch therefore varies significantly even if the contention is constant, i. e. a fixed co-running application. In such heavy GPU contention it is thus better to use the CPU which provides a guaranteed running time of 25ms rather than using the GPU which may lead to the kernel having to wait for 120ms before even entering the execution phase.

This unpredictable behaviour of OpenCL applications in the presence of GPU contention provides a big challenge to schedulers trying to find the best partitioning of a kernel across devices. Dynamic schemes generally rely on *feedback* from previous executions on a device. As is shown in figure 5.4b however, the execution time of a single kernel launch on the GPU is not necessarily a good indicator for the execution



(a) CPU.



(b) GPU.

Figure 5.4: Profiling of kernel launches in the presence of device contention. Each bar shows the total running time of a kernel launch; broken down into queueing, ready and execution times. During the first half the system is idle, then another application is launched competing for resources of the same device.

time of the next launch. Dynamic schemes thus fail to adapt to resource contention on the GPU as was shown in section 5.2.2. A more detailed evaluation is given in section 5.6.2.

5.4 A PREDICTIVE MODEL FOR OPENCL TASK PARTITIONING IN THE PRESENCE OF GPU CONTENTION

This section describes how a machine learning-based model can be built for determining a good partitioning for OpenCL tasks in the presence of GPU contention. The input to the predictive model contains information on both the OpenCL kernel and the GPU contention. Its output is a ratio describing the amount of work to map to the CPU and the GPU.

5.4.1 *The Features of the Model*

The input features of the predictive model are constructed from two sets of features. The first set of features describe the OpenCL kernel itself. They are extracted using the static analysis tool introduced in section 2.4.1.1. The second set of features characterize the contention on the GPU device. These features are constructed from information readily available via the OpenCL API.

5.4.1.1 *Program Features*

Similar to the features used by the predictive model in the previous chapter the program features contain information about the static number and types of instructions in a kernel. Additionally, the number of coalesced memory accesses is determined. The benchmarks used for this study, as described in section 5.5.1, often contain vector data types because they were targeted towards both CPUs and GPUs. Another feature is thus the number of vector operations in the code.

In chapter 4 a crucial feature was the amount of data that needed to be transferred to the GPU memory and back. Using the GPU was only useful if there is enough computation to outweigh the overheads of data transfers. The heterogeneous system being targeted in this chapter contains memory that is *shared* by the CPU and the GPU, eliminating the need for data transfers. This significantly reduces the overhead of GPU execution. However, it also removes a feature that previously provided important information for determining the best work partitioning between the CPU and the GPU.

As before, individual features are aggregated to provide more information to the model. The full list of program features is shown in table 5.1. Features 1 - 6 are all extracted at compile time while feature 7, the number of work-items, can only be known at run-time.

	Program Features
1:	# global memory accesses
2:	# compute operations
3:	# conditionals and loops
4:	communication-to-computation ratio
5:	percentage of vector operations
6:	percentage of coalesced memory accesses
7:	# work-items

Table 5.1: List of program features used by the predictive model.

5.4.1.2 *Contention Features*

Contention on the GPU is experienced by increased delays in the ready phase waiting for the device to become available (see section 5.3). The specific GPU kernel causing the contention does not have any other influence on the remaining programs because access to the GPU is exclusive. The best way to characterize GPU contention is thus to quantify this delay. Since the delay exhibits a significant variance a single observation does not carry much information. Therefore, to characterize the contention the average delay (using the arithmetic mean) is computed over time.

The OpenCL API provides profiling information for kernel executions via the `clGetEventProfilingInfo` function. The delay waiting for the GPU to become available is computed by subtracting the value for `CL_PROFILING_COMMAND_START` from the value of `CL_PROFILING_COMMAND_SUBMIT`, as shown in section 5.3.

5.4.2 *Collecting Training Data*

A set of training programs and "workload" programs are used to collect training data for the predictive model. The workload programs introduce contention on the GPU by either using it for graphics or by executing OpenCL kernels on it. A detailed discussion of which programs were used is given in section 5.5.1.

Each combination of training and workload program is executed with different partitionings of the training program. In total eleven different partitionings are evaluated, ranging from CPU-only to GPU-only execution in steps of 10% as shown in figure 5.1. To ensure a constant degree of contention the workload program is started a few seconds ahead of the training program and input parameters are chosen so that it only finishes after the training program has finished execution.

The best partitioning for each scenario is computed by finding the one with the shortest overall running time. Since some benchmarks contain multiple kernels this

computation is done on a per-kernel basis using OpenCL profiling information. The running time of a kernel that is partitioned across the CPU and GPU is defined as the maximum running time across the two devices.

The optimal partitionings form the targets of the predictive model. The features are collected by performing static analysis on the training program and by recording the incurred delay using the profiling functions provided by OpenCL.

5.4.3 *Building the Model*

In order to model the problem of task partitioning, a multi-class classification model is used. Specifically, the model is based on support vectors machines (SVMs) as described in section 2.3.3.1.

As opposed to the hierarchical model in chapter 4.3 the model used in this chapter is a single SVM. The motivation behind the hierarchical model in the previous chapter was to have very accurate predictions for benchmarks that should be either executed completely on the CPU or the GPU. Due to the separate memory spaces it was important to not use the GPU at all if the overheads are too high. Conversely, the GPU was significantly faster than the CPU for some of the programs, meaning that any work scheduled to the CPU would lead to a slow-down. The system targeted in this chapter is more balanced. Firstly, GPU execution overheads are much smaller due to the shared physical memory. Secondly, the peak performances of the two processors are much closer than on systems with discrete GPUs (compare table 4.2 to table 5.2). A prediction that is off by only a small percentage is thus less likely to incur significant load imbalance, even if CPU-only or GPU-only execution is optimal. This is why a simple model, containing only a single SVM, suffices to achieve good accuracy.

5.4.4 *Deployment of the Model*

The model can only be evaluated at run-time because it partially relies on run-time information. On the one hand, the number of work-items needs to be known to compute the program features. This is often only known at run-time. Furthermore, the current GPU contention can obviously only be determined when the program is actually running.

Computing the contention features involves monitoring the waiting times on the GPU and computing the average waiting time as described in section 5.4.1. Only a window of waiting times of the last few kernel executions should be used, however, to be able to adapt to changes in the contention. This information can either be obtained through previous kernel executions of the program (assuming it launches a sequence of kernels) or by sharing this information across programs.

	CPU	GPU
Model	Intel Core i5-3570K	Intel HD Graphics 4000
Core Clock	3.40 GHz	1.15 GHz
Core Count	4	16
Peak Performance	108.8 GFLOPS	147.2 GFLOPS
System Memory	8 GB	
Operating System	Windows 7 Professional SP 1	
OpenCL SDK	Intel SDK for OpenCL Applications 2013	

Table 5.2: Experimental Setup.

5.5 EXPERIMENTAL METHODOLOGY

This section describes the setup used for evaluating the approach presented in this chapter. It details how and which aspects of the model were evaluated and describes which other methods the model was compared against.

5.5.1 *Experimental Setup*

PLATFORM In the evaluation of this approach all experiments were carried out on a system comprising an Intel IvyBridge chip, specifically an Intel Core i5-3570K. This chip contains a quad-core CPU and an integrated Intel HD 4000 graphics chip. Full details are shown in table 5.2. There was no discrete GPU and work was thus partitioned between the CPU and the integrated GPU. The system was running on Windows 7 and the Intel SDK for OpenCL Applications 2013 (Intel, 2013) was used.[2] Each measured run was repeated 10 times and the average execution time was recorded.

Integrated platforms such as the Intel IvyBridge chip are increasingly common in the desktop and mobile computing space. The trend is to further integrate the CPU and GPU in order to allow close cooperation between the two types of processors. The experiments in this chapter are thus evaluated on such a system to account for this change in heterogeneous, GPU-based systems.

BENCHMARKS 22 different benchmarks from the Intel SDK (Intel, 2013) and the AMD SDK (AMD, 2013) were evaluated. These benchmarks were chosen because they are not specifically tuned for GPUs but for use on both CPUs and GPUs, e. g. by using vector data types. They thus provide for more interesting partitioning scenarios. In

2 As opposed to the other experiments in this thesis, the setup in this chapter uses the Windows operating system. The reason is that Intel only provides Windows versions of the OpenCL drivers that support both the CPU and the GPU on the IvyBridge chips.

Benchmarks	
God Rays (Intel)	Floyd Warshall (AMD)
Median Filter (Intel)	Histogram (AMD)
Tone Mapping (Intel)	Mandelbrot (AMD)
AES (AMD)	Matrix Multiplication (AMD)
Binomial Option (AMD)	Matrix Transpose (AMD)
Blackscholes (AMD)	Monte Carlo Asian (AMD)
Box Filter - SAT (AMD)	NBody (AMD)
Box Filter - Separable (AMD)	Quasi Random Sequence (AMD)
Convolution (AMD)	Recursive Gaussian (AMD)
DCT (AMD)	Reduction (AMD)
Fast Walsh Transform (AMD)	Sobel Filter (AMD)

Table 5.3: Benchmarks used for evaluation.

order to increase the set of training points each benchmark was used with multiple input sizes. A list of all benchmarks is given in table 5.3.

The main computational parts of the benchmarks were executed repeatedly to ensure a minimum running time of around 500ms. This was done to expose each benchmark to the fluctuations of GPU contention as shown in section 5.3. It further allows the online search method to find a good partitioning.

CONTENTION SCENARIOS To introduce contention to the system a range of applications using GPUs was used. These mainly include OpenCL benchmarks targeting the GPU but also a video player application (VLC). Additionally, the scenario of the idle system, i. e. without any contention, was evaluated. A list of all contention scenarios is given in table 5.4. The entries are ordered by how disruptive they are in terms of the average waiting caused as described in section 5.4.1.2.

Some programs were used across multiple contention scenarios. In this case, the input size of the program was varied to achieve varying average waiting times. Using the same program but with different input sizes provides a controlled way to cover a range of average waiting times. The method is valid because the only feature used to distinguish contention scenarios is the waiting time and features of the actual programs do not matter.

Name	Type	Waiting Time (μs)
none	no contention	65
vlc	video player	68
sobel-512	OpenCL application	759
monte_carlo	OpenCL application	1,306
sobel-1024	OpenCL application	3,228
aes-512	OpenCL application	8,471
sobel-2048	OpenCL application	12,584
aes-1024	OpenCL application	16,259
aes-2048	OpenCL application	21,944
sort	OpenCL application	36,585

Table 5.4: Contention scenarios. Ordered from lowest to highest contention.

5.5.2 *Evaluation Methodology*

The cross-validation method described in section 2.5 was used to evaluate the model. When predicting for a certain benchmark and contention scenario, no data from that benchmark, including data from runs with different input sizes, nor data from that contention scenario were used in building the model. It is assumed that information about the GPU contention is available. Section 5.4.4 provides a brief discussion on how this information can be obtained.

The approach presented in this chapter is compared to a number of competing approaches. Unless stated otherwise, performance is shown as the speedup over CPU-only execution. This is a useful baseline because the CPU is always available during the experiments whereas there may be heavy contention on the GPU.

Initially, the approach is compared to an "oracle" which is a theoretical scheduler that always picks the best static partitioning. It thus provides an estimate of the upper bound performance available in each scenario. Since exhaustive experiments were carried out on each benchmark to collect training data, the performance of the oracle is readily available.

Additionally, the approach is compared to two dynamic mapping approaches. The first one, "task farm", splits each task into a fixed number of chunks. Initially, one chunk is sent to each device and devices request more work after they have finished processing their chunk. For this evaluation (as for the one in the previous chapter) we specified the number of chunks to be 8, which provides a good trade-off between the ability to load-balance and reducing overheads.

The second dynamic mapper, "online search", finds a good partitioning over time. For each kernel the scheme keeps track of what the partitioning between the CPU and GPU is. The partitioning is represented by a *split value* which is the percentage of work mapped to the CPU. This value is adjusted over time to balance the running times on the two processors. At the first kernel launch the work is split in half, i. e. the split value is 50%. When the kernel execution has finished the split value may be adjusted by a certain *step value* depending on which processor has finished earlier. If the CPU finished before the GPU the split value is incremented by the step value (more work is assigned to the CPU), otherwise it is decremented. If both the CPU and GPU finished within the same amount of time the optimal split value has been found and it remains unchanged. The initial step value is 10% but it is divided by 2 whenever the *direction* of the adjustment changes. This ensures that initial large imbalances can be overcome swiftly while a precise split value can still be found.

Each benchmark run contains multiple iterations of the main computational phase of the program. This is a common scenario in, for example, linear algebra applications or video processing. Having multiple iterations allows the online search method to find a good partitioning between the CPU and GPU. Furthermore, due to the variable waiting times shown in section 5.3 it is needed to achieve consistent results.

5.6 RESULTS

This section evaluates and analyses the performance of the predictive modeling approach presented in this chapter. First performance is compared to an oracle scheduler. Then the performance of the two dynamic schemes is evaluated and compared to the predictive model.

5.6.1 *Comparison to Oracle*

Figure 5.5 shows the performance of the oracle and the predictive model. Additionally, the performance of a GPU-only approach, which executes all kernels on the GPU, is shown because OpenCL applications typically target the GPU. The performance on each benchmark is shown, averaged across all ten contention scenarios. The numbers are normalized to (parallel) CPU-only execution.

With the exception of five benchmarks, the GPU-only approach leads to slowdowns over CPU-only execution. On average the speed-up is 0.64. As shown in section 5.2 GPU execution can be severely delayed in contention leading to increased running times. The oracle results demonstrate, however, that when using the GPU in the right way, significant speed-ups over CPU-only execution can be achieved: up to 2.47 when averaged across contention scenarios. Averaged across all benchmarks *and* contention scenarios, a speed-up of 1.43 is achievable.

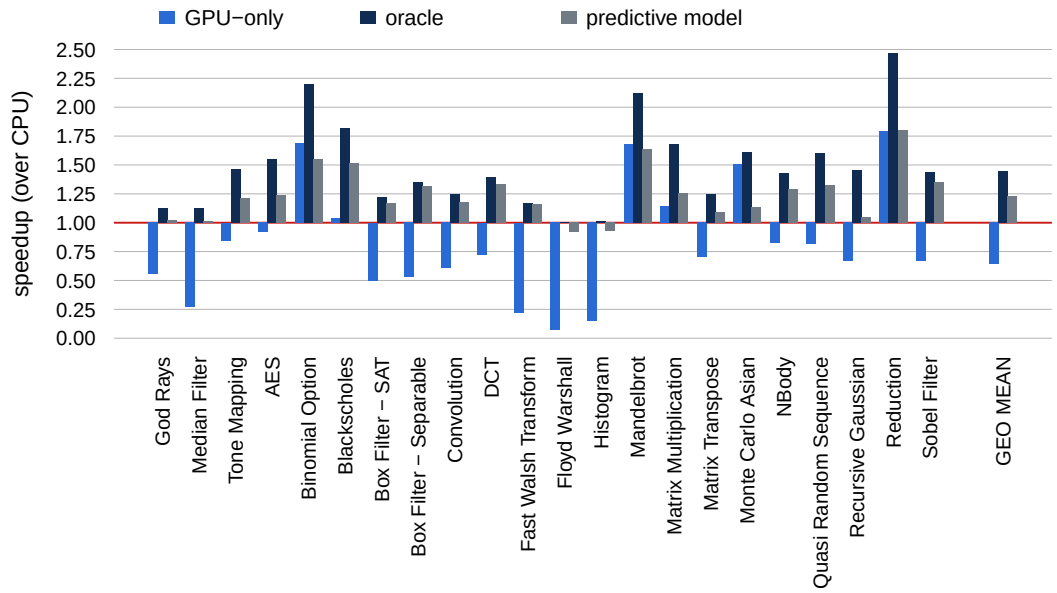


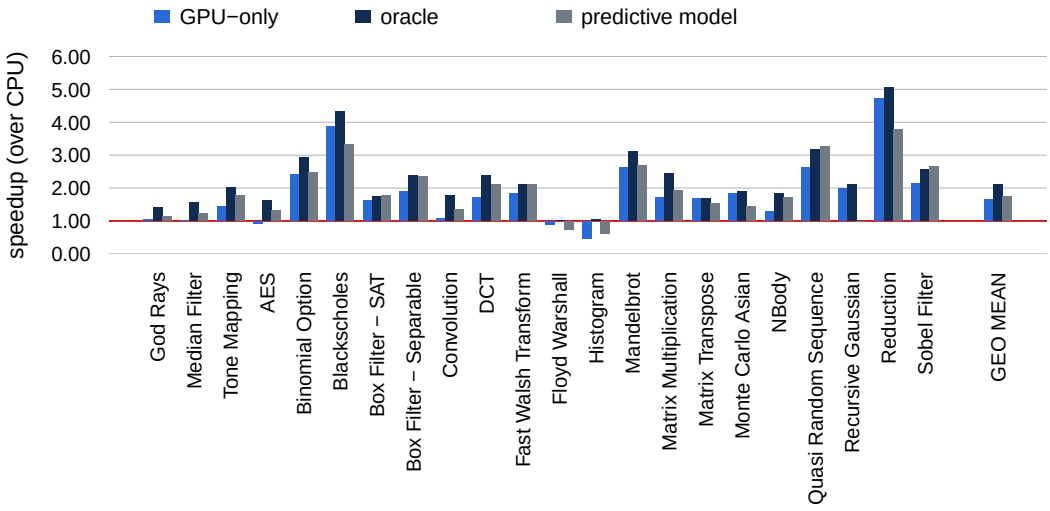
Figure 5.5: Speed-up over CPU-only execution averaged across all ten contention scenarios. The GPU-only, oracle and predictive model achieve average speed-ups of 0.61, 1.48 and 1.24 respectively.

The predictive modeling approach achieves performance close to the oracle for a number of benchmarks, e. g. `DCT` or `Sobel Filter`. For all but three benchmarks it outperforms the GPU-only approach and in only two cases can slow-downs over the CPU be observed. On average, the predictive model achieves a speed-up of 1.23 which is 86% of the oracle performance, compared to 45% and 70% for the GPU-only and CPU-only approaches, respectively. These results demonstrate that the predictive model manages to adjust well to different contention scenarios.

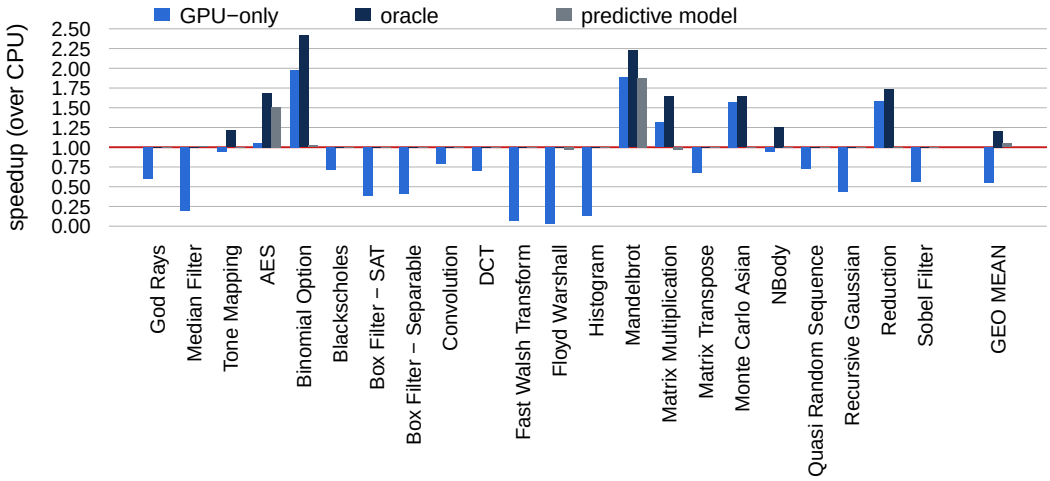
The overall accuracy of the model is 47.8%, i. e. in almost half of all cases the model picks the correct partitioning out of the 11 possibilities. A wrong prediction does not necessarily lead to bad performance though. If the prediction is only slightly off, the performance is often close to the optimum as can be seen in figure 5.1.

To gain a better understanding of the results, figure 5.6 shows performance results for three of the ten contention scenarios, namely `none`, `aes-512` and `sort`, representing no, medium and heavy contention respectively.

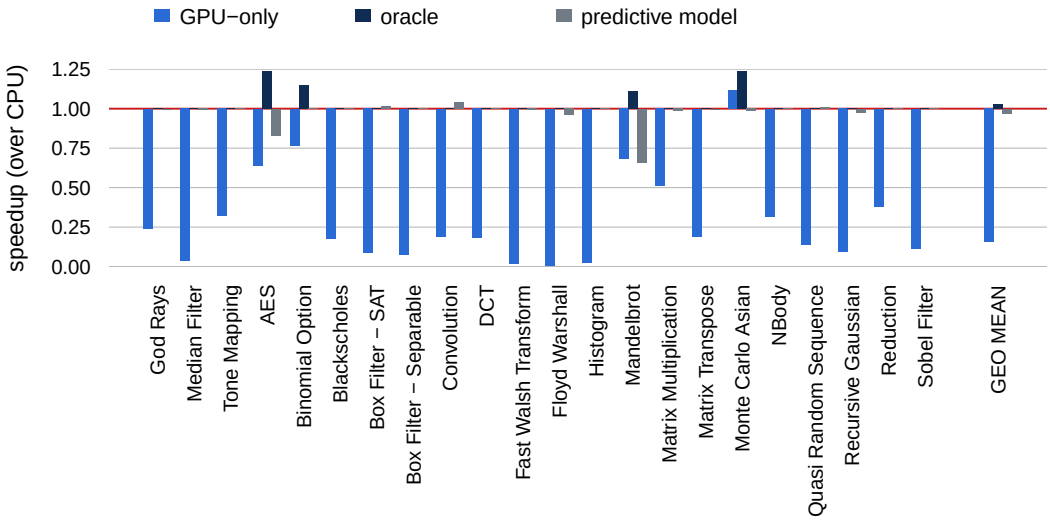
When there is no contention on the GPU, the GPU-only approach leads to good performance because the GPU generally outperforms the CPU. On average it achieves a speed-up of 1.64. The oracle results show, however, that partitioning the work between the CPU and GPU improves performance for all but one benchmark (`Floyd Warshall`) over using only a single device. For `Convolution`, for example, the CPU and the GPU have very similar performance. Splitting the work between the two processors in an optimal way leads to a speed-up of 1.77. The average performance of the oracle in no contention is a speed-up of 2.11. The predictive model is not quite



(a) none (no contention)



(b) aes-512 (medium contention)



(c) sort (heavy contention)

Figure 5.6: Speed-up over CPU-only execution in three different contention scenarios: no contention (a), medium contention (b) and heavy contention (c).

able to match that but with an average speed-up of 1.74 it outperforms both the CPU-only and GPU-only approach. The predictive model tries to find a good partitioning between the CPU and the GPU which, if the prediction is correct, can lead to significant speed-ups, e. g. for `Sobel Filter`, but sometimes leads to slow-downs, e. g. `Histogram`.

When introducing medium contention on the GPU (figure 5.6b) performance of the GPU-only method suffers significantly for some benchmarks, e. g. `Median Filter` or `Floyd Warshall`, while staying strong for others, e. g. `Binomial Option` or `Mandelbrot`. On average, it achieves a speed-up of only 0.55 compared to 1.21 of the oracle. The predictive modeling approach leads to an average 1.05 speed-up across the benchmarks.

In a heavy contention scenario the GPU-only approach performs poorly (figure 5.6c). For only one benchmark, `Monte Carlo Asian`, an improvement over CPU execution can be observed with some benchmarks achieving speed-ups as low as 0.01. On average, the GPU-only method leads to a 0.16 speed-up in this contention scenario. As shown by the oracle performance, the best mapping for the majority of benchmarks is to use the CPU. With only two exceptions, namely `AES` and `Mandelbrot`, the predictive model is at least close to CPU-only execution. On average, the predictive model achieves a speed-up of 0.97 compared to an oracle performance of 1.03.

Given these results one may consider that a simple policy would achieve good performance: if there is no contention, use the GPU, otherwise use the CPU. However, this leaves the problem of defining a threshold at which a certain delay is considered contention. Furthermore, some kernels cope better with contention than others. So a policy that is agnostic to kernel characteristics is bound to deliver suboptimal performance.

The predictive model is able to adapt to contention on the GPU and outperforms single-device approaches. When there is no contention the GPU typically outperforms the CPU but this is reversed when contention is introduced. In all scenarios the predictive model is able to at least match the performance of the fastest single-device strategy. The next section investigates the performance compared to two dynamic mapping approaches.

5.6.2 *Comparison to Dynamic Mapping Schemes*

Figure 5.5 shows the performance of the two dynamic approaches, task farm and online search, as well as that of the predictive model. As before, the performance of each benchmark is shown, averaged across all ten contention scenarios. The numbers are normalized to (parallel) CPU-only execution.

It can be seen immediately that both dynamic approaches fail to achieve good performance in the presence of GPU contention. With a few exceptions, e. g. `Binomial`

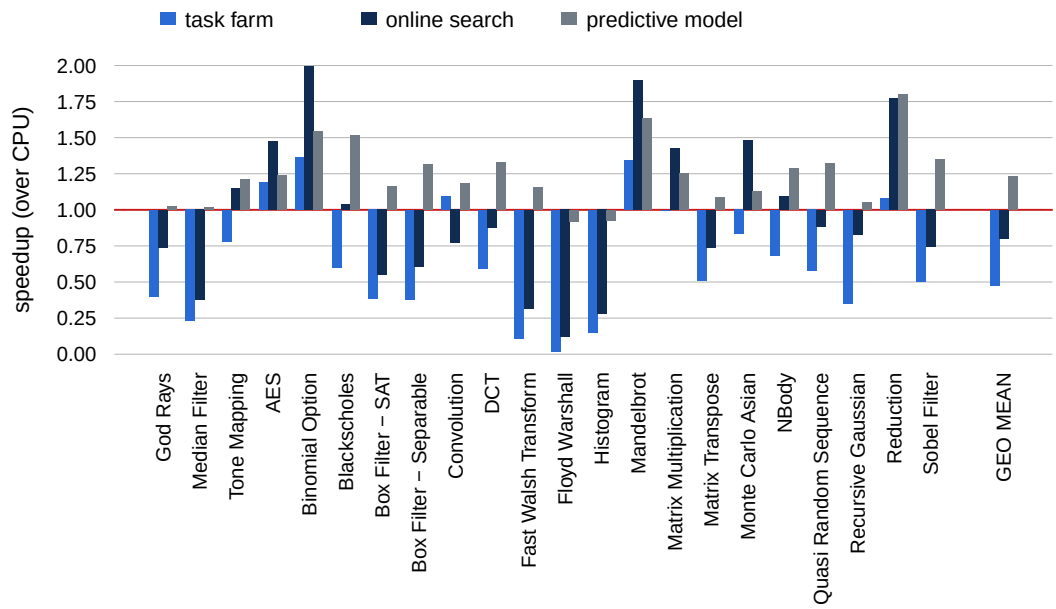


Figure 5.7: Speed-up over CPU-only execution averaged across all ten contention scenarios. The task farm, online search and predictive modeling approaches achieve average speed-ups of 0.45, 0.73 and 1.24 respectively.

`Option` or `Mandelbrot`, both approaches are not able to outperform the CPU-only approach. Especially the task farm mapper leads to slow-downs in most cases. For all but one benchmark, namely `Convolution`, the online search approach beats the task farm mapper. In only 5 of the 22 benchmarks does either of the dynamic approaches outperform the predictive modeling approach.

On average, the task farm method achieves a speed-up of 0.45 and the online search approach achieves a speed-up of 0.73. In other words, if there is possible contention on the GPU it is better to use only the CPU rather than any of these two dynamic approaches. The predictive model, on the other hand, achieves a speed-up of 1.24, demonstrating that using the GPU in the right way can be very beneficial.

The benchmarks where the dynamic approaches, especially the online search method, do well are the ones where GPU execution performs strongly even in heavy contention scenarios, e. g. `Binomial Option` or `Monte Carlo Asian`, as can be seen in figure 5.6c. Conversely, benchmarks where a GPU-only approach performs poorly in heavy contention, e. g. `Fast Walsh Transform` or `Floyd Warshall`, also show huge slow-downs on the dynamic approaches.

Figure 5.8 shows results for three individual contention scenarios: no contention, medium contention and heavy contention. On idle machines (no contention) the dynamic approaches perform well. This was already indicated in section 5.2. The online search method performs best, even outperforming the predictive model. However, the online search method relies on repeated kernel executions whereas the predictive model finds a good partitioning straightaway. The task farm mapper performs worst

out of all approaches but it still achieves an average speed-up of 1.46. This compares to a speed-up of 2.01 of the online search method and 1.74 of the predictive model.

In a medium contention scenario (figure 5.8b) the results are different. The online search method delivers good performance on a number of benchmarks, e. g. `Binomial Option` or `Mandelbrot`, even outperforming the predictive model. For many other benchmarks, however, big slow-downs can be observed. The task farm mapper leads to even worse performance. On average, the task farm and online search method achieve speed-ups of 0.35 and 0.61, respectively. The predictive model, on the other hand, achieves better results with an average speed-up of 1.05. The predictive model is able to use the GPU selectively leading to speed-ups over CPU-only execution for benchmarks such as `Mandelbrot` while not suffering the slow-downs for others.

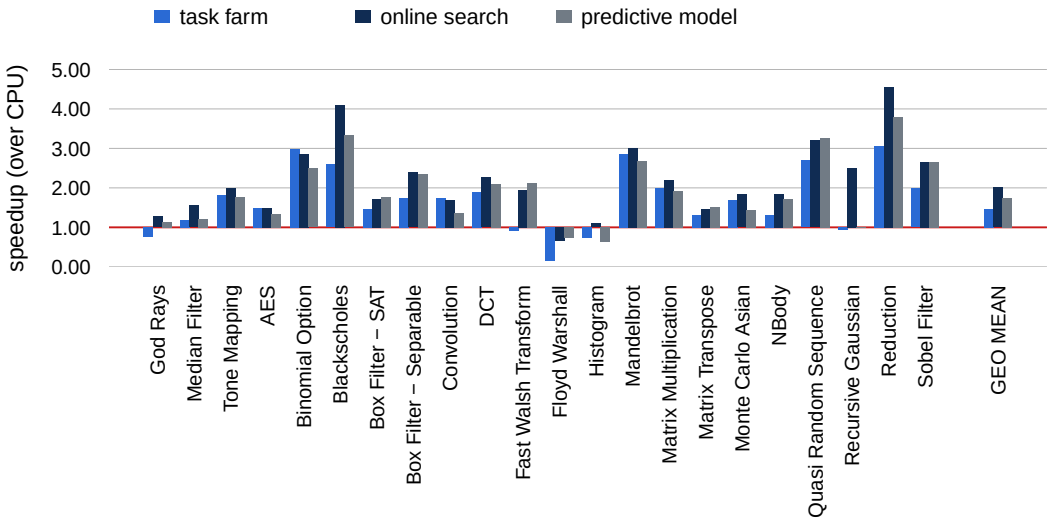
In the heavy contention scenario (figure 5.8c) similar observations can be made. However, the slow-downs of the dynamic approaches are even more drastic on some benchmarks, e. g. `Floyd Warshall`. This leads to average speed-ups of just 0.14 and 0.23 of the dynamic approaches. The predictive model is not able to match the performance of the CPU-only approach either but with an average speed-up of 0.97 it significantly outperforms the dynamic approaches. Furthermore, only minimal improvements over the CPU-only approach are possible in heavy contention as was shown in figure 5.6c.

5.7 SUMMARY

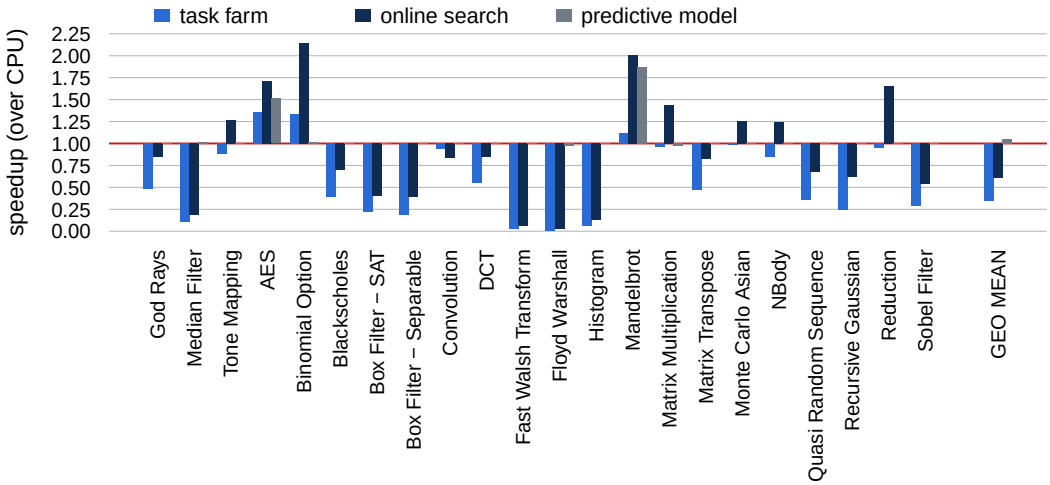
This chapter has investigated the impact of contention for GPU resources on mapping OpenCL programs to CPU-GPU systems. Standard mapping techniques fail to adapt to this type of contention because, unlike on the CPU, kernels have exclusive access to the GPU and cannot be preempted. This can lead to very unintuitive behaviour as was shown in section 5.2. It is possible, however, to adapt mapping decisions to GPU contention by explicitly taking the contention into account. A machine learning-based approach has been presented that uses information about the contention as well as program characteristics to decide how to partition an OpenCL kernel across the CPU and GPU.

Across a set of 22 benchmarks and 10 different contention scenarios this method achieved a speed-up of 1.23 over CPU-only execution. This corresponds to 86% of the performance of an oracle approach. Two dynamic mappers, task farm and online search, only achieve speed-ups of 0.48 and 0.80 respectively, thus actually slowing down the execution time compared to the CPU-only method.

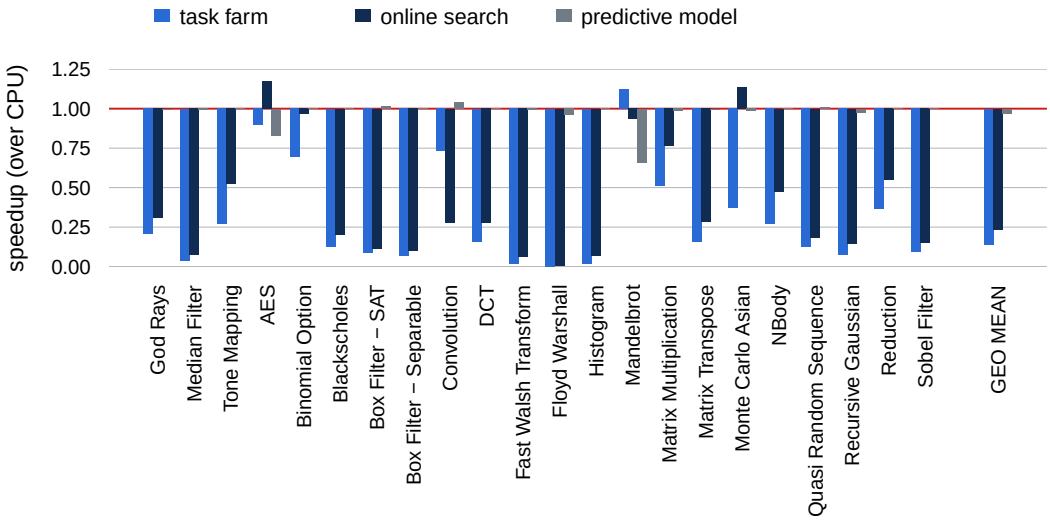
Similar to the previous chapter, this chapter has dealt with the problem of task mapping. It was assumed that programs were already written in OpenCL so that they can be readily executed on both CPUs and GPUs. Writing OpenCL code requires a lot of effort from the programmer though and can thus be a barrier for het-



(a) none (no contention)



(b) aes-512 (medium contention)



(c) sort (heavy contention)

Figure 5.8: Speed-up over CPU-only execution in three different contention scenarios: no contention (a), medium contention (b) and heavy contention (c).

erogeneous computing. The next chapter considers automatically generating and optimising OpenCL code from a higher-level, parallel programming language, OpenMP. Because not all programs are equally suited for GPU execution it also incorporates a predictive model for choosing the optimal device for a given program.

6

PORTABLE MAPPING OF DATA PARALLEL PROGRAMS TO OPENCL FOR HETEROGENEOUS SYSTEMS

The last two chapters focused on the problem of task mapping and partitioning. In this chapter this idea is combined with automatic code generation and optimization for GPUs. A compiler-based approach is presented that automatically generates optimized OpenCL code from OpenMP code and predicts whether it is faster to execute the OpenMP code on the CPU or the OpenCL code on the GPU.

Section 6.2 motivates the need for compiler optimizations when generating GPU code as well as a method for determining the optimal device based on the quality of the generated code. The overall scheme is presented in section 6.3 followed by a description of the code generation and optimization in section 6.4. The predictive model for determining the optimal device is described in section 6.5. Section 6.6 presents a technique for determining whether or not to apply dynamic index reordering, a data layout transformation that can provide performance benefits but which should be used selectively. The experimental methodology is shown in section 6.7 followed by an analysis of the results in section 6.8. Section 6.9 takes a closer look at the performance impact of dynamic index reordering before section 6.10 concludes the chapter.

6.1 INTRODUCTION

Heterogeneous systems consisting of a host multi-core CPU and a GPU are highly attractive as they give potentially significant performance improvements at little cost. Realizing such potential, however, is challenging due to the complexity of programming. Users typically have to identify sections of their code that are potentially suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems. While OpenCL provides functional portability across devices, in practice programs have to be rewritten and tuned to deliver performance when targeting new processors (Komatsu et al., 2010). OpenCL thus does little to reduce the programming complexity barrier for users.

High level shared memory programming languages such as OpenMP are more attractive. They give a simple upgrade path to parallelism for existing programs using pragmas. Although OpenMP is mainly used for programming shared memory multi-cores, it is a high-level language with little hardware specific information and

can be targeted to other platforms. What users would like is the ease of programming of OpenMP with the GPU availability of OpenCL that is then optimized for a particular platform and gracefully adapts to GPU evolution. The method presented in this chapter achieves this by developing a compiler-based approach that automatically generates optimized OpenCL from a subset of OpenMP. This allows the user to continue to use the same programming language, with no modifications, while benefiting automatically from heterogeneous performance.

The first effort in this direction is by Lee et al. (2009) who present a compiler that generates CUDA code from OpenMP programs. While promising, there are two significant shortcomings with this approach. Firstly, their compiler does not apply data transformations. As shown in this chapter, data transformations are crucial to achieve good performance on GPUs. Secondly, the programs are always executed on GPUs. While GPUs may deliver improved performance, they are not always superior to CPUs (Bordawekar et al., 2010; Lee et al., 2010). A technique for determining when GPU execution is beneficial is needed. This chapter addresses both of these issues and when evaluated on the full NAS parallel benchmarks our technique outperforms the approach by Lee et al. (2009) by a factor of 10.

A key feature of the scheme presented here is that it uses predictive modeling to automatically determine if it is worthwhile running the code on the GPU or the multicore CPU. Furthermore, it can adapt this model to GPU evolution. This means that the user can use the same OpenMP code on different platforms with the compiler determining the best place for code to run and optimize it accordingly. As opposed to the approaches in the two previous chapters, the model in this chapter only chooses between GPU-only and CPU-only execution rather than attempting to split the work across devices. Partitioning the *work* across devices also means partitioning the *data* accordingly because GPUs often have their own memory space. In a compiler setting the mapping between work and data would have to be determined automatically which requires complex analysis of the code if it is possible at all, e. g. when dealing with indirect memory accesses. In chapter 4 it was assumed that the user provides this information to the system and generally the focus was on building a model for predicting the best partitioning rather than the infrastructure of actually partitioning the work. In a compiler setting this cannot be ignored, however. Chapter 5 avoided the problem of data partitioning by only considering systems with integrated GPUs that share the CPU memory. These systems are increasingly common in desktop and mobile computing. The work presented in this chapter targets benchmarks from the high-performance computing domain, however, where discrete GPUs are commonplace. For complexity reasons the approach thus only either executes the entire computation on the CPU or the GPU.

6.2 MOTIVATION

With the significant interests in GPUs, it is important to know that GPUs are not always the most suitable device for scientific kernels. This section provides a simple example demonstrating whether or not it is profitable to use a GPU depends on the original program, data size and the transformations available.

Consider the OpenMP fragment in figure 6.1a from the NAS parallel benchmark `bt`, a benchmark containing over 50 parallel loops potentially suitable for offloading to a GPU. Using the basic OpenMP to OpenCL translator introduced in section 6.4 yields the code shown in 6.1b (simplified for presentation). The parallel loop has been translated into a kernel where each of the loops is parallelized forming a 3D parallel work-item space, each point of which is accessed through a call to `get_global_id()` for dimensions 0, 1 and 2.

This code if executed on a GPU with the small W data size however gives disappointing performance when compared to executing the code on a multi-core CPU as shown in figure 6.1c. If the same code is executed with a larger data size A, the GPU performance improves but is still less than the performance achieved on the CPU. The main reason is the memory access pattern of the kernel which does not allow for memory coalescing on the GPU (see section 2.1.2). This can be changed by performing global index reordering as shown in figure 6.1d, transforming the data layout of array `lhs`. This gives the new OpenCL program shown in figure 6.1e. Here the most rapidly varying indexes of the array correspond to the tile IDs giving coalesced memory accesses. Figure 6.1f shows that the resulting performance of the GPU code improves substantially for data size W but is still below the performance of the original OpenMP code on the multi-core CPU. If this transformed code is executed with the larger data size A, however, the GPU performance further improves and both GPUs now outperform the multi-core OpenMP implementation.

This example shows that the translated OpenCL code can give better performance than the original OpenMP code depending on the data size and transformations available. As described in section 6.8, this decision varies from program to program and across different platforms and data sizes. What is needed is a system that learns when to use the GPU, changing its decision based on the availability of underlying optimizations such as data layout transformations. The remainder of this chapter describes the translation and optimizations applied and develops a predictor that determines whether to exploit the GPU depending on circumstances.

6.3 OVERALL SCHEME

The compiler presented in this chapter automatically translates OpenMP programs to OpenCL-based code, performing loop and array layout optimizations along the

```
#pragma omp parallel for
for (i=1; i<grid_points[0]-1; i++) {
  for (j=1; j<grid_points[1]-1; j++) {
    for (k=1; k<grid_points[2]-1; k++) {
      ...
      lhs[i][j][k][0][0][0] = ...;
      lhs[i][j][k][0][0][1] = ...;
      ...
    }
  }
}
```

(a) Original OpenMP

```
#pragma omp parallel for
for (i=1; i<grid_points[0]-1; i++) {
  for (j=1; j<grid_points[1]-1; j++) {
    for (k=1; k<grid_points[2]-1; ++) {
      ...
      lhs[0][0][0][i][j][k] = ...;
      lhs[0][0][1][i][j][k] = ...;
      ...
    }
  }
}
```

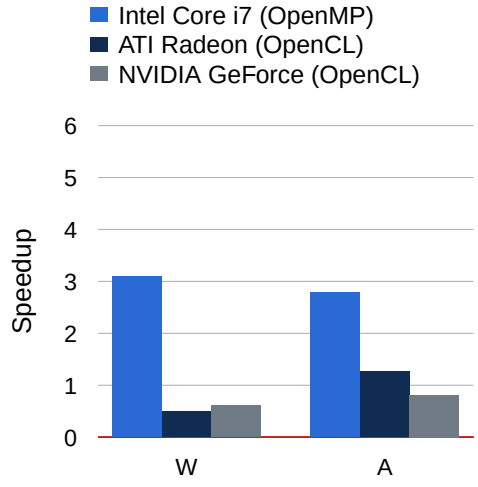
(d) Transformed OpenMP

```
__kernel void lhsy_L1 (...) {
  int k = get_global_id (0) + 1;
  int j = get_global_id (1) + 1;
  int i = get_global_id (2) + 1;
  ...
  lhs[i][j][k][0][0][0] = ...;
  lhs[i][j][k][0][0][1] = ...;
  ...
}
```

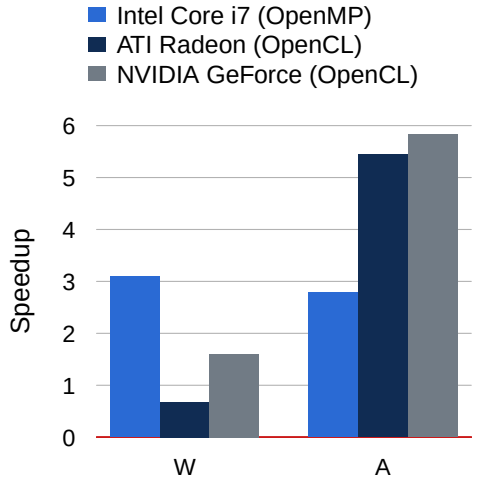
(b) Non-optimized OpenCL

```
__kernel void lhsy_L1 (...) {
  int k = get_global_id (0) + 1;
  int j = get_global_id (1) + 1;
  int i = get_global_id (2) + 1;
  ...
  lhs[0][0][0][i][j][k] = ...;
  lhs[0][0][1][i][j][k] = ...;
  ...
}
```

(e) Optimized OpenCL



(c) Before optimizations



(f) After optimizations

Figure 6.1: Simplified example of generating OpenCL code from OpenMP code. The top left code (a) snippet is taken from `bt`. The corresponding OpenCL code (b) delivers poor performance on both GPUs (c). After applying data transformation to the OpenMP code (d), the new OpenCL code shown in (e) is obtained. The performance of both GPUs improves significantly, but only for large inputs can they outperform the multi-core CPU (f).

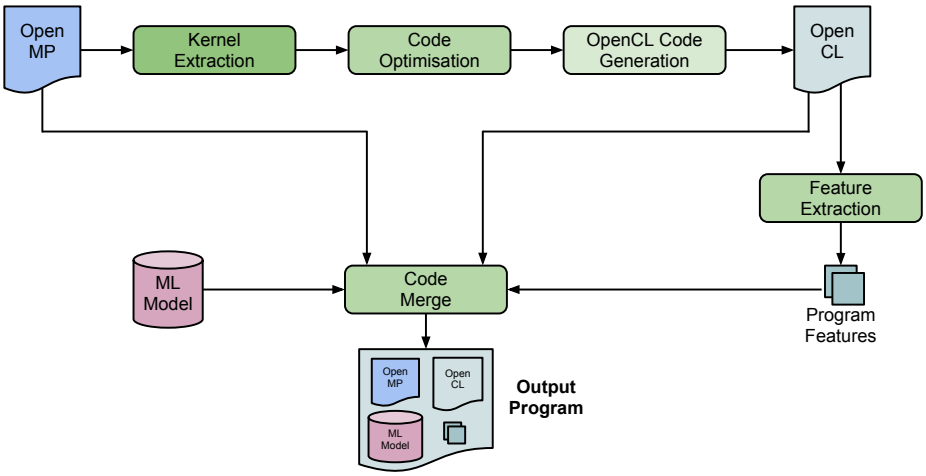


Figure 6.2: The compiler identifies kernels within the OpenMP program and performs several optimizations before generating the OpenCL code. This code is passed to our feature extraction tool to collect code features. In the final step the original OpenMP code, generated OpenCL code, code features and a machine learning model that is built off-line are merged into a single output program.

way. It generates multi-versioned code, the original OpenMP parallel loop and an optimized OpenCL kernel alternative. At runtime, a predictive model decides which version to use for execution. The prototype compiler is implemented using Clang and LLVM.

COMPILE-TIME Figure 6.2 gives an overview of our approach. The OpenMP program is read in and parallel loops are optimized and translated to OpenCL kernels. The generated kernels are passed to a feature extraction phase which collects characteristics or features of the generated code. These features are later used by the predictive model to select whether the OpenMP loop or OpenCL kernel version is best (see section 6.5). The features, together with the generated OpenCL code, the original OpenMP code and a machine learning predictor built off-line are merged into a single output program.

RUN-TIME At execution, the generated program first updates the parameterized features based on the runtime values of parameters and passes the updated features to the predictive model. The built-in model then predicts where to run the program and to pick either the OpenMP code for the multi-core CPU or the OpenCL code for the GPU. Evaluating the model at runtime involves on the order of tens of operations and is thus negligible.

This is a high-level overview of the compilation framework. The following sections describe each of the stages in more detail.

6.4 CODE GENERATION AND OPTIMIZATION

The compiler framework currently converts OpenMP parallel loops, i. e. loops that are annotated with `omp for` or `omp for reduction`, into OpenCL kernels. A standard two-stage algorithm (AMD/ATI, 2013) is used to translate a parallel reduction loop. Other parallel OpenMP directives associated with task parallelism are not currently supported. Each parallel OpenMP loop is translated to a separate kernel using the OpenCL API where each iterator is replaced by a global work-item ID. The compiler currently does not split candidate loops into smaller kernels nor merge loops to generate larger kernels.

For each parallel loop, the loop body is outlined and two versions are generated for it: an OpenCL and an OpenMP version. The original loop body is replaced with a function pointer which points to either the OpenCL or the OpenMP version of the loop. Each generated program has a prediction function that decides where the code is to run and which sets the function pointers to the corresponding version. This is described in section 6.5.

For each array that is used by both the host and the GPU two copies are managed: one in the host memory and the other in the GPU memory. The runtime records the status of each variable and checks whether the copy on a device memory space is valid or not. No memory transfer is needed as long as the copy in the target memory space is valid.

6.4.1 *OpenCL Code Optimization*

The compiler performs a number of optimizations to improve the performance of the OpenCL code on the GPU. The optimizations are discussed below, presented in the order in which they are applied.

The first two transformations, loop interchange and global index reodering, both aim to improve memory access behaviour. These transformations typically have the largest impact on performance. Loop interchange is applied first, because it is a local and thus less radical transformation. Global index reordering is only performed if it is possible to further improve memory access behaviour. The remaining optimizations perform local transformations to improve the performance of the OpenCL kernel on the GPU.

LOOP INTERCHANGE High memory bandwidth on GPUs can only be achieved when memory accesses are coalesced (see section 2.1.2). Memory coalescing happens when adjacent work-items access adjacent memory locations in the GPU off-chip memory. OpenMP programs are typically written for CPUs though which achieve better performance when adjacent memory locations are accesses by the *same thread*

because of caching. The framework thus applies loop interchange when optimising loop nests for GPU execution. It places outermost those iterators that occur most frequently in the innermost array subscripts, as shown in figure 6.1b.

Loop interchange can only be performed safely if there are no dependencies across iterations of the body of the loop nest. Otherwise the transformation would change the semantics of the program. A loop dependence algorithm (Kennedy and Allen, 2002) can be used to detect to which level the nested loop can be interchanged. For simplicity, this analysis was performed manually in this thesis.

GLOBAL INDEX REORDERING Global index reordering is the data structure equivalent of loop reordering. This transformation is necessary when loop interchange cannot provide memory coalescing. Similarly to loop nests, multi-dimensional arrays in OpenMP programs are typically laid out so that a single thread accesses adjacent elements of the array. The compiler changes the layout of these arrays to enable memory coalescing on the GPU. An example of this transformation was shown in figure 6.1 where the layout of the `lhs` array was transformed: $[i, j, k, 0, 0, 0] \mapsto [0, 0, 0, i, j, k]$.

MEMORY LOAD REORDERING In the original OpenMP programs, some accesses of read-only buffers can be reordered to form a sequence of consecutive load operations which can be vectorized. The compiler automatically detects those candidates and replaces scalar load operations with an OpenCL vector load. This can improve the memory performance of the generated OpenCL code.

REGISTER PROMOTION On many occasions, a global memory scalar variable (or array) is accessed multiple times by a single OpenCL kernel. To reduce global memory latencies, the compiler automatically creates a private register object for such variables. It generates code to load the data from the global memory to the private register copy (or write back to the global memory from the register for the last store operation). This allows the generated OpenCL kernel to reuse the object in the private register multiple times and the global memory operation only needs to be performed once. Doing so can eliminate redundant global memory loads and stores and thus improve performance.

PREFETCHING AND LOCAL MEMORY OPTIMIZATION The compiler automatically identifies read-only buffers that are used by multiple OpenCL work-items and generates code to prefetch those buffers to local memory. Exploiting local memory further reduces the memory latencies for GPU kernels.

6.4.1.1 *Dynamic Index Reordering*

In conjunction with loop interchange, global index reordering is often sufficient to achieve memory coalescing. However, in some cases there is no clear best global data layout, e.g. when different loops in a program require different layouts of an array to achieve memory coalescing. In this case *dynamic* index reordering (Che et al., 2011) is considered.

Before entering a code region containing loops that prefer a certain index order for an array X (different from the global one), a reordered copy of the array, X', is created. Within the code region all references to X are redirected to X' and the indexes are reordered appropriately. At the end of the region the data gets reordered and copied back to the original array.

Dynamic index reordering for GPU computing can often be prohibitive due to the high costs of reordering data on the fly. The transformation should only be applied if the benefits of data coalescing outweigh the costs of data reordering. In a compiler setting, a mechanism is therefore needed to *automatically* determine when this transformation should be applied. Section 6.6 describes a data-driven approach that solves this problem.

6.5 PREDICTING THE MAPPING

A crucial part of the approach presented in this chapter is to automatically determine the best execution target for the input program, i. e. should it be run on the multi-core CPU or translated to OpenCL and executed on the GPU. The idea presented here is to generate the OpenCL-based code and then use a predictive model to see if this is profitable to run on a GPU. If it is not profitable the original OpenMP code will be executed on the CPU. As this decision will vary greatly depending on GPU architecture and the maturity of the OpenCL runtime, a portable model that can adapt to the change of the architecture and runtime is desirable.

The model used to make that choice is a decision tree classifier where at every node of the tree a decision is made whether to follow the left or the right child. The C4.5 algorithm (Quinlan, 1993) is used to automatically construct the decision tree from training data by correlating the features to the best device. Decision trees have been introduced in section 2.3.3.2.

The predictive models built in the two previous chapters were all based on Support Vector Machines (SVMs). Both types of models, SVMs and decision trees, can be used for classification but they have different advantages and disadvantages. Decision trees are conceptually simpler than SVMs, while SVMs are more suitable for modeling complex problems (Huang et al., 2003). This comes at the cost, however, that instances of SVMs are hard to examine once they have been built. Decision trees, on the other hand, are intuitive to understand.

The modeling problem in this chapter involves two classes, CPU or GPU execution, compared to the eleven classes in the two previous chapters. Due to the problem's lesser complexity, decision trees are able to achieve a similar accuracy to SVMs in this case. Since decision trees are easier to analyse (an example will be shown in section 6.8.4) they were chosen instead of SVMs.

As in the previous chapters the predictive model is based on code features extracted by the static analysis tool introduced in section 2.4.1.1. This avoids the need for any additional profiling runs or exhaustive search over different data sets. The exact set of features used is presented below.

6.5.1 *Training the Predictor*

The training process involves the collection of training data which is used to fit the model to the problem at hand. In this case a set of programs is used that are each executed on the CPU and the GPU to determine the best device for each one of them. Furthermore, static code features are extracted for each program. The features together with the best device for each program form the training data used to build the model. Since training is only performed once at the factory, it is a one-off cost.

6.5.2 *Code Features*

Programs are characterized by analysing the OpenCL code generated by the compiler using the static feature extraction tool introduced in section 2.4.1.1. As before the raw features are combined to provide more meaningful information to the model. Various combinations of features were evaluated and the best performing set was chosen for the final evaluation. The list of features is shown in table 6.1.

The `comp` feature used in F1 and F4 represents the total number of compute operations. Double precision floating point operations are given a higher weight (4x) than single precision operations. A potential feature is the amount of control flow in an application. While this feature can have an impact on performance on the GPU it was not relevant for the benchmarks we considered. It is thus not included in our feature set.

OpenMP programs often contain multiple parallel loops that are translated to OpenCL kernels by the compiler. In this case the features of all kernels are aggregated into a single set of features by adding up the individual kernel's features.

6.5.2.1 *Collecting Training Data*

Two sets of benchmarks are used to train the model in this chapter. First we use a collection of 47 OpenCL kernels (previously used in chapter 4) taken from various sources: SHOC (Danalis et al., 2010), Parboil (of Illinois at Urbana-Champaign, 2013),

Code Features		
F1:	`transfer/(comp+mem)`	communication-computation ratio
F2:	`coalesced/mem`	% coalesced memory accesses
F3:	`(localmem/mem)` × `avgws`	ratio local to global mem accesses × avg. # work-items per kernel
F4:	`comp/mem`	computation-memory ratio

Table 6.1: List of code features.

NVIDIA CUDA SDK (NVIDIA Corp., 2013) and AMD Accelerated Parallel Processing SDK (AMD/ATI, 2013). These benchmarks are mostly single precision with only one kernel in each program whereas the NAS benchmarks are double precision and have multiple kernels. We thus also add the NAS benchmarks to our training set, but exclude the one that we make a prediction for (see section 6.7.2).

6.5.3 *Runtime Deployment*

Once the machine learning-based model has been built, it is inserted together with the code features to the generated code for any new programs, so that the model can be used at runtime.

UPDATING FEATURES At compile time the OpenCL kernel code is analysed and code features are extracted and inserted to the generated program together with the trained model. As some loop bounds are dependent on the tasks' input data, the compiler may not be able to determine the value of certain features. In this case the compiler represents these features with static symbolic pre-computation of loop bound variables. At execution time, these features are updated using runtime values.

VERSION SELECTION The first time a kernel is called the built-in predictive model selects a code version for execution. It uses updated features to predict the best device to run the program on and sets the function pointer of each parallel loop to the corresponding code version. In the current implementation, prediction happens once during a program's execution. The overhead of prediction is negligible (a few microseconds). This cost is included in our later results.

6.6 A MODEL FOR DYNAMIC INDEX REORDERING

Section 6.4.1.1 described the dynamic index reordering transformation. This transformation can greatly improve performance on the GPU but it can also lead to slow-

downs if the cost of reordering the data is higher than the benefits. Because the point at which the benefits outweigh the costs is highly machine-dependent, a portable machine learning-based approach is used that can be easily adapted to different systems. Similar to predicting the mapping it is based on a decision tree classifier. The features of the model are the size of the data structure and the ratio between the number of accesses to the data structure and its size.

We use micro benchmarks to obtain the training data for this problem.[1] Given a simple kernel accessing an array in a non-coalesced way we vary the array size and the number of times the kernel is called, thereby changing the number of accesses to the array. We measure the execution time with and without applying dynamic index reordering to determine whether it is beneficial in each case. Evaluating the benchmarks and then building the decision tree model takes less than half an hour.

The resulting model is embedded into each output program because array dimensions and loop bounds may not be known at compile time. We thus keep two versions of each candidate kernel: the original one and one with accesses reordered. At runtime one of them gets chosen by the model.

6.7 EXPERIMENTAL METHODOLOGY

This section introduces the experimental setup used in the evaluation of the approach presented in this chapter, including the platforms and benchmarks used. It further describes the methodology used in evaluating the approach.

6.7.1 *Experimental Setup*

PLATFORMS The approach presented in this chapter is primarily evaluated on two CPU-GPU systems, both of which use an Intel Core i7 6-core CPU. One system contains an NVIDIA GeForce GTX 580 GPU, the second an AMD Radeon 7970. They both run with the Ubuntu 10.10 64-bit OS. The compiler was GCC with the -O3 option enabled. Table 6.2 gives detailed information on our platforms.

In section 6.8.5 a brief evaluation on systems with integrated GPUs is shown. The first system contains an AMD Llano (A8-3850) chip comprising a quad-core CPU and an AMD Radeon HD 6550D GPU. The second system is based on an Intel Ivy Bridge (Core i5 3570K) chip comprising a quad-core CPU and an Intel HD Graphics 4000 GPU.

BENCHMARKS All eight of the NAS parallel benchmarks (v2.3) (NASA, 2013) were used for evaluation. The NAS benchmarks were written for general-purpose machines and therefore present a good insight into the feasibility of running general-

1 We opted for micro benchmarks because the amount of training data from real applications is limited.

	Intel CPU	NVIDIA GPU	AMD GPU
Model	Core i7 3820	GeForce GTX 580	Radeon 7970
Core Clock	3.6 GHz	1544 MHz	925 MHz
Core Count	4 (8 w/ HT)	512	2048
Memory	12 GB	1.5 GB	3 GB
Peak Performance	122 GFLOPS	1581 GFLOPS	3789 GFLOPS
Operating System	Ubuntu 10.10 64-bit		
Compiler	GCC 4.4.1 w/ -O3		

Table 6.2: Hardware platform

purpose code on heterogeneous systems. For each benchmark a maximum of five different input sizes (S, W, A, B, C) were evaluated subject to the data fitting in the GPU's memory.

6.7.2 *Evaluation Methodology*

The model is trained and evaluated using the leave-one-out cross-validation method introduced in section 2.5. This means that the target program to be predicted is removed from the training program set and we then build a model based on the remaining programs. This procedure is repeated for each NAS benchmark in turn. This approach is not necessary for the dynamic index reordering model because we use micro benchmarks as training data rather than the programs themselves.

In addition to evaluating the accuracy of the predictive model, the performance of the whole approach is compared to two other methods. Firstly, it is compared to SNU (Seo et al., 2011), a manual OpenCL implementation of the NAS benchmark suite. Secondly, the performance is compared to OpenMPC (Lee et al., 2009), a compiler translating OpenMP code to CUDA code. In both approaches, the programs are always executed on the GPU.

6.8 EXPERIMENTAL RESULTS

This section evaluates the approach presented in this chapter on two separate heterogeneous systems for the NAS parallel benchmark suite. First the performance of the predictive modeling approach is shown compared to always using the multi-core CPU or always the GPU. This is followed by a comparison to a manual OpenCL implementation of the NAS benchmark suite (Seo et al., 2011) and OpenMPC (Lee et al., 2009), the closest related prior work, showing performance improvements of

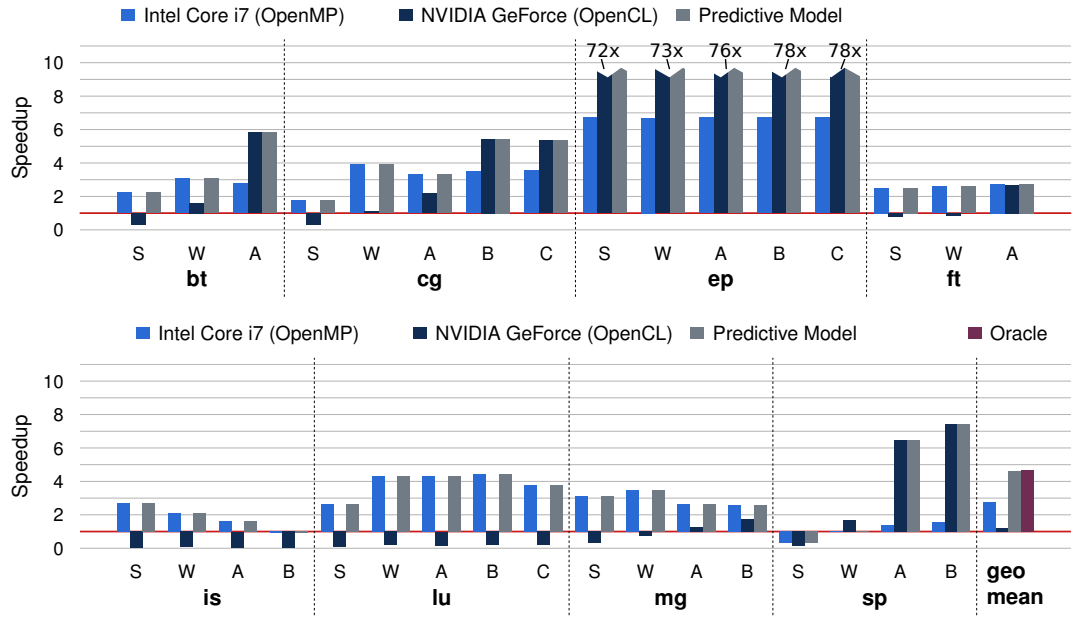


Figure 6.3: Performance of OpenMP on the Intel CPU, OpenCL on the NVIDIA GeForce GPU and the version selected by the predictive model. The predictive model outperforms the CPU-only approach by 1.69x and the GPU-only approach by 3.9x.

1.4x and 10x respectively. The section then takes a closer look at the predictive model and finally concludes with a brief evaluation of the approach on two heterogeneous systems with integrated GPUs.

6.8.1 *Performance Evaluation*

Figures 6.3 and 6.4 show speed-ups for the NAS benchmarks on the two heterogeneous systems described in section 6.7. For each benchmark-input pair the multi-core CPU performance, the GPU performance and the performance of the device selected by our predictor is shown. The last set of columns represents the average performance (using the geometric mean) of each approach as well as of the oracle which always picks the best device in each case. The performance numbers presented are speed-ups over single-core execution.

On both systems significant speed-ups can be achieved by selecting the right device, CPU or GPU. When always selecting the faster of the two, speed-ups of 4.70x on the NVIDIA system and 4.81x on the AMD system can be achieved. This compares to 2.78x and 2.74x when always using the multi-core CPU[2] and 1.19x and 0.71x on the GPU.

The results show that speed-ups vary dramatically between CPU and GPU and none of the devices consistently outperforms the other. On `ep`, for example, an embar-

2 Even though the same CPU was used in both cases the numbers vary slightly because the benchmarks sets are different due to memory constraints on the GPUs.

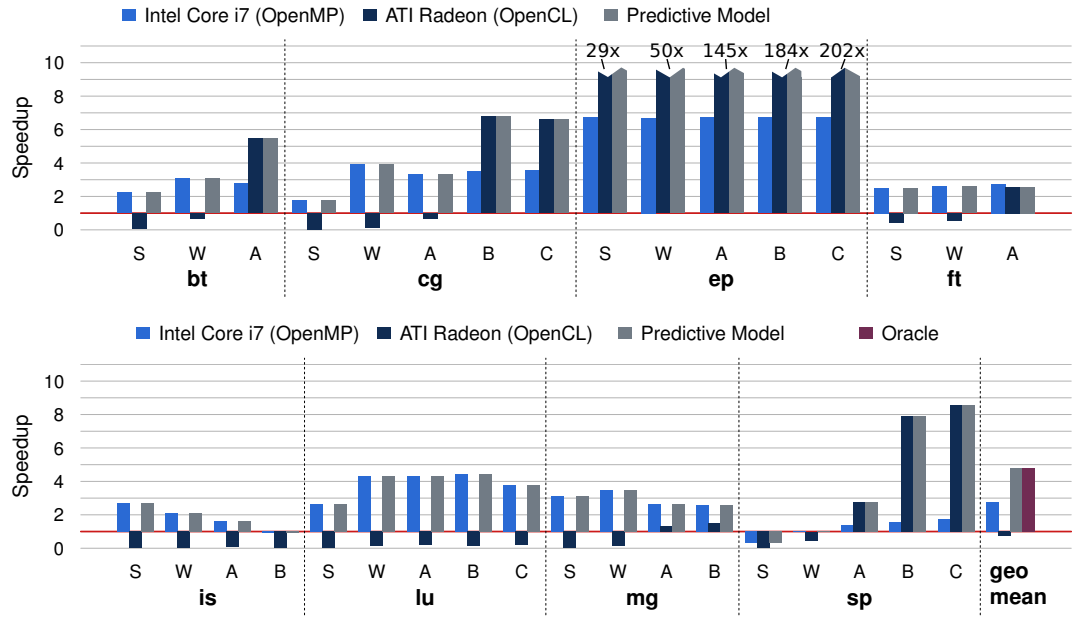
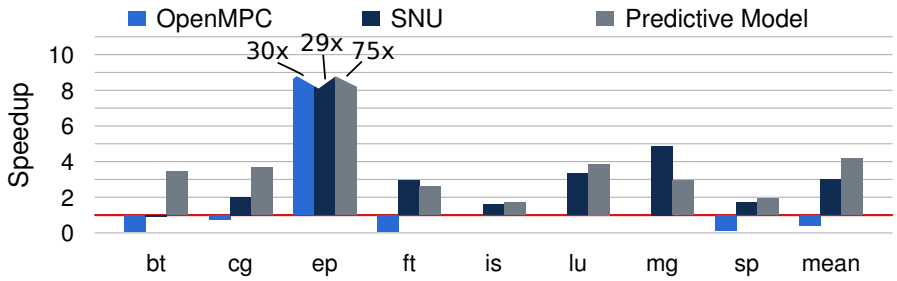


Figure 6.4: Performance of OpenMP on the Intel CPU, OpenCL on the AMD Radeon GPU and the version selected by the predictive model. The predictive model outperforms the CPU-only approach by 1.76x and the GPU-only approach by 6.8x.

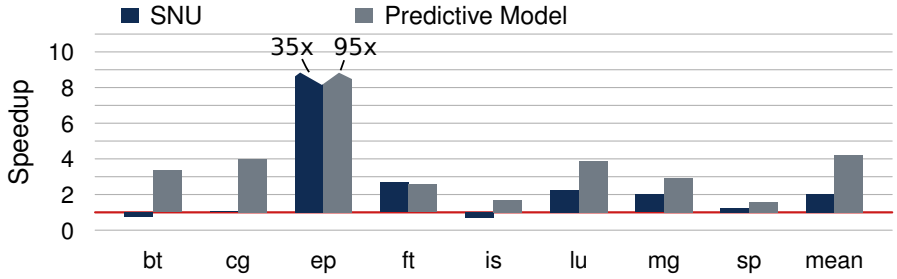
rassingly parallel benchmark, the GPU clearly outperforms the multi-core CPU: up to 11.6x on NVIDIA and 30.2x on AMD. However, on other benchmarks, such as `is` or `lu` the CPU is significantly faster. In the case of `lu` this is because the OpenMP version exploits pipeline parallelism using a combination of asynchronous parallel loops and a bit-array to coordinate pipeline stages. The current SIMD-like execution models of GPUs are not designed to exploit this type of parallelism. The `is` benchmark does not perform significant amounts of computation and GPU execution is dominated by communication with the host memory. This leads to underutilization of the GPU and thus poor performance.

For benchmarks `bt`, `cg` and `sp` we observe that the CPU is faster for small inputs but the GPU is better on larger input sizes. This behaviour is to be expected because GPUs require large amounts of computation to fully exploit their resources. On small inputs the overheads of communication with the host dominate the overall runtime when using the GPU. A similar pattern is shown for `ft` and `mg`: GPU performance is stronger for larger inputs. However, the GPU is not able to beat the CPU even for the largest input sets. For *is*, because the program does not have enough parallelism, it is actually not worthwhile to run it in parallel for any given data set on our platforms. This is also reported in other studies (Tournavitis et al., 2009).

These observations show the need for a careful mapping of applications to devices. The predictive model is able to choose the correct device almost all of the time. On the NVIDIA system it incorrectly picks the GPU for benchmark `sp.W` and on the AMD system it picks the GPU for `ft.A` even though the CPU is faster. Overall it is able



(a) NVIDIA GeForce



(b) AMD Radeon

Figure 6.5: Speed-up averaged across inputs of the OpenMPC compiler (NVIDIA only), the manual SNU implementation of the NAS parallel benchmarks and our predictive model.

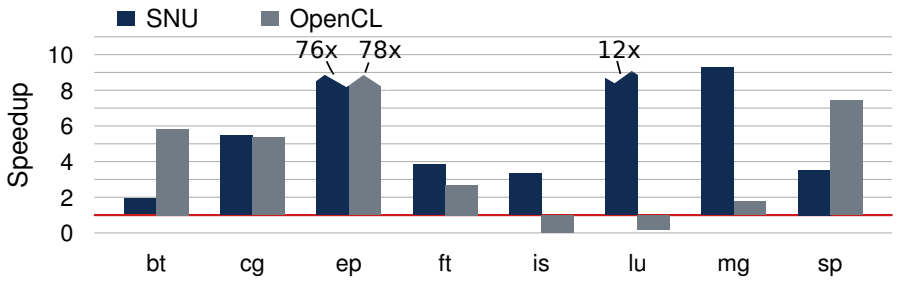
to achieve speed-ups of 4.63x and 4.80x respectively. This is significantly better than always choosing the same device and not far off the performance of the "oracle".

6.8.2 *Comparison to State-of-the-Art*

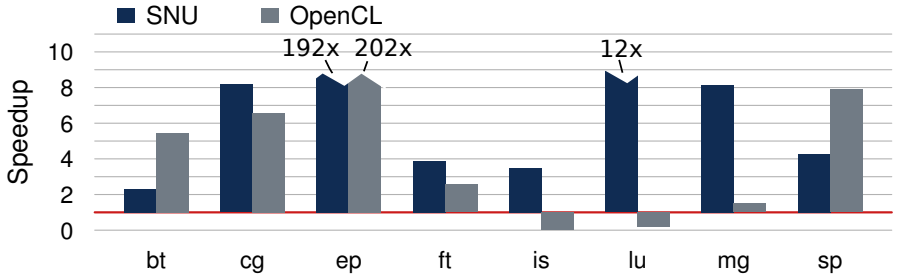
The approach presented in this chapter is compared to two others: OpenMPC (Lee et al., 2009), a compiler translating OpenMP to CUDA, and the SNU NPB suite (Seo et al., 2011) which provides independently hand-written OpenCL implementations of the NAS parallel benchmarks. The results are shown in figure 6.5. Because OpenMPC generates CUDA code, we only evaluated it on the NVIDIA platform. We were unable to generate code for benchmarks `is`, `lu` and `mg` using OpenMPC.

With the exception of `ep`, the OpenMPC-generated code performs poorly compared to the other approaches. It only achieves a mean speed-up of 0.42x, i. e. slower than sequential execution. The main reason is that OpenMPC does not perform any kind of data transformation, leading to uncoalesced memory accesses in many cases. On average, our approach outperforms OpenMPC by a factor of 10.

The SNU implementation shows significantly better performance than OpenMPC but it is only able to outperform this chapter's approach on `ft` and `mg` on the NVIDIA platform and only on `ft` on the AMD platform. On average it achieves a speed-up of 3.01x on NVIDIA and 2.02x on AMD compared to 4.18x and 4.22x of the predictive



(a) NVIDIA GeForce



(b) AMD Radeon

Figure 6.6: Speed-up of the SNU implementation and our generated OpenCL code for the largest possible input size.

modeling approach. There are two reasons why the approach presented here is able to outperform the hand-written versions of the benchmarks. Firstly, it performs aggressive data restructuring, including dynamic index reordering, which is especially important for benchmarks `bt` and `sp`. Secondly, it is able to detect when using the CPU over the GPU is the better option. So while the SNU code may be faster than the OpenCL code generated by our approach for some benchmarks (as we will show in the next section), we are often able to provide better performance by choosing *not* to use the GPU but the CPU.

We also tried to evaluate the benchmarks using the OpenACC standard. However, some OpenMP features, e. g. threadprivate variables, are not supported by OpenACC and the compiler infrastructure is not very mature. Substantial effort in rewriting the code as well as more mature compilers are required to perform a fair comparison.

6.8.3 *OpenCL Code Performance Comparison*

Figure 6.6 compares the generated OpenCL code to the hand-written SNU implementation. It factors out the predictive model and focuses solely on the quality of the generated OpenCL code. The largest possible input size is selected for each benchmark.

The data show mixed results. For benchmarks `bt` and `sp` the generated code outperforms the hand-written code. This is mainly due to the data restructuring by the compiler. For `cg`, `ep` and `ft` the speed-ups are similar. On the remaining benchmarks, `is`, `lu` and `mg`, the generated code is not as good as the SNU implementation.

The SNU version of `lu` uses a different algorithm than the original OpenMP code (Seo et al., 2011, Section 3.1). Their implementation uses a hyperplane algorithm which is much more suited for GPU execution. Changing the algorithm is obviously out of the scope of a compiler-based approach. For `is` the SNU implementation uses atomic operations to compute a histogram and a parallel prefix sum algorithm which is not exposed in the OpenMP code. These types of optimizations are beyond the scope of a compiler. The code for `mg` works on highly irregular multi-dimensional data structures. In the generated code these data structures are flattened and indirection in each dimension is used to navigate through the data. The SNU implementation uses a different approach that requires a single level of indirection which leads to vastly improved performance. For `ft`, the SNU code removes the array boundary checks in the primary reduction loop because the programmer knows it is safe to do so for certain inputs. This results in better GPU performance as branch overhead is eliminated.

Overall our generated OpenCL code performs well. The hand-coded versions generally only perform better when algorithmic changes or code restructuring is performed which is beyond the scope of a compiler.

6.8.4 *Analysis of Predictive Model*

Figures 6.7 and 6.8 show the decision trees constructed for the two systems by excluding `bt` from the training set. The learning algorithm automatically places the most relevant features at the root level and determines the architecture-dependent threshold for each node. All this is done automatically without the need of expert intervention.

As an example, the features for benchmark `bt` are shown in table 6.3.[3] The features are shown both before and after applying data transformations according to the example shown in section 6.2. This demonstrates the impact of the transformations on the mapping decision.

At the root of the tree in figure 6.7 is the value for the communication-computation ratio (F1). In all versions the value is far below the threshold. The evaluation thus proceeds to the left subtree until reaching the fourth level of the tree. This node looks at the percentage of coalesced memory accesses (F2). Without data transformations none of the accesses are coalesced and the left branch is taken, eventually leading to CPU execution. With data transformations memory coalescing has been improved (see bold values in table 6.3b). For input sizes S and W the percentage of coalesced

3 The feature values for all benchmarks can be found at `http://homepages.inf.ed.ac.uk/s0898672/cgo2013-features.tgz`.

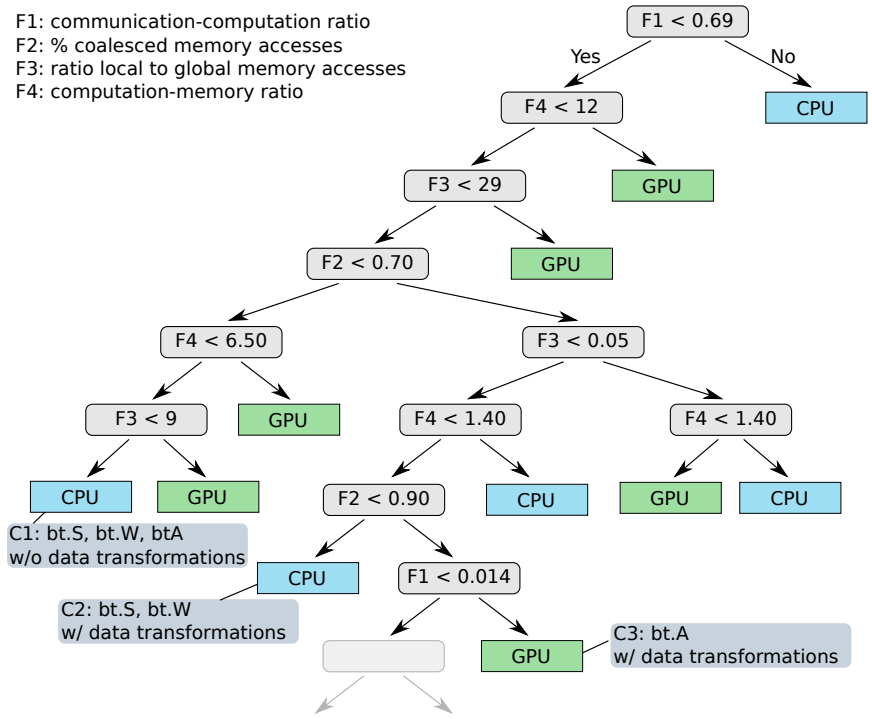


Figure 6.7: The model used for `bt` on NVIDIA GeForce. Predictions for `bt` with and without data transformations are marked as C1, C2 and C3.

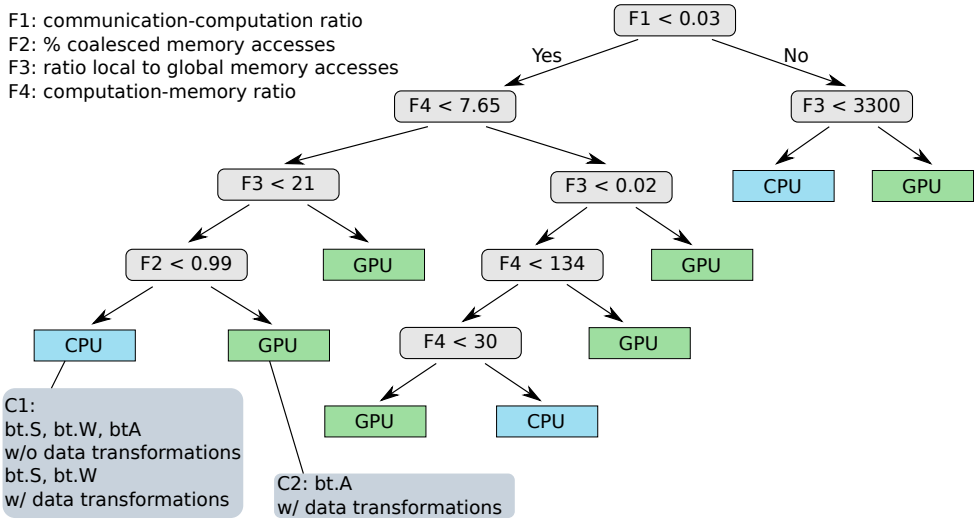


Figure 6.8: The model used for `bt` on AMD Radeon. Predictions for `bt` with and without data transformations are marked as C1 and C2.

	F1	F2	F3	F4
S	0.0020	0	0	0.81
W	0.0004	0	0	0.82
A	0.0003	0	0	0.82

(a) w/o data transformations

	F1	F2	F3	F4
S	0.0020	*0.731*	0	0.81
W	0.0004	*0.780*	0	0.82
A	0.0003	*0.999*	0	0.82

(b) w/ data transformations

Table 6.3: Features of `bt`

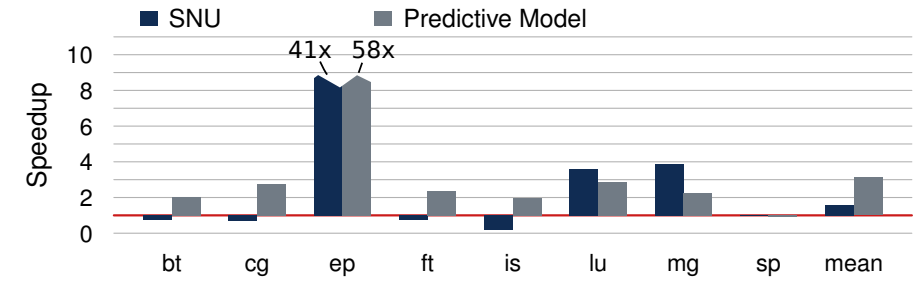
accesses is less than 80%. For A almost all accesses are coalesced due to dynamic index reordering (see section 6.4.1.1). All values are above the threshold so the right branch is taken. The evaluation follows the same branches until another node of F2 is reached. This time the threshold is higher, namely 0.9. For input sizes S and W the left branch is taken which leads to execution on the CPU. For the larger input size A the right branch is taken and eventually a leaf node is reached predicting to run on the GPU. All programs get mapped to the correct device.

Figure 6.8 shows the decision tree constructed for the AMD Radeon system. The first node also looks at the ratio between communication and computation. The threshold is lower (0.03), but for all versions of `bt` the ratio is still below the threshold. The same path is followed by all versions until the fourth level of the tree is reached. At this point the percentage of coalesced accesses is considered. The versions without data transformations are mapped to the CPU because none of the accesses are coalesced. Even when applying data transformations, for input sizes S and W the value is below the threshold and the code gets mapped to the CPU. Only input size A is mapped to the GPU. All programs are again mapped to the right device.

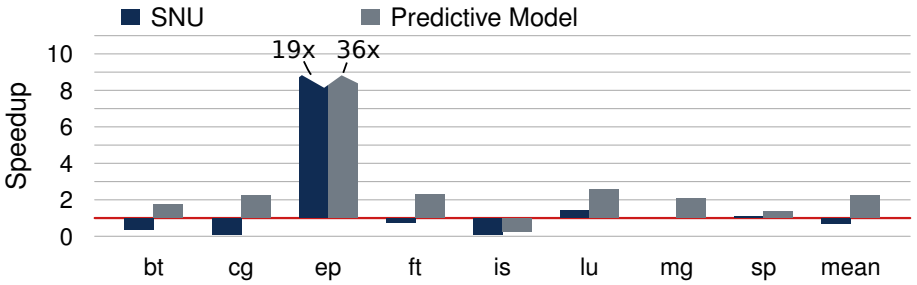
6.8.5 *Performance on Integrated Systems*

GPU technology is constantly evolving. To check the approach presented in this chapter also works on new devices it was evaluated on two systems with integrated GPUs, AMD Llano and Intel Ivy Bridge. Figure 6.9 shows a summary of the results using the SNU implementation and our predictive modeling approach. The manual SNU code only achieves speed-ups of 1.6x and 0.7x on average compared to the 3.1x and 2.2x of our approach.

The integrated GPUs on these systems are less powerful than the discrete GPUs evaluated before. This demonstrates even more the need for a model that only maps code to the GPU when it is beneficial. Integrated GPUs share the system memory with the CPU, making data movements between the devices cheaper or even unnecessary in the case of Intel IvyBridge. Because most benchmarks in the NAS parallel



(a) AMD Llano



(b) Intel Ivy Bridge

Figure 6.9: Speed-up averaged across inputs of the manual SNU code and our model on systems with integrated GPUs.

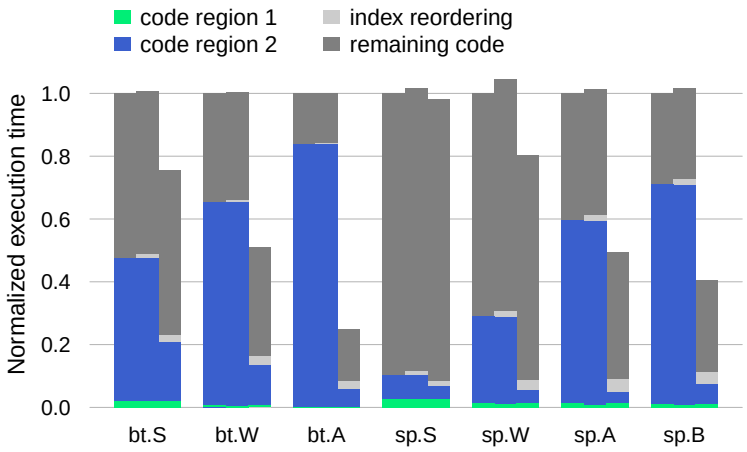
benchmark suite are compute-intensive this advantage does not lead to improved performance overall.

6.9 EVALUATION OF DYNAMIC INDEX REORDERING

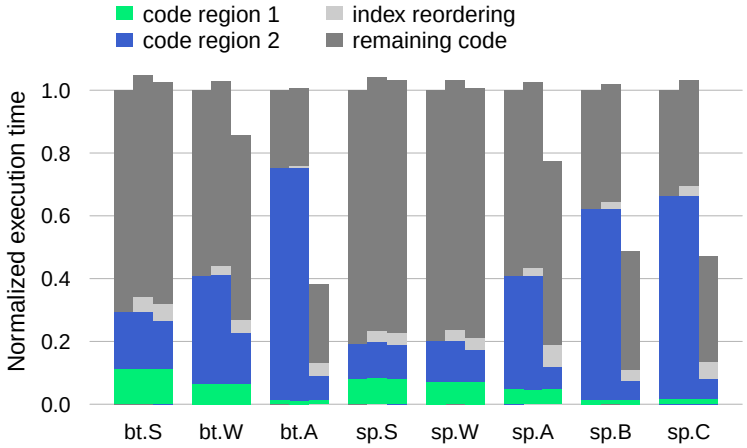
This section evaluates the dynamic index reordering transformation introduced in section 6.6. First, the performance impact of applying the transformation is considered, followed by a study of the predictive model used to decide when to apply the transformation.

6.9.1 *Performance Impact*

The benchmarks `bt` and `sp` contain candidate regions for dynamic index reordering. Figure 6.10 shows the performance of the benchmarks with different input sizes when applying dynamic index reordering to none, the first or the second of those regions. The performance is normalized to the runtime when the transformation is not applied. In each case the runtime is broken up into the runtimes for the two candidate code regions, the overhead of the transformation (if applicable) and the rest of the program.



(a) NVIDIA GeForce

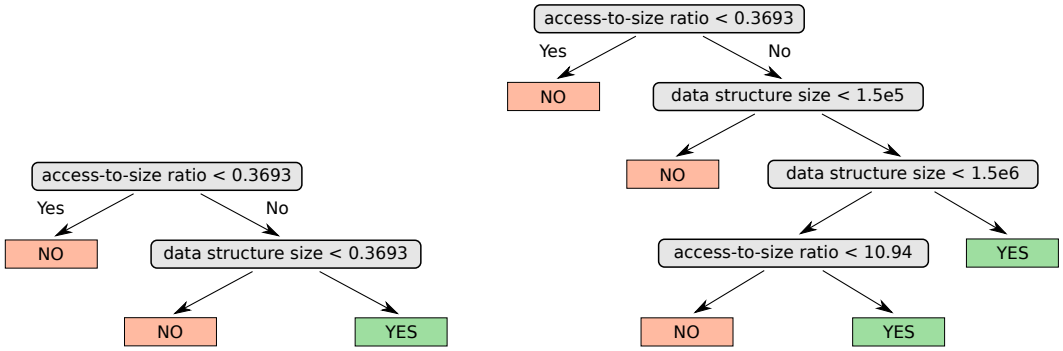


(b) AMD Radeon

Figure 6.10: Performance impact of dynamic index reordering when applying the transformation to none, the first or the second of the candidate regions. The runtime is broken up into the runtimes for the two candidate code regions, the runtime of the transformation (if applicable) and the rest of the program.

The first candidate region makes up only a small fraction of the overall runtime of both benchmarks; 1-3% on the NVIDIA and 1-11% on the AMD system. When applying dynamic index reordering here the performance of this region barely improves, because there are not many memory accesses that benefit from the transformation. The cost of reordering the data thus often outweighs the benefits which leads to minor slow-downs overall.

The second region, on the other hand, makes up a larger chunk of the overall runtime. Applying dynamic index reordering significantly reduces the runtime of this region. Since the overhead of data reordering is comparatively small, big overall runtime reductions are achieved by applying the transformation to this region: up to 75% on the NVIDIA system and 62% on the AMD system.



(a) NVIDIA GeForce (b) AMD Radeon

Figure 6.11: Decision trees for dynamic index reordering.

6.9.2 *Analysis of Predictive Model*

Similar to predicting which device to run a program on, decision trees are used to determine when dynamic index reordering is beneficial (see section 6.6 for details). The decision trees for the NVIDIA and the AMD system are depicted in figure 6.11. In both cases the most important feature (at the root of the tree) is the ratio of accesses to the size of the data structure. If this ratio is below 0.37 the transformation is not worthwhile because the benefit is too small to outweigh the cost. If the ratio is above the threshold the size of the data structure is considered next. For small data structures the transformation is not applied because the overhead of the transformation is too high. The threshold for NVIDIA (570000) is higher than the one for AMD (150000) which is consistent with the data shown in figure 6.10. On NVIDIA it is always worthwhile applying the transformation in code region 2 whereas on AMD it is only beneficial for large data sizes. The tree for NVIDIA ends here but for AMD up to two more levels have to be evaluated before a decision is reached.

Table 6.4 shows the features as described in section 6.4.1.1 for the `sp` benchmark with input size `B`. There are two regions of code which each have two candidate arrays for dynamic index reordering. While the corresponding data sizes are very similar the access-to-size ratio varies a lot between the two code regions. None of the arrays in the first region are being transformed because the access-to-size ratio is too small (0.128 and 0.073). The other ones are transformed, however, because of the higher access-to-size ratio (5.948 and 7.626) and a large enough size to hide any overheads. As was shown in figure 6.10 these choices are correct on both systems. Transforming the arrays in region 1 leads to a slow-down but for region 2 dynamic index reordering improves performance significantly.

On average across the `bt` and `sp` benchmarks, the predictive model achieves an accuracy of 79% on the NVIDIA and 94% on the AMD system.

	Region 1		Region 2	
data structure size	5.5e6	5.5e6	5.5e6	16.4e6
access-to-size ratio	0.128	0.073	5.948	7.626

Table 6.4: Features for dynamic index reordering on benchmark `sp` with input size `B`.

6.10 SUMMARY

This chapter has described a compilation approach that takes shared memory programs written in OpenMP and outputs OpenCL code targeted at GPU-based heterogeneous systems. The proposed approach uses loop and array transformations to improve the memory behaviour of the generated code. The performance of this method is evaluated on two different platforms, NVIDIA GeForce and AMD Radeon, on the whole NAS parallel benchmark suite where the OpenCL code generated can produce significant speedups (up to 202x). However, GPUs are not best suited for all programs and in some cases it is more profitable to use the host multi-core CPU instead. This chapter has developed an approach based on machine learning that determines for each new program whether the multi-core CPU or the GPU is the best target. If the multi-core is selected the appropriate OpenMP code is executed as it currently outperforms OpenCL on multi-cores. This model is learned on a per platform basis and we demonstrate that the model adapts to different platforms and achieves consistent prediction accuracy. The approach achieves 1.39x and 2.09x improvement over a hand-coded, GPU-specific implementation.

In this chapter the last contribution of this thesis has been presented. The next chapter concludes this document by providing a brief summary of all contributions as well as a critical analysis and ideas for future work.

7

CONCLUSION

This thesis has introduced several methods for tackling two of the challenges of heterogeneous computing. Solutions for the task mapping problem have been discussed in chapters 4 and 5. They both use machine learning techniques and static code features to predict the optimal mapping of OpenCL programs. While the method introduced in chapter 4 assumes that no other programs are running on the system the technique from chapter 5 can handle contention on the GPU. The challenge of code generation and tuning is tackled in chapter 6. It presents a compiler-based method for translating OpenMP code to OpenCL which is tuned for GPU execution. At the beginning of each execution a predictive model is evaluated that determines whether to execute the original OpenMP code on the multi-core CPU or the generated OpenCL code on the GPU.

The structure of this chapter is as follows. Section 7.1 briefly summarizes each of the contributions presented in this thesis. These are critically analyzed in section 7.2. Section 7.3 concludes the chapter with a discussion of possible future work.

7.1 CONTRIBUTIONS

This section summarizes the main contributions presented in the previous three chapters. Both the task mapping and the code generation and tuning challenge have been addressed.

7.1.1 *Task Mapping*

Two machine learning-based techniques for partitioning OpenCL kernels across CPUs and GPUs have been developed in this thesis.

The first one, introduced in chapter 4, uses static code features and minimal runtime information, namely the input size, to determine the optimal mapping of work between the two devices. A predictive model determines how to statically split the work between the devices, thus minimizing scheduling overheads. Other approaches either require expensive offline profiling of each application (Luk et al., 2009) or they rely on dynamic mapping techniques (Ravi et al., 2010). Dynamic mapping techniques increase the scheduling overhead and, as was shown in section 4.5, they can perform poorly, especially when the entire workload should be mapped to a single device. The technique proposed in this thesis avoids all of those drawbacks.

The predictive model used to determine the mapping is built automatically using training data. It is thus easily portable to new hardware by simply repeating the fully automatic training process on the new machine.

Across a suite of 47 OpenCL kernels, each with multiple input sizes, the proposed method achieved an average speed-up of 14.3 over single-core execution. This compares to a speed-up of 9.13 of a dynamic task farm mapper and of 4.73 and 9.21 when only using the multi-core CPU or the GPU, respectively.

7.1.1.1 *Task Mapping in the Presence of GPU Contention*

Chapter 5 considered task mapping in situations where other applications are competing for the GPU. This case has been neglected by task mapping approaches thus far, which typically assume that the system is idle. Since heterogeneous computing is becoming increasingly popular in desktop and mobile computing, this assumption is often no longer valid.

Intuitively, dynamic mapping approaches, such as the task farm mapper presented by Ravi et al. (2010), should be able to deal with resource contention. While this may be true for contention on the CPU it was shown that these techniques fail to achieve good performance in GPU contention. This is because GPU tasks have exclusive access to the GPU and cannot currently be preempted.

This thesis developed a task mapping approach that explicitly takes GPU contention into account. Using minimal information on the GPU contention in addition to static code features it determines the optimal partitioning between the CPU and GPU for a kernel in the current environment.

This approach was evaluated over a set of 22 benchmarks and 10 different contention scenarios. Compared with two dynamic mappers, task farm and online search, it achieved speed-ups of 2.56 and 1.54 respectively.

7.1.2 *Code Generation and Tuning*

A compiler-based approach for executing OpenMP programs on heterogeneous systems has been proposed in chapter 6. It combines code generation and tuning with task mapping.

The proposed method translates parallel loops in the input OpenMP program into OpenCL kernels. It then applies several transformations with the aim of improving performance on GPUs. The main focus is on data layout transformations which have a significant impact on GPU performance. Where static layout transformation do not suffice dynamic transformations that change the layouts of arrays at runtime are considered. Since these transformations incur a cost, a novel technique has been developed to determine whether or not dynamic layout transformations are beneficial or not.

Once the OpenCL code has been generated and tuned for GPU execution both the original OpenMP and the generated OpenCL versions are merged into an output binary. At runtime a predictive model, trained offline, determines whether to execute the OpenMP version on the multi-core CPU or the OpenCL version on the GPU. This decision is based on static code features as well as information about input sizes.

The method has been evaluated on the entire NAS benchmark suite and compared to both a competing compiler-based approach (Lee et al., 2009) and hand-tuned OpenCL implementations of the benchmarks (Seo et al., 2011). On two different platforms, NVIDIA-based and AMD-based, the approach achieves average speed-ups of 1.39 and 2.09 over the hand-written implementation. Compared to the competing compiler-based approach it achieves a speed-up of 10 on the NVIDIA-based systems.

7.2 CRITICAL ANALYSIS

While this thesis has presented some significant contributions to the field of heterogeneous computing, some aspects of the used methods demand for a critical analysis. This section briefly discusses these issues.

7.2.1 *Machine Learning-Based Task Mapping*

All task mapping methods presented in this thesis have been based on machine learning techniques. One can argue that these techniques are overly complex for task mapping. If only 11 different partitionings are considered, for example, one could simply evaluate them all initially and then use the fastest one from there on. If the same program is run many times with the same input size and on the same machine this may be a feasible solution. However, if the input size or hardware is changing frequently this process would have to be repeated many times. Furthermore, if more complex scenarios are considered, e. g. the possibility of contention on the GPU, this approach becomes infeasible.

A shortcoming of machine learning-based task mapping, and indeed of any fully static task mapping approach, is that the mapping is not adjusted over time even if it is known to not be optimal. After a task has been executed for the first time one could measure the running times on the different devices and check if the mapping was optimal, i. e. the workload was balanced across the devices. There are two possible fixes to this problem. Firstly, the observed running times could be fed back into the machine learning model, thus improving the model over time. Secondly, the predicted mapping could be adapted over time, similar to the online search method introduced in chapter 5.

Two other aspects related to machine learning techniques, training data and static code features, are discussed in the following two sections. These discussions are not

specific to using machine learning for task mapping but apply to machine learning-based program optimization in general.

7.2.2 *Training Data*

Machine learning techniques rely heavily on the quality of the training data. It is crucial that the training data cover a wide range of the input feature space of the predictive model. Otherwise the model is not able to make accurate predictions for data points that lie well outside the region covered by the training data. Machine learning models are thus typically trained with hundreds or thousands of data points. When used for program optimization, however, it is difficult to find this many training benchmarks. A trick that can be used (and has been used in this thesis) is to use the same benchmark multiple times but with different input sizes. This is only a partial solution, however, because it does not add to the variety of training benchmarks.

A possibly better solution would be to generate training data automatically so as to not rely on the limited supply of actual programs. The difficulty with this approach is that the generated benchmarks must be as complex as actual programs. Automatically generated benchmarks have been used in section 6.6 for training the model for dynamic data layout transformations. In this simple case it was possible to generate these benchmarks automatically. However, for more complex problems this will be significantly harder. To the best of the author's knowledge automatic benchmark generation has never been used for applying machine learning techniques to complex optimization problems.

Another issue is the cost of collecting the training data. The training benchmarks do not immediately form the training data, rather the data is *derived* from the benchmarks. This process involves both extracting features from the benchmarks and finding the target values, i. e. the optimal configuration for each training benchmark. In the case of static code features the feature extraction phase is very short. However, finding the optimal configuration can be time-consuming. In the case of program partitioning 11 different partitionings have been evaluated in this thesis. Each partitioning has been further executed multiple times in order to ensure stable results. It can therefore take several hours or even days to collect the training data, depending on the number of training benchmarks. It has to be noted, however, that this process only happens once, namely when the model is trained. It is a one-off cost paid "at the factory" and thus not paid by the user.

7.2.3 *Static Code Features*

Throughout this thesis static code features have been used to characterize pieces of code, primarily OpenCL kernels. Static code features are cheap to collect as they

do not require any profiling runs. However, they may not be very accurate in some situations.

Firstly, static code features are only accurate when the number of instructions (including loops) can be statically determined. While this is true for the majority of OpenCL benchmarks there are cases where, for example, a loop bound depends on the input data values. In this case static code features cannot accurately characterize the code without inspecting the input data.

Secondly, there are some characteristics that are difficult to capture using only static analysis, e. g. memory behaviour. Cache hit and miss rates, for example, can be collected during profiling runs but they are hard to extract statically. However, some things can be captured, at least to some extent, such as the number of coalesced memory accesses.

7.2.4 *Data Transformations*

Chapter 6 presented data layout transformations. These transformations are crucial to achieve good performance when transforming OpenMP code to GPU-specific OpenCL code. However, they can be difficult to implement. The benchmarks used in chapter 6 were primarily array-based, i. e. the data was stored in statically allocated (multi-dimensional) arrays. For programs with more complex data structures, e. g. pointer-based data, these transformation will require thorough analysis techniques to understand and manipulate the layout of the data.

7.2.5 *Choice of High-Level Language*

Chapter 6 presented an approach for targeting heterogeneous systems from OpenMP. OpenMP was chosen because it is an established standard. There are many benchmarks available and it is heavily used, especially in the high-performance computing domain. Furthermore OpenMP is very convenient for the user because it is based on existing languages, C and Fortran, and allows users to easily upgrade applications by simply annotating existing programs.

The drawback of OpenMP is that, partially because it is based on existing languages, programs can be very complex to analyze. Performing transformations on the code can therefore be difficult as the previous section discussed. Furthermore, it is hard to distribute the work of single kernels across devices with separate memory spaces because it requires the runtime to know how to split the data accordingly. For this reason the mapping method used in chapter 6 only ever maps kernels entirely to the CPU or the GPU.

The alternative is to use more abstract languages, e. g. based on functional programming languages, or even domain-specific languages. These languages provide

more information to the compiler and can therefore be analyzed more easily. The disadvantage is, however, that many programmers are not familiar with these kinds of languages. In addition, these languages are typically less expressive, thus restricting the programs that can be developed.

The choice of the high-level language when targeting heterogeneous systems is important. However, there is no right or wrong answer. Different approaches come with different strengths and weaknesses. When choosing a language one should consider the type of programmer the language is targeted at as well as the performance one wants to achieve.

7.3 FUTURE WORK

This section briefly discusses ideas on how the work presented in this thesis could be extended in the future.

7.3.1 *Different Optimization Metrics*

All approaches discussed in this thesis had the goal of improving the performance of applications. However, other optimization metrics can be considered as well, e. g. energy consumption which is becoming increasingly important in modern computer systems.

The machine learning-based techniques presented in this thesis can be easily adapted to other optimization targets. It only involves picking the target configuration of an application that optimizes the chosen metric and retraining the model. This way various metrics, such as energy or energy-delay product (EDP), could be optimized for.

7.3.2 *Hybrid Task Mapping*

The presented task mapping approaches in this thesis were static, i. e. the work was fully divided between the CPU and GPU before a kernel was launched. The advantage of this approach is that it minimizes scheduling overheads. However, if a wrong decision is made it cannot be rectified. The task mapping methods could be made more flexible by combining them with elements of dynamic mapping strategies, thus making them hybrid static-dynamic techniques.

A possible strategy would be to use a dynamic scheduler whenever the static predictor is not confident enough. Many machine learning techniques attach a confidence value to their predictions. If, for example, an application is very different from any of the training applications the confidence of the predictor is likely to be low. In these situations a dynamic mapper could be used as a fallback strategy.

Another possibility would be to use machine learning models to predict parameter values of the dynamic strategies, such as the number and size of chunks in a task farm. A model could also be used to address other shortcomings of task farm mappers. It could, for example, predict whether a task should be mapped entirely to the CPU or the GPU, similar to the first level of the predictor shown in chapter 4. Only if the model predicts that work should be assigned to both devices would the task farm mapper be used to balance the work.

7.3.3 *Mapping Programs with Multiple Kernels*

Two different ways of dealing with programs with multiple kernels have been used in this thesis. In chapter 4 only single kernels have been considered so this issue did not need to be addressed.

In chapter 5, however, programs with multiple kernels were allowed. In this case a prediction for each kernel was made individually. Because the targeted system allowed for the CPU and GPU to share the same memory no data needed to be transfered when successive kernels used different mappings. But even on these systems it may be beneficial to make application-wide mapping decisions rather than per-kernel decisions, e. g. in order to improve cache behaviour.

The majority of benchmarks used in chapter 6 contained multiple kernels. Since the main targets in that chapter were systems with discrete GPUs, using different mappings for kernels in the same applications is likely to incur significant data transfer costs. Therefore a single mapping, CPU-only or GPU-only, was used for all kernels in an application. On systems where the CPU and GPU share the same memory kernel-specific mapping decisions could further improve performance. Even on systems with discrete GPUs this could be beneficial if the cost of data transfers is taken into account.

7.4 SUMMARY

This chapter concludes the thesis. The chapter has provided a summary of all the main contributions of this thesis to the field of heterogeneous computing. Various aspects of this work have been critically analyzed and ideas for future work have been presented.

A

BENCHMARKS USED IN CHAPTER 4

Tables A.1 to A.8 present the complete list of the 220 kernel and input size pairs used for evaluation in chapter 4.

Kernel	Input sizes
stream_sdk_AES_encrypt	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
	width=16384 height=16384
stream_sdk_AES_decrypt	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
	width=16384 height=16384
stream_sdk_binarySearch_binarySearch	length=1048576 subdivSize=8
	length=2097152 subdivSize=8
	length=4194304 subdivSize=8
	length=8388608 subdivSize=8
	length=16777216 subdivSize=8
	length=33554432 subdivSize=8
	length=67108864 subdivSize=8
stream_sdk_blackscholes_blackscholes	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
stream_sdk_boxFilter_boxFilter	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
stream_sdk_boxFilter_horizontalSAT0	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
stream_sdk_boxFilter_horizontalSAT	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
stream_sdk_boxFilter_verticalSAT	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096

Table A.1: OpenCL kernels and input sizes used in chapter 4.

Kernel	Input sizes
stream_sdk_boxFilter_vertical	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
stream_sdk_boxFilter_horizontal	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
stream_sdk_boxFilter_horizontalLocal	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
stream_sdk_DCT_DCT	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
stream_sdk_histogram_histogram256	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
stream_sdk_histogramAtomics_histogramGlobal	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192
stream_sdk_histogramAtomics_histogramLocal	width=1024 height=1024
	width=2048 height=2048
	width=4096 height=4096
	width=8192 height=8192

Table A.2: OpenCL kernels and input sizes used in chapter 4.

Kernel	Input sizes
stream_sdk_	width=1024 height=1024 maxIter=32
mandelbrot_	width=2048 height=2048 maxIter=32
mandelbrot	width=4096 height=4096 maxIter=32
	width=8192 height=8192 maxIter=32
stream_sdk_	widthB=1024 widthA=1024 heightA=1024
matrixMul_	widthB=2048 widthA=2048 heightA=2048
matrixMul	widthB=4096 widthA=4096 heightA=4096
	widthB=8192 widthA=8192 heightA=8192
stream_sdk_	widthB=1024 widthA=1024 heightA=1024
matrixMul_	widthB=2048 widthA=2048 heightA=2048
matrixMul_local	widthB=4096 widthA=4096 heightA=4096
	widthB=8192 widthA=8192 heightA=8192
stream_sdk_	widthB=1024 widthA=1024 heightA=1024
matrixMul_	widthB=2048 widthA=2048 heightA=2048
matrixMul_local2	widthB=4096 widthA=4096 heightA=4096
	widthB=8192 widthA=8192 heightA=8192
stream_sdk_	width=1024 mulFactor=2 height=1024
mersenneTwister_	width=1024 mulFactor=4 height=1024
gaussianRand	width=2048 mulFactor=2 height=2048
	width=2048 mulFactor=4 height=2048
stream_sdk_monteCarloAsian_	noOfSum=12 width=1024 height=1024
calPriceVega	noOfSum=12 width=2048 height=2048
stream_sdk_	numBodies=32768
nbody_	numBodies=65536
nbody_sim	numBodies=131072
	numBodies=262144
	numBodies=524288
	numBodies=1048576
shoc_fft_fft1D	n_ffts=16384
	n_ffts=32768
	n_ffts=65536

Table A.3: OpenCL kernels and input sizes used in chapter 4.

Kernel	Input sizes
shoc_fft_ifft1D	n_ffts=16384
	n_ffts=32768
	n_ffts=65536
shoc_md_computeAccel	nAtom=32768
	nAtom=65536
	nAtom=131072
	nAtom=262144
	nAtom=524288
	nAtom=1048576
shoc_md_applyBoundary	nAtom=1048576
	nAtom=2097152
	nAtom=4194304
	nAtom=8388608
	nAtom=16777216
	nAtom=33554432
	nAtom=67108864
shoc_md_updateVelocities	nAtom=1048576
	nAtom=2097152
	nAtom=4194304
	nAtom=8388608
	nAtom=16777216
	nAtom=33554432
shoc_md_updateCoordinates	nAtom=1048576
	nAtom=2097152
	nAtom=4194304
	nAtom=8388608
	nAtom=16777216
	nAtom=33554432

Table A.4: OpenCL kernels and input sizes used in chapter 4.

Kernel	Input sizes
shoc_scan_addUniform	numElements=1048576
	numElements=2097152
	numElements=4194304
	numElements=8388608
	numElements=16777216
	numElements=33554432
	numElements=67108864
shoc_scan_scan	numElements=1048576
	numElements=2097152
	numElements=4194304
	numElements=8388608
	numElements=16777216
	numElements=33554432
	numElements=67108864
shoc_sgemm_sgemmNT	N=1024
	N=2048
	N=4096
	N=8192
shoc_sgemm_sgemmNN	N=1024
	N=2048
	N=4096
	N=8192
nvidia_sdk_blackscholes_blackscholes	optn=10000000
	optn=15000000
	optn=20000000
	optn=25000000
	optn=30000000
	optn=35000000
	optn=40000000

Table A.5: OpenCL kernels and input sizes used in chapter 4.

Kernel	Input sizes
nvidia_sdk_ convolutionSeparable_ convolutionColumns	imageWidth=1024 kernelRadius=8 imageHeight=1024 imageWidth=2048 kernelRadius=8 imageHeight=2048 imageWidth=4096 kernelRadius=8 imageHeight=4096 imageWidth=8192 kernelRadius=8 imageHeight=8192
nvidia_sdk_ convolutionSeparable_ convolutionRows	imageWidth=1024 kernelRadius=8 imageHeight=1024 imageWidth=2048 kernelRadius=8 imageHeight=2048 imageWidth=4096 kernelRadius=8 imageHeight=4096 imageWidth=8192 kernelRadius=8 imageHeight=8192
nvidia_sdk_dotProduct_ dotProduct	length=262144 length=524288 length=1048576 length=2097152 length=4194304 length=8388608 length=16777216
nvidia_sdk_matrixMul_ matrixMul	widthB=1024 widthA=1024 heightA=1024 widthB=2048 widthA=2048 heightA=2048 widthB=4096 widthA=4096 heightA=4096 widthB=8192 widthA=8192 heightA=8192
nvidia_sdk_matVecMul_ matVecMulUncoalescedo	width=1024 height=1024 width=2048 height=2048 width=4096 height=4096 width=8192 height=8192
nvidia_sdk_matVecMul_ matVecMulUncoalesced1	width=1024 height=1024 width=2048 height=2048 width=4096 height=4096 width=8192 height=8192

Table A.6: OpenCL kernels and input sizes used in chapter 4.

Kernel	Input sizes
nvidia_sdk_	width=1024 height=1024
matVecMul_	width=2048 height=2048
matVecMulCoalesced0	width=4096 height=4096
	width=8192 height=8192
nvidia_sdk_	width=1024 height=1024
matVecMul_	width=2048 height=2048
matVecMulCoalesced1	width=4096 height=4096
	width=8192 height=8192
nvidia_sdk_	width=1024 height=1024
matVecMul_	width=2048 height=2048
matVecMulCoalesced2	width=4096 height=4096
	width=8192 height=8192
parboil_	voly=1024 volx=1024 atoms=2000
cp_	voly=2048 volx=2048 atoms=2000
cuenergy	voly=4096 volx=4096 atoms=2000
	voly=8192 volx=8192 atoms=2000
parboil_	numx=1048576 numk=1024
mri-fhd_	numx=2097152 numk=1024
computeFH	numx=4194304 numk=1024
	numx=8388608 numk=1024
	numx=16777216 numk=1024
	numx=33554432 numk=1024

Table A.7: OpenCL kernels and input sizes used in chapter 4.

Kernel	Input sizes
parboil_mri-fhd_computeRhoPhi	numk=32768
	numk=65536
	numk=131072
	numk=262144
	numk=524288
	numk=1048576
	numk=2097152
	numk=4194304
	numk=8388608
	numk=16777216
	numk=33554432
parboil_mri-q_computeQ	numx=1048576 numk=1024
	numx=2097152 numk=1024
	numx=4194304 numk=1024
	numx=8388608 numk=1024
	numx=16777216 numk=1024
	numx=33554432 numk=1024

Table A.8: OpenCL kernels and input sizes used in chapter 4.

BIBLIOGRAPHY

Agakov, F. V., Bonilla, E. V., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 295–305. (Cited on page 53.)

Alpaydin, E. (2004). *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press. (Cited on page 17.)

AMD (2013). Accelerated parallel processing (APP) SDK. `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`. (Cited on page 85.)

AMD/ATI (2013). ATI Stream SDK. `http://www.amd.com/stream/`. (Cited on pages 66, 102, and 106.)

Amini, M., Coelho, F., Irigoin, F., and Keryell, R. (2013). Static compilation analysis for host-accelerator communication optimization. In *Languages and Compilers for Parallel Computing*, pages 237–251. Springer. (Cited on page 47.)

Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. (2010). Data-aware task scheduling on multi-accelerator based platforms. In *Proceedings of IEEE 16th International Conference on Parallel and Distributed Systems*, pages 291–298. (Cited on page 36.)

Augonnet, C. and Namyst, R. (2008). A unified runtime system for heterogeneous multi-core architectures. In *Euro-Par Workshops*, pages 174–183. (Cited on page 36.)

Augonnet, C., Thibault, S., and Namyst, R. (2009a). Automatic calibration of performance models on heterogeneous multicore architectures. In *Proceedings of the Euro-Par Parallel Processing Workshops*, pages 56–65. (Cited on page 36.)

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009b). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference*, pages 863–874. (Cited on pages 36, 37, and 51.)

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198. (Cited on pages 36, 37, and 51.)

Barros, R. C., Cerri, R., Jaskowiak, P. A., and de Carvalho, A. (2011). A bottom-up oblique decision tree induction algorithm. In *Proceedings of 11th International Conference on Intelligent Systems Design and Applications*, pages 450–456. (Cited on page 23.)

Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. (2008). A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 225–234. (Cited on pages 45 and 49.)

Baskaran, M. M., Ramanujam, J., and Sadayappan, P. (2010). Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th International Conference on Compiler Construction*, pages 244–263. (Cited on page 49.)

Beaumont, O., Legrand, A., and Robert, Y. (2003). Optimal algorithms for scheduling divisible workloads on heterogeneous systems. In *Proceedings of 17th International Parallel and Distributed Processing Symposium*. (Cited on page 33.)

Beaumont, O., Legrand, A., Robert, Y., Carter, L., and Ferrante, J. (2002). Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *Proceedings of 16th International Parallel and Distributed Processing Symposium*. (Cited on page 33.)

Becchi, M., Byna, S., Cadambi, S., and Chakradhar, S. T. (2010a). Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 82–91. (Cited on page 37.)

Becchi, M., Cadambi, S., and Chakradhar, S. T. (2010b). Enabling legacy applications on heterogeneous platforms. In *Proceedings of 2nd USENIX Workshop on Hot Topics in Parallelism*. (Cited on page 38.)

Becchi, M. and Crowley, P. (2006). Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 29–40. (Cited on page 35.)

Belviranli, M. E., Bhuyan, L. N., and Gupta, R. (2013). A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization*, 9(4):57. (Cited on pages 42 and 43.)

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. (Cited on pages 17, 20, 22, 59, 62, and 63.)

Bordawekar, R., Bondhugula, U., and Rao, R. (2010). Believe it or not!: mult-core CPUs can match GPU performance for a FLOP-intensive application! In *Proceedings of the*

19th International Conference on Parallel Architecture and Compilation Techniques, pages 537–538. (Cited on page 98.)

Boser, B. E., Guyon, I., and Vapnik, V. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the 5th Annual ACM Conference on Computational Learning Theory*, pages 144–152. (Cited on page 22.)

Bottou, L., Cortes, C., Denker, J. S., Drucker, H., Guyon, I., Jackel, L. D., LeCun, Y., Muller, U. A., Sackinger, E., Simard, P., and Vapnik, V. (1994). Comparison of classifier methods: a case study in handwritten digit recognition. In *Proceedings of the 12th International Conference on Pattern Recognition*, pages 77–82. (Cited on page 23.)

Boyer, M., Skadron, K., Che, S., and Jayasena, N. (2013). Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the 10th ACM International Conference on Computing Frontiers*. (Cited on pages 42 and 43.)

Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D. A., and Freund, R. F. (1999). A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Proceedings of the 8th Heterogeneous Computing Workshop*, pages 15–29. (Cited on page 32.)

Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D. A., and Freund, R. F. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837. (Cited on pages 31 and 32.)

Chakravarty, M. M. T., Keller, G., Lee, S., McDonell, T. L., and Grover, V. (2011). Accelerating haskell array codes with multicore GPUs. In *DAMP*. (Cited on page 5.)

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. (Cited on page 23.)

Charles, P., Grothoff, C., Saraswat, V. A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538. (Cited on page 48.)

Che, S., Sheaffer, J. W., and Skadron, K. (2011). Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceesings of the Conference on High Perfor-*

mance Computing Networking, Storage and Analysis, page 13. (Cited on pages 45, 49, and 104.)

Cole, M. I. (1989). *Algorithmic skeletons: structured management of parallel computation*. MIT Press. (Cited on page 50.)

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297. (Cited on page 22.)

Curtis-Maury, M., Dzierwa, J., Antonopoulos, C. D., and Nikolopoulos, D. S. (2006). Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 157–166. (Cited on pages 25 and 54.)

Dagum, L. and Menon, R. (1998). OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1). (Cited on page 13.)

Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. (2010). The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*, pages 63–74. (Cited on pages 65 and 105.)

Davis, E. and Jaffe, J. M. (1981). Algorithms for scheduling tasks on unrelated processors. *J. ACM*, 28(4):721–736. (Cited on page 32.)

Diamos, G. F. and Yalamanchili, S. (2008). Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High-Performance Distributed Computing*, pages 197–200. (Cited on page 37.)

Dolbeau, R., Bihan, S., and Bodin, F. (2007). HMPP: A hybrid multi-core parallel programming environment. In *Proceedings of the 1st Workshop on General Purpose Processing Using Graphics Processing Units*. (Cited on page 52.)

Dubach, C., Cheng, P., Rabbah, R. M., Bacon, D. F., and Fink, S. J. (2012). Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. (Cited on page 50.)

Dubach, C., Jones, T. M., Bonilla, E. V., Fursin, G., and O'Boyle, M. F. P. (2009). Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 78–88. (Cited on pages 24, 25, 30, and 54.)

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*. (Cited on page 51.)

Fodor, I. K. (2002). A survey of dimension reduction techniques. Technical report, Lawrence Livermore National Laboratory. (Cited on page 20.)

Garland, M. and Kirk, D. B. (2010). Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66. (Cited on page 11.)

González-Vélez, H. and Cole, M. (2010). Adaptive statistical scheduling of divisible workloads in heterogeneous systems. *Journal of Scheduling*, 13(4):427–441. (Cited on page 35.)

Grama, A., Karypis, G., Kumar, V., and Gupta, A. (2003). *Introduction to parallel computing*. Addison Wesley. (Cited on page 13.)

Greenhalgh, P. (2011). Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*. (Cited on page 9.)

Gregg, C., Brantley, J., and Hazelwood, K. (2010). Contention-aware scheduling of parallel code for heterogeneous systems. Technical report, Department of Computer Science, University of Virginia. (Cited on pages 38 and 39.)

Grewe, D. and O'Boyle, M. F. (2011). A static task partitioning approach for heterogeneous systems using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction*. (Cited on page 6.)

Grewe, D., Wang, Z., and O'Boyle, M. F. (2013). Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the 11th International Symposium on Code Generation and Optimization*. (Cited on page 7.)

Han, T. D. and Abdelrahman, T. S. (2009). hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. (Cited on page 51.)

Han, T. D. and Abdelrahman, T. S. (2011). hiCUDA: High-level GPGPU programming. *IEEE Transaction on Parallel and Distributed Systems*, 22(1):78–90. (Cited on page 51.)

Hennessy, J. L. and Patterson, D. A. (2012). *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann. (Cited on page 11.)

Hormati, A., Samadi, M., Woh, M., Mudge, T. N., and Mahlke, S. A. (2011). Sponge: portable stream programming on graphics engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–392. (Cited on page 49.)

Horowitz, E. and Sahni, S. (1976). Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327. (Cited on page 32.)

Hsu, C.-W. and Lin, C.-J. (2002). A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425. (Cited on page 23.)

Huang, J., Lu, J., and Ling, C. X. (2003). Comparing naive bayes, decision trees, and svm with auc and accuracy. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 553–556. (Cited on pages 24 and 104.)

Hwu, W.-M. W., Rodrigues, C. I., Ryoo, S., and Stratton, J. A. (2009). Compute unified device architecture application suitability. *Computing in Science and Engineering*, 11(3):16–26. (Cited on pages 10, 16, 43, and 44.)

Ibarra, O. H. and Kim, C. E. (1977). Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289. (Cited on pages 31 and 32.)

Intel (2013). Intel SDK for OpenCL applications 2013 | Intel developer zone. `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`. (Cited on page 85.)

Jablin, T. B., Jablin, J. A., Prabhu, P., Liu, F., and August, D. I. (2012). Dynamically managed data for CPU-GPU architectures. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174. (Cited on page 48.)

Jablin, T. B., Prabhu, P., Jablin, J. A., Johnson, N. P., Beard, S. R., and August, D. I. (2011). Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151. (Cited on pages 43, 47, and 48.)

Jiang, W. and Agrawal, G. (2012). MATE-CG: A map reduce-like framework for accelerating data-intensive computations on heterogeneous clusters. In *26th IEEE International Parallel and Distributed Processing Symposium*, pages 644–655. (Cited on pages 41 and 75.)

Jiménez, V. J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., and Navarro, N. (2009). Predictive runtime code scheduling for heterogeneous architectures. In *High Performance Embedded Architectures and Compilers, Fourth International Conference*, pages 19–33. (Cited on pages 38 and 39.)

Jon Peddie Research (2013). Nvidia winner in Q1, AMD flat, Intel down. (Cited on page 1.)

Kennedy, K. and Allen, J. R. (2002). *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers. (Cited on page 103.)

Khokhar, A. A., Prasanna, V. K., Shaaban, M. E., and Wang, C.-L. (1993). Heterogeneous computing: Challenges and opportunities. *IEEE Computer*, 26(6):18–27. (Cited on page 57.)

Khronos (2013). OpenCL: The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`. (Cited on pages 14, 16, and 36.)

Kim, J.-K., Shivle, S., Siegel, H. J., Maciejewski, A. A., Braun, T. D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R. B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., and Yellampalli, S. S. (2003). Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 98. (Cited on page 32.)

Knerr, S., Personnaz, L., and Dreyfus, G. (1990). Single-layer learning revisited: a stepwise procedure for building and training a neural network. In *Neurocomputing*, pages 41–50. Springer. (Cited on page 23.)

Knijnenburg, P. M. W., Kisuki, T., and O'Boyle, M. F. P. (2002). Iterative compilation. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*, pages 171–187. (Cited on pages 24 and 52.)

Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., and Takizawa, H. (2010). Evaluating performance and portability of OpenCL programs. In *Proceedings of the 5th International Workshop on Automatic Performance Tuning 2010*. (Cited on page 97.)

Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. (2003). Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 81–92. (Cited on page 39.)

Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P. (2005). Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38. (Cited on pages 34 and 57.)

Lee, S. and Eigenmann, R. (2010). OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. (Cited on page 51.)

Lee, S. and Eigenmann, R. (2013). OpenMPC: extended OpenMP for efficient programming and tuning on GPUs. *Internation Journal on Computer Science and Engineering*, 8(1):4–20. (Cited on page 51.)

Lee, S., Min, S.-J., and Eigenmann, R. (2009). OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 101–110. (Cited on pages 48, 51, 98, 108, 111, and 123.)

Lee, S. and Vetter, J. S. (2012). Early evaluation of directive-based GPU programming models for productive exascale computing. In *SC Conference on High Performance Computing Networking, Storage and Analysis*, page 23. (Cited on page 51.)

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 451–460. (Cited on page 98.)

Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H. Y. (2008). Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–296. (Cited on page 41.)

Liu, C., Li, J., Huang, W., Rubio, J., Speight, E., and Lin, X. (2012). Power-efficient time-sensitive mapping in heterogeneous systems. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 23–32. (Cited on page 39.)

Liu, Y., Zhang, E. Z., and Shen, X. (2009). A cross-input adaptive framework for GPU program optimizations. In *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10. (Cited on page 46.)

LLVM (2013). Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`. (Cited on page 25.)

Luk, C.-K., Hong, S., and Kim, H. (2009). Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55. (Cited on pages 40, 41, 43, 57, 60, 65, 75, and 121.)

Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D. A., and Freund, R. F. (1999). Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 59(2):107–131. (Cited on page 34.)

Maheswaran, M. and Siegel, H. J. (1998). A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 57–69. (Cited on pages 34 and 37.)

McIntosh-Smith, S. (2012). Adaptive heterogeneous computing with OpenCL: Harnessing hundreds of GPUs and CPUs. In *Los Alamos National Laboratory Accelerated High Performance Computing Symposium*. (Cited on page 35.)

Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International*

Conference on Artificial Intelligence: Methodology, Systems, and Applications, pages 41–50. (Cited on page 53.)

Moss, J. E. B., Utgoff, P. E., Cavazos, J., Precup, D., Stefanovic, D., Brodley, C. E., and Scheeff, D. (1997). Learning to schedule straight-line code. In *Advances in Neural Information Processing Systems*. (Cited on page 53.)

NASA, Advanced Supercomputing Division. (2013). NAS parallel benchmarks. `http://www.nas.nasa.gov/publications/npb.html`. (Cited on page 107.)

Nielson, F., Nielson, H. R., and Hankin, C. (2005). *Principles of program analysis*. Springer. (Cited on page 26.)

NVIDIA (2009). NVIDIA's next generation CUDA architecture: Fermi. *NVIDIA White Paper*. (Cited on page 10.)

NVIDIA Corp. (2013). NVIDIA Developer Zone. `http://developer.nvidia.com/category/zone/cuda-zone`. (Cited on pages 66 and 106.)

of Illinois at Urbana-Champaign, U. (2013). Parboil benchmark suite, `http://impact.crhc.illinois.edu/parboil.php`. (Cited on pages 65 and 105.)

Olson, D. L. and Delen, D. (2008). *Advanced Data Mining Techniques*. Springer. (Cited on page 22.)

OpenACC (2013). OpenACC: Directives for accelerators. `http://www.openacc-standard.org/`. (Cited on pages 5 and 52.)

Pai, S., Govindarajan, R., and Thazhuthaveetil, M. J. (2012). Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 33–42. (Cited on page 48.)

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. (2013). Improving GPGPU concurrency with elastic kernels. In *Proceedings of the 18th Internation Conference on Architectural Support for Programming Languages and Operating Systems*, pages 407–418. (Cited on page 39.)

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572. (Cited on page 20.)

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106. (Cited on page 23.)

Quinlan, J. R. (1993). *C4.5: programs for machine learning*. (Cited on pages 24 and 104.)

Raman, A., Zaks, A., Lee, J. W., and August, D. I. (2012). Parcae: a system for flexible parallel execution. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 133–144. (Cited on page 77.)

Ravi, V. T. and Agrawal, G. (2011). A dynamic scheduling framework for emerging heterogeneous systems. In *Proceesings of the 18th International Conference on High Performance Computing*, pages 1–10. (Cited on page 42.)

Ravi, V. T., Ma, W., Chiu, D., and Agrawal, G. (2010). Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th International Conference on Supercomputing*, pages 137–146. (Cited on pages 42, 57, 60, 66, 121, and 122.)

Rifkin, R. M. and Klautau, A. (2004). In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141. (Cited on page 63.)

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and mei W. Hwu, W. (2008a). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82. (Cited on pages 43 and 44.)

Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S.-Z., Stratton, J. A., and mei W. Hwu, W. (2008b). Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th International Symposium on Code Generation and Optimization*, pages 195–204. (Cited on pages 44 and 45.)

Ryoo, S., Rodrigues, C. I., Stone, S. S., Stratton, J. A., Ueng, S.-Z., Baghsorkhi, S. S., and mei W. Hwu, W. (2008c). Program optimization carving for GPU computing. *J. Parallel Distrib. Comput.*, 68(10):1389–1401. (Cited on pages 44, 45, and 49.)

Sato, S. and Iwasaki, H. (2009). A skeletal parallel framework with fusion optimizer for GPGPU programming. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, pages 79–94. (Cited on page 50.)

Scogland, T., Rountree, B., chun Feng, W., and de Supinski, B. R. (2012). Heterogeneous task scheduling for accelerated OpenMP. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, pages 144–155. (Cited on page 43.)

Seo, S., Jo, G., and Lee, J. (2011). Performance characterization of the NAS parallel benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, pages 137–148. (Cited on pages 108, 111, 113, and 123.)

Shirahata, K., Sato, H., and Matsuoka, S. (2010). Hybrid map task scheduling for GPU-based heterogeneous clusters. In *Cloud Computing, Second International Conference*, pages 733–740. (Cited on page 41.)

Spafford, K., Meredith, J. S., Lee, S., Li, D., Roth, P. C., and Vetter, J. S. (2012). The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proceedings of the Computing Frontiers Conference*, pages 103–112. (Cited on page 13.)

Stephenson, M. and Amarasinghe, S. P. (2005). Predicting unroll factors using supervised classification. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pages 123–134. (Cited on page 53.)

Stotzer, E., Beyer, J., Das, D., Jost, G., Raghavendra, P., Leidel, J., Duran, A., Narayanaswamy, R., Tian, X., Hernandez, O., Terboven, C., Wienke, S., Koesterke, L., Milfeld, K., Jayaraj, A., and Dietrich, R. (2012). OpenMP technical report 1 on directives for attached accelerators. Technical report, The OpenMP Architecture Review Board. (Cited on page 52.)

Sun, E., Schaa, D., Bagley, R., Rubin, N., and Kaeli, D. (2012). Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 84–93. (Cited on page 38.)

Thies, W., Karczmarek, M., and Amarasinghe, S. P. (2002). Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196. (Cited on page 49.)

Topcuoglu, H., Hariri, S., and Wu, M.-Y. (1999). Task scheduling algorithms for heterogeneous processors. In *Heterogeneous Computing Workshop*, pages 3–14. (Cited on pages 33 and 37.)

Topcuoglu, H., Hariri, S., and Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274. (Cited on pages 33, 36, and 37.)

Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. F. P. (2009). Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–187. (Cited on pages 54 and 110.)

Udupa, A., Govindarajan, R., and Thazhuthaveetil, M. J. (2009). Software pipelined execution of stream programs on GPUs. In *Proceedings of the 7th International Symposium on Code Generation and Optimization*, pages 200–209. (Cited on page 49.)

Ueng, S.-Z., Lathara, M., Baghsorkhi, S. S., and mei W. Hwu, W. (2008). Cuda-lite: Reducing GPU programming complexity. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, pages 1–15. (Cited on pages 44 and 49.)

Wang, Z. and O'Boyle, M. F. P. (2009). Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 75–84. (Cited on page 54.)

Wang, Z. and O'Boyle, M. F. P. (2010). Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*, pages 307–318. (Cited on page 54.)

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D. B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P. P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, 7(3). (Cited on page 39.)

Wolfe, M. (2010). Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*, pages 43–50. (Cited on page 52.)

Wu, B., Zhao, Z., Zhang, E. Z., Jiang, Y., and Shen, X. (2013). Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 57–68. (Cited on page 46.)

Yang, Y., Xiang, P., Kong, J., Mantor, M., and Zhou, H. (2012). A unified optimizing compiler framework for different GPGPU architectures. *ACM Transactions on Architecture and Code Optimization*, 9(2):9. (Cited on pages 45 and 49.)

Yang, Y., Xiang, P., Kong, J., and Zhou, H. (2010). A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 86–97. (Cited on page 45.)

Zhang, E. Z., Jiang, Y., Guo, Z., and Shen, X. (2010). Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th International Conference on Supercomputing*, pages 115–126. (Cited on page 46.)

Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. (2011). On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the 16th International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 369–380. (Cited on page 46.)