Analysis and Optimization of Parallel Programs under the Insieme Compiler and Runtime System

PhD thesis in computer science

by

Klaus Kofler

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of doctor of philosophy

advisor: Univ.-Prof. Dr. Thomas Fahringer, Institute of Computer Science
second advisor: Univ.-Prof. Dr. Dr. h.c. Michael Oberguggenberger

Innsbruck, February 16, 2017

Certificate of Authorship and Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Klaus Kofler, Innsbruck, February 16, 2017

Abstract

Recent developments in computer hardware have resulted in increasingly parallel and often heterogeneous computing systems combining several processing units of different kinds. Such systems impose new challenges to the programmers, as writing parallel, heterogeneous programs is not only more error prone and typically also more time consuming than writing programs for sequential systems, but a programmer must also take into account all levels of parallelism to obtain high performance. Not only do the different processing units in a heterogeneous system favor different code optimizations, the workload distribution among the available computational resources is also crucial.

To support the programmer and making the creation of parallel heterogeneous programming more efficient, it is desirable to automate the program optimization process as much as possible. However, current compiler and runtime systems still have a large potential for improvements in that regard. The purpose of this thesis is to advance the state of the art in automatic optimization of parallel heterogeneous programs. It presents a detailed survey of modern parallel and heterogeneous systems and describes the challenges that programmers need to overcome to achieve good performance. It introduces several novel approaches to automate parts of the optimization process, thereby making programming parallel, heterogeneous system easier and more efficient. To perform automatic code analysis and transformations, we use the Insieme compiler and runtime system which has been extend with new features as part of the research described within this thesis.

To guide our research and help programmers understand what design patterns work best for which processing unit, we analyze the different performance characteristics of various OpenCL devices using a set of microbenchmarks. Additionally, we present two novel techniques to speed up the execution of OpenCL programs. The first technique automatically distributes the workload of OpenCL programs over several processing units while the second one automatically determines the tile size for a given OpenCL program/GPU combination. Furthermore, we introduce an auto-tuner which optimizes OpenMP programs with regard to several objectives using iterative compilation. Finally, we demonstrate the benefits of GPGPU computing using OpenCL to execute scientific simulations on the example of two domain applications.

Acknowledgements

During the years of study for my PhD degree, I enjoyed the support of many people, both study related and in my private life. I would like to thank all of them in the following paragraphs. Foremost, my advisor Univ.-Prof. Dr. Thomas Fahringer, for his support and guidance as well as for providing the funds to make my studies possible. I would also like to thank my second advisor Univ.-Prof. Dr. Dr. h.c. Michael Oberguggenberger as well as the thesis committee, and in particular the external reviewers.

I am also thankful for the support I received from my family, my parents Josef and Notburga, and my siblings Helga, Peter, Siegfried, and Maria, who always believed in me. Special thanks also go to the current and former members of the Distributed and Parallel Systems group, mainly the members of the Insieme team, with whom I have been working and therefore spent a lot of my time in the last few years. Among them I would like to highlight, sorted by seniority, Dr. Peter Thoman, Herbert Jordan PhD, Philipp Gschwandtner, and Ivan Grasso. They have been a big help for me when I needed advices of any kind and provided a great working environment. The Postdoctoral researchers on our institute, Dr. John Thomson, Dr. Biagio Cosenza, and Dr. Juan J. Durillo, who consecutively acted as my mentor during the various stages of my PhD studies have also been a great support for me and enriched my research and studies. I also appreciate the support I received from my colleges during my research visit at the University of Notre Dame, especially from Jarek Nabrzyski PhD and Gregory Davis PhD who advised me during that time.

I gratefully acknowledge my roommate throughout my entire PhD studies and co-author of my interdisciplinary publication, Dominik Steinhauser who played a non-negligible role in my decision to try to acquire a PhD degree. I would also like to thank the members of the IT-Boyz IBK, mostly for their support throughout my Bachelor and Master studies, but also for interesting discussions and shared information, even though it was mostly over mailing list, in the recent years. Furthermore, I would like to thank some friends with whom I spend some quality time with and who often cheered me up when I was in need, namely (in alphabetic order) Cen Wenkai, Florian Gatterer, Iwí Leskovec, Johann Voppichler, Julia Bezrukova (a.k.a. Julez Photos), Klaus Brugger, Markus Voppichler, Meinhard Oberhollenzer, Rachel Mazurek, Thomas Ennemoser, Ulrich Leiter and Zoe Yang.

Contents

Chapter 1

Introduction

Since the advent of computers, researchers have desired for more powerful machines in order to let their simulations scale to bigger sizes or make them more accurate. Over the last years, mainly due to the limits introduced by the power wall and frequency wall [88], sequential performance increases been very limited. Therefore, parallelism has evolved as the main source for performance increases. This lead to supercomputers with large numbers of connected compute nodes, each of them housing several processors, as well as to the advent of highly parallel processors, such as multi-core CPUs, massively parallel GPUs and CPU-like many core accelerators. As each of the aforementioned processing units has their own advantages and disadvantages, processors of different kind are often combined within a single compute node, forming heterogeneous systems. The list of the 500 fastest supercomputers of the world from November 2016 [6] lists 86 supercomputers that are based on heterogeneous compute nodes.

This hardware development led to several levels of parallelism in modern supercomputers as described in Section 2.1.1. Programming modern parallel computers is far from trivial, as a programmer has to take all levels of parallelism into account in order to achieve good performance. This makes programming and optimizing programs very complex and time consuming. New tools such as optimizing compilers (see Definition 1.1) and runtime systems (Definition 1.2) are needed to minimize the complexity for the programmers and maximize their productivity. The research presented in this thesis aims at improving the state of the art of such optimizing compilers and runtime systems.

There exist a number of different programming languages that target one or several parallel architectures [22, 118, 3, 123, 89]. MPI has been established as a quasi-standard for programs that are distributed over several compute nodes (see Section 2.1). It is a low level library defining a message passing interface, supporting point-to-point communications.

OpenMP (see Section 3.2) has been established as a quasi-standard for shared memory parallel programming on CPUs (defined in Section 2.1). Hence, part of this thesis is dedicated to the optimization of OpenMP programs. It uses pragma directives to control the parallel execution of shared memory programs, which makes it easy to use.

OpenCL (see Section 3.1) is probably the most versatile of the aforementioned languages. It supports a large number of architectures and offers a high level of control over the hardware. Therefore, this thesis will target mostly OpenCL programs. The primary use case of OpenCL is programming GPUs (as described in Section 2.1), as well as heterogeneous systems consisting of CPUs and GPUs.

Definition 1.1 (Compiler)**.** The term *compiler* describes a computer program that is designed to read and process a program p and generate a semantically equivalent program p'. The program p can be transformed from one language to another language (in most cases from a programming language to a machine language), p can be restructured to improve its non-functional behavior, or a combination of both.

Definition 1.2 (Runtime System)**.** A *runtime system* is a computer program that implements control over the order in which a program p is executed to optimize the non-functional behavior of p. In some cases, the runtime system is also responsible for an efficient distribution of the data. A runtime system dynamically collects information during the execution of p and uses this information to optimize its non-functional behavior.

Writing high performance OpenCL programs can be especially challenging as the language offers a very detailed level of control to the user. This level of control is needed to leverage the full potential of the various supported processing units which feature very different performance characteristics. However, this high level of control results in a rather low level language that makes writing programs which fully exploit the performance of a single, or even more complex, several target architectures very complex. The goal of this thesis is to simplify parallel programming by adding a source-to-source compiler and a sophisticated runtime system to the programming framework. The source-to-source compiler in combination with the runtime system should be responsible for low level optimizations and applying sophisticated scheduling techniques in order to achieve good performance while freeing the programmer from the labor intensive and error prone task of program optimization and thereby raising the programmers productivity.

Among the most important decisions in order to reach high performance on modern, heterogeneous architectures are data placement and work distribution over multiple processing units, the degree of parallelism, as well as assigning the workload to the most suitable processing unit. This thesis introduces automatic techniques to provide solutions for these problems for both OpenMP and OpenCL applications.

1.1 Open Problems

The aforementioned challenges result in several problems, many of them still unsolved. To provide a solution to these problems, we need advancements in current compiler and runtime systems. Parallel constructs should be made first class citizens of compilers in order to enable sophisticated analysis and transformations that unleash the full potential of modern, parallel processing units. As many decisions cannot be made at compile time, current runtime systems need to be adapted in order to optimize the parallel execution of programs. The following paragraphs describe in more detail the problems that we are trying to solve as part of this thesis.

Characterization of Heterogeneous Processing Units One problem arises from the differences between the parallel processing units. Each of them has different characteristics and shows best performance with different code optimizations. It is hard for the programmer to determine what code is most suitable for a certain processor. A set of tools (e.g. a processing unit characterizing benchmark suite), that provides information about the performance characteristics of a given processing unit is desirable. This problem will be addressed in Chapter 4.

Heterogeneous Task Partitioning Another problem in heterogeneous parallel processing is the distribution of the workload among the available computational resources. The traditional approach is to manually select one processing unit for each task during the program design phase. This approach does not only increase the complexity for the programmer, it also ignores the influence of changing machines and varying input sizes. Furthermore, it is very complex for a programmer to distribute the workload among several heterogeneous processing units concurrently, mostly because a uniform workload distribution among all available processing units often leads to sub-optimal performance in heterogeneous systems. There is a need for a framework that can automatically distribute the workload among several heterogeneous processing units, taking into account also the impact of the problem size. In Chapter 5 we introduce a system that can automatically distribute the workload of a program over all the available computational resources.

Data Layout Optimization An important decision when designing a program regards the *data layout* (described in Chapter 6) of the data to be processed. The ideal data layout may vary with the used processing unit as well as with the program. To aid the programmer in finding a suitable data layout for a given program/processing unit combination Chapter 6 introduces an automated system to deduce such data layout.

Multi-Objective Optimization of Parallel Programs Due to recent interest in objectives other than execution time, like for example energy consumption of a program, modern program optimization should be able to deal with multiple objectives. Since these objectives may be conflicting, it is often impossible to find a single program configuration that maximizes the performance in every objective. To solve this dilemma, a set of program configurations which represent the best trade-offs between the objectives must be found. This complex and time intensive task should be automatized by a system that can find the best trade-offs between the different objectives and provide them to the user to select the most suitable configuration. An approach to solve this problem using auto-tuning is presented in Chapter 7.

1.2 Organization

This thesis is structured into nine major chapters as follows: The next chapter formally defines the hardware and software model used throughout this thesis. Chapter 3 gives an overview of the programming languages and frameworks used for the research presented in this thesis. In Chapter 4 we introduce a set of microbenchmarks which are designed to identify the most performance relevant features of various processing units. A system to automatically distribute OpenCL kernel functions over a set of devices is presented in Chapter 5, while Chapter 6 shows how a well performing data layout can be determined automatically for a given pair of OpenCL program and device. Chapter 7 presents a multi-objective auto-tuning system for OpenMP programs based on iterative compilation. Finally, Chapter 8 shows how two domain applications were accelerated using OpenCL and GPUs. Chapter 9 concludes the thesis and presents future work.

Chapter 2

Model

This chapter describes the hardware and software model as well as the associated technical terms used throughout this thesis. Section 2.1 presents about the architectures which have been used to run the experiments for this thesis. Section 2.2 outlines the programs targeted by this work.

2.1 Hardware Model

The target architecture for the work in this thesis are all systems whose devices can execute OpenCL and/or OpenMP code. This includes mobile systems on a chip with their embedded CPUs and GPUs, modern laptops, desktops and workstations as well as supercomputers with their multi-socket CPUs and the connected GPUs and other accelerators. The different levels of parallelism of the target hardware are described in Section 2.1.1 while the basic hardware components are detailed in the following paragraphs.

Processing Unit (PU) In this work, a *Processing Unit* designates a set of homogeneous *cores* that have access to the same *DRAM*. The *caches* and *scratchpad memory* associated with those cores are also part of the PU. Heterogeneous PUs may share the same DRAM. In OpenCL, a PU is called a *device*. Examples for a processing unit are Intel Xeon X5650, AMD FirePro S9000 and NVIDIA Tesla k20.

Chip A chip designates an integrated circuit, i.e. a physical microchip [154]. Some PUs are composed of multiple chips and other chips house more than one PU. Examples for a PU with multiple chips are Intel Xeon X5650 and AMD Opteron 2435 whereas AMD FirePro S9000 and NVIDIA Tesla k20 are PUs that consist of a single chip. The case of the IBM PowerXCell 8i, a single chip contains two PUs, the PPE and the SPEs.

CPU The *central processing unit* (CPU) is the PU which executes the operating system [157] and can also be used to perform calculations. CPUs have a sophisticated, multi-level cache hierarchy (three levels in most cases) with relatively large caches (several MB in total). CPUs do not have a scratchpad memory. Examples for a CPU are Intel Xeon E5-2690 v2, AMD Opteron 2435 and Intel Xeon X5650.

GPU A *graphical processing unit* (GPU) is a PU designed for accelerated graphics processing and display output. However also general purpose computing can be done on the GPU (*GPGPU*). In this work, GPUs are used to execute kernels which are offloaded to them by a CPU using the OpenCL programming language (as described in Section 3.1). Some GPUs have no caches, others have a relatively small cache hierarchy (below 1 MB in size and a maximum of two levels). GPUs always have a scratchpad memory of several KB per core. Examples for GPUs are AMD FirePro S9000, NVIDIA GeForce GTX 480 and Tesla k20.

Accelerator Accelerators are PUs designed for special tasks (e.g. highly parallel computational kernels). Typically they deliver a much higher performance than CPUs in the task they are designed for. Similarly to GPUs, work is offloaded to them by the CPU. The cache hierarchy is not as complex as the one of CPUs and has a maximum of two levels. Scratchpad memory is optional. An Example for an accelerator is the SPEs in the IBM PowerXCell 8i.

Core A core is one of the building blocks of a PU. Most modern PUs consist of multiple cores. Each core houses functional units like arithmetic-logic units (ALUs), vector units, and one or more program counters (PC) [156]. A core can execute one or more threads independently. In OpenCL terminology, a core is called a *compute unit*.

Memory Controller The memory controller is a PU's connection to its DRAM. All data that is processed (both application data and the program itself) has to be loaded/written from/to the DRAM via the memory controller. The data loaded by the memory controller is usually automatically buffered in caches for faster data reuse.

Arithmetic-Logic Unit (ALU) An ALU is a basic building block of a core that can perform arithmetic and bitwise logical operations [155].

Vector Unit A vector unit is a special form of an ALU that can execute one instruction on several data elements concurrently either using multiple slots in vector registers (SIMD) or executing the same instructions on different data using multiple threads (SIMT).

Cache The cache is a small, low latency, high bandwidth memory which is used to buffer data from the DRAM. Caches are managed automatically by the hardware. An application cannot directly influence which part of its data is buffered in the cache. However, it can arrange the memory accesses in such a way that the caches can be used more effectively. Often several caches are arranged in multiple levels, usually numbered in ascending order. Lower level caches are smaller and faster and offer higher bandwidth and lower latency. The latency and size of a cache increases with its level, while the bandwidth decreases. Each core always has an exclusive first level cache, while higher level caches may be shared among multiple cores.

Scratchpad Memory In contrast to caches, *scratchpad memory* does not provide cache coherence and must be managed by the application, which makes their hardware implementation much easier. Furthermore, their ability to be fully controlled by the programmer can lead to better performance compared to automatically managed caches in certain situations. In terms of size, latency, and bandwidth scratchpad memories are similar to caches.

DRAM DRAM is a comparably large off-chip memory with significantly higher latency and lower bandwidth than caches and scratchpad memory. It is used to temporarily store programs and data that is too big to fit into the caches. The DRAM of CPUs is usually referred to as *main memory*.

Definition 2.1 (Compute Node)**.** A compute node is an entity that runs exactly one instance of an operating system [157]. It comprises one or more processing units as well as one or more main memories.

Figure 2.1 depicts a generic compute note and its components. It consists of an arbitrary number of PUs which can be homogeneous or heterogeneous. Each PU has one or more cores. Every core has one or more program counters (PCs) as well as several ALUs. In addition to them, many cores also feature vector units for high-throughput computations. In some cases, each of those cores has access to a dedicated scratchpad memory. Typically there are one or more cache levels which are shared among some or all cores of the PU (cache level N+1 to M in Figure 2.1), while there also exist some (lover level) caches with exclusive access of a single core (cache level 1 to N in Figure 2.1). Some PUs share the DRAM (PU 0 to N in Figure 2.1) while others have a dedicated memory (PU N+1 to M in Figure 2.1). Figure 2.2 exemplary shows three PUs that are used for experiments in this thesis. The Intel Xeon X5650 in Figure 2.2a is a CPU while Figure 2.2c and Figure 2.2b show an AMD FirePro S9000 and NVIDIA Tesla k20 GPU, respectively.

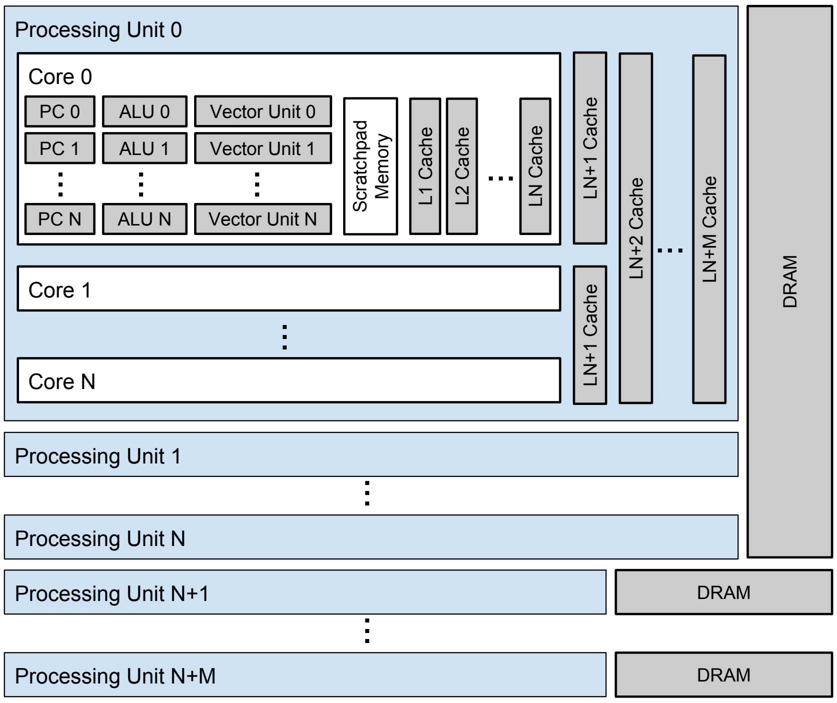
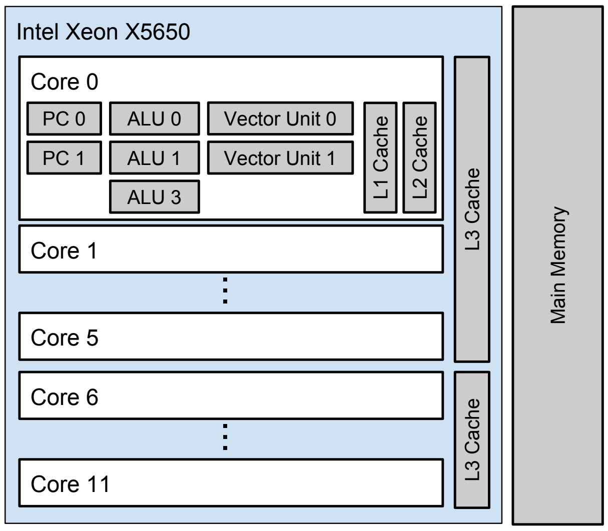
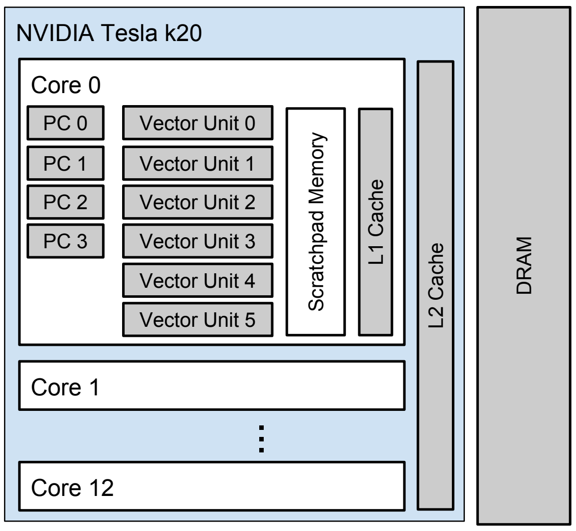


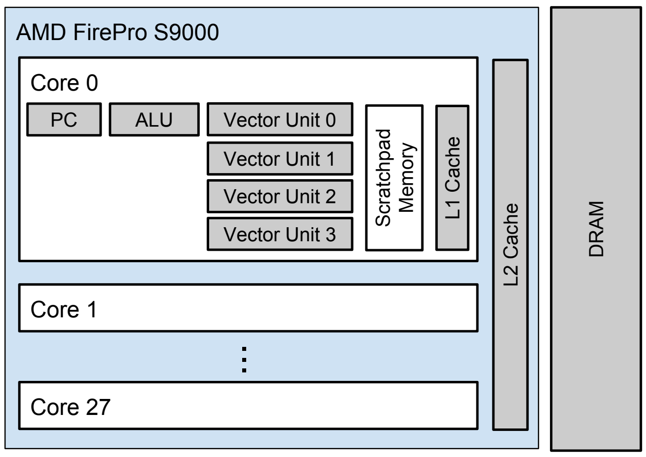
Figure 2.1: Generic compute node



(a) Intel Xeon X5650



(b) NVIDIA Tesla k20



(c) AMD FirePro S9000

Figure 2.2: Examples for PUs used in this thesis

2.1.1 Levels of parallelism

As mentioned earlier, modern computers have several levels of parallelism. Some of those levels involve multiple processing units while others can be found within a single processing unit. The following enumeration lists the most important levels of parallelism in modern computers:

- Multi-PU parallelism
  - Inter-compute node parallelism
  - Intra-compute node parallelism

- Single-PU parallelism
  - Multi-core parallelism
  - Vector parallelism

Multi-PU parallelism involves multiple PUs. Those PUs can either be housed in the same compute node (in case of *intra-compute node parallelism*) or connected via a network (in case of *inter-compute node parallelism*). In both cases, the PUs can be either homogeneous or heterogeneous. As neither OpenCL nor OpenMP are able to address inter-compute node parallelism it will not be discussed in this thesis. In case of intra-compute node parallelism over multiple PUs, OpenCL addresses each PU as a separate device. The task of distributing work among them is solely left to the programmer. Therefore part of this thesis is dedicated to develop an automated system to find a high performing work distribution over several devices in a heterogeneous system (see Chapter 5).

Single-PU parallelism exploits the parallelism inside a single PU. Most modern PU consist of several cores, which can operate independently and only share the DRAM and some parts of the cache hierarchy. In most architectures each core has one or more dedicated caches, where some higher level caches are shared among several or all cores. The DRAM is always shared among all cores of a PU (and in some cases even among several heterogeneous PUs). The decision which part of the DRAM is buffered in the various caches is taken automatically by the hardware. The programmer-managed *scratchpad memory* is always exclusive for each core.

In OpenCL, cores are called *compute units*. The individual cores often feature some form of vector unit which performs the same instruction multiple times, either on different slots of vector registers (SIMD) or in different threats (SIMT), operating on different data elements. How these hardware units are addressed by an OpenCL program is not specified by the standard and varies between different hardware and sometimes even between different compiler/runtime versions for the same hardware. The two most common ways are:

- Bundling a set of *work items* (described in Section 3.1) together and map their instructions to the vector unit. Only work items within the same *work group* (see Section 3.1) can be run in parallel on one vector unit. Divergent branches within one work group may further reduce the vectorization potential. Therefore, this technique requires sufficient number of non divergent work items in each work group.

- Mapping OpenCL operations based on *vector types* [89] to the vector units. In this case, the programmer is required to use vector types in order to take advantage of the vector units.

OpenMP can only address the CPUs in a system. The default behavior is to assign equally sized chunks of work to each CPU core. However, the number of those chunks doesn't have to match the number of cores and using less chunks, and therefore, utilizing only a subset of the available cores, can be advantageous in terms of energy consumption and resource usage, as it will be discussed in Chapter 7.

2.2 Software Model

This thesis presents methods and techniques to transform parallel programs in order to minimize their execution time. To achieve this, we use the Insieme source-to-source compiler/runtime system as described in Section 3.3. The Insieme compiler takes a parallel program as input and outputs an optimized version of it. The resulting program is then mapped to the hardware by the Insieme Runtime System. The software model comprises the program and data model as described in the following sections.

2.2.1 Program Model

A program p can be modeled as a graph $(S,\ E)$ where S is a set of statements (e.g. arithmetic operations, I/O operations, branches, function calls, etc.) and E consists of directed edges between those statements that model the control flow. The number of outgoing edges $\#\{(s_0, s_1) \in E\}$ is called the *out-degree* of the statement s_0, the number of incoming edges $\#\{(s_1, s_0) \in E\}$ is called *in-degree* of the statement s_0. A program has exactly one statement with an in-degree equal to 0, the so called *entry point.* This is where the program execution starts. A program has one or more statements with an out-degree of 0, the so called exit points. When the program flow reaches such a statement, the program terminates.

Definition 2.2 (Program)**.**

$$S = \{s_0, s_1, ..., s_n\} \tag{2.1}$$

$$E \subseteq S^2 \tag{2.2}$$

$$p = (S, E) \tag{2.3}$$

S is a set of statements, E are directed edges between statements and p a program that is represented by a graph formed by the statements (i.e. vertices) S and the edges E. P is the set of all programs p.

Parallel Program Model

The set of parallel programs P_p is a subset of all programs P where the set S also contains the statements *fork* and *merge.* A fork statement f starts a parallel execution, as all its direct successors $\{s \in \mathrm{S} | (f, s) \in \mathrm{E}\}$ are executed concurrently. This parallel execution continues along the graph until a merge statement is reached. A merge statement m waits until all its (parallel) predecessors $\{s_i \in \mathrm{S} | (s_i, m) \in \mathrm{E}\}$ complete their executions. After the merge statement, the program continues to run sequentially. The sub-graph between a fork statement and the corresponding merge statement is called a *parallel region.* Figure 2.3 shows an example for a parallel program. The three paths between the fork and the merge node can be executed in parallel. This thesis focuses on the optimization of parallel programs. For those programs, the parallel regions are defined either using OpenMP (described in Section 3.2) or OpenCL (see Section 3.1).

In the OpenMP programs which are covered in this thesis, parallel regions are defined using the pragma `#pragma omp parallel`. OpenCL programs consist of a host and a device part as described in Section 3.1. The device part is inherently parallel and always starts with two nested fork instructions to represent its implicit, two level parallelism as described in Section 3.3.2.

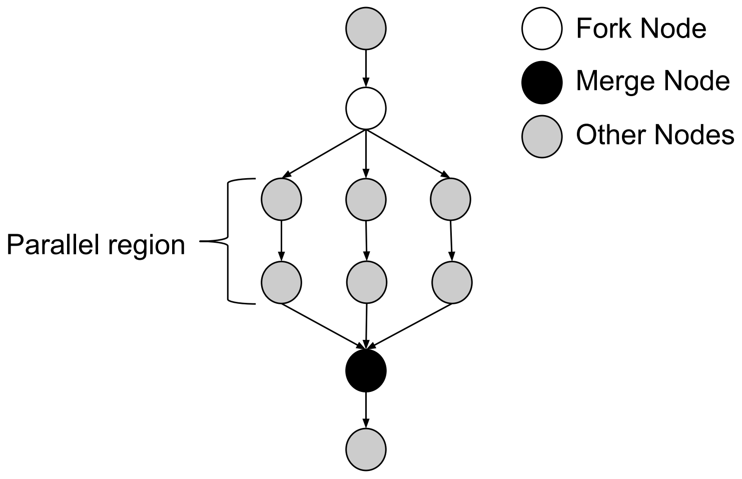


Figure 2.3: A parallel program represented as a graph.

2.2.2 Data Model

The basics of the type system in C, OpenCL and also INSPIRE (as described in Section 3.3.1) are *scalar types* such as integer and floating point numbers. In addition to scalar types we define *array*, *struct* and *tuple* types. Array types are formed by a tuple of a data type and a natural number specifying the array's dimensionality. An array contains an unspecified number of elements of the given data type, arranged in the given number of dimensions. The elements in the array can be accessed via an index vector that specifies the element's position in each dimension. A special case are one dimensional arrays as their elements can be accessed using a scalar as index. Struct types consist of an arbitrary number of tuples, each of them containing a name and a data type. Tuple types are similar to struct types, however their elements do not have a name and are differentiated solely by their position. Array, struct, and tuple types can be nested in an arbitrary way.

Definition 2.3 (Types)**.**

$$T ::= T_s | Array | Struct \tag{2.4}$$

$$T_s ::= \text{scalar}(\text{signed integer}|\text{unsigned integer}|\text{real}, l \in \{1,2,4,8\}) \tag{2.5}$$

$$Array ::= \text{array}(t \in T, d \in \mathbb{N}^+) \tag{2.6}$$

$$Struct ::= \text{struct}\{(t \in T, name)^*\} \tag{2.7}$$

$$Tuple ::= \text{tuple}\{(t \in T)^*\} \tag{2.8}$$

T_s is the set of all scalar types which can be signed integer, unsigned integer or real numbers and have a size of $l \in \{1,2,4,8\}$ bytes. *Array* is a collection of an arbitrary number of elements of type $t \in T$ arranged in d dimensions with $d \in \mathbb{N}^+$. *Struct* designates a collection of tuples consisting of a name and a data field of type $t \in T$. *Tuple* defines a collection of data fields with varying types $t \in T$ with T denoting the set of all types.

Figure 2.4 shows examples for scalar types, an array, a struct as well as nested type called an *array of structs* (AoS). Especially when data types with nested arrays and structs are used, the data layout has a significant impact on the performance as it will be discussed in Chapter 6.

```
int := scalar(signed integer, 4)
double := scalar(real, 8)
oneDimensionalArray := array(int, 1)
twoElementStruct := struct{(int, member0), (double, member1)}
AoS := array(twoElementStruct, 1)
```

Figure 2.4: Examples for scalar types, an array, a struct and an array of structs (AoS).

Chapter 3

Background

This chapter introduces the parallel programming languages and frameworks used for the experiments in this thesis. These tools comprise the programming languages used for the applications that we are optimizing as well as the Insieme compiler and runtime system that is used to perform code analysis and transformations. While OpenCL and OpenMP, described in the following sections, are industry standards for parallel programing, the Insieme framework is a research compiler and runtime system designed and implemented by the Distributed and Parallel Systems group at the University of Innsbruck.

3.1 OpenCL

OpenCL (Open Compute Language) is a parallel programming language, introduced by the Kronos Group [158] as an open standard for cross-platform parallel computing. Several hardware vendors such as AMD, ARM, IBM, Intel, and NVIDIA provide an OpenCL capable compiler/runtime infrastructure, thus OpenCL supports a wide range of hardware. This has the advantage that OpenCL code can be executed on almost any existing hardware. However, it also comes with the disadvantage, that OpenCL doesn't support any low level access routines to the hardware, as the supported hardware varies a lot.

The OpenCL programming model splits the programs in two parts: a host part and a device part [89]. The host part is a "normal" C-program that uses a specific runtime API to access the devices. A *device* can be any processing unit that supports OpenCL. The host part is executed by the operating system and therefore it runs on the CPU, sometimes referred to as *host* in this context. It is responsible for setting up the devices, loading the program code to be executed on the device, distributing the data to the devices and starting the device part of the program. It is also responsible for collecting the results after the devices finish their computations.

The device part of the program is represented by so called *kernel functions*. For those kernel functions, OpenCL defines a C99-based language with OpenCL specific keywords. These functions are executed on the devices in parallel and should therefore contain all computational expensive parts of the program. To be executed in parallel, the kernel functions are mapped to a three dimensional grid of independent threads, called the *NDRange*. Every element of the NDRange, called a *work item*, executes one instance of the kernel function. Devices are not capable of dynamically changing the

NDRange, allocate and free memory, or transfer data from/to the host or other devices. These tasks have to be performed by the host.

A kernel function can be executed by any device available, whether it is a CPU, a GPU or an accelerator. The characteristic that any kernel function can be executed by devices of any kind is called *code portability.* To enable this characteristic, the OpenCL kernel code has to be compiled by a compiler provided by the device's manufacturer. This compiler is different from the one used to compile the host code (which can be compiled with any C-compiler). To maintain maximum flexibility, the kernel code is usually compiled at run-time. This means, that the host and device code is compiled by different compilers at different points in time. Obviously, this makes optimizations that incorporate changes in both, the host and the device part of a program, very challenging. Furthermore, OpenCL is not ideal for *performance portability*, as different devices often require very different parallelization strategies and memory access patterns in order to achieve their maximal performance. This will be discussed in detail in Chapter 4.

In order to maximize the throughput for a given task it can be beneficial to distribute it over all available devices. OpenCL does not provide any built-in functionality to distribute work over multiple devices, hence this task is left to the programmer. Modern compute nodes with accelerators typically offer a heterogeneous set of devices. This makes an ideal work distribution over the given devices challenging. In Chapter 5 we present a system that not only simplify the work distribution over multiple devices, but also automatically finds good work distribution for a given kernel/devices combination.

Another important factor for the performance of OpenCL programs, especially when executed on GPUs, is the data layout. However, OpenCL does not provide any utilities that facilitate the selection of the best suited data layout. Chapter 6 presents a system that helps the programmer to select a well suited data layout for a given kernel/device combination.

3.1.1 OpenCL Hardware View

Due to the large variety of the supported hardware, the OpenCL hardware view is rather abstract. This abstract view allows the use of the same hardware model for all supported devices. As mentioned in Section 3.1, OpenCL distinguishes between the host and the device. The host runs a standard C-program and is not intended to do any computational expensive tasks. Therefore, from an OpenCL point of view, the host is sequential (although the host part may be parallelized by some programming models other than OpenCL) and has access to the host memory. The host also can access all the devices attached to the system and provide them with data or computational tasks. A single host can control an arbitrary number of devices. Each device has its own memory, which is subdivided in three address spaces as described in Section 3.1.3. Each Device consists of several *compute units*, which can execute different instructions independently. Depending on the actual device, they may be able to execute different kernel functions at the same time. Each compute unit has a *local memory* and consists of several *processing elements*, which can execute a single work item at a time and have access to a small amount of fast private memory. Figure 3.1 illustrates the hardware view of OpenCL.

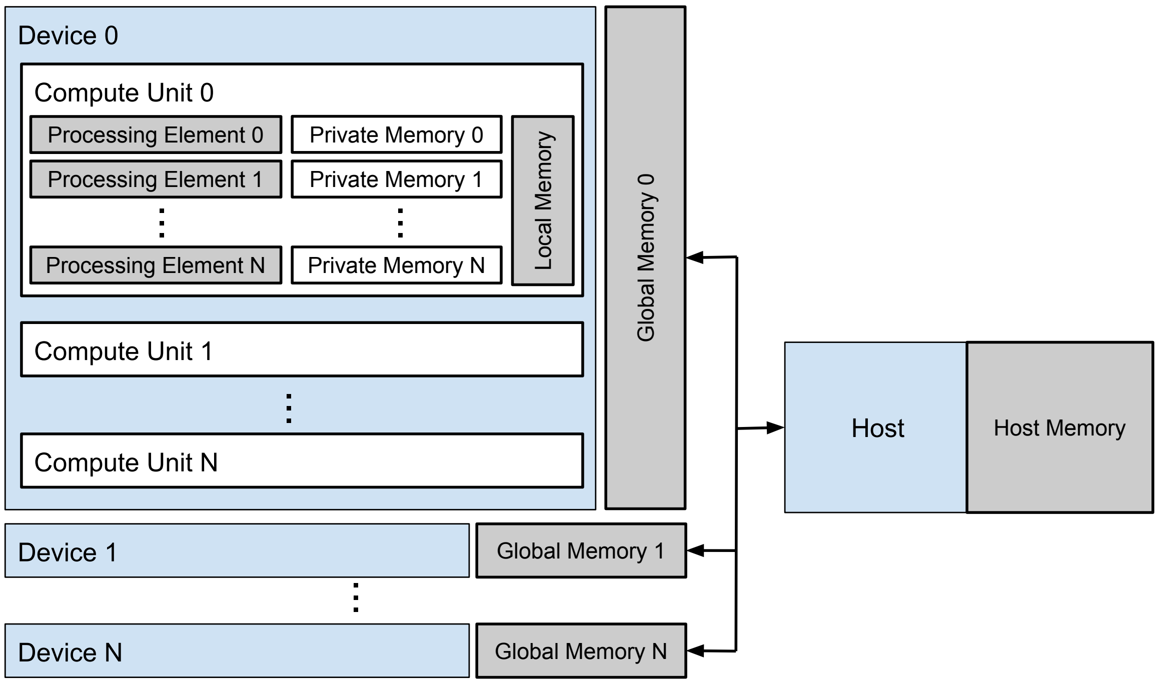


Figure 3.1: OpenCL view of the underlying hardware

3.1.2 The OpenCL NDRange

As mentioned above, OpenCL kernels are mapped to a three dimensional grid of work items, the NDRange. The NDRange consists of several *work groups*. These work groups are further subdivided into a predefined number of work items. These two levels of parallelism influence how the individual work items are mapped to the hardware. One important property of OpenCL is, that synchronizations in the kernel using the `barrier` function only synchronize the work items within the same work group, not the entire NDRange. The entire NDRange is only synchronized when the kernel function terminates.

All work items of the same work group are always executed on the same compute unit. The mapping of work groups and work items to the hardware depends on the actual hardware and the used OpenCL runtime system. The number of work groups and work items in the NDRange is determined by the parameters of the OpenCL library function `clEnqueueNDRangeKernel`, which generates a new NDRange and executes a kernel function within it. The parameter `global_work_size` determines the total number of work items, while the parameter `local_work_size` prescribes how many of them form one work group. Figure 3.2 shows an example for an NDRange. In this case the `global_work_size` is 4,8,8 and the `local_work_size` is 2,4,4. The work groups are labeled with the group id the dimensions 0, 1, and 2 while the labels on the two highlighted work items are the local ids in the various dimensions. The global id of the highlighted work item in work group 0,1,1 is 0,7,7; for the work item in work group 1,1,1 it is 2,6,5.

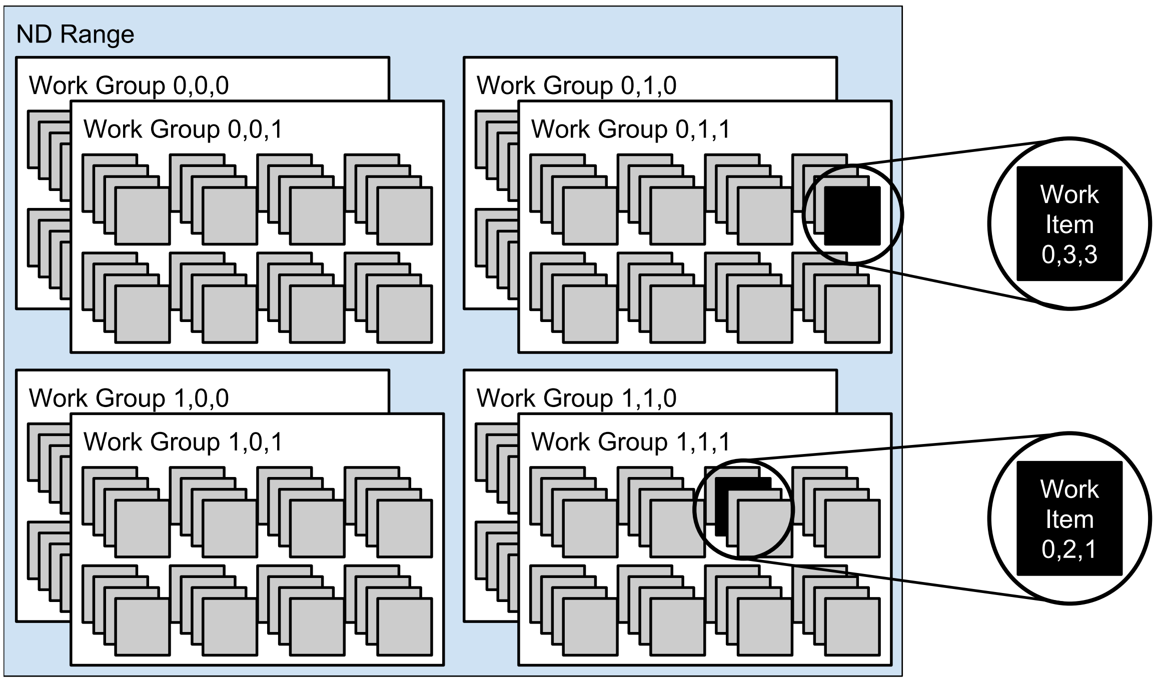


Figure 3.2: Example for an OpenCL NDRange with 256 work items, arranged in $2 \times 2 \times 2$ work groups with $2 \times 4 \times 4$ work items each.

3.1.3 OpenCL Address Spaces

OpenCL distinguishes three address spaces in the device code: *private*, *local*, and *global*. The address space of a variable is selected by using the corresponding qualifier at variable declaration. If no qualifier is used, the variable resides in the private memory space. These three address spaces are accessible from the device code. In addition to that, the *main memory* (see Section 2.1) of the CPU acting as the host is a fourth address space, often called the *host memory*. Out of the three device address spaces, the host can read/write data only from/to the global memory using the corresponding OpenCL library functions. The relationship between the four address spaces is illustrated in Figure 3.3.

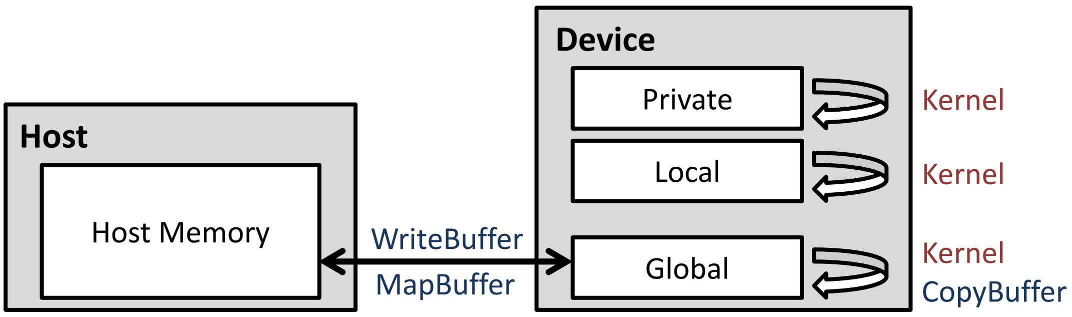


Figure 3.3: OpenCL memory architecture, taken from [161]

How the three device address spaces are mapped to the various available memories of a device depends on the used hardware and OpenCL implementation. However, the accessibility of the address spaces is defined by the OpenCL standard:

Private memory is exclusive to each work item. Arrays (see Definition 3.1) in private memory can only be allocated statically. It is usually mapped to registers or a fast *cache* (described in Section 2.1).

Local memory is shared among the work items of one work group. Arrays have to be allocated statically when they are declared inside the kernel function or dynamically when passed as an argument to the kernel function. If the device features a *scratchpad memory* (described in Section 2.1), the scratchpad memory is used for local memory, otherwise the local memory is mapped to the cache.

Global memory is shared among all work items of a kernel. All global memory variables are pointers (see Definition 3.2) . Global memory cannot be allocated in the device code, it has to be allocated in the host code using the corresponding OpenCL library call and then passed to the kernel function as an argument. All data that is exchanged between host and device has to be stored in global memory. Global memory uses the *DRAM* (described in Section 2.1) of the device.

Definition 3.1 (Array)**.** An *array* describes a regular arrangement of storage locations for a fixed number of values. All these values are of the same type and can be accessed via an integral index.

Definition 3.2 (Pointer)**.** The term *pointer* defines a variable whose value is a reference to a memory location. Pointers are often used to indicate the starting point of a series of values of the same type, i.e. *arrays.*

3.2 OpenMP

OpenMP (Open Multi-Processing) describes an open interface to create parallel programs for shared memory parallel architectures [123]. It is an extension for existing programming languages, mainly being used for C, C++ and Fortran. It mostly consists of a set of directives, called `pragmas`, which are added to programs written in a supported language, in order to instruct the compiler how to parallelize certain parts of that program. Important OpenMP directives are:

omp parallel starts a parallel section of the program. The code block following an `omp parallel` is executed by all available threads concurrently.

omp for is used to mark for-loops inside an `omp parallel`. The iterations of such a loop are distributed among all threads. Each iteration is executed by exactly one thread, where the execution order of the individual iterations is not defined.

omp barrier causes all threads of a parallel region to synchronize. When a thread executing a parallel region reaches the barrier, it waits until all threads executing the parallel region have reached the barrier before progressing.

3.3 Insieme

The main goal of the Insieme project [1] is to optimize parallel programs, relying on a source-to-source compiler and a parallel runtime system which will be described in more detail in the following sections. This thesis is based on the Insieme branch *inspire_1.3* which is freely available at [2]. My main contribution to the Insieme compiler is the Insieme OpenCL frontend extension.

3.3.1 Insieme Compiler

The Insieme Compiler is implemented in C++. It accepts C and several of its derivatives (including OpenCL) as an input language.It consists of three major components:

- The frontend, translating a program from the input language to the Insieme parallel intermediate representation (INSPIRE described in Section 3.3.1).
- The core, performing analysis and transformations on the program in INSPIRE representation
- The backend, translating the program from INSPIRE to the output language.

Figure 3.4 shows how these three parts interact during a program compilation using the Insieme Compiler. As all analysis and transformations are performed on code in INSPIRE representation, it did not have to be adapted in order to process OpenCL programs. The frontend on the other hand, had to be extended in order to translate OpenCL code to INSPIRE. In a similar way, the backend had to be adapted in order to translate INSPIRE back to OpenCL code, especially in order to create OpenCL code in which the kernel can be distributed among several devices as described in Chapter 5.

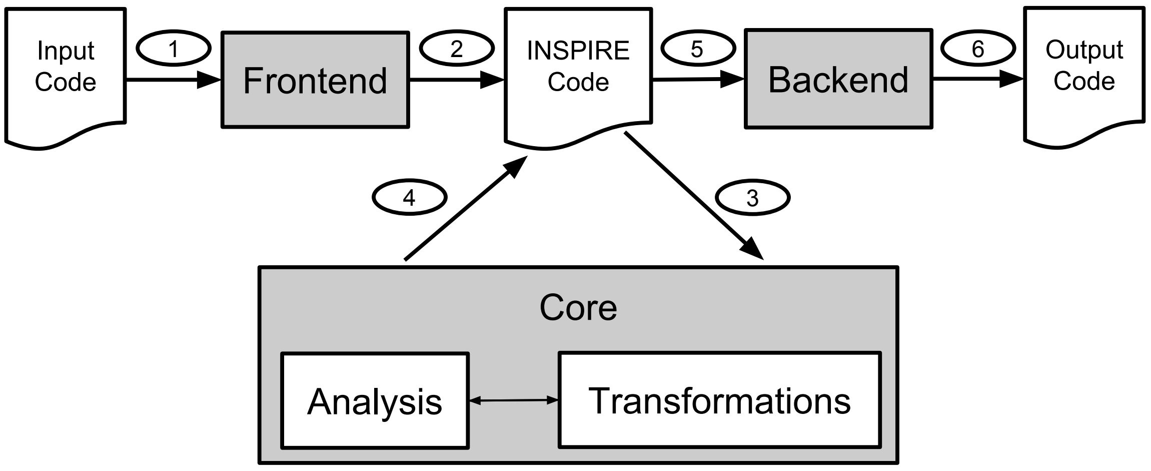


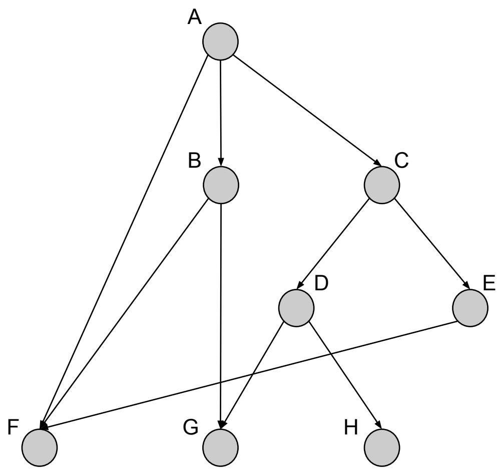
Figure 3.4: Interactions of the Insieme Compiler's main components

Insieme Parallel Intermediate Representation (INSPIRE 1.3)

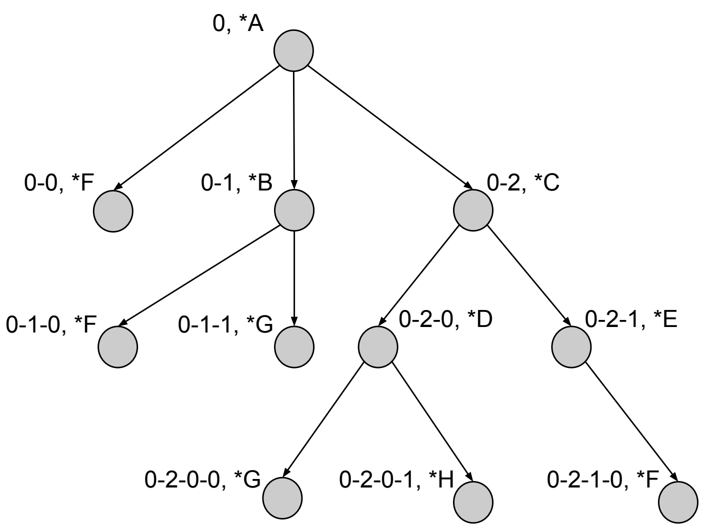
As mentioned in the previous paragraphs, the Insieme compiler works on programs in INSPIRE representation. INSPIRE is a high-level, unified, language and API independent program representation.

INSPIRE was developed to simplify analysis and code transformations within the Insieme compiler. INSPIRE represents a program as a directed acyclic graph (DAG). In order to minimize the memory footprint of the intermediate representation, nodes in this graph may be shared. For example, every type is present only once in the DAG, and every node in the entire program which refers to the corresponding type (e.g. a variable of this type) will refer to this single node. However, this representation may cause troubles when analyzing or transforming the code. For example, when replacing the type of a variable, replacing the node representing the variable's type would change also the type of all other variables, functions, etc. of the corresponding type. Therefore, the so called *Address View* of the DAG exists. The nodes in the address view refer to one node in the DAG and additionally contain an address, where shared nodes are referenced multiple times. Thereby, the Address View unfolds the DAG to an abstract syntax tree (AST). An address is always relative to its root node and is represented by the path from the root to the corresponding node (the address of the root node is always 0). Every node in the DAG can be used as root for addresses for all its child nodes.

Figure 3.5 shows an example of the DAG of a program as well as a possible AST created from it. The AST in Figure 3.5b shows a possible INSPIRE address view of the DAG in Figure 3.5a, where node **A** acts as a root address. The nodes in the AST in Figure 3.5b are formed by pairs containing the address, which is determined by the path from the root to the corresponding node, and a reference to a node in the INSPIRE DAG in Figure 3.5a. As this example shows, nodes with multiple input edges in the DAG are referenced multiple times in the address AST.



(a) INSPIRE node DAG



(b) INSPIRE address AST

Figure 3.5: Example program representation in INSPIRE

INSPIRE defines its own language and provides a large set of primitives. The following primitives are relevant for this thesis:

`parallel` The primitive `parallel` starts a new *thread group* i.e. a set of newly created threads. It initiates a parallel section.

`job` A `job` takes a function as argument. This function is then executed by each thread of the current thread group. It also specifies the lower and upper bound of the number of threads that execute the given function in parallel. The primitive `job` is often used in conjunction with the primitive `parallel`, to specify that a function should be executed concurrently by a newly create thread group.

`pfor` A `pfor` is a for loop within a `job`. Each iteration of it is executed once by any thread within the thread group in arbitrary order.

`redistribute` The blocking primitive `redistribute` collects information from each thread and passes this information to a user defined function, aggregated in an array. As `redistribute` is a blocking primitive, it can be used to implement barriers. In this case, the user defined function is a *no-op*.

`bind` The expression `bind` generates a closure. Its parameters are used as arguments on the enclosed function call while eventual additional arguments are captured from the surrounding context.

`CAST` To perform conversions from one datatype to another datatype, INSPIRE provides the primitive `CAST`. Casts using this primitive are restricted to scalars with the same size in bytes.

`ref_reinterpret` Arrays of a given type can only be converted into arrays of another type using the `ref_reinterpret` construct in INSPIRE. When reinterpreting an array using another type, the binary representation is unchanged, but it is interpreted as another type. Therefore, also the number of elements may changes.

Additional details about the primitives defined in INSPIRE can be found in [81]. The INSPIRE type system is analog to the one introduced in Section 2.2.2. Table 3.1 shows INSPIRE syntax for types.

Type	INSPIRE syntax
scalar(signed integer, l)	`int<l>`
scalar(unsigned integer, l)	`uint<l>`
scalar(real, l)	`real<l>`
array(t, d)	`array<t, d>`
struct{(t_0, m0), (t_1, m1)}	`struct`{$\mathtt{t}_0$ m0`,` $\mathtt{t}_1$ m1}
tuple{t_0,t_1}	`(`$\mathtt{t}_0$`,` $\mathtt{t}_1$`)`

Table 3.1: Type syntax of INSPIRE

3.3.2 Insieme Compiler Frontend

The Insieme compiler frontend is responsible for translating the input code to INSPIRE. To parse the input code, it relies on Clang [13]. Therefore, the frontend is limited to input languages which are supported by Clang, which is the case for most C-derivatives, including OpenCL. The frontend consists of a generic implementation, which is used to translate C to INSPIRE. In order to translate non-standard C (e.g. OpenCL, Cilk, etc.), an extension to the generic frontend is needed. Such an extension can either replace the default handling for specific Clang nodes, add a post-processing step for some INSPIRE nodes, or add a post-processing step for the entire program, after it has been translated by the generic frontend.

Insieme OpenCL Frontend Extension

The OpenCL part of the Insieme frontend has two major responsibilities: On the one hand, it implements the semantics of many OpenCL library calls (in the OpenCL host code), OpenCL built-in functions, and implicit semantics (in the device code) to enable detailed program analysis. On the other hand, it connects the host and device code of an OpenCL program to form one single instance. By doing so, several analysis and transformations can be done, which would not be possible when analyzing at each part individually. To further ease analysis, the OpenCL frontend makes the semantics of the OpenCL library and built-in functions explicit, so that an OpenCL program could be translated to a C program that does not depend on the OpenCL library.

The OpenCL frontend performs all transformations on the INSPIRE representation. This means, that the generic C frontend generates an INSPIRE DAG from the source code on which the OpenCL frontend performs some transformations, as shown in Figure 3.6. The result of the OpenCL Frontend is a program in INSPIRE that contains the host and device part of an OpenCL program and where the implicit OpenCL functionalities (e.g. parallelism, data transfer, etc.) are made explicit. OpenCL specific qualifiers (e.g. `__kernel`, `__local`, `__global`, etc.) which are not part of standard C are defined by an automatically injected header file and eliminated (if irrelevant for the INSPIRE representation) or translated to annotations which are attached to the corresponding INSPIRE node. The generic frontend is also responsible for translating OpenCL vector types to INSPIRE vectors with the appropriate type as well as translating vector accesses to subscript operations and operations on vectors (e.g. addition) to calls to functions generated using `vector.pointwise`, which performs the corresponding operation on each vector element in a loop. This is done directly when translating the Clang AST to the INSPIRE DAG.

Since the OpenCL host code is standard C code which includes some specific headers, whereas the device or kernel code uses an extension of a subset of C, the OpenCL frontend extensions for both parts are implemented in two independent compiler components which will be described in the following paragraphs.

OpenCL Device Frontend

To compile an OpenCL kernel function with Insieme, the OpenCL Device Frontend automatically adds the header file *ocl_device.h*. This header file declares (but does not implement) most of the OpenCL built-in functions (in both scalar and vector version), so that Clang will be able to parse the kernel file without errors and a valid INSPIRE program can be generated from it.

Since OpenCL kernels do not have a `main` function, the compiler can't find the entry points automatically. Therefore, the programmer must mark all kernel functions that should be compiled with Insieme using `#pragma insieme mark`. The generic frontend will then generate a program with a separate entry point for each kernel function and add an annotation to each of them, so they can be identified as kernel functions by the OpenCL Device Frontend.

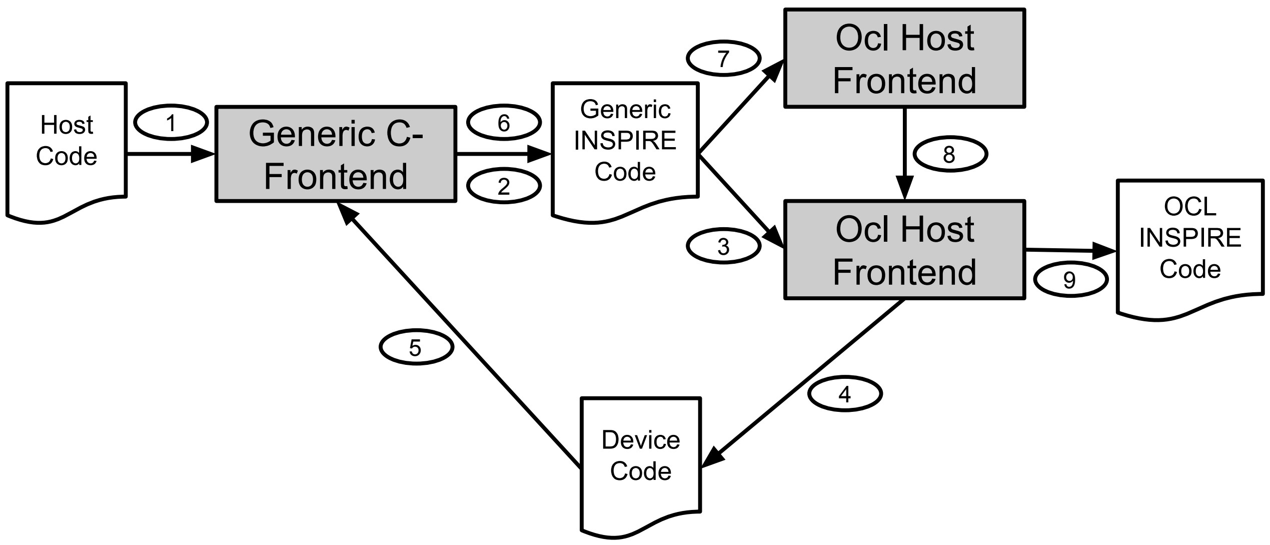


Figure 3.6: Schematic view of the individual stages of the Insieme frontend when compiling an OpenCL program.

When an INSPIRE node with the aforementioned annotation is found, the OpenCL Device Frontend transforms the attached function and all its sub-functions as follows: It makes the implicit parallelism of OpenCL explicit. When calling an OpenCL kernel function, several instances of it will be started in parallel in a two level hierarchy of threads. The instances of the first level are called *work groups*. Work groups consist of several *work items* (which are similar to threads), forming the second level. Both levels can have up to three dimensions. For a detailed description of the two level hierarchy see Section 3.1. In order to capture this semantics in INSPIRE, two nested `parallel/job` constructs [81] are wrapped around the kernel function's body, one covering the parallel work groups, the other one representing the parallel work items as it can be seen in Figure 3.8 which shows the INSPIRE code generated from the OpenCL kernel function performing a vector addition in Figure 3.7. The `job` constructs have a fixed number of threads which is determined by the arguments `global_work_size` and `local_work_size` of the function `clEnqueueNDRangeKernel` in the host code. In OpenCL, these values are passed to the kernel function implicitly, therefore in INSPIRE two additional arguments are added to the kernel, passing them explicitly. The number of threads of the outer `job` construct is equal to `global_work_size` divided by `local_work_size`. Therefore, at the beginning of the kernel function (at line 36 in Figure 3.8), a statement performing exactly this calculation is added and the result is used as argument to the outer `job` construct. The number of threads for the inner `job` construct is equal to `local_work_size`.

The `parallel/job` constructs in INSPIRE are only one dimensional. Therefore the three dimensional NDRange of OpenCL is flattened to generate two one dimensional spaces. In order to maintain the semantics of the input program, the three dimensional group, local, or global id has to be restored at any place where an instance of `get_[group|local|global]_id` is called. In order to do so, these functions are replaced with new functions, implemented in INSPIRE. Like their OpenCL counterparts, those new functions take the dimension as an argument. Additionally, also the local and global

```
#pragma insieme mark
__kernel void vec_add(__global int* input1, __global int* input2, __global int*
    output, unsigned int size) {
  int gid = get_global_id(0);
  if (gid >= size) return;
  output[gid] = input1[gid] + input2[gid];
}
```

Figure 3.7: OpenCL kernel performing a vector addition.

size (passed as arguments to the kernel function as described in the previous paragraph) and/or the number of groups (calculated at the first line of the kernel, as described in the previous paragraph) is added to the argument list, depending on the actual function. From these parameters the actual, three dimensional index is calculated using division and modulo calculations as the example in Figure 3.9 shows. Similarly, calls to `get_[local|global]_size` and `get_num_groups` are replaced with functions implemented in INSPIRE to yield the same return value as the original functions.

When a function is called inside a kernel, the interface of this function may be changed. As mentioned before, in the INSPIRE representation some functions need the local size, global size or number of groups as argument which is not contained in the OpenCL input code. Therefore, if anything inside a sub-function needs one of those variables, they will be added to the interface and to all its call sites.

Arguments of the kernel function which use the `__local` qualifier are declared between the two `parallel`/`job` constructs, so they are shared among the threads of the inner parallel, but each thread of the outer parallel has its own instance of those arguments, as it is defined in [89]. In case of kernel arguments with the `__local` qualifier, their size is not known at compile time. Therefore, they are replaced with arguments of type `uint<8>`, which are used to pass their size from the host code to the device code, where they are declared between the two `parallel`/`job` constructs. Variables with a `__global` qualifier don't have to be re-declared inside of the `parallel`/`job` constructs, since they are always pointers. The pointer itself is private to each thread, only the data it is pointing to is shared among all threads. Arguments using the `__private` qualifier are passed by value so that every thread has a private copy of them.

In OpenCL, all math functions (e.g. sin, cos etc.) can be marked as `native_` by the programmer. This keyword means that, if available, a faster and less accurate version of the function (usually implemented in hardware) shall be used. To cover this semantic, the OpenCL host frontend extension embeds marked function calls in a call to the function `accuracy_fast`, to keep the information that accuracy should be traded for speed if possible. This is also done for math functions marked with `half_`, which work with half floating point precision in OpenCL. Since Insieme does not support two byte floating point numbers, it translates them to four byte floats. However, the information that this function does not need high accuracy is preserved. The built in OpenCL function `mad` is expanded into a multiplication and an addition since INSPIRE does not support *fused multiply-add.*

`mem_fence` in OpenCL is directly translated into calls to `barrier` in INSPIRE. They only synchronize the inner parallel region, as synchronizations on the outer parallel region are not supported by OpenCL. While `mem_fence` is only synchronizing accesses to either the local or global memory, depending on the argument, the INSPIRE `barrier` always synchronizes all instructions. This may

```
let vecAddWorkItems = fun(
  ref<array<int<4>,1>> input1,
  ref<array<int<4>,1>> input2,
  ref<array<int<4>,1>> oputput,
  int<4> size, vector<uint<8>,3> numGroups,
  vector<uint<8>,3> localSize
) -> unit {
    decl ref<int<4>> id = ( var(uint_to_int(getGlobalId(0u, numGroups, localSize),
        4)));
    if((( *id)>=size)) {
        return unit;
    };
    oputput&[int_to_uint(( *id), 4)] := *(input1&[int_to_uint(( *id), 4)]) + *
        input2&[int_to_uint(( *id), 4)];
};

let vecAddWorkGroups = fun(
  ref<array<int<4>,1>> input1,
  ref<array<int<4>,1>> input2,
  ref<array<int<4>,1>> oputput,
  int<4> size, vector<uint<8>,3> numGroups,
  vector<uint<8>,3> localSize
) -> unit {
    parallel(job([vector_reduction(localSize, 1, uint_mul)-vector_reduction(
        localSize, 1, uint_mul)]){
        bind(){vecAddWorkItems(input1, input2, oputput, size, numGroups, localSize)
            }
    });
    mergeAll();
};

let vecAdd = fun(
  ref<array<int<4>,1>> input1,
  ref<array<int<4>,1>> input2,
  ref<array<int<4>,1>> oputput,
  int<4> size,
  vector<uint<8>,3> globalSize,
  vector<uint<8>,3> localSize
) -> unit {
    decl vector<uint<8>,3> numGroups = vector_pointwise(uint_div)(globalSize,
        localSize);
    parallel(job([vector_reduction(numGroups, 1, uint_mul)-vector_reduction(
        numGroups, 1, uint_mul)]){
        bind(){vecAddWorkGroups(input1, input2, oputput, size, numGroups, localSize
            )}
    });
    mergeAll();
};
```

Figure 3.8: INSPIRE code performing a vector addition, created from the OpenCL kernel function in Figure 3.7. The INSPIRE representation of the function `getGlobalId` can be found in Figure 3.9.

```
let getGlobalId = fun(uint<4> dimindx, vector<uint<8>,3> globalSize, vector<uint
    <8>,3> localSize) -> uint<8> {
    decl uint<8> localId = getThreadID(0);
    decl uint<8> groupId = getThreadID(1);
    switch(dimindx) {
        case 0: {
            return (((localId/(localSize[2]))/(localSize[1]))+((localSize[0])*((
                groupId/(globalSize[2]))/(globalSize[1]))));
        }
        case 1: {
            return (((localId/(localSize[2]))%(localSize[1]))+((localSize[1])*((
                groupId/(globalSize[2]))%(globalSize[1]))));
        }
        case 2: {
            return ((localId%(localSize[2]))+((localSize[2])*((groupId/(globalSize
                [2]))%(globalSize[1]))));
        }
        default: { }
    };
    return 0;
};
```

Figure 3.9: INSPIRE code of the OpenCL built-in function `get_global_id`.

introduce an overhead due to unnecessary synchronizations, but it never affects the program's correctness.

In OpenCL, it is legal to assign a scalar pointer (e.g. of type `int`) to a pointer of vector-type (e.g. `int4`) by using a simple cast. However this does not correspond with the semantics of a `CAST` in INSPIRE. Therefore, such operations are replaced with a call to the function `ref_reinterpret`. This is already done during the translation from the Clang AST to the INSPIRE DAG by an OpenCL specific extension to the generic frontend. When one of the OpenCL built-in functions starting with `convert_` is used to transform scalar arrays to vectors, this function is not affected by the generic frontend but passed to the OpenCL device frontend extension, which replaces it with a function which iterates over the array and constructs the desired vector element wise.

OpenCL Host Frontend Extension

The host code of an OpenCL program is standard C code, therefore the generic C frontend can translate it to INSPIRE. However, three transformations need to be applied before any meaningful analysis on this code can be performed:

- Replacing OpenCL library functions with INSPIRE implementations.
- Typing of `cl_mem` objects.
- Integrating the device code with the host code.

```
1 let createBuffer = fun(
2   type<'a> t,
3   uint<8> size,
4   ref<array<int<4>,1>> err
5 ) -> ref<array<'a,1>> {
6   err&[0u] := 0;
7   return new(array_create_1D(t, size));
8 };
```

Figure 3.10: INSPIRE code of the OpenCL function `clCreateBuffer`.

Replacing OpenCL Functions with INSPIRE Implementations Most of the functionalities of an OpenCL host code are performed by some functions defined in the header file *cl.h*, defined by the OpenCL standard. However, the actual implementations of those functions are not accessible, since they are vendor specific and not publicly available. Therefore, the generic frontend only adds the function's prototype in the generated INSPIRE code. The OpenCL host frontend extension replaces them with implementations in INSPIRE, so that Insieme can analyze the program. Two exemplary examples are described in the following paragraphs.

The INSPIRE implementation of `clCreateBuffer` can be found in Figure 3.10, which is used to allocate memory on the device. In line 6 the function sets the fourth argument, which corresponds to the error code of `clCreateBuffer` to 0 to indicate a successful execution. Line 7 returns a new array, allocated in the heap, of the requested type and size.

Figure 3.11 shows the implementation of `clEnqueueWriteBuffer` in INSPIRE. This function copies data from host memory to device memory. The arguments `command_queue`, `blocking_write`, `num_events_in_wait_list`, and `event` are dropped since they are not needed in INSPIRE. The argument `devicePtr` is the sink of the copy operation, i.e. the device buffer, while offset represents the starting point for the write operation relative to the buffer's first element in bytes. `cb` is the number of bytes being copied where `hostPtr` is a memory address in the host memory starting from where `cb` bytes will be copied to the buffer. The lines 8–9 show a conversion of `hostPtr` to an array with the same type as the buffer which is needed to perform the copy operation without violating any type constraints. Line 10 converts the `offset` from bytes into number of elements of the corresponding type. The loop in line 11–13 copies the data element wise to the device buffer. The function always returns 0, which is equivalent to `CL_SUCCESS` in order to mimic the behavior of the original OpenCL function.

Many other OpenCL library functions are processed similarly. Some are dropped since they are not needed. Most of the dropped functions deal either with synchronization of out-of-order/non-blocking calls to OpenCL functions (not needed since the INSPIRE code is always assumed to be blocking/in-order) or with gathering information about the device. The code to gather information about the available devices can be dropped from the application code as this task will be taken over by the Insieme Runtime.

Typing of `cl_mem` Objects In OpenCL, all data transfer between the host and the device is done with `cl_mem` buffers. In the host code, they represent a typeless memory area. Such objects are represented in Inspire as abstract data types, which prohibits most analysis. Therefore, the Insieme

```
let writeBuffer = fun(
  ref<array<'a, 1> >  devicePtr,
  uint<8>        offset,
  uint<8>        cb,
  ref<any>       hostPtr
) -> int<4> {
  decl ref<array<'a,1>> hp =
    ref_reinterpret(hostPtr, lit(array<'a, 1>));
  decl uint<8> o = offset / sizeof( lit('a) );
  for(uint<8> i = 0u .. cb) {
    devicePtr[i + o] = *(hp[i]);
  }
  return 0;
};
```

Figure 3.11: INSPIRE code of the OpenCL function `clEnqueueWriteBuffer`.

```
unsigned int n = ...
int* input = (int*)malloc(sizeof(int) * n);
cl_command_queue queue;
...
cl_mem* buf_input = clCreateBuffer(context, CL_MEM_READ_ONLY, n*sizeof(int), NULL,
    NULL);
clEnqueueWriteBuffer(queue, buf_input, CL_TRUE, 0, n*sizeof(int), input, 0, NULL,
    NULL);
...
```

Figure 3.12: Exemple for a buffer initialization in OpenCL.

OpenCL host frontend extension translates all `cl_mem` objects to variables of type `array<a,1>` where `a` is the actual datatype of the elements and `1` specifies the array as one dimensional. In order to identify the datatype, the compiler searches for calls to `clCreateBuffer`. In the `size` argument of those calls it tries to find a `sizeof([type])` call to extract the type from it. If this cannot be found, the compilation fails. The extracted type is then used for the corresponding buffer. Using a buffer twice with two different types is not supported. Updating the type of all buffers implies also updating all function interfaces which use a buffer as an argument. Figure 3.12 shows an example how a buffer may be initialized in OpenCL while the resulting INSPIRE code is shown in Figure 3.13. The INSPIRE code uses the functions `createBuffer` (see Figure 3.10) and `writeBuffer` (see Figure 3.11). The example shows that the variable `queue` is dropped during the translation to INSPIRE as it is not relevant for the functionality of the resulting INSPIRE program.

Integrating the Device Code with the Host Code A very important feature of Insieme is the ability to combine the host and device code to one single program in order to perform analysis on the entire program. When compiling an OpenCL program with a traditional compiler, the integration is only established at run-time, which prohibits many compile time optimizations. Obviously, this connection can only be performed if the kernel function can be determined at compile time. Dynamic

```
decl ref<uint<4>> n = ...
decl ref<ref<array<int<4>,1>>> input = var(new(array_create_1D(type<int<4>>, (
    uint_precision(*n, 8)*sizeof(type<int<4>>))/sizeof(type<int<4>>))));
...
decl ref<ref<array<int<4>,1>>> buf_input = var(createBuffer(type<int<4>>,
    uint_precision(*n, 8), ref_reinterpret(ref_null, type<array<int<4>,1>>)));
writeBuffer(*buf_input, 1u, 0ul, (uint_precision(*n, 8)*sizeof(type<int<4>>)), *
    input);
...
```

Figure 3.13: INSPIRE representation of the code shown in Figure 3.12.

kernel selection at run-time is not supported by Insieme. The compiler reads the name of the kernel function directly form the `kenrel_name` argument of the function `clCreateKernel` which means, that it must contain the kernel function's name as a string. Finding the file, which contains the kernel code is not trivial, since this is normally done at run-time. Therefore, Insieme requires the programmer to add the pragma `#pragma insieme kernelFile "[pathToKernelFile]"` to the call to the OpenCL function `clCreateProgramWithSource`, providing the path to the kernel file. Kernel code which is embedded in the host source file as a string is not supported by Insieme.

Once the kernel code is loaded, it is translated to INSPIRE as described in Section 3.3.2. Each occurrence of `clEnqueueNDRangeKernel` in the host code is then replaced with a call to a function which executes the appropriate kernel function and passes all the arguments to the kernel. It always returns zero, which corresponds to `CL_SUCCESS` in order to match the return value of the OpenCL function `clEnqueueNDRangeKernel`. Figure 3.15 demonstrates what INSPIRE code results from the code fragments in Figure 3.14. The arguments to the kernel are not part of the replaced call, but are specified somewhere in the host code by calls to `clSetKernelArg`. In order to collect all the arguments, the `cl_kernel` variable used in these calls is replaced with a variable of type `tuple`, which is used to collect all the arguments. In the source code shown in Figure 3.15, this tuple is declared at line 32. The calls to `clSetKernelArg` are replaced with a call to a function that stores the argument in the tuple and returns 0 to mimic the behavior of `clSetKernelArg` (line 34). In the call to the kernel function (line 23 – 28), the single elements of the tuple are passed as arguments. If a kernel has an argument using the `__local` qualifier, it is replaced by an unsigned integer, holding the size of the requested array in bytes. This variable is then used to allocate a memory region that represents the behavior of OpenCL variables with the `__local` qualifier as it is described in Section 3.3.2.

Insieme OpenMP Frontend Extension

The Insieme OpenMP frontend extension consists of two components: The first component translates the OpenMP pragmas in the input source code to INSPIRE annotations during the generation of the INSPIRE DAG. The second component generates parallel INSPIRE constructs out of these annotated INSPIRE nodes.

The most important OpenMP pragmas are `omp parallel` and `omp for`. `omp parallel` will be translated to a `parallel/job` construct in INSPIRE, with a variable number of threads. For-loops that are marked with the `omp for` pragma are converted to `pfor` loops which act as a work

```
cl_kernel kernel = clCreateKernel(program, "vec_add", NULL);
...
cl_mem* buf_input = ...
...
clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf_input);
...
size_t localSize = ...
size_t globalSize = ...
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &localSize, 0, NULL,
    NULL);
```

Figure 3.14: Exemplary code fragments setting the arguments for and calling the OpenCL kernel function in Figure 3.7.

sharing construct in INSPIRE as described in [82]. OpenMP barriers are implemented using the `redistribute` function in INSPIRE. Implicit barriers, e.g. at the end of an `omp for` loop are made explicit in their INSPRIE representation. Figure 3.16 shows a simple parallel loop in OpenMP while its INSPIRE representation can be found in Figure 3.17. The last argument of the `pfor` in line 34 expects a function taking the start, end and step size of the loop to be executed in parallel as arguments. To pass additional arguments to the function in line 1, the `bind` expression is used to generate a closure taking the three arguments provided by the `pfor` and capturing the remaining arguments form the surrounding context. Further details about the Insieme OpenMP Frontend can be found in [159].

3.3.3 Insieme Backend

The Insieme Backend translates the INSPIRE code to C code that is suitable for the Insieme Runtime System (see Section 3.3.4). For this purpose, the INSPIRE code is translated to a C-AST which is then converted to C. The backend is written in an extendable way, thus additional modules can be added easily. One of those modules is used to generate OpenCL code as described in the following paragraph. Another one is designed to generate calls to the Insieme Runtime System out of parallel INSPIRE constructs to enable the parallel execution on multi-core CPU architectures. Detailed information about the Insieme backend can be found in [81].

Insieme OpenCL Backend Module

The backend module for OpenCL adds three functionalities to the basic Insieme Backend:

- Generating special OpenCL keywords/functions in the OpenCL kernel code.
- Identifying the OpenCL kernel functions and including them in the output code as a string.
- Adding the needed OpenCL code to the host code.

Generating special OpenCL keywords/functions in the OpenCL kernel code The OpenCL kernel code has several syntax extensions over the C language. These consist of special keywords (e.g.

```
let kernelTupleType = (ref<array<ref<array<int<4>,1>>,1>>,ref<array<ref<array<int
    <4>,1>>,1>>,ref<array<ref<array<int<4>,1>>,1>>,ref<array<uint<4>,1>>);

let setArg = fun(
  ref<kernelTupleType> kernel,
  ref<array<ref<array<int<4>,1>>,1>> argument
) -> int<4> {
    (tuple_ref_elem(kernel, 0, type<ref<array<ref<array<int<4>,1>>,1>>>) :=
        argument);
    return 0;
};

letConvertToVector = fun(
  ref<uint<8>> scalar
) -> vector<uint<8>,3> {
    decl vector<uint<8>,3> vec = [(*scalar), 1, 1];
    return vec;
};

let callKernel = fun(
  kernelTupleType kernel,
  vector<uint<8>,3> globalSize,
  vector<uint<8>,3> localSize
) -> int<4> {
  vec_add(
    *(tuple_member_access(kernel, 0, type<ref<array<ref<array<int<4>,1>>,1>>>)&[0])
        ,
    *(tuple_member_access(kernel, 1, type<ref<array<ref<array<int<4>,1>>,1>>>)&[0])
        ,
    *(tuple_member_access(kernel, 2, type<ref<array<ref<array<int<4>,1>>,1>>>)&[0])
        ,
    *(tuple_member_access(kernel, 3, type<ref<array<uint<4>,1>>>)&[0]),
    type<int<4>>), globalSize, localSize);
  return 0;
};

ref<ref<kernelTupleType>> kernel = var( new(undefined(type<kernelTupleType>)));
...
setArg(*kernel, scalar_to_array(buf_input));
...
decl ref<ref<array<int<4>,1>>> buf_input = ...
...
decl ref<uint<8>> localSize = ...
decl ref<uint<8>> globalSize = ...
callKernel(*( *kernel), convertToVector(globalSize), convertToVector(localSize)));
```

Figure 3.15: INSPIRE representation of the code fragments shown in Figure 3.14, calling the kernel function in Figure 3.8.

```c
int main() {

  unsigned int size = 100;
  int* input1 = malloc(size * sizeof(int));
  int* input2 = ...
  int* output = ...

#pragma omp parallel for
  for(unsigned int i = 0; i < size; ++i) {
    output[i] = input1[i] + input2[i];
  }

  return 0;
}
```

Figure 3.16: Exemple of a simple C program with a OpneMP parallel for loop, performing a vector addition.

`__kernel`, `__global`, etc.) and special data types, especially the OpenCL vector types. Those are generated by this backend module when needed. Furthermore, the module is also responsible to convert matching INSPIRE constructs to function calls such as `get_global_id` and other OpenCL built in functions.

Identifying the OpenCL kernel functions and including them in the output code as a string The output of the Insieme Backend is only one source file. Therefore, the kernel code is embedded in it as a string and then compiled and executed with the corresponding OpenCL utility functions.

Adding the needed OpenCL code to the host code OpenCL needs some code to move data to the devices, start the kernel execution, and retrieve the result. Calls to these functions are also inserted by the OpenCL backend extension. Furthermore, the code for distributing the input data among several devices as well as merging together the results of those devices as described in Chapter 5 is performed at this point. To ease this tasks, Insieme OpenCL Backend Module uses the OpenCL utility library presented in [58].

Insieme Runtime Backend Module This module of the Insieme backend is designed to create programs that can be executed within the Insieme runtime system. The generated C code contains a call to the Insieme Runtime library in order to generate *Insieme work items* [159] which can be executed in parallel. Parallel INSPIRE constructs, which are not intercepted by the OpenCL Backend Module, are translated to calls to the Insieme Runtime Library which trigger the parallel execution of these work items. For example, input programs which are parallelized using OpenMP will be translated to programs using such calls by the Insieme compiler. Which INSPIRE statement is mapped to what call to the Insieme Runtime Library is described in [159].

```
let function = fun(
  ref<ref<array<int<4>,1>>> input1,
  ref<ref<array<int<4>,1>>> input2,
  ref<ref<array<int<4>,1>>> output,
  uint<4> start,
  uint<4> end,
  uint<4> stepsize
) -> unit {
    for(decl uint<4> i = start .. end : stepsize) {
        *output&[i] :=  *( *input1)&[i] + *( *input2&[i]);
    };
};

let body = fun(
  ref<uint<4>> size,
  ref<ref<array<int<4>,1>>> input1,
  ref<ref<array<int<4>,1>>> input2,
  ref<ref<array<int<4>,1>>> output
) -> unit {
    {
        pfor(getThreadGroup(0), 0u, ( *size), 1, bind(start, end, stepsize){
            function(input1, input2, output, start, end, stepsize)});
        barrier();
    };
    mergeAll();
};

let main = fun() -> int<4> {
    decl ref<uint<4>> size = var(100u);
    decl ref<ref<array<int<4>,1>>> input1 = ( var( new(array_create_1D(type<int
        <4>>, (100ul*sizeof(type<int<4>>))/sizeof(type<int<4>>)))));
    decl ref<ref<array<int<4>,1>>> input2 = ...
    decl ref<ref<array<int<4>,1>>> output = ...
    {
        merge(parallel(job(([1-inf])){
            bind(){body(size, input1, input2, output)}
        }));
    };
    return 0;
};
```

Figure 3.17: INSPIRE representation of a program with a parallel loop performing a vector addition, generated from the C code in Figure 3.16.

3.3.4 Insieme Runtime System

Programs compiled with the Insieme compiler are designed to be executed by the Insieme runtime system. Those programs are typically multi-threaded using calls to the Insieme runtime library as described in Section 3.3.3. The Insieme runtime system provides a parallel execution environment with customizable parallelization strategies. It is responsible for mapping the executed program to OS-level threads. For each OS-level thread used, one *worker* is created and started [159]. Every worker is mapped to a specific CPU core in ascending order. Each worker has its own queue of Insieme work items that it processes. The runtime system manages the work distribution among those queues and can apply different distribution strategies to them (e.g. work stealing or static work distribution). If the work item queue is empty, the worker is put to sleep and woken up by the runtime system when new work items are assigned to the corresponding worker. The Insieme runtime system is designed in a modular way. OpenMP programs that have been compiled with the Insieme Compiler are mapped to this customizable infrastructure to maximize their performance. A detailed description of the Insieme runtime system can be found in [159].

Insieme OpenCL Runtime System

The main difference of the Insieme OpenCL runtime system to the generic runtime system implementation is that it queries for and initializes the OpenCL devices and distributes the work among them instead among the CPU cores only. The Insieme runtime system allocates one CPU thread for each OpenCL device. This thread is responsible for all the communication between the host and the corresponding device. Using a separate CPU thread, i.e. worker, for each device is needed to enable efficient overlapping computation of several devices. To find an efficient distribution of the given work over the available, typically heterogeneous, devices, it can use information gathered from an external source, e.g. a database or the Insieme Compiler, and feed them into a previously generated model that suggests a workload distribution.

3.4 Summary

This chapter introduced the programming languages and frameworks which will be used in the following chapters. Two of them, namely OpenCL and OpenMP, are well known and established themselves as quasi standards in their respective fields. While OpenMP is very common in parallel programs for CPUs, OpenCL is mostly used for programming GPUs and hybrid computing nodes. Insieme, on the other hand, is framework designed for research purposes at the University of Innsbruck. It provides researchers a toolkit that can be easily extended and adapted for their needs. The following chapters are examples how the Insieme compiler and runtime system may be used to extend the current state of the art.

Chapter 4

Automatic OpenCL Device Characterization

This chapter presents a benchmark suite (uCLbench) that analyzes the strengths and weaknesses of OpenCL devices. It presents measurements for eight hardware architectures, four GPUs, three CPUs, and one accelerator, and illustrates how the results accurately reflect unique characteristics of the respective device. In addition to measuring quantities traditionally benchmarked on CPUs like arithmetic throughput or the bandwidth and latency of various address spaces, uCLbench also includes code designed to determine parameters unique to OpenCL like the dynamic branching penalties prevalent on GPUs. The contributions in this chapter are a joint work with Dr. Peter Thoman. My main task was the implementation, execution and evaluation of the benchmarks. The results of the research presented in this chapter have been published in [161].

As mentioned in Chapter 1, the search for higher sustained performance and efficiency has led to increasing use of highly parallel architectures. This movement includes GPU computing, accelerator architectures like the Cell Broadband Engine, but also the increased thread- and core-level parallelism in classical CPUs [122]. As described in Section 3.1, OpenCL provides a unified programming environment which is capable of effectively targeting this variety of devices.

Broad acceptance of OpenCL leads to the interesting situation where vastly different hardware architectures can be targeted with essentially unchanged code. However, as mentioned in Section 3.1, implementations that yield good performance on one platform may – because of seemingly small architectural differences – fail to perform well on other platforms. The large and increasing number of hardware and software targets and the complex relationships between code and performance changes make it hard to predict how some algorithm will perform across the full range of platforms.

In order to enable automated in-depth characterization and comparison of OpenCL hardware and software platforms, we have created a suite of microbenchmarks – uCLbench. It includes programs measuring the following data points:

Arithmetic Throughput Parallel and sequential throughput for all basic mathematical operations, and many built-in functions defined by the OpenCL standard. When available, native implementations (with reduced accuracy) are also measured.

Memory Subsystem Host to device, device to device and device to host copying bandwidth. Streaming bandwidth for on-device address spaces. Latency for memory accesses to global, local and constant address spaces. Also determines existence and size of caches.

Branching Penalty Impact of divergent dynamic branching on device performance, particularly pronounced on GPUs.

Runtime Overheads Kernel compilation time and queuing delays incurred when invoking kernels of various code volume.

4.1 Benchmark Design and Methodology

Before examining the individual benchmarks composing the uCLbench suite, the basic goals that shaped our design decisions need to be established. The primary purpose of the suite is to characterize and compare the low-level performance of OpenCL devices and implementations. As such, we did not employ device-specific workarounds to ameliorate problems affecting performance on some particular device, since the same behavior would be encountered by actual programs. Another concern is providing programmers with useful information that can support them in achieving good performance over a broad range of devices. Particularly the latency and branching penalty benchmarks are designed with this goal in mind.

There are three main implementation challenges for uCLbench:

1. **Ensure accuracy**. The benchmarks need to actually measure the intended quantity on all devices tested, and it must be possible to verify the computations performed.
2. **Minimize overheads**. Overheads are always a concern in microbenchmarks, but with the variety of devices available to OpenCL they are difficult to avoid. E.g. a simple loop whose performance impact is negligible on a general purpose CPU can easily dominate execution time on a GPU.
3. **Prevent compiler optimization**. Since kernel code is compiled at run-time using the compiler provided by the OpenCL implementation, we have no control over the generated code. Thus, it is imperative to design the benchmarks in a way that does not allow the compiler to perform unintended optimizations. Such optimizations could result in the removal of operations that should be measured.

There is an obvious area of conflict between these three goals. It is particularly challenging to prevent compiler optimization while not creating significant overheads that could compromise accuracy – even more so when the same code base is used on greatly differing hardware and compiled by different closed-source optimizing compilers.

4.1.1 Arithmetic Throughput

As a central part of the suite, this benchmark measures the arithmetic capabilities of a device. It includes primitive operations as well as many of the complex functions defined in the OpenCL standard. Two distinct quantities are determined: the device-wide throughput that can be achieved

```
__kernel void arith_float(__global float* input, __global float* output, const
    unsigned iterations)
{
  float a = input[0];
  for(unsigned i = 0; i < iterations; i++) {
        a = sin(a);
        a = sin(a);
        // ... repeated 100 times
        a = sin(a);
  }
  output[get_global_id(0)] = a;
}
```

Figure 4.1: Arithmetic Testing Kernel. Example to test the throughput of the sinus operation.

by independent parallel execution as well as the performance achieved for sequentially dependent code. All measurements are taken for scalar and OpenCL vector types.

To enable result checking and prevent compiler optimization, input and output are performed by means of global memory pointers (see Section 3.1), and the result of each operation is used as input for the subsequent ones. The loop is manually unrolled to minimize loop overheads on all devices. Automatic unrolling cannot be relied upon to achieve repeatable results for all platforms and data/operation types. Figure 4.1 shows the kernel function of this benchmark.

The kernel is invoked with a local and global size of one work item to determine the sequential time required for completion of the operation, and with a local size of $loc =$ `CL_DEVICE_MAX_WORK_GROUP_SIZE` (as defined in [89]) and a global size of `CL_DEVICE_MAX_COMPUTE_UNITS`$*loc$ items to calculate device-wide throughput.

4.1.2 Memory Subsystem

Current GPUs and accelerators have a memory design that differs from the deep cache hierarchies common in CPUs.

Bandwidth While global GPU memory bandwidth per-chip is high, due to the degree of hardware parallelism, the memory bandwidth available per compute unit can be insufficient [168]. Another bottleneck for current GPUs is the limited bandwidth between host and device.

As mentioned in Section 2.1, the memory subsystems of CPUs, GPUs and accelerators show significant differences. In OpenCL, the use of scratchpad memory and, to some degree, caches can be controlled by using variables of different address spaces: private, local, constant, global, and host memory.

For this reason, the memory subsystem benchmarks are divided in two major parts: one for on-device memory layers and one for memory traffic between host and device. To test the bandwidth of on-device memory, the benchmark invokes kernels which stream data from one layer in the memory hierarchy back into the same layer. We also discern differences between scalar and various OpenCL vector types, as the latter might show higher performance.

For the streaming kernel, overheads were a major concern. This was addressed by using fast add operations to forestall optimization, and by maximizing the ratio of read/write memory accesses.

Host $\leftrightarrow$ device bandwidth measurement does not require any kernel, instead it uses OpenCL library calls to copy data from/to the device's global memory or inside device's global memory. For device/host communication, two options are considered: the first generates a buffer and commands the OpenCL runtime to transfer it (`clEnqueueWriteBuffer`), the second *maps* a device buffer into the host memory and works directly on the returned pointer.

Latency In addition to bandwidth, knowledge about access latency is essential to effectively utilize the available OpenCL memory spaces. Depending on the device used, only some or none of the accesses may be cached, and latency can vary by two orders of magnitude, from a few cycles up to several hundreds.

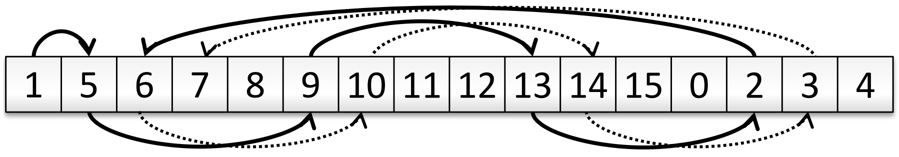


Figure 4.2: Latency benchmark offset array for a cache line size of 4 elements used for latency benchmark.

The latency benchmark uses a specifically designed index array to perform a large number of indirect memory accesses. The index array contains address offsets chosen to cause jumps larger than the cache line size, and end on a zero entry after traversing the entire array, as illustrated in Figure 4.2. The measurement kernel is relatively simple, as shown in Figure 4.3. It is always invoked using a single work item. Constant address space is tested similarly.

Some input-dependent computation and output has to be performed to prevent optimization, which is achieved by accumulating the offsets. Writing the sum to global memory allows correctness checking. To sufficiently reduce the impact of loop overheads when measuring the latency of global memory on all platforms required a manual unrolling of 64 iterations. Note that the resulting program only works correctly for `input` sizes evenly divisible by 64.

```
__kernel void latency(__global uint* input, __global uint* result)
{
  uint res = 0, next = 0;
  do {
    next = input[next];
    // ... 63 repetitions of
    //   "next = input[next];"
    res += next;
  } while(next);
  *result = res;
}
```

Figure 4.3: Memory latency kernel. Example for testing the latency of global memory.

```
__kernel void
branchPenalty(__global float* brancharray, __global float* outarray)

{
  int id = get_global_id(0);
  if(brancharray[id] >= 63.9f && brancharray[id] <= 64.1f) {
    outarray[id] = work(64.0f*brancharray[id]);
  }
  // ... 63 to 2 skipped

  if(brancharray[id] >= 0.9f && brancharray[id] <= 1.1f) {
    outarray[id] = work(1.0f*brancharray[id]);
  }
}
```

Figure 4.4: Branch penalty testing kernel

Measuring local memory latency is slightly more involved, since it requires copying the input array as a first step inside the kernel. This introduces some delay which cannot be measured reliably and exactly. We compensate this by repeating the traversal of the memory segment 10000 times in each invocation. This number of repetitions has proven sufficient to reduce the impact of the copying operation to less than one percent on all platforms.

4.1.3 Branching Penalty

On some OpenCL devices divergent dynamic branching on work items leads to some or all work being serialized. The impact can differ with the amount and topological layout of diverging branches on the work items. Since the effect on algorithm performance of this penalty can be severe [54] we designed a microbenchmark to determine how devices react to various branch counts and layouts.

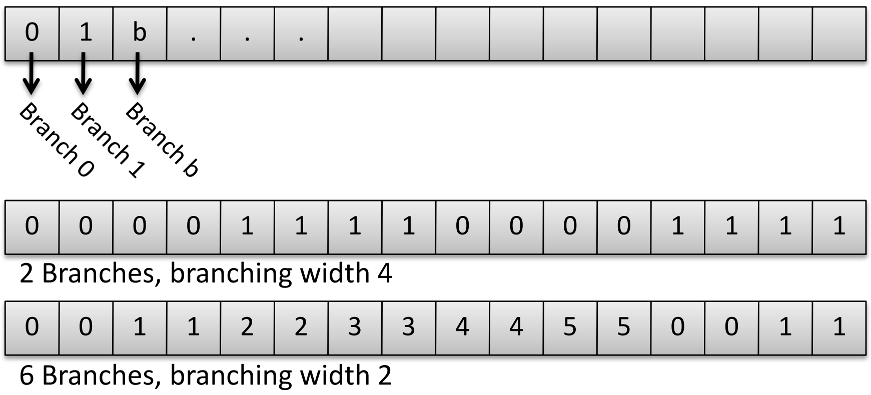


Figure 4.5: Measurement layout options used for branch penalty benchmark.

The benchmark kernel is provided with an array of floating point numbers equal in length to the amount of work items. Each item then takes a branch depending on the number stored in its assigned location. Figure 4.4 shows the kernel code for this test. The `work` function performs busy work independent of the input in order to increase the run time of each branch and make any dynamic branching penalties more easily measurable. Figure 4.5 illustrates how `brancharray` configurations can be used to test a varying number of branches and different branch layouts.

For this benchmark, dealing with the issues described at the beginning of this section was relatively easy. Since only relative performance with different branching behavior is of interest, the function `work` can be scaled without influencing the outcome of the benchmark. This makes overheads a non-issue and allows the benchmark design to concentrate on ensuring that no unwanted compiler optimizations can take place. This is achieved by calling the function `work` with a number partly composed of the dynamic input, and saving the result to global memory.

4.1.4 Runtime Overheads

Compared to traditional program execution, the OpenCL model introduces two potential sources of overhead. Firstly, it is possible to compile kernels at run-time, and secondly there is an amount of time spent between queuing a kernel invocation and the start of computation. These overheads are measured in uCLBench using the OpenCL profiling event mechanism – we define the invocation overhead as the elapsed time between the `CL_PROFILING_COMMAND_QUEUED` and `CL_PROFILING_COMMAND_START` events, and the compilation time as the time spent in the `clBuildProgram` call. The actual kernel execution time is disregarded for this benchmark, and the accuracy of the profiling events is implementation defined (see Table 4.1).

4.2 Device Characterization – Results

To represent the broad spectrum of OpenCL-capable hardware we selected eight devices, comprising four GPUs, three CPUs and one accelerator. Their device characteristics, as reported by OpenCL are summarized in Table 4.1. The following paragraphs describe the used processors in detail.

NVIDIA TESLA 2050 The GF100 Fermi chip in this GPGPU device contains 14 compute units with a load/store unit, a cluster of four special function units as well as two 16 element wide SIMT vector units each. These execution units are fed with instructions by two independent scheduling units.

AMD Radeon HD5870 The Cypress GPU on this card has 20 compute units containing 16 Very Long Instruction Word (VLIW) [51] processors with an instruction word length of five. To benefit from the VLIW architecture in OpenCL the programmer should use a vector data type such as `float4`.

NVIDIA GeForce GTX460 The GTX460 contains a GF110 Fermi GPU which comprises 7 compute units. These compute units are similar to the ones on the TESLA 2050, with one important difference: Each compute unit consists of 3 SIMT vector units fed by 2 superscalar scheduling units.

Device	Tesla2050	Radeon5870	GTX460	GTX275	2x X5570	2x Opt.2435	2xCellPPE	2xCellSPE
Implementation	NVIDIA	AMD	NVIDIA	NVIDIA	AMD	AMD	IBM	IBM
Operating System	CentOS5.3	CentOS5.4	CentOS5.4	Win 7	CentOS5.4	CentOS5.4	YDL 6.2	YDL 6.2
Host Connection	PCIe 2.0	PCIe 2.0	PCIe 2.0	PCIe 2.0	-	-	-	On-chip
Type	GPU	GPU	GPU	GPU	CPU	CPU	CPU	ACCEL
# Chips	1	1	1	1	2	2	2	2
# Compute Units	14	20	7	30	16	12	4	16
Max Workgroup	1024	256	1024	512	1024	1024	256	256
Vect.Width Float	1	1	4	1	4	4	4	4
Clock (MHz)	1147	1400	850	1404	2933	2600	3200	3200
Max.Alloc. (MB)	671	256	512	220	1024	1024	757	757
Images	Yes	Yes	Yes	Yes	No	No	No	No
Max. Kernel Args	4352	1024	4352	4352	4096	4096	256	256
Alignment	64	128	128	16	128	128	1	1
Cache	R/W	None	R/W	None	R/W	R/W	R/W	None
Cache Line	128	-	128	-	64	64	128	-
Cache Size (KB)	224	-	112	-	64	64	32	-
Global Mem (MB)	3072	1024	2048	877	3072	3072	3072	3072
Constant (KB)	64	64	64	64	64	64	64	64
Local Type	Scratch	Scratch	Scratch	Scratch	Global	Global	Global	Scratch
Local (KB)	48	32	48	16	32	32	512	243
Timer Res. (ns)	1000	1	1000	1000	1	1	37	37

Table 4.1: OpenCL devices benchmarked in this chapter

NVIDIA GeForce GTX275 This graphics card is based on the GT200 GPU which has 30 compute units, each containing one eight element wide SIMT vector unit.

Intel Xeon X5570 The Intel Xeon X5570 features 4 physical CPU cores with simultaneous multithreading (SMT) leading to a total of 8 logical cores. The Xeons used in our benchmarks are mounted on an IBM HS22 Blade featuring two Intel Xeon X5570 with shared main memory, resulting in a single OpenCL device with 16 compute units. Each compute unit houses one four element wide vector unit.

AMD Opteron 2435 The Opteron 2435 CPUs used in this chapter are mounted on a dual-socket IBM LS22 Blade. Each Opteron 2435 contains 6 cores leading to a total of 12 compute units with one four element wide vector unit each.

IBM PowerXCell 8i The accelerator device in our benchmarks consists of two PowerXCell 8i mounted on an IBM QS22 Blade. In OpenCL a Cell processor comprises two devices: A CPU (the PPE of the Cell) and an accelerator (all SPEs of the Cell). The two Cell PPEs, each featuring SMT, contain four compute units, the eight SPE cores of the two Cell chips add up to 16 compute units. Every compute unit, both PPE and SPE, contains one eight element vector unit each.

All benchmarked devices in this chapter, except the NVIDIA GPUs, feature SIMD vector units that can be exploited in OpenCL by using vector data types such as `float4`. The vector units of the NVIDIA GPUs can only be addressed with OpenCL using multiple work items within the same work group.

4.2.1 Arithmetic Throughput

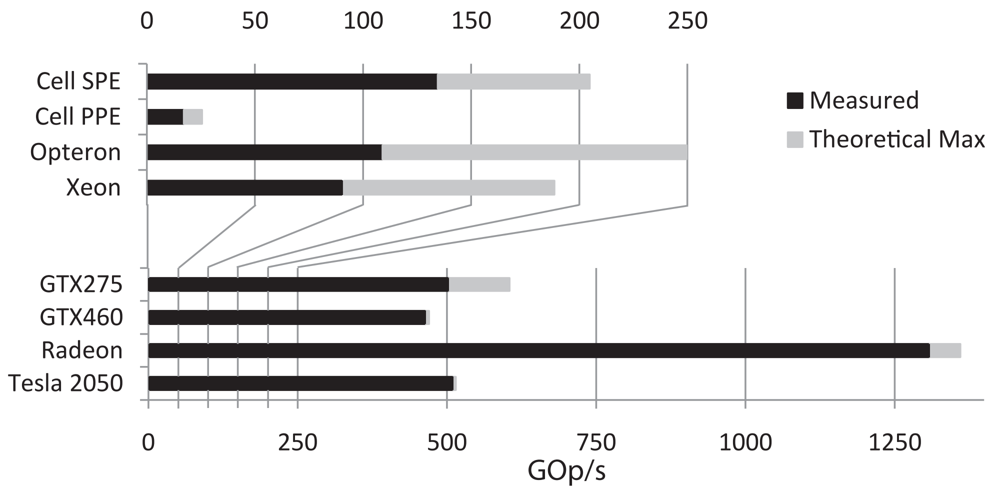
We have gathered well over 3000 throughput measurements using the uCLBench arithmetic benchmark. A small subset that provides an overview of the devices and contains the most significant and interesting results will be presented in this section.

Figure 4.6a shows the number of single precision floating point multiplications per second measured on each device and the theoretical maximum calculated from the hardware specifications. The first thing to note is the large advantage of GPUs in this metric, which necessitates the use of separate scales to portrait all devices meaningfully.

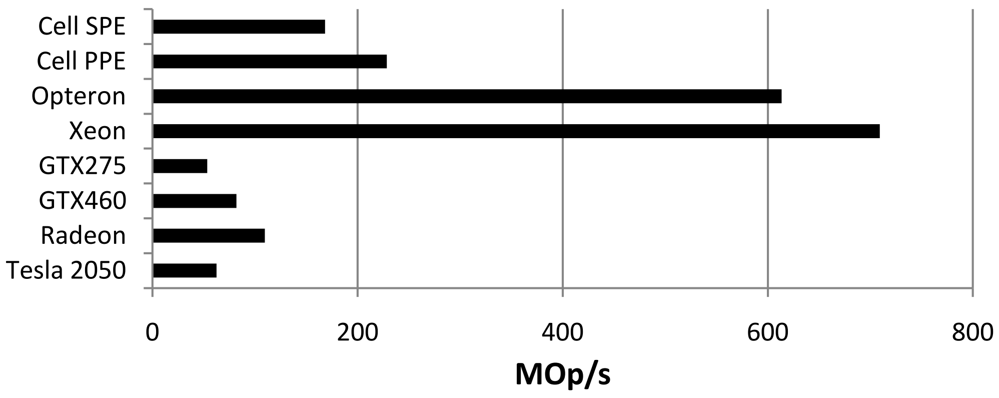
Looking at the effective utilization of hardware capabilities, the GPUs also do well. The Fermi cards reach over 99% utilization. The other GPUs still go over 80% while the two x86 CPUs fail to reach the 50% mark. IBM's OpenCL devices perform a bit better, achieving slightly over 65% of the theoretical maximum throughput on both the PPEs and SPEs.

While throughput of vectorized independent instructions is important for scientific computing and many multimedia workloads, some problems are hard to parallelize. The performance in such cases depends on the speed at which sequentially dependent calculations can be performed, which is summarized in Figure 4.6b. The CPUs clearly outperform GPUs and accelerators here, providing a solid argument for the use of heterogeneous systems.

Vectorization Figures 4.7a and 4.7b show the relative performance impact of manual vectorization using the `floatN` OpenCL datatypes. With a single work item all devices benefit from vectorization to some extent. Since all three CPUs deliver the same relative performance, they are consolidated.



(a) Parallel



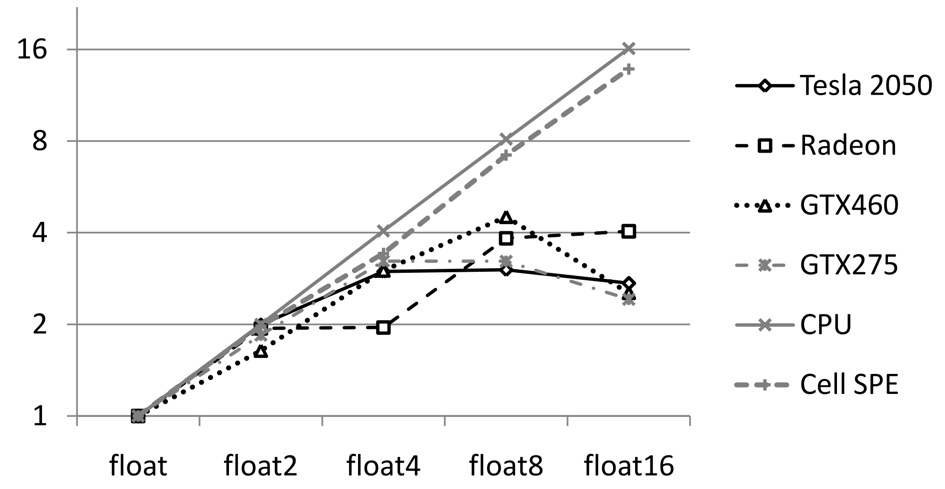
(b) Sequential

Figure 4.6: Floating point multiplication throughput

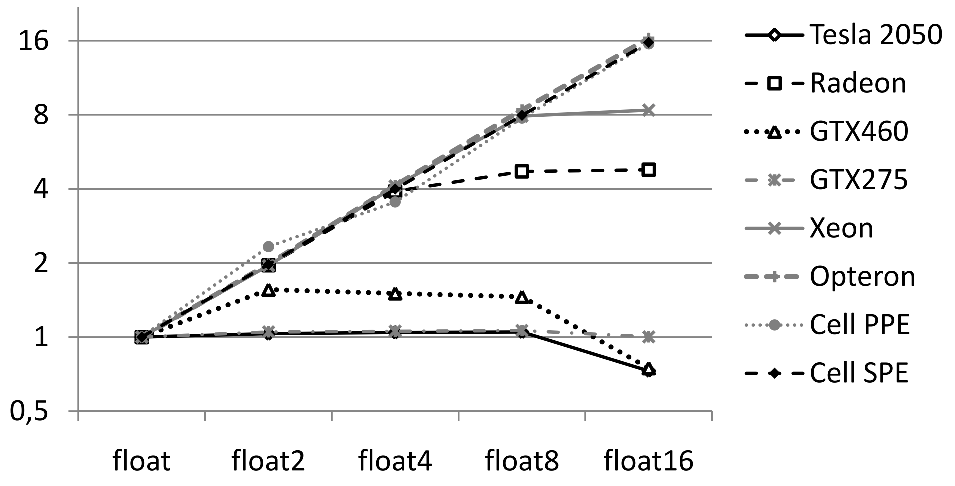
When the full amount of work items is used there are two clearly visible categories: The NVIDIA GPUs effectively gather individual work items into SIMD groups and thus show no additional benefit from manual vectorization, vectors with 16 elements even slow down the execution. The GTX460 result is counter-intuitive, but can be explained by scheduling constraints introduced by the superscalar architecture.

4.2.2 Memory Subsystem

The memory subsystems of the benchmarked OpenCL devices diverge in two areas – availability of dedicated global device memory and structure of the on-chip memory. The GPU devices feature dedicated DRAM (i.e. global memory) while for all other devices the global device address space resides in the host's main memory. Furthermore, the local memory on GPUs and Cell SPUs is a manually managed scratchpad while on CPUs it is placed inside the cache hierarchy.



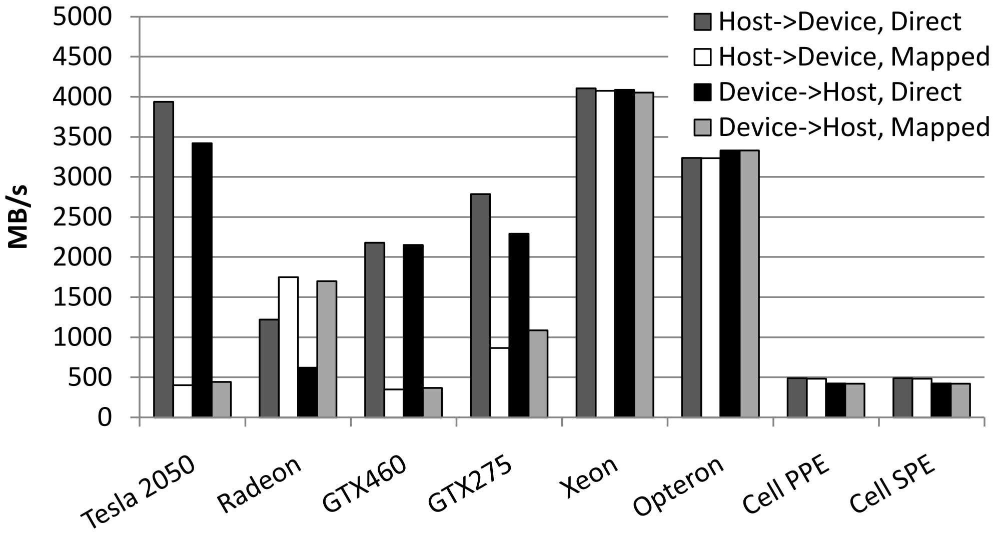
(a) Using a single work item



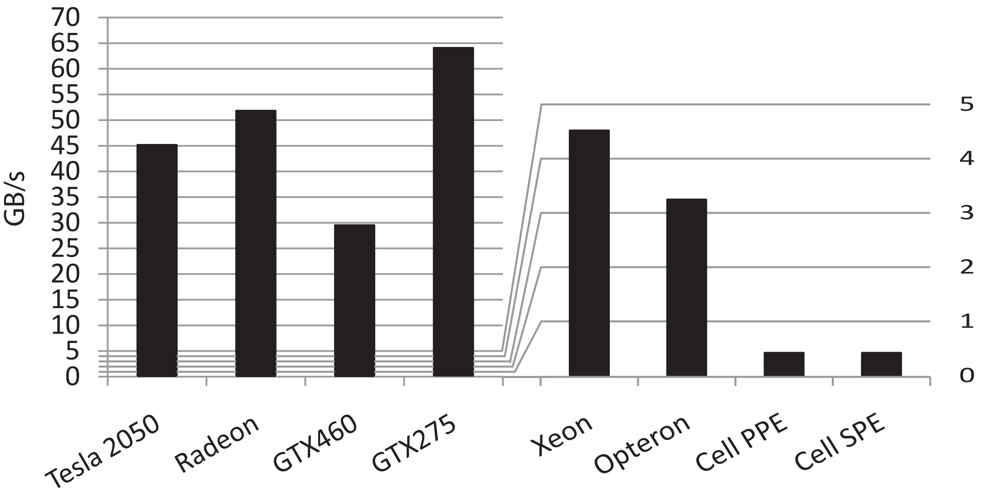
(b) Using many work items

Figure 4.7: Vectorization Impact

Bandwidth The bandwidth measured between host and devices is shown in Figure 4.8a. For CPUs, data is simply copied within their main memory, while for GPUs it has to be transferred over the PCIe bus. Therefore, the bandwidth measured for the CPUs is higher in this benchmark,and the results of the two CPUs using the AMD implementation correspond to their main memory bandwidth. All NVIDIA GPUs perform similarly, whereas the Radeon is far behind them when using direct memory while it is faster when using mapped memory. The Cell processor achieves very low bandwidth although it is equipped with fast memory, a result that we attribute to an immature implementation of the IBM OpenCL runtime.



(a) Bandwidth transferring data from/to the device

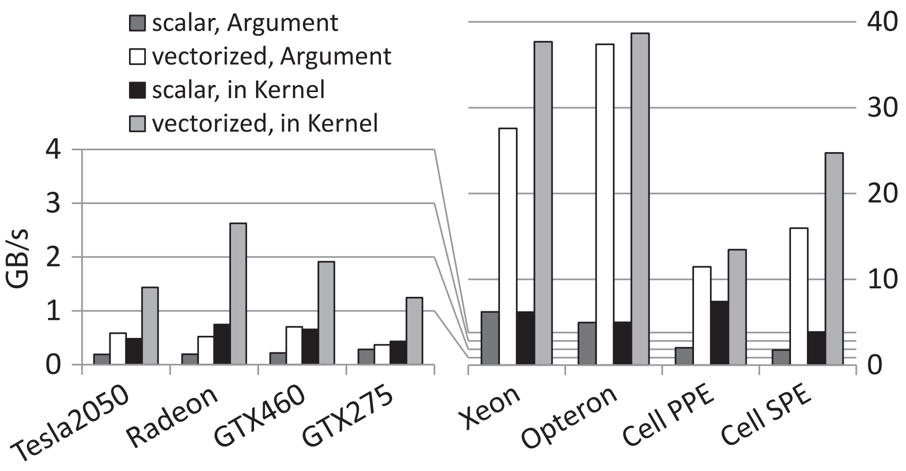


(b) Global device memory bandwidth

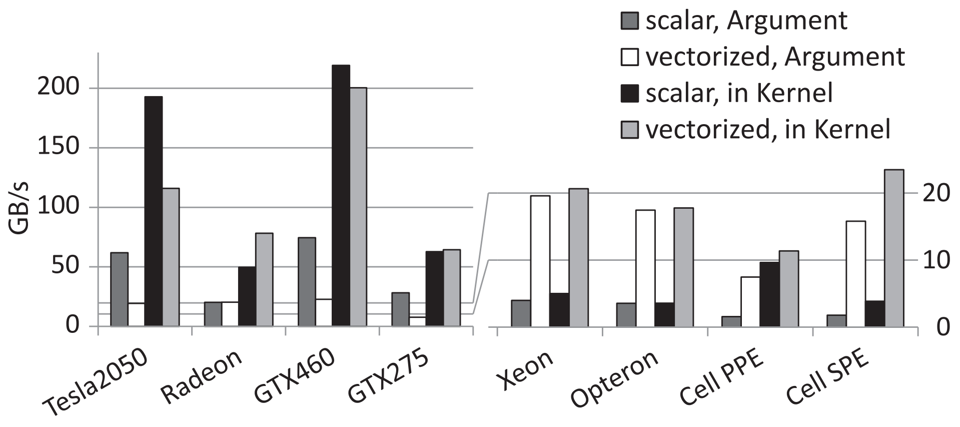
Figure 4.8: Bandwidth measurements

A second property we measured is the bandwidth of the devices' global memory. As shown in Figure 4.8b the GPUs outperform all other architectures in this benchmark due to their wide memory interfaces. The GTX275 outperforms the Radeon as well as the newer NVIDIA GPUs although the theoretical memory bandwidth of the latter ones is slightly higher. All CPUs achieve the same bandwidth as in the host $\leftrightarrow$ device benchmark since host and device memory are physically identical.

Looking further into the memory hierarchy we measure the bandwidth of a single compute unit to its local memory. Since all compute units on a device can access their local memory concurrently, the numbers provided need to be multiplied by the compute unit count in order to calculate the local memory bandwidth of the whole device. We measured the bandwidth in four ways: in the first case



(a) Using a single work item



(b) Using multiple work items

Figure 4.9: Bandwidth of one compute unit to its local memory

only one work item accesses the memory, in the second case the maximum launchable number of work items is used. These two variants were used on local memory that has been statically declared inside a kernel function as well as to local memory passed as an argument to the kernel function. Furthermore, all benchmarks were performed using scalars and vector data types. Figure 4.9a shows the result of the benchmarks using only one work item while Figure 4.9b displays the values for the maximum amount of work items. GPU scratchpad memories are clearly designed to be accessed by multiple work items, and with parallel access their performance increases by up to two orders of magnitude. All GPUs benefit from using vector data types when accessing the memory sequentially, while in the parallel version only the Radeon shows a higher bandwidth when using OpenCL vector types than with scalars.

In contrast to the GPUs, the Cell SPE scratchpad memory can be used efficiently in the sequential benchmark, parallelizing the access has only a minor impact on the speed. On the CPU side, all systems exhibit unexpected slowdowns with multiple work items. We believe that this is caused by

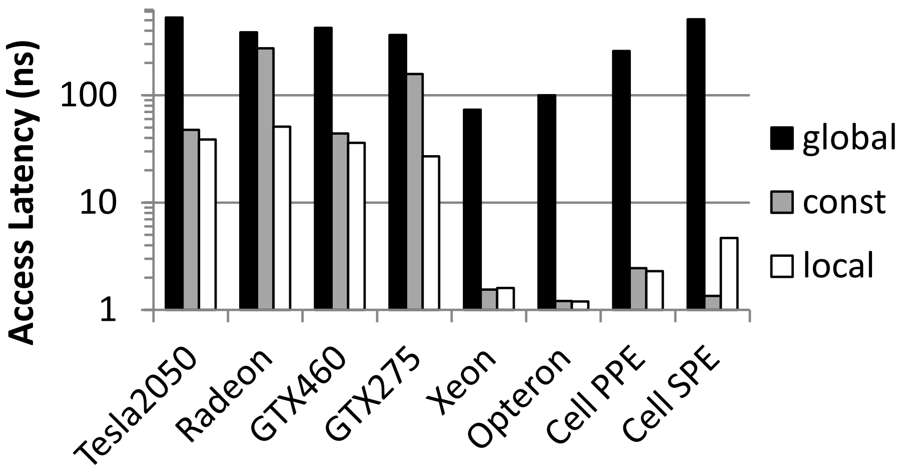


Figure 4.10: Memory access latency of various OpenCL address spaces

superfluous cache coherency operations due to false sharing [163]. All CPUs benefit from OpenCL vector types. All devices show a higher bandwidth to the local memory when it is statically declared inside the kernel function.

Latency One purpose of the multiple address spaces in OpenCL give the programmer access to lower latency memory pools. This is particularly important on GPUs and accelerators, where global memory is un-cached or the caches involved are relatively small. As shown in Figure 4.10, absolute access latency to global memory, i.e. to the DRAM, is almost an order of magnitude larger on GPUs and accelerators than on CPUs. Additionally CPUs can rely on their highly sophisticated cache hierarchies to reduce the access times even further. The impact of caching is shown in Figure 4.11 which shows the relative time to access a data item of a certain size in comparison to the previously measured latency to the global memory. If the data fits in a cache, not the access time to the DRAM, but to the cache is measured.

This depiction clearly identifies the number of caches featured by a device, as well as their usable size in OpenCL. The caches on the CPUs are the most efficient. Since the Fermi GPUs feature a L1 and L2 cache, accesses to data in those caches have a much lower latency than an access to the DRAM. The Radeon and GTX275 as well as the Cell SPE do not feature any automated caching of data in global memory resulting in equal access time for all tested sizes.

Local and constant memory latency is significantly smaller than the latency of the global memory on all devices. On the CPUs it corresponds to L1 cache latency as expected. All four GPUs show very similar performance to access the local memory, while the Fermi based chips outperform the Radeon and GTX275 in accessing the constant memory by approximately six and three times, respectively. The accelerator's behavior more closely resembles a CPU than a GPU regarding local latency, resulting in the largest difference between global and local timings. The SPEs are the only device to achieve significantly lower latency for constant memory than for local memory accesses.

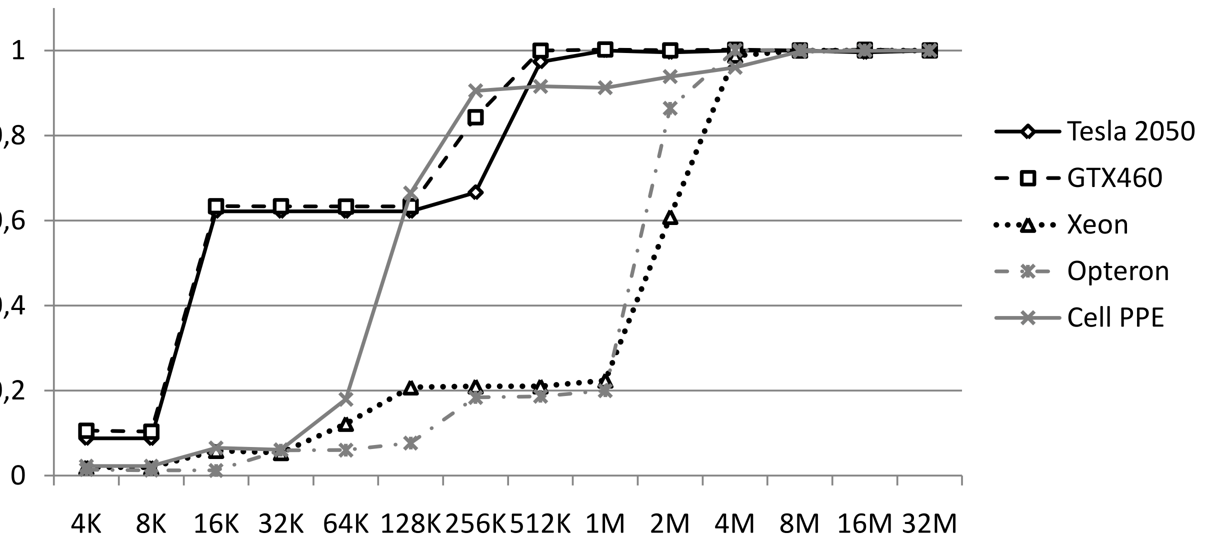


Figure 4.11: Memory access latency relative to un-cached global memory accesses depending on data size demonstrating the impact of caching on global memory latency.

4.2.3 Branching Penalty

We measured the time taken to process the branch penalty testing kernel with one to 128 branches relative to the time required to complete a single branch. All CPUs remain at the same performance level regardless of the number of divergent branches. This is expected, as CPUs do not feature the SIMT execution model that results in a branching penalty. Also the Cell SPE accelerator does not exhibit any penalty. The situation is more interesting for the GPUs, which show a linear increase in run-time with the number of branches until a certain cutoff point. In case of all NVIDIA GPUs, this point is reached at 32 divergent branches, on the Radeon it takes 64 branches. This measurement coincides perfectly with the *warp size* [118] reported for each GPU, which is the number of work items that are grouped together for SIMT execution.

Figure 4.12 summarizes the results obtained varying both branch count and topological layout of branches in the local size. A darker color indicates longer kernel run-time, and the lower right part is black since it contains infeasible combinations of branching width and branch count. Generally, grouping branches together improves the performance. In fact, the hardware behaves in a very predictable way: if the condition $branchingWidth * branchCount \geq warpSize$ is fulfilled, further increases in the branch count will not cause performance degradation. On NVIDIA GPUs, multiples of 8 for the branch width are particularly advantageous, and the same is true for multiples of 16 on the AMD Radeon 5870. For GTX275 and AMD Radeon 5870 this value is equal to the reported width of the architecture's vector units. This is not the case for the Fermi-based NVIDIA GPUs, where a vector unit width of 16 is generally assumed, yet their behavior remains unchanged compared to the older NVIDIA GPU.

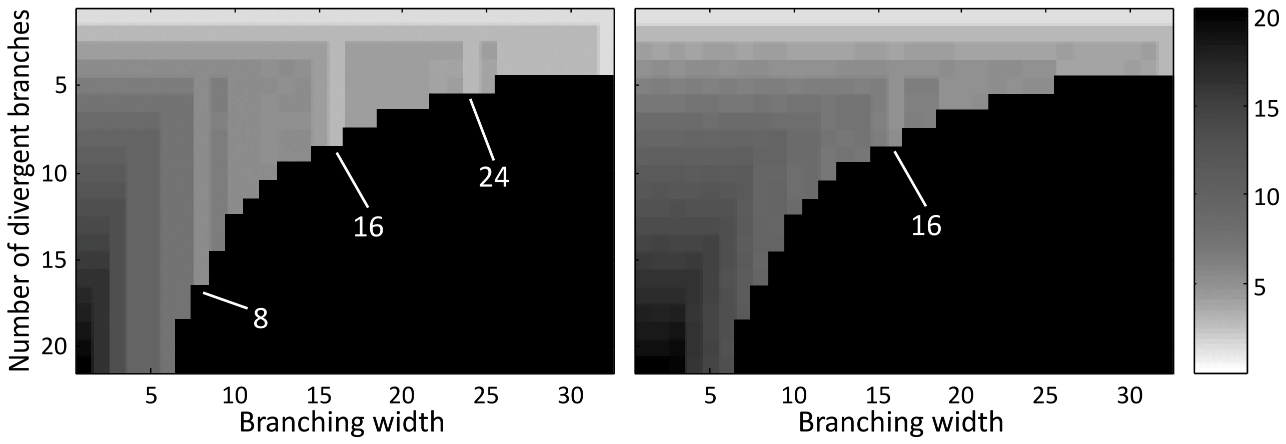


Figure 4.12: Branching penalty with varying branch width for NVIDIA GPUs (left hand side) and the AMD Radeon 5870 (right hand side).

4.2.4 Runtime Overheads

Invocation overheads of a kernel function, as depicted in Figure 4.13, remain below 10 microseconds on the tested x86 CPUs as well as the Fermi GPUs. The two IBM devices and the GTX275 take approximately 30 and 50 microseconds, respectively. The Radeon HD5870 requires approximately 450 microseconds from enqueueing to kernel startup.

We measured compilation times below 1 second for all mature platforms, scaling linearly with code size. The IBM platform has larger compilation times, particularly for the SPEs, reaching 30 seconds and more for kernels beyond 200 lines of code.

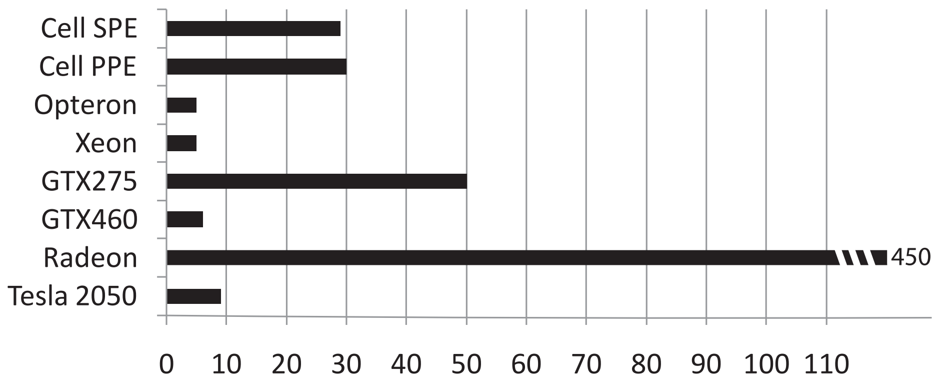


Figure 4.13: Kernel invocation overhead

4.3 Related Work

Microbenchmarks have a long history in the characterization of parallel architectures. The Intel MPI Benchmarks (IMB) [77] are often used to determine the performance of basic MPI operations on clusters. For OpenMP, the EPCC suite [27] measures the overheads incurred by synchronization, loop scheduling and array operations. Bandwidth is widely measured using STREAM [109], and our memory bandwidth benchmark implementation is based on its principles.

A major benefit of using OpenCL is the ability to target GPU devices. Historically these were mostly used for graphics rendering, and benchmarked accordingly, particularly for use in computer games. A popular tool for this purpose is 3DMark [139]. When GPU computing first became widespread Stanford University's GPUbench suite [24] provided valuable low-level information. However, it predates the introduction of specific GPU computing languages and platforms, and therefore only measures performance using the restrictive graphics programming interface. In depth performance analysis of one particular GPU architecture has been performed by Wong et al. [177].

Recently, the SHOC suite of benchmarks for OpenCL was introduced [39]. While it contains some microbenchmarks, it is primarily targeted at measuring mid- to high-level performance. It does not try to identify the individual characteristics of mathematical operations or measure the latency of accesses to OpenCL address spaces. Conversely, our suite is aimed at determining useful low-level characteristics of devices and includes exhaustive latency and arithmetic performance measurements as well as a benchmark investigating dynamic branching penalties. We also present results for a broader range of hardware, including an accelerator device.

The Rodinia Heterogeneous Benchmark Suite [29] predates wide availability of OpenCL, therefore separately covering CUDA, OpenMP and other languages with distinct benchmark codes. Also, unlike uCLbench, Rodinia focuses on determining the performance of high-level patterns of parallelism.

The performance characteristics of additional OpenCL devices which are not covered in this thesis, investigated using uCLbench, can be found in [152]. Additionally, uCLbench has been used to guide device specific optimizations for OpenCL programs in [110, 111, 164, 112]. The authors of [108] and [86] utilized uCLbench to generate hardware features for automatic scheduling of OpenCL kernel functions in heterogeneous nodes, using a model generated with machine learning model and an analytical model, respectively.

4.4 Summary

The uCLbench suite provides tools to accurately measure important low-level device properties including: arithmetic throughput for parallel and sequential code, memory bandwidth and latency to several OpenCL address spaces, compilation time, kernel invocation overheads and divergent dynamic branching penalties. We obtained results on eight compute devices which reflect important hardware characteristics of the platforms.

Our benchmark suite is useful to quickly gaining an in-depth understanding of new hardware and software OpenCL platforms. We demonstrated that microarchitectural features such as warp sizes or memory layout can be accurately deduced from the results of the respective benchmarks. Additionally, uCLbench can help in assessing the viability of a GPU or accelerator version of a certain program, and inform decisions on what device to use for which parts of an algorithm in a heterogeneous system.

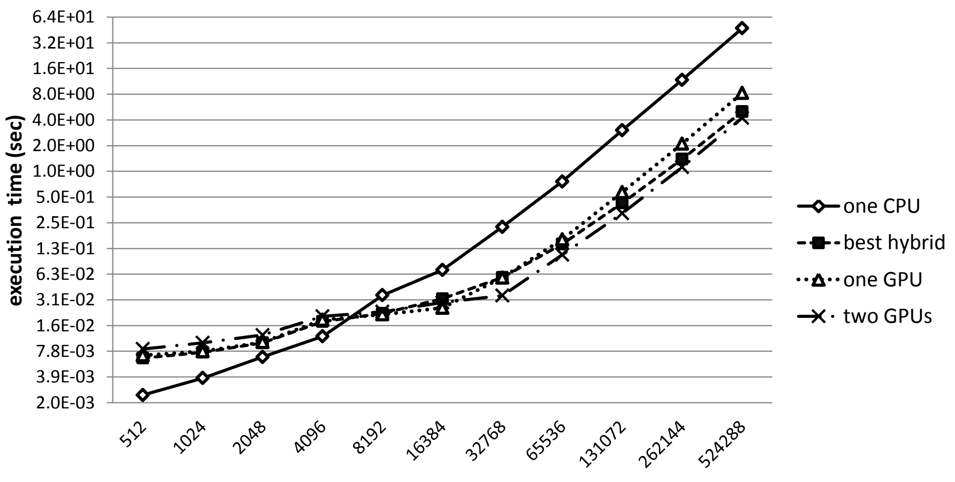
Chapter 5

Automatic Input-Sensitive Heterogeneous Task Partitioning

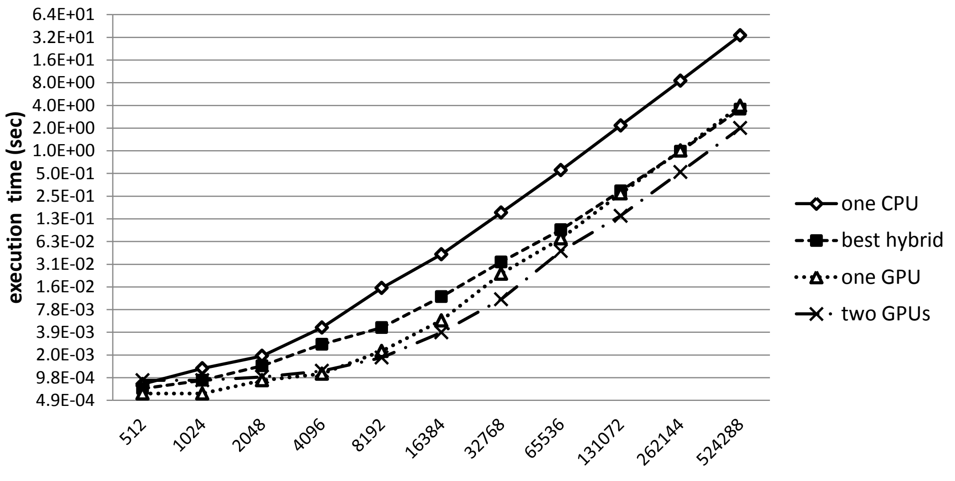
This chapter examines the problem of distributing the workload of an OpenCL program over all available processing units in order to minimize the execution time. The target architecture is a single compute node consisting of three different OpenCL-capable processing units: two (equal) GPUs and one multi-core CPU. The system is executing only one OpenCL kernel at a time, the input codes are standard OpenCL programs. In contrast to previous work, the system presented in this chapter is designed to find an efficient workload distribution for previously unseen programs. This chapter presents a joint work with Ivan Grasso. My contribution was the translation of the input programs to INSPIRE as well as tuning and experimenting with the artificial neural networks, including the work distribution and decision making process. The contributions presented in this chapter have been published in [95].

In the past few years, heterogeneous computing systems (as described in Section 2.1) have emerged as cost-effective means for scaling. The transition from homogeneous to heterogeneous architectures is challenging with respect to the efficient utilization of the hardware resources and the reuse of the software stack. As heterogeneous computing opens many new opportunities for developing parallel algorithms, our work is motivated by the additional challenges and complexity that it introduces. One of the challenges is the distribution of tasks (i.e. task partitioning) among the available OpenCL devices in order to maximize the system performance. Task partitioning defines how the total workload is distributed among several computational resources.

It is important to understand that the best performing task partitioning is likely to change with different applications, different (input) problem sizes, and different hardware configurations. We justify our statement presenting a case study with two programs which are part of our test cases: *linear regression* and *reduction*. The programs have been executed with different problem sizes and varying task paritionings. We measured the execution times on two heterogeneous target architectures consisting of one CPU and two GPUs. The results of these experiments are shown in Figures 5.1 and 5.2. Each chart shows execution times (in seconds) in logarithmic scale (y-axis) with different number of work items (x-axis). Detailed hardware descriptions of the two target architectures *mc1* and *mc2* are shown in Table 5.1.



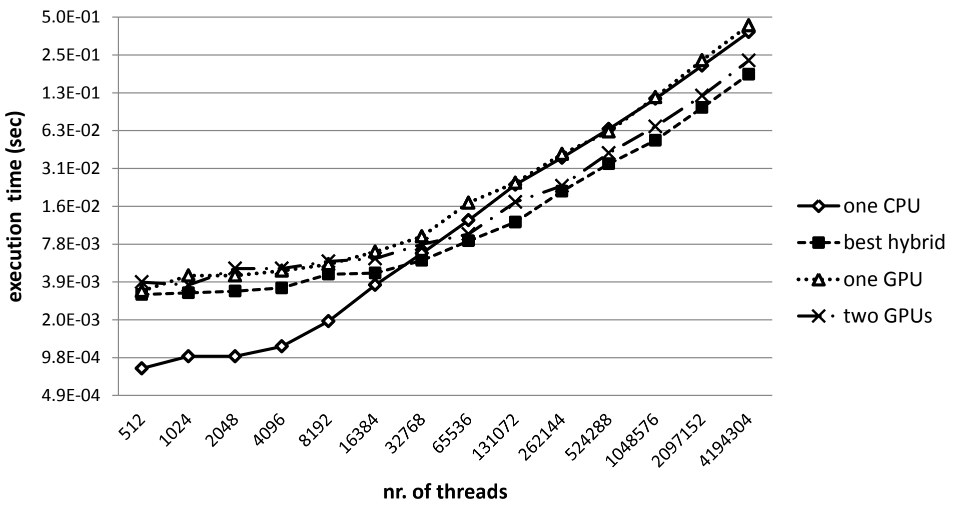
(a) Linear regression on *mc1*



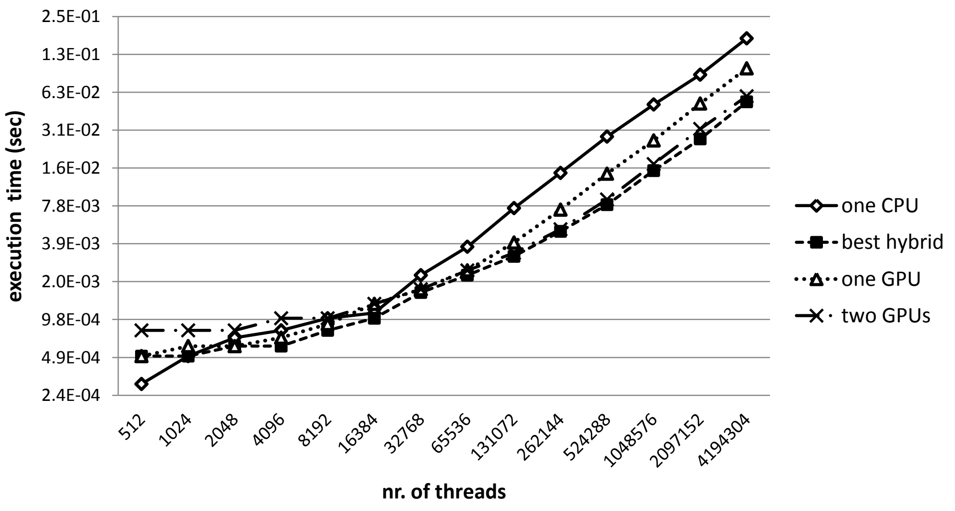
(b) Linear regression on *mc2*

Figure 5.1: Performance behavior of linear regression on different target architectures with varying problem size (i.e. work items).

On our first target architecture, *mc1*, for small problem sizes, the GPU is less effective and the *one CPU* task partitioning delivers the best performance for both applications. However, for some sizes, a *hybrid* task partitioning (using the CPU as well as one or two GPUs) or a *GPU only* task partitioning is preferable. On the second target architecture, *mc2*, *linear regression* performs best on one GPU for smaller problem sizes while *reduction* reaches the best performance with one CPU. For increasing problem size the GPUs become more effective and *linear regression* should be distributed over two GPUs for both *mc1* and *mc2*. The *reduction* program exhibits a different behavior for larger problem sizes, favoring hybrid solutions which outperform any homogeneous configuration by up to 44% and 19% on *mc1* and *mc2*, respectively.



(a) Reduction on *mc1*



(b) Reduction on *mc2*

Figure 5.2: Performance behavior of a parallel chunked reduction on different target architectures with varying problem size (i.e. work items).

These experiments demonstrate that even for a single application, the optimal partitioning considerably depends on the problem size and the capabilities of the hardware.

Another important aspect of heterogeneous computing is the difficulty of writing *multi-device* programs (i.e. a single program which can be executed on multiple devices concurrently). Since current state-of-the-art compilers are not capable of automatizing this complex task, new tools are needed in order to facilitate the conversion of existing programs to heterogeneous systems.

In this chapter we present an automatic, problem size sensitive method for task partitioning of OpenCL programs on heterogeneous systems. Our work is based on machine learning which effectively combines compile time analysis with run-time feature evaluation to predict the optimal

	Machine	
Name	*mc1*	*mc2*
CPU manufacturer	AMD	Intel
CPUs	2x Opteron 6168	2x Xeon X5650
#CPU cores (HT)	24	12 (24)
CPU frequency	1.9 GHz	2.67 GHz
#Parallel Ops (SP)	96	48
Peak Performance	364 GFLOPS	256 GFLOPS
Memory	32 GB	24 GB
Memory Bandwidth	83 GB/s	62 GB/s
Compiler	GCC 4.6.3 w/ ”-O3”	
Operating System	CentOs 5.8	
OpenCL version	AMD APP SDK 2.7	
GPU manufacturer	Ati	NVIDIA
GPUs	Radeon HD 5870	GeForce GTX 480
#GPU cores	20	15
Core frequency	850 MHz	1401 MHz
#Parallel Ops (SP)	1600	480
Peak Performance	2.7 TFLOPS	1.3 TFLOPS
Memory	2 GB	1.5 GB
Memory Bandwidth	153 GB/s	177 GB/s
Connection	PCIe 2.0 x16	PCIe 2.0 x16
OpenCL version	AMD APP SDK 2.7	CUDA 4.1.1

Table 5.1: Experimental target architectures

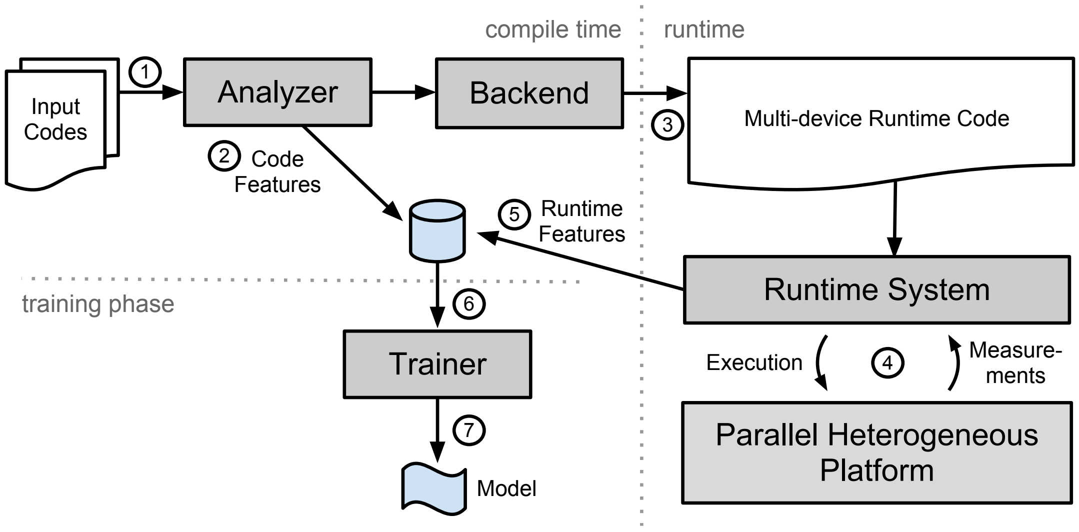
task partitioning for every combination of program, problem size and hardware configuration. The contributions of this chapter are as follows:

- We propose and implement a novel compiler-runtime framework for auto-generation of multi-device OpenCL code and optimized task partitioning on heterogeneous systems. Our framework is portable to any OpenCL environment with an arbitrary number of devices. Its task partitioning system is based on an off-line generated, problem size sensitive model, which is capable of outperforming the CPU/GPU only strategy by 22% and 25%, respectively. Our experimental results demonstrate the capabilities of our approach using 23 different applications on two different heterogeneous multi-device systems.

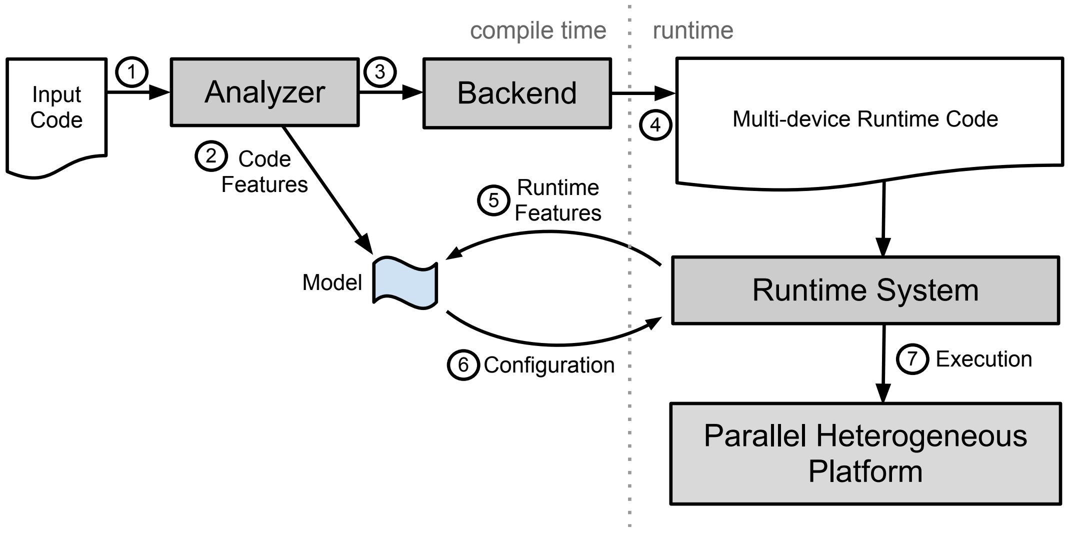
- We show that Principal Component Analysis (PCA) improves the performance of dynamic task partitioning system by 2% to 7%, depending on the used machine learning technique and target architecture.

- We present an analysis of different machine learning techniques suitable to solve the automatic task partitioning problem and show that Artificial Neural Networks (ANN) outperform Support Vector Machines (SVM) in the presented use case.

- We empirically demonstrate the benefits of our machine learning based approaches compared to traditional static task partitioning techniques.



(a) Training phase



(b) Deployment phase

Figure 5.3: Framework Overview

5.1 Framework Overview

Heterogeneous systems are difficult to program, and moreover the performance capability of individual devices can vary significantly across different applications and problem sizes which often makes static, problem size insensitive distribution techniques unsuitable. We use the Insieme Compiler and Runtime framework relieve the developer from this difficult task. The Insieme Compiler translates single-device OpenCL programs (i.e. an OpenCL program which uses only one processing uint) into multi-device OpenCL programs. The Insieme runtime system distributes the computation among the available devices to effectively exploit the performance capabilities of a heterogeneous system.

5.1.1 Architecture

Figure 5.3 illustrates the architecture of the proposed framework, highlighting two main phases: *training* and *deployment*. The labels (1-7) in Figure 5.3a and Figure 5.3b explain the processing of a program within the Insieme framework.

The goal of the training phase is to build a task partitioning prediction model. Any previously unseen target architecture can be supported by generating a new model for it. Since the model generation is done automatically, our approach can be ported to any heterogeneous system without user intervention. To build a model, a set of OpenCL programs are provided to the system and translated to INSPIRE (see Section 3.3.1) by the code analyzer (1). From this representation, the features of the program (the *static program features*) are extracted and stored in a database (2). The intermediate representation of the program is then passed to the backend which generates multi-device OpenCL code (3). Once generated, the new program will be executed with various problem sizes and the available task partitionings. The obtained performance measurements (4), together with the problem size dependent features of the program (the *run-time features*), are collected and added to the database (5). After these steps have been accomplished for all programs, the trainer uses the features and the performance measurements stored in the database (6) to generate a task partitioning prediction model (7).

In the deployment phase a new OpenCL program is provided to the analyzer (1) for optimizations, the static features are extracted (2) and the intermediate representation is passed to the backend (3) which generates a multi-device OpenCL program (4). When the program is executed, the run-time features are provided to the previously trained model (5), which combines them with the static program features to predict the best task partitioning for the current program with the selected problem size (6). Finally, the runtime system executes the program on the given hardware using the predicted task partitioning (7).

5.1.2 Implementation

In this chapter, the input for the Insieme Compiler is a single-device OpenCL program. An OpenCL program consists of a host and a device part as described in Section 3.1. The goal of this work is to reduce the executing time of the device part by distributing the kernel function over multiple devices.

In order to distribute a kernel function, the Insieme Compiler analyzes the generated IR of the input program. It collects the subscripts of all buffer accesses in order to derive the buffer's access pattern. This analysis identifies whether a buffer should be replicated or distributed among several devices. If the buffer can be distributed among several devices, we call it *splittable*. If every device running a fraction of the original kernel function requires the entire buffer we refer to it as *non-splittable*. All buffers with reading accesses in the kernel function have to be splittable in order to allow our system to distribute the kernel over several devices. Due to the limited synchronization capabilities of OpenCL, this is the case for most kernel functions.

The access pattern analysis is based entirely on the device code. However, also the host code has to be adapted according to the results of the access pattern analysis in order to guarantee the correct distribution of data. For this reason, the Insieme source-to-source compiler connects host and device code during the translation of the Clang AST into IR, enabling the analysis of the entire program.

After the analysis, the IR is translated by the backend to a multi-device OpenCL program. The generated code is semantically equivalent to the input code, but its kernels can be distributed among a generic number of devices by the Insieme runtime system. To select the task partitioning *a priori*, the runtime system employs a model generated by machine learning. This model is based on static program features extracted at compile time and problem size sensitive features collected at run-time. A detailed description of how we extract features and build this model can be found in Section 5.2.

5.1.3 Limitations

While the Insieme framework can be used to optimize the performance of many programs on heterogeneous systems, it also has limitations that leave room for future improvement. At the current stage, the buffer analysis and task partitioning are executed individually on each kernel. In programs with multiple kernels, this can cause unnecessary data transfers since the output of each of them must be copied back to the host in order to be redistributed with a new task partitioning.

Device-specific optimizations are not in the scope of this publication. This work aims at distributing a given kernel over a set of devices in the best performing way, not at generating a specifically tuned version of the kernel for each device. Although this issue is not addressed in the current publication, our system is able to handle multiple versions of a kernel.

Other restrictions are related to scattered data accesses and atomic operations, both performed on buffers in global memory. For scattered accesses on buffers, the analysis distinguishes two cases: read-only and read-write buffers. In the first case, the entire buffer will be copied to each device including data that is not needed. In the second case the kernel will be not distributed, since the gathering and merging of writes from different devices is not yet supported. Regarding the use of atomic operations on buffers, OpenCL does not provide any means to implement such operations over multiple devices, therefore the Insieme framework currently does not support kernels with atomic operations.

Our approach cannot deal with irregular workloads due to the difficulty to statically predict an optimized task partitioning for such cases.

5.2 Partitioning Data-Parallel Task

Data-parallel tasks can often be split into smaller sub-tasks and distributed across multiple devices. However, finding an efficient partitioning is not trivial. As will be explained in Section 5.4 and also pointed out by other studies [62], a dynamic scheduling approach may not lead to an optimal solution, mostly due to the large difference in performance and transfer bandwidth of the single devices. Therefore, our approach, based on analysis of the program structure and input data, tries to predict the optimal partitioning for an OpenCL program *a priori*. This section describes the extraction of features and the construction of the machine learning model, used to predict a partitioning.

5.2.1 Predicting the Optimal Partitioning

Our overall approach requires to build a model using machine learning in order to predict a task partitioning a from a vector of features that describes the essential characteristics of a program as well as the current problem size. Each task partitioning is characterized by a tuple of n integer values for a target architecture with n devices. Each value represents the percentage of work that is executed on a particular device. The set A contains all possible partitionings over the available devices with a granularity of 10% and the predicted task partitioning a should be as near as possible to the best task partitioning in terms of performance. As done in [62], we choose a granularity of 10% since this is a good compromise between granularity and number of classes.

5.2.2 Extracting Features

The feature extraction consists of two phases. In the first phase, all the features that can be statically inferred from the intermediate representation are extracted by the Insieme compiler. This phase takes place during the source-to-source compilation step of the Insieme Compiler. In the second phase, the Insieme runtime system determines the values of all problem size dependent run-time features. The second phase takes place when a program is executed, since the problem size is unknown at compile time.

The feature extractor needs to know the execution count of each feature relevant statement. If it is not possible to derive the execution count at compile time (for instance, if loop bounds depend on input data), the feature extractor assumes a loop iteration count of 100. This means that every static feature that appears in a loop is multiplied by 100. If loops are nested, this rule is applied recursively. The resulting value may not be realistic in many cases. However, our goal is not to estimate the absolute execution times but instead compare relative execution times for different devices. Therefore, it is sufficient to consider whether feature relevant statements occur outside, inside or within nested loops. The compiler is also responsible for the generation of one univariate linear polynomial for each run-time feature, which takes the problem size as input. The generated polynomials are evaluated during the second phase of the feature extraction to calculate the actual values of the run-time features.

The features we used to train our framework are sub-divided in static program features (extracted from the intermediate representation during the source-to-source compilation process) and run-time features (calculated by the runtime system when the program is executed). Most static program features count the occurrence of certain activities, like arithmetic operations, memory accesses, or OpenCL built in functions (e.g. *log* or *cos*). Others describe the ratio between two characteristics (e.g. the ratio between computation and memory accesses or the ratio between number of branches and all instructions).

All run-time features depend on the problem size. Apart from the problem size itself, they describe how much data has to be transferred between the host and the devices. We differ between device-to-host and host-to-device transfers and between transfer size for splittable and non splittable buffers. Since splittable buffers are distributed over all devices, the total amount of data to be copied is independent from the number of devices used. In contrast to that, the transfer size of non splittable buffers scales with the number of devices, since each device must hold a copy of the entire buffer in its memory.

Algorithm 5.1 Greedy Feature Selection algorithm. F denotes the set of all features and the features to be used are collected in the set G.

```
 1: function GreedyFeatureSelection($F$:set)
 2:     $F \leftarrow$ non empty set of all features
 3:     $G \leftarrow \emptyset$
 4:     $mse \leftarrow \infty$
 5:     improved $\leftarrow true$
 6:     while improved do
 7:         improved $\leftarrow false$
 8:
 9:         for all $f \in F$ in parallel do
10:             model $\leftarrow$ trainModel($f \cup G$)
11:             mse$_{tmp}$ $\leftarrow$ evaluate(model)
12:             if mse$_{tmp}$ $< mse$ then
13:                 $mse \leftarrow$ mse$_{tmp}$
14:                 $g \leftarrow f$
15:                 improved $\leftarrow true$
16:             end if
17:         end for
18:         if improved then
19:             $F \leftarrow F \setminus g$
20:             $G \leftarrow G \cup g$
21:         end if
22:     end while
23: end function
```

We used the Greedy Feature Selection described in [144] and illustrated by Algorithm 5.1 to select the most important features out of a set of 24 static code features and 9 dynamic run-time features. To select the most important ones, a separate model is trained for each single feature $f \in F$. The feature which generates the model that gives the lowest mean squared error mse is added to the set of selected features G. In the next step, a separate model for each remaining feature f and the already selected ones in set G is trained. Again, the feature which gives the model with the lowest error is added to the set of selected features G. We repeated this step until adding another feature would not further reduce the error.

We performed this greedy algorithm on both target architectures using an SVM. During the feature selection phase, static code features and dynamic run-time features were treated equally. Table 5.3 lists the features that we used to train models on our two target architectures. The column Rank indicates the order in which the features where added by the Greedy Feature Selection. The column MSE shows the mean squared error of the model using the corresponding feature and the ones with a lower rank. The selected features clearly show that on *mc1* the dynamic run-time features have a bigger influence on the result, while on *mc2* the static features are more important. This underlines the necessity to select the features individually for different target architectures. As it will be shown in Section 5.4, the combination of the selected static program features and run-time features seem to

Training codes description	Performance[1] on *mc1*				Performance[1] on *mc2*			
Application	CPU	GPU	SVM	ANN	CPU	GPU	SVM	ANN
Data Transfer to/from Device	90	37	92	**98**	84	72	88	**94**
Vector Addition	77	40	93	**94**	71	69	**87**	85
Matrix Multiplication	64	49	78	**87**	45	79	**98**	90
Black-Scholes Option Pricing	82	41	91	**93**	65	76	93	**95**
Vertex positions in Sine Wave Pattern	15	**70**	34	47	7	70	83	**95**
2D 3x3 Convolution	70	50	94	**98**	38	82	95	**96**
Molecular Dynamics Simulation	81	57	94	**99**	68	87	83	**94**
Sparse Matrix Vector Multiplication	96	59	97	**100**	82	93	**98**	96
Linear Regression	51	59	51	**60**	22	74	70	**83**
K-Means clustering	86	48	97	**98**	76	80	85	**88**
K-Nearest-Neighbor Classification	22	**68**	45	48	5	68	69	**87**
Symmetric Rank-2k Operations	**95**	24	87	78	**94**	49	51	54
Sobel Filter	75	58	91	**97**	51	**90**	85	85
Median Filter	82	54	96	**98**	56	93	90	**96**
Ray-triangle Intersection	90	62	94	**97**	74	**98**	89	94
Finite-time Lyapunow Exponent Field Calculation	77	56	**95**	94	59	82	**85**	84
Flow Map Calculation	91	35	60	**92**	75	81	81	**88**
Chunked Reduction	72	41	84	**89**	61	73	**88**	87
Perlin Noise Generator	**94**	17	81	73	83	49	84	**85**
Chunked Calculation of the Geometric Mean	68	45	81	**92**	54	81	**94**	93
Mersenne Twister Random Number Generator	79	41	**91**	89	67	72	90	**91**
Bytewise Integer Compression	77	39	90	**94**	70	69	89	**95**
Simulation of a Swinging Pendulum	20	**75**	20	20	19	70	58	**76**

[1] Achieved performance compared to the maximum performance in percentage as described in Section 5.4.

Table 5.2: Description of test cases used for model training and performance of various task partitioning strategies.

Rank	static program features	MSE
2	OpenCL built-in functions	76.3
3	Number of branches / number of statements	64.4
4	Scalar float operations / number of statements	61.1

Rank	run-time features	MSE
1	Data transfer size for *splittable buffer* (device to host)	99.7
5	Number of global work items	60.0
6	Data transfer size for *splittable buffer* (host to device)	47.6
7	Data transfer size for *non splittable buffer* (h2d) / number of arith. ops	47.5

(a) Features selected by Greedy Feature Selection for *mc1*

Rank	static program features	MSE
1	Number of branches / number of statements	91.6
2	Scalar float operations / number of statements	75.8
4	OpenCL built-in functions / number of statements	66.9
6	Scalar int operations / number of statements	56.5
7	Vector float operations / number of statements	52.2
8	Number of loops / number of statements	48.6
9	Scalar int operations	47.5
10	Vector float operations	46.9

Rank	run-time features	MSE
3	Data transfer size for *splittable buffer* (host to device)	69.6
5	Data transfer size for *splittable buffer* (device to host)	64.0

(b) Features selected for *mc2*

Table 5.3: Static program and run-time features used by our approach determined using the Greedy Feature Selection [144].

carry enough information to characterize the behavior of our tested programs.

5.2.3 Generating Training Data

To train and validate our model we use the set of codes listed in Table 5.2. As shown in Figure 5.3a, all training codes are compiled with the Insieme source-to-source compiler and their static program features are collected in a database. After the compilation, the programs are executed with various problem sizes (9 to 17 problem sizes, depending on the program) and task partitionings, adding to the database information about run-time features and execution times. The set of explored task partitionings depends on the number of available devices in the system, as described in Section 5.2.1.

In order to generate the training patterns needed for the model generation, we perform an exhaustive search on that set, finding the task partitioning with the best execution time. The size of the search space is defined by the number of experiments multiplied with the number of possible task partitionings.

For the number of training codes and the target architectures considered in our study, the search space consists of $355 \times 21 = 7455$ elements, where 355 corresponds to all problem size/program combinations and 21 is the number of task partitionings.

For each combination of test case and problem size we generate one training pattern that combines static and dynamic program features with the best performing task partitioning. Such task partitioning will then be used as target value during the training of our model.

5.2.4 Building the Model

Based on the training patterns we build a model with one input for each feature (listed in Table 5.3) and one output, which represents the task partitioning predicted by the model. In our framework, the user can choose between Support Vector Machines [36] (SVMs) and Artificial Neural Networks [36] (ANNs). As shown in Table 5.4, SVMs have a much lower training time, while ANNs introduce a lower overhead during the deployment phase and deliver a higher performance.

During the construction of the model we also evaluate the effect of Principal Component Analysis [36] (PCA) on the classification result. PCA can be described as the linear projection that minimizes the average projection cost, defined as the mean squared distance between the data points and their projections [125]. In our case this means that a certain number of features is reduced to a smaller number of new features in a lossy way, conserving as much of the original features' variance as possible. PCA can help to increase the predictive accuracy of models. However, calculating the PCA, which includes the calculation of the features' eigenvalues and eigenvectors, introduces a notable overhead. This means that applying PCA to all our features, which include some values only available at run-time, would substantially increase the execution time. In order to eliminate this additional overhead, we apply PCA only to the static program features, leaving the run-time features unchanged. In this way we move the overhead of calculating the principal components to the source-to-source compilation phase, not affecting the execution time. The effect of PCA on our models' performance is described in Section 5.4.2.

5.3 Experimental Methodology

This section describes the test cases and the target architectures used in our experiments as well as the evaluation methodology.

5.3.1 Test Cases

To evaluate the performance of our approach we used a selection of 23 programs (see Table 5.2). These programs have been drawn from OpenCL vendors example codes, applications from our department and VRC at the Universität Stuttgart [124], and benchmark suites [40, 60, 30]. After translating the OpenCL input program with the Insieme compiler, the Gnu Gcc Compiler version 4.6.3 was used to convert the resulting code to binary.

In order to examine the impact of problem sizes on task partitioning we executed each benchmark with varying problem sizes on two target architectures. For each test case we examined 9 to 17 different problem sizes (depending on the amount of memory needed by the program), resulting in 355 training patterns. Each training pattern consists of the static features of a program, its run-time features for a certain problem size as well as the best task partitioning for the given program with the current problem size. To ensure a fair comparison between different task partitionings, we measured the execution time of the kernels including the memory transfer overhead [61].

5.3.2 Experimental Setup

The experiments were performed on two different heterogeneous target architectures composed of three OpenCL devices: two GPU devices and one CPU device.The first platform, *mc1*, consists of two AMD Opteron CPUs and two Ati Radeon GPUs, while the second, *mc2*, holds two Intel Xeon CPUs and two NVIDIA GeForce GPUs. Table 5.1 gives a more detailed listing of the two systems' characteristics.

As already mentioned in Section 5.2.1 we use a set of task partitionings. For the target architectures used in this study, consisting of one CPU device and two GPU devices, we characterize each task partitioning with a tuple of three numbers representing the percentage of the workload executed on a specific device. The first number represents the portion to be executed on the CPU while the second and third number represent the percentage for the first and second GPU, respectively. Task partitioning $(100, 0, 0)$, for example, means that the entire workload is assigned to the CPU, while $(0, 50, 50)$ means that the work is distributed evenly among the two GPUs while nothing is assigned to the CPU. The entire set of task partitionings A is constructed as follows:

$$\begin{aligned} X =& \{0, 10, 20, ..., 100\} \\ A =& \bigcup_{x \in X} \{(x, 100 - x, 0), (x, \tfrac{100-x}{2}, \tfrac{100-x}{2})\} \end{aligned}$$

Where X is the set of different percentage values of the workload considered to be executed by the CPU. The remaining workload is then executed either by the first GPU or distributed evenly among the two GPUs. The resulting set A consists of 21 different task partitionings.

From this set A our runtime system tries to select the optimal task partitioning using the prediction model as described in Section 5.2. To evaluate the performance of our approach we compare the execution times of a program with two different task partitionings. The first task partitioning is proposed by The Insieme Runtime System and the second one is found by an exhaustive search over all task partitionings in the set A.

In order to evaluate the quality of our models we perform a leave-one-out cross validation [47] on all our training programs of the set H listed in Table 5.2. To evaluate the model's performance for a particular program $h \in H$, we train the model with all programs except h. Obviously, this means not leaving out only one training pattern, but all training patterns related to program h (i.e. all different problem sizes).

5.4 Experimental Results

In this section we report the performance of our approach. As performance metric we use the achieved percentage of the optimal performance, which can be reached by applying the best task partitioning. We calculate the performance of a task partitioning as follows

$$b = t_{best}/t_{actual} * 100$$

where b is the achieved performance in percentage, t_{best} is the execution time of the best task partitioning (identified with an exhaustive search over all task partitionings used) and t_{actual} is the actual execution time of the selected task partitioning. To combine the performance for several experiments in one value (e.g. the performance for a specific test case using different problem sizes), we simply calculate the average of the performance across these experiments.

5.4.1 Performance Results

As shown by the measurements presented in Figure 5.4, depending on the target architecture, the problem size, and the program, it can be important to select a certain task partitioning, whereas in other cases, several different task partitionings may deliver similarly good performance. The diagrams in this figure list the various task partitionings on the x-axis while the y-axis shows the achieved performance in %, relative to the best task partitioning (as described in Section 5.4). As it can be seen in Figures 5.4a and 5.4b, when executing *matrix multiplication* with large problem sizes it is very important to distribute the workload over both GPUs. Furthermore, for hybrid solutions it is not important if one or two GPUs are used, since the CPU is always the limiting factor. For smaller problem sizes, in particular for *mc2*, several task partitionings yield good performance. In contrast to that, on *mc1* small matrices should be multiplied on the CPU alone. The penalty for selecting a non-optimal task partitioning on intermediate problem sizes on *mc1* is less severe than on *mc2*.

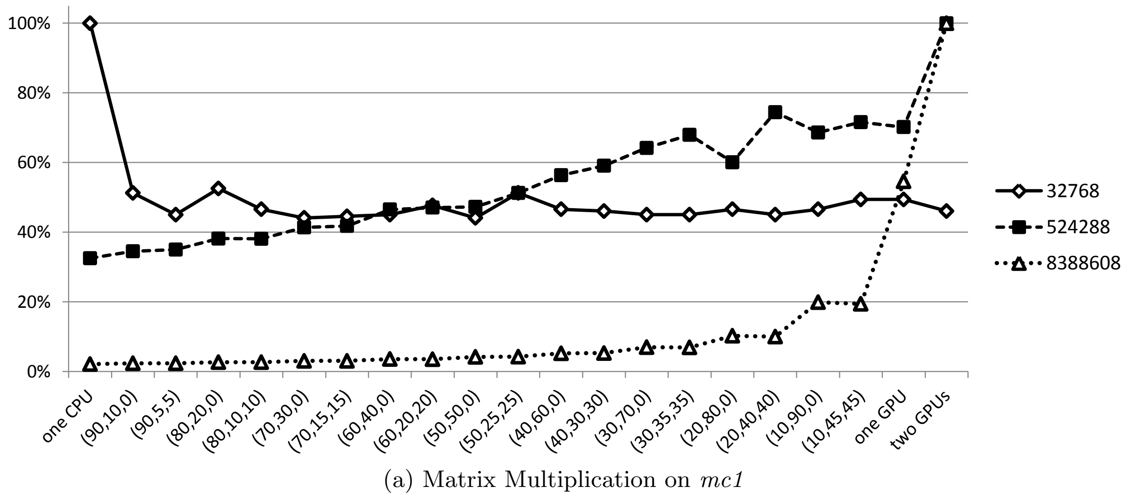
The situation is different when running the *integer compression* benchmark. Figure 5.5a shows that on *mc1* with a problem size of 16384 work items, CPU only substantially outperforms all other task partitionings, while on *mc2* the difference is much smaller and all task partitionings deliver 40% or more of the maximum performance, as revealed in Figure 5.5b. For the larger problem sizes, on both target architectures a hybrid task partitioning delivers the best performance. However, the best performing task partitioning is different for each problem size and target architecture. In this test case, using a heterogeneous distribution can reduce the execution time by up to 23% over any homogeneous task partitioning (including the dual GPU task partitioning).

As shown in Figures 5.1 and 5.2, there are cases in which a single GPU performs better than two GPUs. This behavior can be observed in some data transfer dominated scenarios and is mainly related to the shared connection of the GPUs to the CPU's main memory.

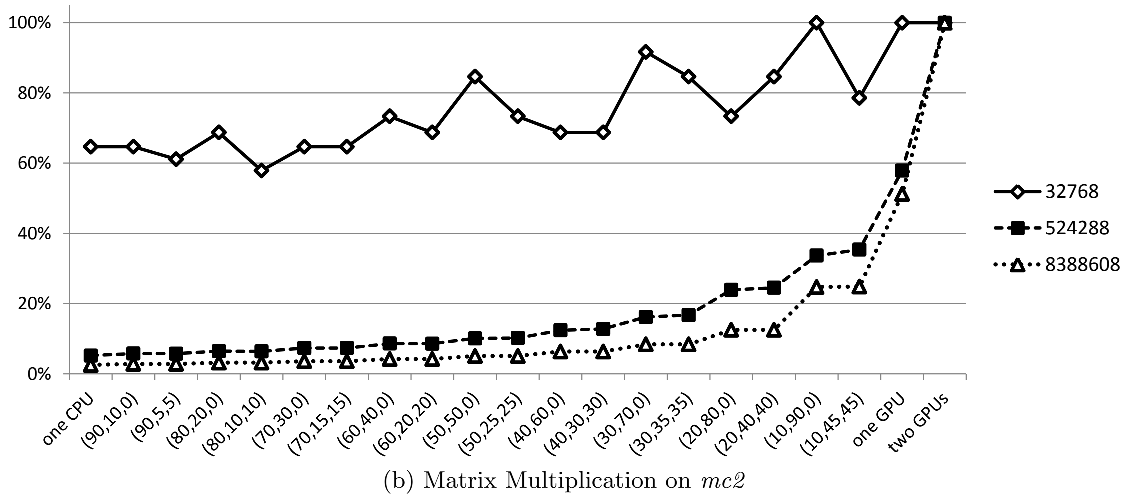
From the 355 training patterns considered for this study, more than 25% deliver best performance when using a hybrid task partitioning.

5.4.2 Comparison of Different Models/Techniques

To select the best partitioning we tested a variety of models, generated either with a Support Vector Machine [36] (SVM) or an Artificial Neural Network [36] (ANN). For both techniques we used the



(a) Matrix Multiplication on *mc1*



(b) Matrix Multiplication on *mc2*

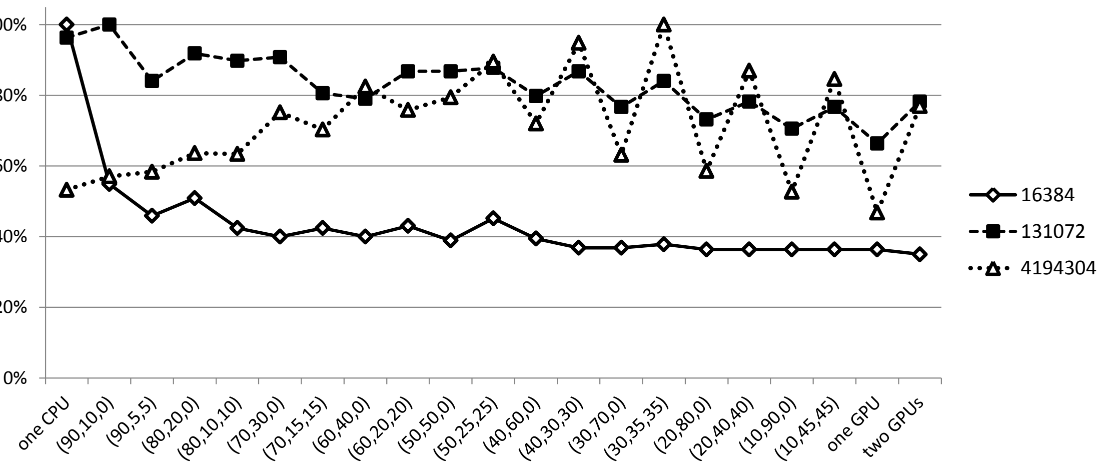
Figure 5.4: Performance behavior of matrix multiplication on two target architectures with different workload distributions.

implementation provided by the Shark library [76]. In this section, we compare the performance of our model-guided runtime system with the performance of the two default strategies which use either one CPU or one GPU. These are the only available options when using the unchanged input programs, without the generation of multi-device code by the Insieme Compiler.

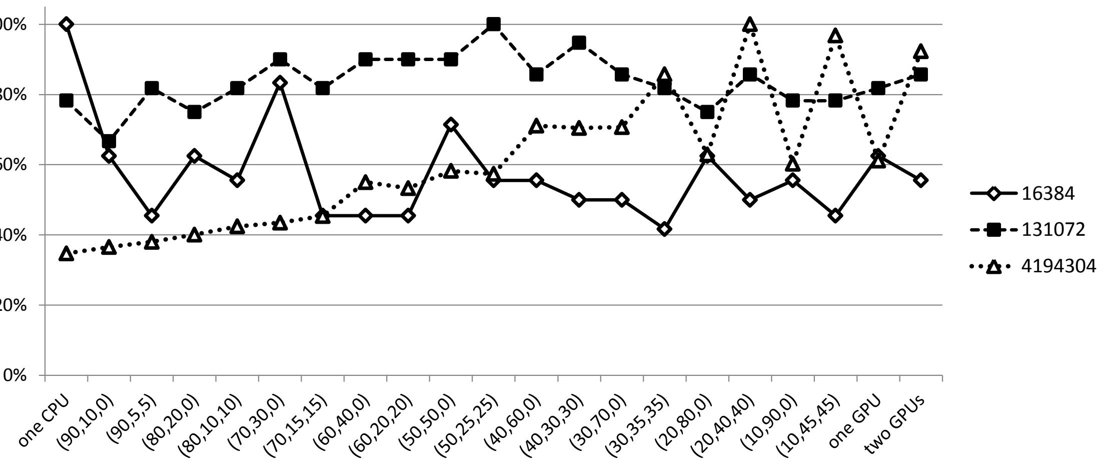
Furthermore, without using Insieme framework, the challenging task of choosing the most appropriate device is left to the user.

We also show the advantage of our approach over the expected performance of a random scheduler, calculated by taking the average execution time over all task partitionings in our set A (described in Section 5.3.2).

Table 5.4 shows the average performance for a cross validation over all test cases in Table 5.2 using different scheduling approaches. On *mc1* the CPU-only strategy outperforms the GPU-only strategy



(a) Bytewise Integer Compression on *mc1*



(b) Bytewise Integer Compression on *mc2*

Figure 5.5: .
Performance behavior of bytewise integer compression on two target architectures with different workload distributions.

while on *mc2* we observe the opposite behavior. This underlines the complexity of choosing the most appropriate device in a heterogeneous environment. On average, over the two target architectures, both default strategies fail to reach 70% of the optimal performance. In most cases there are only few well performing task partitionings while the others show rather poor performance. Therefore, the random scheduler is not a good solution and even lags behind the two default strategies.

Our SVM approach uses the MulticlassSVM implementation of [76]. As kernel function we used Radial Basis Function [36] (RBF). This kernel function is the most widely used for classification with SVMs. The parameter γ of the RBF was set to 2.5, the regularization parameter c was set to 15 for both positive and negative examples. We observed, that the performance does not vary more than 4 - 5% when changing these values, which demonstrates the robustness of SVMs with regard to these

Task Partitioning Approach	Execution Time: Training (sec) *mc1*	*mc2*	Deployment (ms) *mc1*	*mc2*	Performance[1] *mc1*	*mc2*	**Avg.**
CPU only	-	-	-	-	73	58	**65.5**
GPU only	-	-	-	-	48	77	**62.5**
Random	-	-	0.12	0.09	44	55	**49.5**
SVM[2]	8	8	0.31	0.23	80	78	**79.0**
ANN[2]	248	421	0.07	0.07	84	84	**84.0**
SVM[3]	22	19	0.28	0.18	82	85	**83.5**
ANN[3]	317	201	0.07	0.06	86	89	**87.5**

[1] Percentage of maximum performance as described in Section 5.4.
[2] Using all static features listed in Table 5.3.
[3] Using static features generated form the static features listed in Table 5.3 with PCA.

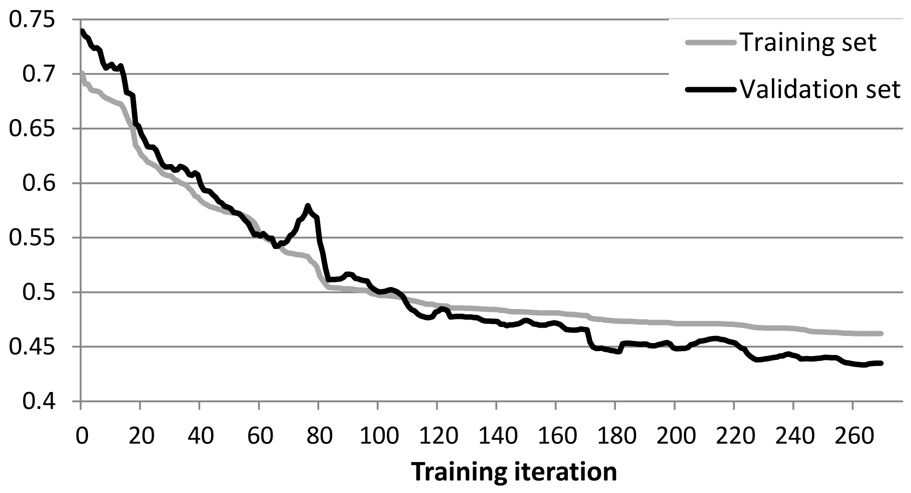
Table 5.4: Properties and performance of different machine learning algorithms.

parameters.

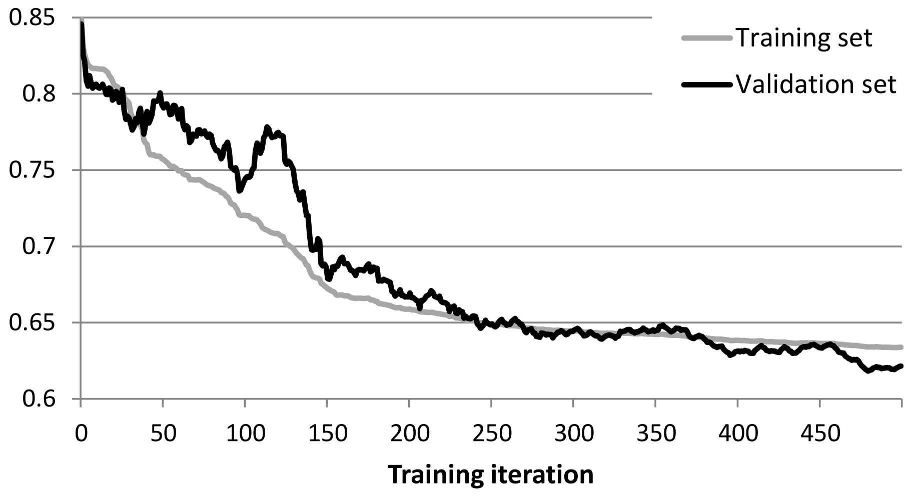
The ANNs used for our study are three-layer feed-forward perceptron networks with a sigmoid activation function and five neurons in the hidden layer [36]. All three layers are fully connected with their neighboring layers. For our ANN we use the FFNet implementation of [76]. All weights inside an ANN are initialized randomly within the same range, equal to ± 0.125.

As training algorithm we used the conjugate gradient method provided by Shark, which automatically adapts the training rate. To determine the number of training iterations for the neural network, we use the *early stopping* method which terminates the training automatically after a certain level of convergence is reached. The training data is split into a training set, used to train the model, and a validation set which is not used for training. The level of convergence is measured by observing how the error on the validation set evolves over consecutive training iterations [36]. Depending on what test case is removed from the training set to perform the cross validation, the training is stopped after 36 to 749 iterations. The training times shown in Table 5.4 refer to the training for all test cases without cross validation. Figure 5.6 shows how the mean squared error evolves on the training set and the validation set during the training (without cross validation) on both of our target architectures using our best performing ANN. In both cases the error curves on the training set are very smooth and converge to a minimum. As usual, the error curves on the validation set are more uneven, but they also converge during the training. Surprisingly, in both cases the mean squared error on the validation set was lower than the one on the training set when the training was stopped.

As explained in Section 5.2.4, we apply PCA to our static program features. On both target architectures we use the first n principal components of the static code features listed in Table 5.3 in order cover 100% of the static program features' total variance (calculated in single precision floating point). For the static features used on *mc1*, this resulted in using only the first principal component. For the static features used on *mc2*, two principal components were needed to cover all their variance. Our results in Table 5.4 clearly show that PCA improves the accuracy of our models and shortens the



(a) Best performing ANN on *mc1*



(b) Best performing ANN on *mc2*

Figure 5.6: Error curves showing the means squared error after each training iteration on our two experimental target architectures.

deployment times. PCA is only applied to static code features, so it is not part of the execution time of the programs. It is noticeable that the models used on *mc2* benefit more from the PCA than the models used on *mc1*. This is most likely related to the relative large number of static code features used on *mc2*, which can be reduced to one quarter of the original number using PCA without loosing any information.

Our task partitioning approach which assigns one portion of the task to each device, has some significant advantages over a dynamic scheduler. A dynamic scheduler has to split a task into a large amount of small chunks. At the beginning of the execution, each device receives one chunk. When a device has finished its assigned work, it will receive another chunk until the entire task has been processed. The chunk size is a very important factor for such an approach. Smaller chunks are better for load balancing, but they reduce the parallelism inside one chunk and suffer from higher data transfer and kernel invocation overhead. Larger chunks reduce the load balancing, but also the number of kernel invocations and data transfers, resulting in a lower overall overhead. On the one hand, a scheduler for OpenCL task partitioning should use large chunks, because the kernel invocation and data transfer overhead are relatively high, compared to the execution time. For example, executing two *vector addition* chunks with a size of 65536 on a GPU in *mc1* takes 71% longer than running one chunk of twice the size. On the other hand, a scheduler for OpenCL task partitioning requires small chunks, due to the high differences in performance of the heterogeneous devices. As it can be seen in Figure 5.4a, with a problem size of 838868 running only 10% of the task on the CPU reduces the performance to 20% of the performance that can be reached by distributing the task evenly over both GPUs. Based on this observation, we believe that dynamic schedulers cannot efficiently solve the task partitioning problem described in this chapter.

5.4.3 Analysis of the Results

In Table 5.2 we compare the performance of the task partitionings predicted by the Insieme Runtime System based on an SVM and ANN using PCA (listed in Table 5.4), with the performance delivered by the CPU/GPU only strategy for each code and each target architecture individually. For almost all test cases, the CPU-only strategy delivers a higher performance on *mc1* than on *mc2*, while the GPU-only strategy usually performs better on *mc2*. This is related to the weaker performance of the GPU (Ati Radeon HD 5870) in *mc1*. Its VLIW architecture with a very wide instruction width and high branch miss penalty would require specific fine-tuning of each code to perform well [161]. However, none of our test cases was tuned for a specific device.

On average, considering both target architectures, our machine learning guided approaches deliver a significant better performance than the two default strategies for most test cases. Our models are capable of representing the target architecture's characteristics in order to find performance efficient task partitionings. Our approaches also determine which device is to be favored on a specific target architecture. This is underlined by the fact that our machine learning guided approaches show their worst performance in atypical test cases, i.e. test cases which perform better on the GPU than on the CPU on *mc1* (e.g. Simulation of a Swinging Pendulum) or vice versa on *mc2* (e.g. Symmetric Rank-2k Operations on *mc2*).

In most cases the ANN shows a better performance than the SVM. The ANN is also faster to predict the task partitioning of a program, as shown in Table 5.4. For both of our approaches the time to predict the task partitioning is negligible (in the range of 0.06 to 0.31 ms). The downside of ANN is the corresponding relatively long training time as well as the associated sensitivity regarding the tuning parameters like network structure or weight initialization range. SVMs do not have this many tuning parameters and the quality of the result does not depend that much on the parameters' value.

5.5 Related Work

In recent years, heterogeneous systems have received great attention from the research community. Several projects [149, 12, 17, 147, 91, 90] mainly focused on OpenMP, CUDA, and OpenCL extensions, have investigated how to facilitate the programming of clusters with heterogeneous compute nodes.

Our work, while following the same idea, is focused on automatic management of multiple devices in a single compute node. A similar study was done by Chen et al. [33]. The authors introduce an automatic parallelization process to use multiple GPUs. This work targets mainly the analysis of access patterns for data decomposition, showing that many applications can be parallelized automatically. Our approach, based on a similar analysis, not only derives the data partition schemes, but also provides a solution for optimal task partitioning on heterogeneous devices.

In a different perspective, much work has been done to address mapping or scheduling of tasks to heterogeneous systems. Several frameworks [149, 15, 104] have been created to support the developer in the use of all available computing resources of a heterogeneous system. Although these studies propose several possible solutions to the problem, they are mostly based on performance estimations provided by the user. On the contrary, our approach is automatic and does not require any additional user-supplied information. Furthermore, these approaches focus on optimizing the scheduling of multiple available tasks, assuming that several parallel tasks are available. Our system is designed to optimize the execution of a single task and can therefore optimize also programs with a single task.

Other works have investigated the problem of automatic task partitioning. Luk et al. [105] introduced an adaptive mapping approach based on a regression model. Their system considers every first run of a program as a training run that can then be used to determine the computation-to-processor mapping for the same program with a new input problem. This approach expects that a program is trained once and then used many times afterward. In contrast to our work, they only show results of one target architecture equipped with only one CPU and one GPU.

A similar approach was adopted by Kai et al. [84]. They proposed a holistic energy management framework for heterogeneous architectures which dynamically splits and distributes the workload over GPU and CPU based on the observed performance. Their algorithm dynamically adjusts the task partitioning based on the run-time difference between devices. Our approach, on the other hand, does not require any profiling or training runs of the program to be optimized. We can derive an optimized task partitioning during the first run of a new program by using a previously, off-line trained model.

Hong et al. [72] proposed MapCG, a framework that supports source code level portability between CPU and GPU. By incorporating a MapReduce programming model, a program can be compiled and executed on either CPUs or GPUs without modification. However, they observed that CPU/GPU combinations did not yield significant performance improvement for the 8 test cases they examined. In contrast to this work, as already described in Section 5.4.1, on our target architectures, we observed the important role of the hybrid task partitioning to achieve the best performance for our test cases.

Grewe et al. [62] developed a purely static task partitioning approach based on predictive modeling and program features. Starting from a multi-device OpenCL code, the authors predict the partitioning of a task with a machine learning model based on static features analysis for fixed problem sizes. Our work uses a similar machine learning approach, but combines static program features detected at compile time with dynamic features collected at run-time that allow the adaptation of the task partitioning to different problem sizes. We test our approach for different target architectures emphasizing the importance of the problem size and the hardware configuration for the tuning of the task partitioning. Furthermore, our system is not limited to a CPU-GPU configuration but can handle an arbitrary number of heterogeneous devices in a single compute node. In another work by Grewe et al. [63] the impact of GPU contention was investigated. The work demonstrated how SVMs and a combination of program and contention features can be used to predict the task partitioning for heterogeneous systems in a scenario where multiple programs compete for the GPU. While such a scenario was not in the scope of our work, their experiments only included a machine with a CPU and a single, integrated GPU.

The authors of [45] extended the framework OmpSs [26] to handle multiple OpenCL devices and concurrently execute tasks among them. They show the efficiency of their approach with a static and a work-stealing scheduling approach. While their approach reaches significant speedups when distributing OpenCL kernel functions among multiple OpenCL devices, their system relies on the availability of multiple, independent OpenCL kernel functions which can be executed concurrently. The approach presented in this work on the other hand, is able to distribute a single kernel function over multiple OpenCL devices.

The framework presented in [57] is able to automatically partition OpenCL kernel functions and distribute them among a CPU and a single GPU, using a machine learning model to predict the partitioning. The authors identified control-flow divergence as an important feature, which is combined with the same static code features as used in [62]. Their experiments demonstrate the benefits of using the control-flow divergence as a feature for their model. While achieving good results, their experiments are limited to a single system using a CPU and a single GPU and do not take into account the effect of varying input sizes.

5.6 Summary

In this chapter we proposed a novel approach which can automatically distribute OpenCL programs on heterogeneous CPU-GPU systems. It consists of a source-to-source compiler, which translates a single-device OpenCL program into a multi-device OpenCL program and a runtime system which distributes the workload over all heterogeneous resources using a machine learning based, off-line generated prediction model.

Our measurements demonstrate that the optimal task partitioning does depend on the program, the target architecture, and the problem size. To accommodate this observation, we use two classes of features: static program features, whose values can be extracted from the source code at compile time, and problem size dependent run-time features, whose values are collected during program execution.

We compared different machine learning techniques, showing that ANNs can reach a higher overall performance, while SVMs can be trained much faster and are less sensitive with respect to their intrinsic parameters. We observed, that the importance of features varies between different platforms. We also demonstrated that PCA applied to the static program features increases the models' accuracy while reducing its run-time overhead.

To demonstrate the portability of our system, all tests were performed on two different target architectures. On average, over those target architectures, the Insieme framework reached up to 87.5% of the optimal performance across 23 programs. Our approach outperforms the default strategies of using only the CPU or only the GPU, which achieve 65.5% and 62.5% of the optimal performance, respectively. In addition, we outperform a random heterogeneous scheduler which delivers only 49.5% of the optimal performance.

Chapter 6

Automatic Data Layout Optimizations for GPUs

This chapter studies the effect of the data layout on the performance of GPUs. Since memory optimizations have become increasingly important in order to fully exploit the computational power of modern GPUs it presents an automatic approach to find an efficient data layout for a given application/GPU pair. This approach is based on hardware features of the GPU, the data structures used by the application as well as decision trees. This tree is built a-priori for each GPU using performance data gathered from a set of predefined training kernels. The results presented in this chapter originate from a collaboration with Dr. Biagio Cosenza from TU Berlin. My contributions are the development of the memory distance metric as well as the decision tree for the structure splitting. I was also responsible for carrying out the needed experiments. The research presented in this chapter has been published in [93].

With the advent of new massively parallel architectures such as GPUs, data layout optimizations have become increasingly important; modern processing units can take 200 clocks to access the DRAM, while a floating point multiply may take only four clock cycles [14]. Because of this *Memory wall*, many research projects focus on memory optimizations. In order to exploit the properties of the memory hierarchy, a key aspect is to maximize the reuse of data.

In this context, **data layout transformation** represents a very interesting class of optimizations. Two typical examples are: organizing data with similar access patterns in structures or rearranging array of structures (AoS) as structure of arrays (SoA). Recent work extends the classical SoA layout by introducing AoSoA (Array of Structure of Array) [175], also called ASA [148]. In this work we prefer the term **tiled-AoS**, but all approaches exploit the same idea: mixing AoS and SoA in a unique data layout. Our work considers tiled layout as intermediate representation between AoS and SoA: if the tile-size is 1, we have an AoS layout; if the tile-size is N, where N is the total number of elements in the array, then we have a SoA layout. Figure 6.1 shows an example with three different variable declarations, one for each AoS, SoA, and tiled-AoS.

In this work, we investigate an automatic memory optimization method which can be easily ported to different GPU architectures, using OpenCL as programming model. We combine two different optimization strategies: we try to group together data fields with similar data access patterns and

```
struct AoS{
  float a;
  float b;
};
AoS aos[N];
```

```
struct SoA{
  float a[N];
  float b[N];
};
SoA soa;
```

```
struct TiledAoS{
  float a[T];
  float b[T];
};
TiledAoS tAoS[N/T];
```

Figure 6.1: Declaration of variables containing two arrays of N elements with different memory layout.

find the best data layout for each of these clusters.

Considering SAMPO [94] as an example, using a struct containing twelve fields. The number of possible ways to partition these twelve fields is equal to $4,213,597$. Considering that this program has minimum run-time of 65 seconds on an AMD FirePro S9000, depending on the data layout, just evaluating all the possible partitions (i.e. clusters) would take more than eight years.

At the same time, for each field cluster with at least two elements (a single-field cluster is naively in SoA layout) we can apply different data layouts, including AoS, SoA, and tiled-AoS with different tile-sizes (in our experiments we considered up to twelve possible tile-sizes).

The exploration of the whole search space, including both fields' clustering and data tiling (i.e. finding the best data layout for each of these clusters) would take more than 400 years.

Figure 6.2 shows a subset of the optimization space for SAMPO. The heat-map on top depicts all possible data tiling for the one-cluster grouping of all the twelve data fields. For this partition, the un-tiled AoS layout is slow (blue); by increasing the data tile-size the run-time decreases (shown in red), and with data tile-size bigger than $12K$ it also outperforms the SoA layout. The lower heat-map shows the performance results while applying the specific data tiling suggested by our algorithm (Section 6.1.1). The fastest version of the shown optimization sub-space is achieved when we use a tile-size of 16 for the smaller struct containing two fields, 24 for the bigger struct with six fields, and having the other fields in a SoA layout. This example program also shows that the best tile-size can be different within the same code and different clusters: when using only one cluster, the highest performance is achieved with large data tiles; however, different clustering delivers better performance with smaller data tiling sizes. This suggests that the optimal data tile-size highly depends on the size of the individual cluster.

Our work is the first approach which automatically tackles the two problems mentioned above. Our contributions are:

- A Kernel Data Layout Graph (*KDLG*) model extracted from an input OpenCL kernel; each vertex weight represents a structure field's size and the edge weight expresses intra-data field memory distance.
- A two-phase algorithm: first, a *KDLG* partitioning algorithm – driven by a device-dependent graph model – splits the original graph into partitions with similar data access patterns; second, for each partition we exploit a data layout selection method – driven by a device-dependent layout calculation – selects the most suitable layout from AoS, SoA and tiled-AoS layouts.
- An evaluation of five OpenCL applications on three GPUs (AMD FirePro S9000, NVIDIA GeForce GTX 480, NVIDIA Tesla k20m) showing a speedup of up to 2.83.

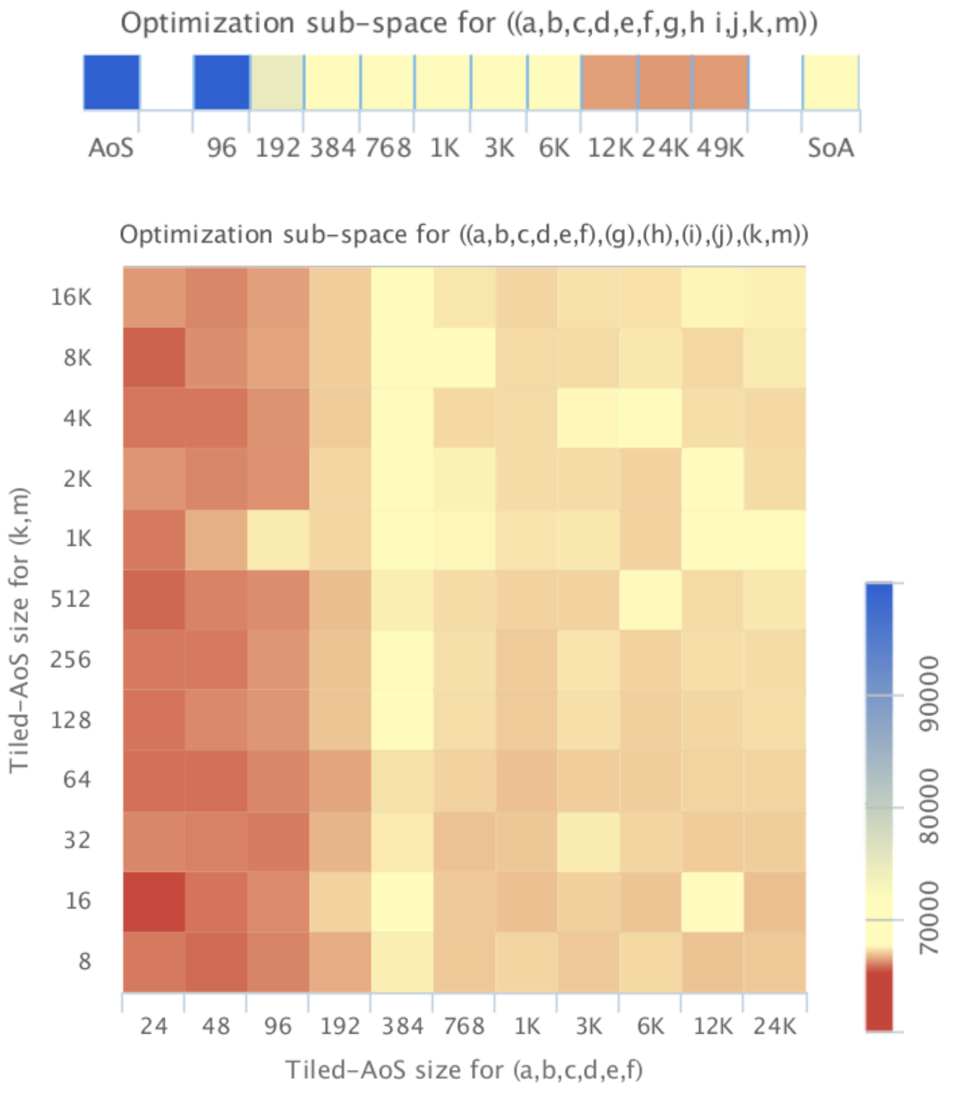


Figure 6.2: Excerpt of SAMPO's Optimization Space. Execution times vary from 65 seconds (in red) to 104 seconds (in blue).

6.1 Method

Our approach tries to answer two questions: (1) What is the best way to group data fields? (2) For each field cluster, what is the best data layout?

We determine a performance efficient way to group data fields, such that all fields of a cluster have a similar *memory behavior*, hence they access memory in a similar way. Once clusters have been identified, for each cluster we try to find the best possible layout within that cluster. Our model supports AoS, SoA, as well as tiled-AoS with different tile-sizes.

In the next section we introduce a novel graph based model, where we encode data layout, field's size and field locality information. The presented two-step approach (1) identifies field partitions (i.e. clusters of fields) with high locality within intra-partition fields and (2) determines an efficient data layout for each partition.

6.1.1 Kernel Data Layout Graph Model

We define a Kernel Data Layout Graph (*KDLG*) as an undirected, complete graph whose nodes represent fields of the input *struct* (assumed to have AoS layout). The *KDLG* has two labeling functions: σ for vertices, representing the field's data size; δ for edges, representing the memory distance (or inverse-affinity) between fields. Formally, a *KDLG* is a quadruple defined as follows:

$$KDLG = (F, E, \sigma, \delta)$$

where F is the set of all fields of the struct, which corresponds to the set of nodes in the *KDLG*. $E = F^2 \setminus \{(x,x)|x \in F\}$ is the set of all edges $e = \{(f_1, f_2)|f_1, f_2 \in F\}$. The mapping function $\sigma : F \to \mathbb{N}$ returns the size of a field f in bytes, e.g. if f refers to a field of type *int*, then $\sigma(f) = 4$, according to the OpenCL specifications. $\delta : E \to \{\mathbb{N} \cup \infty\}$ returns the weight of an edge e. The mapping function $\delta((f_1, f_2))$ is defined as the *memory distance* between the two fields f_1 and f_2 by counting the number of unique memory locations, in bytes, touched by the program between the instruction where they are accessed.

Figure 6.3 shows how a kernel code is converted to a *KDLG*. The edge labels δ represent the *memory distance* between two fields by counting the number of unique memory locations, in bytes, touched by the program between the instructions where they are accessed. We borrow the idea of memory distance from [133] and extend it with the actual data type size, which is important to distinguish between different memory behaviors.

The *KDLG* is based on an OpenCL kernel. The set F will have a vertex for each field defined in the structure, which is passed as an argument to the device kernel function. For each vertex f, the σ function returns the actual type's size in bytes of the corresponding field of f; e.g. according to the OpenCL specifications, if f refers to a field of type *int*, then $\sigma(f) = 4$.

Figure 6.3b displays the *KDLG* generated from the code shown in Figure 6.3a: The fields a and b are always accessed consecutively, therefore $\delta(a, b)$ is 4 bytes. c is accessed after the for loop with 32 iterations, therefore $\delta(c, b) = 252$ and $\delta(c, a) = 256$ bytes, resulting from the 32 iterations that access $2 \cdot 4$ bytes in each iteration. d is never accessed in this kernel, therefore its distance from other fields is ∞.

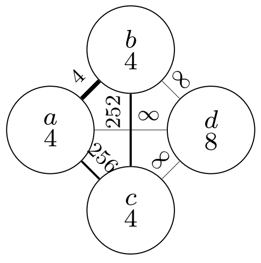
Our graph based model unrolls all loops before starting the analysis. Therefore, it assumes that loop bounds are known at compile time. If not known, we use an OpenCL kernel specific loop size inference heuristic to have a good approximation (see Section 6.1.1). Our analysis focuses on global memory operations, as they are considerably slower than local and private memory operations

Let $MI(f)$ define the set of all global memory instructions (loads and stores) involving the data field f. Our distance function δ between two fields f_1 and f_2 is defined by taking into account the maximum-memory-distance path between the accessing instructions $i_1 \in MI(f_1)$ and $i_2 \in MI(f_2)$.

In order to calculate δ, we use a data flow analysis where each node of the control flow graph (CFG) consists of a single instruction. The function $\sigma(i)$ returns the number of bytes which are written to/read from the global memory in instruction i. We define IN and OUT as

```
struct T {
    float a, b, c;
    double d;
};
__kernel fun(__global T *t) {
    float a, b, c;
    double d;
    int id = get_global_id(0);
    double sum = 0;
    for(int i=id; i<id+32; i++)
        sum += t[i].a * t[i].b;
    t[id].c = sum;
};
```

(a) Kernel code



(b) Generated *KDLG*

Figure 6.3: A *KDLG* generated by a sample input data layout and kernel. Darker edges show fields that are closer in memory (smaller δ).

$$IN_i[j] = \min_{x \in pred(j)} (OUT_i[x])$$

$$OUT_i[j] = \begin{cases} 0 & \text{if } i = j \\ IN_i[j] + \sigma(j) & \text{if } i \neq j \end{cases}$$

In that way, in branches in our code, we only consider the branch which leads to the minimum memory distance. We define an instruction-memory distance function $MD(i_1, i_2)$ as

$$MD(i_1, i_2) = \max(OUT_{i_1}[i_2], OUT_{i_2}[i_1])$$

so that $MD(i_1, i_2) = MD(i_2, i_1)$. We calculate $\delta(f_1, f_2)$, the memory distance between the fields f_1 and f_2, as the maximum memory distance between all instructions in $MI(f_1)$ and $MI(f_2)$ as follows:

$$\delta(f_1, f_2) = \max\left(\max_{i \in MI(f_1), j \in MI(f_2)} MD(i, j) \right)$$

Therefore, we can use $\delta(f_1, f_2)$ to assign a weight to each edge $(f_1, f_2) \in E$. We conservatively use the maximum, which leads to higher weights on the *KDLG*'s edges and leads to more clusters; since more clusters have a lower risk of performance loss on our target architectures. Our approach conservatively assumes that all iterations of a for loop are executed and loop bounds are approximated with constants, as discussed in Section 6.1.1.

KDLG Partitioning

The first step of our algorithm identifies which fields in the input data structure should be grouped together. Formally, we assume that a field partitioning C of the *KDLG* (i.e. field clusters) is *good* if $\forall e \in C | \delta(e) < \epsilon$, where ϵ is a device dependent threshold. We define ϵ as the L1 cache line size of the

Algorithm 6.2 Algorithm to partition the *KDLG* based on the relation between its nodes.

```
1: function KDLG-Partitioning($F, E, \delta, \epsilon$)
2:     $C \leftarrow \varnothing$
3:     for all $f \in F$ do
4:         $C \leftarrow C \cup \{\{f\}\}$
5:     end for
6:     $E_\epsilon \leftarrow \{e \in E : \delta(e) < \epsilon\}$
7:     for all edge $(f_1, f_2) \in E_\epsilon$ do
8:         $c_1 \leftarrow \{x \in C | f_1 \in x\}$
9:         $c_2 \leftarrow \{x \in C | f_2 \in x\}$
10:        if  then$c_1 \neq c_2$
11:            $C \leftarrow (C \setminus \{c_2, c_1\}) \cup \{c_1 \cup c_2\}$
12:        end if
13:    end for
14:    return $C$
15: end function
```

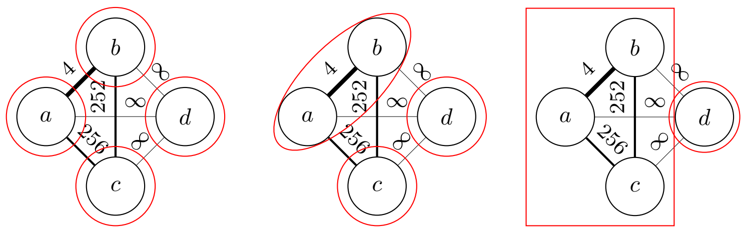
individual GPUs. The values of ϵ are listed in Table 6.1. We use this value as it is the smallest entity that can be loaded from the L1 cache and therefore should be loaded at once.

We propose a strategy based on Kruskal's Minimum-weight Spanning Tree (MST) algorithm [97] that extends the classical MST algorithm with an ϵ-based early termination criteria and multiple clusters of nodes (i.e. struct fields).

As Algorithm 6.2 shows, it takes as input a *KDLG*, previously computed from an input kernel, and a threshold ϵ. It starts by creating a partitioning with $|F|$ sets, each of which contains one field in F (lines 2–5). Line 6 initializes E_ϵ for all edges in E with a weight smaller then ϵ, according to the weighting function δ. The for-loop in lines 7–13 checks, for each edge (f_1, f_2), whether the endpoints f_1 and f_2 belong to the same set. If they do, then the edge is discarded. Otherwise, the two sets are merged in line 11. The complexity of this algorithm is $\mathcal{O}(|E| \cdot |F|)$. Figure 6.4 shows three possible output partitions that can be generated from the graph seen in Figure 6.3b using different ϵ values. Our graph clustering differs from [133] in several aspects: our partitioning algorithm is driven by a hardware-dependent ϵ value; we do not explicitly consider false sharing because our target hardware does not implement a cache coherent protocol; we encode only kernel-dependent field distances in the graph.

Loop Bounds Approximation

When generating the test data to select ϵ we use loops with a fixed number of iterations, to accurately understand the memory distance between two memory accesses. In real world codes, the actual number of iterations is often not known at compile time. Therefore we use a heuristic that is specifically designed for OpenCL kernel codes. If the number of loop iterations are determined by compile-time constants, we use the actual number of iterations. If not, we apply a heuristic to approximate the number of iterations: When a loop performs one iteration for each OpenCL work item of the work group, we estimate it has 256 iterations, as the work group size is usually in this range. When a loop



(a) $\epsilon = 1$ (b) $\epsilon = 100$ (c) $\epsilon = 10000$

Figure 6.4: Different output partitions using different ϵ values on a *KDLG*.

Hardware	ϵ	Decision Tree
AMD FirePro S9000	64	≤ 20 > 48 ≤ 32 32 AoS 16384 ≤ 12 512 1024
NVIDIA GeForce GTX 480	128	≤ 12 ≤ 96 SoA 512 ≤ 8 SoA AoS
NVIDIA Tesla K20m	128	≤ 48 ≤ 96 32768 16384 ≤ 20 8192 SoA

Table 6.1: Properties determined using our algorithms

performs one iteration for each work item of the NDRange, we assume it will have $1 \cdot 10^6$ iterations. If the number of iterations is neither constant nor linked to the work group size or NDRange, we estimate it to have $512 \cdot 10^3$ iterations. The estimation of loop bounds is not very sensitive: we only need to distinguish short loops, which may not completely flush the L1 cache, from long ones.

6.1.2 Per-Cluster Layout Selection

After KDLG-Partitioning was executed, we assume that each field in the same cluster has similar memory behavior. Therefore, all the fields within a cluster should have the same data layout arrangement, e.g. tiled-AoS with a specific tile-size.

To understand what layout is best for a given cluster, we generate different kernels corresponding to a simple one-cluster *KDLG* where δ is roughly the same for each pair of fields. The kernels consist of a single for-loop with a constant number of iterations n. The value of n comprises all powers of two from 128 to 16384. We generated kernels with different combinations of loop size n, number of structure fields m, and tile-size ts. With those kernels we evaluated the performance behavior on various devices.

Algorithm 6.3 Algorithm to optimize the data layout of a struct in two steps: First splitting it into several cluster and then selecting the optimal tile-size for each cluster.

```
1: function LayoutOptimize($F, E, \delta, \epsilon$)
2:     $L \leftarrow \varnothing$
3:     $C \leftarrow$ KDLG-Partitioning($F, E, \delta, \epsilon$)
4:     for all $c \in C$ do
5:         $t \leftarrow$ Select-Tilesize($\sigma(c)$)
6:         $L \leftarrow L \cup \{(c, t)\}$
7:     end for
8:     return $L$
9: end function
```

From the aforementioned performance measurements we derive a device-dependent function Select-Tilesize($\sigma(c)$) which returns the suggested layout for a cluster c, where $\sigma(c) = \sum_{f \in c} \sigma(f)$ and $\sigma(f)$ returns the size of the field f in bytes. Select-Tilesize is implemented using a decision tree, constructed by the C5.0 algorithm [137]. $\sigma(c)$ is the only attribute the decision tree depends on. The potential target classes are AoS , SoA and all powers of two from 2^1 to 2^{15}. The results of those benchmarks are used to generate the training data.For each kernel we create a training pattern for the fastest tile-size as well as all other tile-sizes that are less than 1% slower than the fastest one. These training patterns consist only of the size of the structure $\sigma(c)$, which is the only feature while the used tile-size acts as the target value. Generating training patterns not only for the fastest tile-size but for all which achieve at least 99% of it, as well as several training patterns for different structures with the same size, may lead to contradicting training patterns. However, our experiments demonstrated that the resulting decision tree is more accurate and less prone to overfitting. C5.0 was used with default settings; its run-time was about 1ms, depending on the input.

6.1.3 Final Algorithm

In order to achieve best results, we combine the two algorithms described in Section 6.1.1 and Section 6.1.2. Before applying those algorithms, one has to identify the device dependent factor ϵ and construct a decision tree to be used in function Select-Tilesize, as described in the previous sections. These steps are part of the installation process of our framework. At compile time the *KDLG* graph is constructed and the actual memory layout for the program to be optimized is selected. The selection of the memory layout is described by the pseudo code in Algorithm 6.3. Line 3 calls the KDLG-Partitioning algorithm and returns a set of clusters C in which the corresponding structure should be split. Then the decision tree determines an efficient tiling factor for each of these clusters and stores the resulting pair (cluster, tile-size) (line 4-7).

6.2 Experimental Results

To verify the validity of our approach we implemented a prototype of our framework and observed its performance on several OpenCL applications. The deployment of our system is split into two parts: A device dependent part which has to be performed once for each GPU (installation time), and a

	AMD FirePro S9000	NVIDIA GeForce GTX 480	NVIDIA Tesla k20m
Frequency (MHz)	900	1401	706
Compute Unit	28	15	13
# Parallel Ops	1792	480	2496
FLOPS (SP)	3225	1345	3524
Memory (GB)	6	1.5	5
Memory BW (GB/s)	264	177	208

Table 6.2: Used hardware

program dependent part, which is executed at the compile time of the program. These two parts are depicted in Figure 6.5. The device dependent part consists of identifying the L1 cache line size to be used as ϵ and running a set of training programs to collect the information needed to build the decision tree as defined in Section 6.1.2. Collecting all the necessary data requires to run many benchmarks. Running them takes 196, 158 and 299 minutes on the three evaluated GPUs (described in Table 6.2) FirePro, GeForce and Tesla, respectively. The program dependent part constructs a *KDLG* graph for the structure to be optimized in the corresponding program. This graph is hardware independent and can therefore be shared among different GPUs. By combining the *KDLG* graph with the hardware depended parameter ϵ, we split the struct into several clusters (Section 6.1.1). For each of these clusters we query the hardware dependent decision tree to obtain the tile-size to be used (Section 6.1.2). The resulting program can then be executed on the GPU.

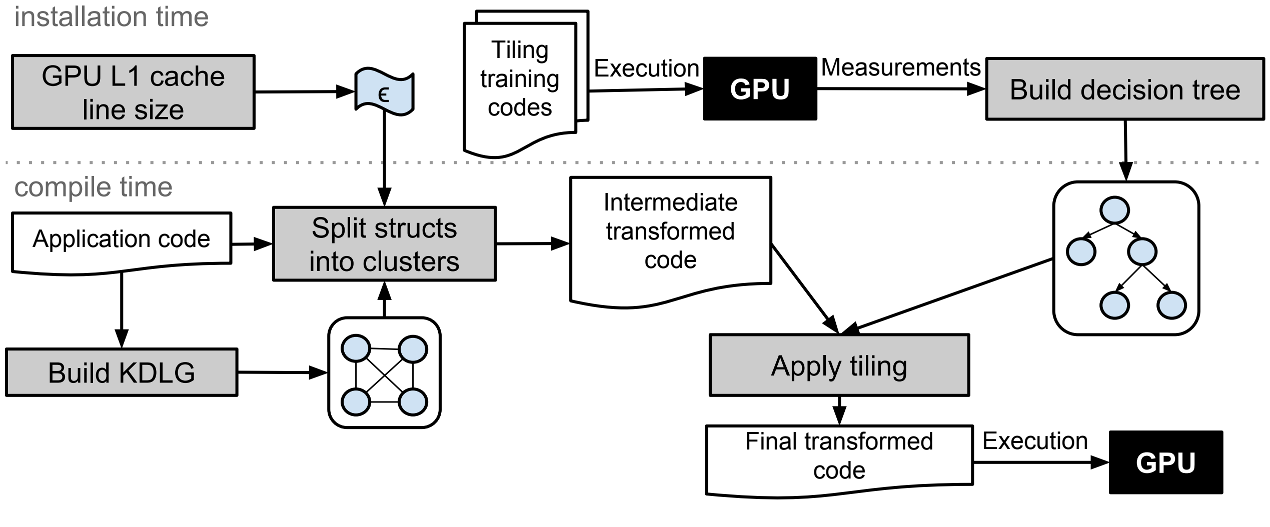


Figure 6.5: Work-flow of our data layout optimization process

To evaluate our framework we run different programs on three different GPUs. Details about the used hardware are listed in Table 6.2 while a short description of the test programs can be found in

Table 6.3. In each program, we focus on the structure with the most instances and try to optimize its layout. The performance of the optimized program is compared to the one using the AoS data layout, which acts as the baseline. The result of our framework on five tested programs is shown in Table 6.3. The data layouts proposed by our system always reach at least the performance of the original implementation using the AoS data layout. In most cases we are able to achieve a considerable speedup (up to 2.83). In the following paragraphs we give more details about three example test cases. For all charts we use AoS as a baseline and report the speedup of four transformed versions: SoA, the version generated after applying the *KDLG* algorithm and splitting the structure if applicable, a tiled AoS version were we use the tile-size proposed by our hardware dependent decision tree-based algorithm, and the final result of our framework as described in Section 6.1.3 which results of first applying the *KDLG* algorithm and then tiling the resulting structures using our decision tree-based algorithm.

	struct size		affected	loop bound	speedup over AoS		
Test codes	bytes	fields	kernels	approx.[1]	FirePro	GeForce	Tesla
N-body	32	2	1	n	1.01	1.06	1.01
BlackScholes [21]	28	7	1	–	1.00	1.43	2.83
Bitonic sorting	16	4	1	u	1.47	1.50	1.38
LavaMD [135]	36	3	1	c,u	2.22	1.89	2.07
SAMPO	48	12	9	w,u	2.19	1.59	1.96

[1] Used Loop bound approximations Section 6.1.1: loop over all work items in the NDRange (n), over the work group size (g), with constant boundaries (c), with unknown boundaries (u).

Table 6.3: Test programs

N-body The first test case that we used to evaluate our framework is N-body, which performs a direct summation of the forces of all particles on every other particle, resulting in a computational complexity of $\mathcal{O}(n^2)$. This implementation contains two structures of the same type, which are switched in each iteration (aka. double buffering). As this exchange is done by simply switching the references, those two structures must have the same type after the transformation of the data layout. Therefore, both instances of this struct are treated equally in our analysis. The struct in the used implementation consists of two fields with a size of 16 bytes each. The edge on the *KDLG* between them is labeled with $1 \cdot 10^6 \cdot 4$ bytes. Considering the ϵ of the used hardware listed in Table 6.1 we can observe, that those fields have a big memory distance and the *KDLG*-based algorithm will split those fields into separate structs. Therefore, the result after applying the *KDLG*-based optimization is the same as when using the SoA layout. Furthermore, applying our tiling algorithm after the *KDLG*-based algorithm has no effect. The speedup achieved is shown in Figure 6.6. It clearly shows that the tiled version of the program is not only slower than the one in SoA data layout, but also slower than the AoS implementation which we use as baseline. This applies to all tile-sizes, not only to the tile-size selected by our decision tree-based algorithm. However, since our framework uses a combination of two layout optimizations, it still correctly selects SoA, which is the data layout with the highest performance for this program on all tested GPUs.

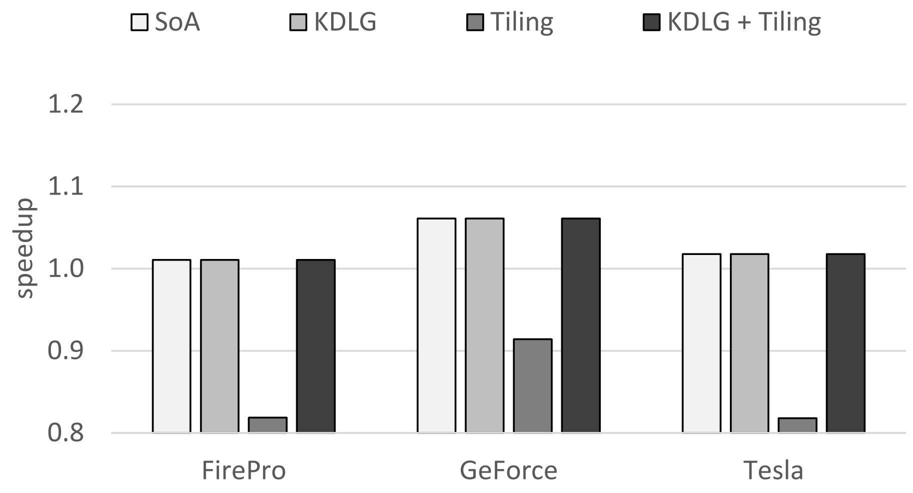


Figure 6.6: Speedup over AoS implementation on N-body using different data layouts.

Bitonic Sort Bitonic Sort [19] is a sorting algorithm optimized for massively parallel hardware such as GPUs. The implementation that we are using sorts a struct of four elements, where the first element acts as the sorting-key. As all elements are always moved together, the *KDLG*-based algorithm results in one single cluster for any $\epsilon \geq 4$. As this is the case on all GPUs that we evaluated, the version generated by the *KDLG*-based algorithm is the same as AoS. The decision-tree-based tiling algorithm converts this layout into a tiled-AoS layout with a tile-size of 512 bytes for the FirePro and GeForce while it suggests to use SoA on the Tesla. The results can be found in Figure 6.7. It clearly shows that, although the *KDLG*-based algorithm fails to gain any improvement over AoS, the decision-tree-based algorithm as well as the combination of both algorithm exceeds the performance of the AoS based implementation by a factor of 1.38 to 1.5. Furthermore, it delivers performance that is comparable or superior than the one achieved by a SoA implementation.

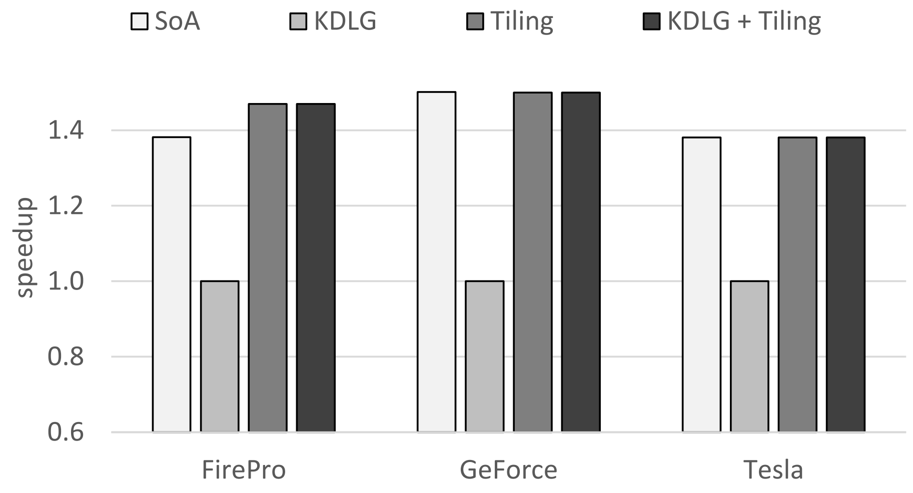


Figure 6.7: Speedup over AoS implementation on Bitonic sorting using different data layouts.

SAMPO SAMPO (described in more detail in Section 8.1) is an agent-based mosquito point model in OpenCL, which is designed to simulate large populations of mosquitoes in order to better understand how vector-borne illnesses (e.g. malaria) may spread. The version available online is already manually optimized for AMD GPUs. Therefore, we ported this version to a pure AoS layout, where each agent is represented by a single struct with twelve fields. We use the AoS version as a baseline for our transformations and compare the result of our framework with the result obtained by the manually optimized version as well as a SoA version. The measurements are displayed in Figure 6.8. The results clearly show, that SoA yields a much better performance than AoS on all tested GPUs, on NVIDIA GPUs it is even better than the manually optimized version. Applying the *KDLG*-based algorithm already results in a speedup between 1.54 and 2.18 on the three tested GPUs, which is within ±10% of the SoA version, depending on the hardware. Applying tiling to the AoS implementation shows good results on the NVIDIA GPUs. Also the AMD GPU benefits from tiling, but it does not reach the performance of the SoA version or the version optimized with the *KDLG*-based algorithm. Applying tiling to the latter version further increases the performance on all evaluated GPUs and leads to a speedup over the AoS version of 2.19, 1.59 and 1.96 on the FirePro, GeForce and Tesla, respectively. This means that the complete version of our algorithm (using both *KDLG*-based algorithm and data tiling) outperforms any other version we tested. Even the manually optimized implementation is outperformed by 7%, 10% and 18% on the FirePro, GeForce and Tesla, respectively.

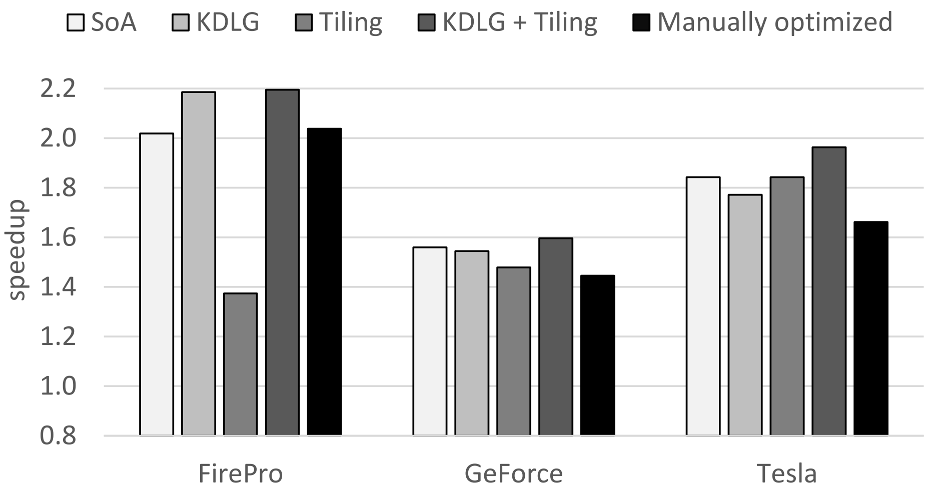


Figure 6.8: Speedup over AoS implementation on SAMPO using different data layouts.

6.3 Discussion

The results clearly show, that programs with AoS data layout are not well suited for GPUs. SoA delivers a much higher performance on all GPU/program combinations we tested. However, also SoA fails to achieve the maximum performance in some cases. We observed that a tiled-AoS can achieve results that are usually equal or better compared to the ones achieved with an SoA layout. But as indicated by our observations on N-body, tiled-AoS is not suited for all programs. Similarly, splitting structures in several smaller ones based on a *KDLG* is beneficial for most programs. However, also this technique fails to improve the performance of some applications, like for example Bitonic Sort. Therefore, combining these two algorithms leads to much better overall results than each of them can achieve individually. This is underlined by the results of SAMPO, where the combination of both algorithms leads to a memory layout that does not only outperform the results of each algorithm applied individually, but also leads to higher performance than obtained by both, a SoA layout and the manually optimized layout.

6.4 Related Work

The problem of finding an optimal layout is not only NP-hard, but also hard to approximate [136]. Such problem has been largely studied in the past [74, 34, 179], often taking into account the target parallel architecture [79, 121]. Raman et al. [133] introduced a graph based model to optimize structure layout for multi-threaded programs. They developed a semi-automatic tool which produces layout transformations optimized for both false sharing and data locality. Our work uses a different graph based model encoding the variables memory distance and data structure size, in order to provide a completely automatic approach; we also support AoS, SoA and tiled-AoS layouts.

Kendermi et al. [85] introduced an inter-procedural optimization framework using both loop optimizations and data layout transformation; our method does not apply to a single function only, but can span over multiple functions in a program and multiple kernels in an OpenCL program.

Data layout transformations such as SoA conversion have been described to be the core optimization techniques for scaling to massively threaded systems such as GPUs [146].

DL presented data layout transformations for heterogeneous computing [150]; DL supports AoS, SoA and ASTA and implements and automatic data marshaling framework to easily change data layout arrangements. Our work supports similar data layouts, but we provide an automatic approach for the layout selection.

Strzodka [148] developed a library that implements multi-valued containers on a multi-GPU system, supporting a wide range of data layouts, including some hybrid AoS/SoA layouts: AoS, SoA, ASA (Array of Structures of Array), and hybrid layouts such as A{ASA}, S{ASA} and {ASA}{ASA}.

Other source-to-source compiler approaches [138] introduced a user specified and automatic data layout selection supporting both SoA and AoS. Their model includes a use graph, an affinity graph and a mathematical model based on affinity index and cache-use factor.

MATOG [175] introduces a DSL-like, library-based approach which optimizes GPU codes using either static and empirical profiling to adjust parameters or to change the kernel implementation. MATOG supports AoS, SoA and AoSoA with 32 threads (to match the warp size on CUDA) on multi-dimensional layouts and builds an application-dependent decision tree to select the best layout.

Dymaxion [32] is an API that allows programmers to optimize memory mapping on heterogeneous platforms. It extends NVIDIA's CUDA API with a data index transformation and a latency hiding mechanism based on CUDA stream. Dymaxion C++ [31] further extends prior work with a clean abstraction backed by a source-to-source code translator, and providing support to map data structures to texture and constant memory. However, it does not relieve the programmer from selecting a good data layout.

The authors of [178] extend the Open64 compiler to enable data layout transformations. They show how structure accesses can be automatically translated into code that can be vectorized on

SIMD architectures. The transformed code shows an average improvement in computation time of 49.5%, the work only targets CPUs.

Tseng et al. [165] proposes a system that allows programmers to write host code in AoS while the data layout is automatically converted to SoA when being transferred to the device. The authors show a performance improvement of up to 177%, even though they do not consider any hybrid data layouts as it is done in our work.

6.5 Summary

We presented a system to automatically determine an improved data layout for OpenCL programs written in AoS layout. Our framework consists of two separate algorithms: The first one constructs a *KDLG*, which is used to split a given struct into several clusters based on the hardware dependent parameter ϵ. The second algorithm constructs a decision tree using to the data gathered from a second benchmark suite. This decision tree is used to determine the tile-size for a certain structure when converting it from AoS to tiled-AoS or SoA layouts.

We found out that the combination of both algorithms is crucial, as using only one of them often leads to no performance improvement over the input code in AoS layout. The layouts proposed by our framework result in speedups of up to 2.22, 1.89 and 2.83 on an AMD FirePro S9000, NVIDIA GeForce GTX 480 and NVIDIA Tesla k20m, respectively and are also able to outperform a manually optimized data layout on a program using a data structure with many fields.

Chapter 7

A Region-Aware Multi-Objective Auto-Tuner for Parallel Programs

This chapter presents a framework for automatic tuning of programs parallelized with OpenMP using iterative compilation and general differential evolution. The framework is implemented within the Insieme compiler and runtime system, described in Section 3.3. The experiments in this chapter show how the Insieme infrastructure can be used to optimize programs for three objectives: wall time, energy consumption and resource usage. The framework extends a the framework published in [83] and was a collaboration with Juan J. Durillo and Philipp Gschwandtner. I was responsible for improving and fine-tuning the auto-tuning method as well as executing and evaluating the experiments. The scientific outcome of the effort made for this chapter has been submitted to the International Conference on Supercomputing (ICS 2017) and is currently under review.

Auto-tuning is increasingly being used to optimize the non-functional parameters for programs. The typically large search space requires sophisticated techniques in order to find well performing parameter values in a reasonable amount of time. Optimizing parallel programs becomes increasingly difficult with rising complexity of modern parallel architectures, as described in Section 2.1. Automatic software tuning, or simply *auto-tuning* [114], arose as an attempt to better exploit hardware features by automatically tuning applications. An auto-tuner tries to find promising *configurations* for a given program. A configuration consists of a set of non-functional parameters with a corresponding value range that can influence a program's performance by transforming the source code of the application, or by finding the right parameter values that govern the execution of an application on a given architecture (e.g. number of threads, frequency per core, etc.).

This chapter describes a novel auto-tuning approach for programs with multiple single-entry single-exit code regions whose non-functional behavior depends on at least one tunable parameter. We assume that we can measure the non-functional behavior of these regions for the optimization objectives (e.g. wall time, energy consumption, etc.). Tuning multi-region applications exposes additional challenges for auto-tuning techniques. Firstly, the same set of parameter values is usually not optimal for all regions of the program. However, setting the parameter values individually for every region leads to a huge search space as it grows exponentially with the number of tuning opportunities, i.e. the number of regions. Secondly, the execution of a region may be influenced by the parameter values applied to neighboring regions. Previous work [102] observed that the optimal parameter values for individual regions of hybrid MPI/OpenMP applications led to sub-optimal overall performance.

Auto-tuning techniques are widely used [176, 169, 129, 53, 151, 10, 9, 35, 11] but are often limited to using the same parameter values for every region, i.e. globally for the entire program, ignoring the fact that different parts of the code may benefit from specific parameter values.

Our auto-tuner can optimize a generic number of objectives which do not necessarily correlate with each other. This lack of correlation makes it impossible to find a single solution which is best in every objective. For example, if some regions are optimized for one objective while others are optimized for different objectives, it is very likely that the overall performance of the program will suffer. It can happen that a configuration exposes low execution time at the cost of high resource usage for one region, but the contrary for another region, resulting in sub-optimal overall performance.

The approach proposed in this chapter extends the method presented in [83], which is limited to optimizing each region in isolation, by adding region-aware auto-tuning support for the entire program. The contributions of this chapter are the following:

- A region-aware multi-objective auto-tuner.
- A compiler-runtime system that automatically identifies regions and enables automatic tuning of their parameters.
- Evaluation of several global and region-aware auto-tuning strategies for several codes on different target architectures which demonstrates the importance of region-aware auto-tuning compared against global optimization.

7.1 Motivation

In this section, we motivate the need for region-aware auto-tuners by using a simple example program consisting of two regions. Both regions perform a parallel matrix multiplication. In the first region, only the outermost loop is parallelized; in the second, we parallelize only the innermost loop. As a consequence we have two regions with different execution behavior: the first one scales well with the number of threads and the second one does not. In order to have similar execution times for both regions, the matrix size in the second region is only one quarter of the matrices in the first region.

The experiments in this section are performed on the Ivy Bridge-EP architecture described in Section 7.5. Our goal is to find the optimal configuration for executing this program. For the sake of performing an exhaustive search of all possible program configurations, we assume that these regions only expose the number of threads as a tunable parameter.

We select configurations for that program using three different approaches:

- Isolated: This approach optimizes both regions in isolation.
- Global: This approach is constrained to find a single set of parameter values to be used in both regions.
- Region-Aware: This approach optimizes both regions using individual parameter values for each of them to maximize the program's overall performance.

The best configurations found by these approaches are summarized in Table 7.1. The first two rows of the table show the number of threads for each region chosen by the approach in the corresponding column. The following two rows include the execution time for each of these regions with the indicated number of threads. The fifth row shows the program's execution time when the regions are executed with the number of threads shown in the first two rows. Finally, the last row shows the relative difference regarding the best found execution time across all three approaches.

The Isolated approach gives the slowest execution times. The reason is that it will run the first region with a large number of threads, as it does scale well. However, this has a negative effect on the second region, which does not scale well. The change from 20 to only 2 threads between the regions introduces a significant overhead by the underlying runtime system, as it also implies a change of the number of sockets where computations are performed on.

The Global approach yields better results since the code regions are not optimized in isolation but regarding the overall program performance. However, as this approach is limited to the same parameter values for both regions, it is unable to exploit the full potential of the hardware. This drawback is overcome by the Region-Aware approach. It does not only take into account eventual performance penalties from varying configurations between the two regions, it can also customize the parameter values for each region individually to comply with their hardware requirements.

	Isolated	Global	Region-Aware
#Threads Region 1	20	10	10
#Threads Region 2	2	10	7
Region 1[1]	546	1075	1075
Region 2[1]	5798	2652	2366
Total[1]	6344	3727	3442
Relative time difference	1.84	1.08	1.00

[1] Execution time in milliseconds.

Table 7.1: Theoretical potential of different auto-tuning approaches, determined by exhaustive search for an example program with two regions.

7.2 Multi-objective Tuning of Multi-Region Programs

In this section we firstly introduce some background related to multi-objective auto-tuning of programs. Afterwards, we state the main challenges when tuning multi-region programs.

7.2.1 Background on Multi-Objective Auto-Tuning

Auto-tuners may optimize several objectives which sometimes conflict with each other. This means that optimizing one of them is only possible by worsening the value of at least one of the other objectives. The mathematical solution to such problems is not defined by a single point, but by a set of points representing a trade-off between these objectives. The set of solutions representing the optimal trade-off between the considered objectives is known as *Pareto set*.

Our approach to apply multi-objective software auto-tuning consists in computing the Pareto set or an approximation of it [38]. Often, related work reduces multi-objective optimization problems to a single objective one by using fixed weights for the individual objectives. Therefore, they try to find a single solution which is near optimal in a pre-defined set of preferences for the objectives. Computing the Pareto set instead of a single solution is often incorrectly cited by related work with the belief that it implies a manual selection of a solution by the user which is just one possibility. Different alternatives comprise a selection based on preferences specified a posteriori, i.e. after the Pareto set has been computed. The latter approach does not require manual interaction of the user, and has the advantage that the computed solution belongs to the optimal trade-off. While computing the Pareto set may seem to require a higher computational effort than computing a single solution, literature in multi-objective optimization shows that this is not necessarily the case [69]. Indeed, the opposite is often true: computing the whole Pareto set may be easier than computing some individual solutions within it, as finding solutions that show a good compromise between two or more objectives implies a different way of navigating the search space.

7.2.2 Challenges in Tuning Multi-Region Programs

We focus on programs composed of multiple regions and tune each of these regions with an individual set of parameter values. A region's performance may depend on the way other regions are executed, what data they access, and other side effects. The non-functional parameters of a region are not always independent of the parameter values of other regions. This issue implies that regions should not be tuned in isolation, which has been observed in [102].

Tuning multi-region applications introduces additional challenges. Firstly, the search space of possible configurations of a program grows exponentially with the number of regions. For example, the *matrix multiplication* kernel considered in [83] consists of a single region. That region requires to tune three tiling dimensions and the number of threads. For a problem size of 14000, i.e. matrices of 14000×14000 elements, and a machine with 32 cores, the search space of possible program executions is $700^3 \times 32 \simeq 10^{10}$. If a program consists of two regions similar to that one, the search space would be $(700^{\times}32)^2 \simeq 10^{20}$. Larger search spaces often reduce effectiveness of search methods [43]. Secondly, in a multi-objective multi-region scenario, it is crucial that the parameter values for the different regions within a program aim the optimization of the same objective. Otherwise, if two regions are assigned parameter values optimizing different objectives, most likely the execution of both regions together will not be optimal for any of these objectives. For example, this is the case when half of the regions within a program would be executed with optimal parameter values for a given objective and the other half of the regions with parameter values optimal for a conflicting objective.

Our goal is to design an auto-tuner that can find a single Pareto set of configurations for a given program with multiple regions. While the parameter values of every region are tuned separately, we measure the effect of changing the parameter values of a region regarding the entire program instead of considering the effect only for individual region executions. In this way, we optimize the whole program execution instead of focusing on specific regions. After the Pareto set for the whole program is computed, a single configuration for the entire program can be selected from the Pareto set, either manually or automatically. This approach differs from the one proposed in [83], which is based on computing an individual Pareto set for every single region in isolation, making this approach prone to the performance penalties described in Section 7.7. Furthermore, computing a Pareto set independently for every region requires a decision making process for every single region. Therefore, the approach presented in [83] is unfeasible for tuning programs with a large number of regions. A feasible way to compare the approach in Jordan et al. [83] to the one presented in this chapter is using the same set of parameters for every region of the program, thereby reducing the search space and producing only a single Pareto set for the entire program. Additionally, the auto-tuner can evaluate the performance of a configuration for all regions at once which makes it aware of eventual performance penalties caused by region interferences. In Section 7.5, we compare this version of the auto-tuner presented in [83], which we call RS-GDE3 Global, against the new version introduced in this chapter.

7.2.3 Method

Our approach extends the RS-GDE3 algorithm presented in [83], which is based on iterative compilation. It uses a fixed size set of different program configurations to be executed on the target architecture in order to determine their performance. RS-GDE3 refers to this set as *population.* Iterative compilation methods update this set across different iterations by generating possibly better performing configurations for the program being tuned. In the case of RS-GDE3, this is done by generating a new population called *offspring population* from the current population as explained later in this section. At the end of every iteration, the current population is updated by considering its content and the content of the offspring population. Details about how configurations are chosen to be part of the population for the next iteration can be found [83].

In order to tune multi-region programs regarding multiple objectives, we need to overcome the following problems:

1. All the parameter values within a configuration should aim for a common goal. If the tuner generates a program where a region r_1 uses parameter values optimized for a given objective, and for a subsequent region r_2 it uses the best parameter values regarding another objective, then the execution of both regions will unlikely be optimal for any of these objectives nor will it represent an optimal trade-off.

2. An intractable large search space, which may reduce the effectiveness of the search performed by RS-GDE3.

3. Existing or changing parameter settings of a predecessor regions that may negatively impact successor regions can cause additional overheads.

4. Well performing sets of parameter values for individual regions may be discarded by the tuner if they are considered in combination with poorly performing parameter values for other regions.

To solve the first problem, our approach does not consider regions in isolation. Instead, our configurations are comprised of the parameter values of all regions and will be kept as part of the population if they contribute to optimize the whole program.

To overcome the second problem, finding a good starting point in this huge search space is crucial to improve the effectiveness of the tuner. For this reason, we perform a *global pre-tuning* phase. We restrict this pre-tuning to use the same set of parameter values for every region within the program. The idea is to reduce the size of the search space, making it easier for tuner to find the best configurations within the limited search space. We use these configurations as the starting point of a second tuning phase where every region can have a different set of parameter values. The global pre-tuning takes place during a few iterations at the beginning of our method. Besides reducing the dimensionality of the search space, the pre-tuning phase also helps to overcome the third problem, since the found configurations avoid overheads caused by changing hardware settings between regions. During the second tuning phase, these overheads may occur due to changing hardware settings. However the auto-tuner will discard these configurations, unless the benefit of the different parameter values for each region outweighs the overheads caused by using different parameter values for individual regions.

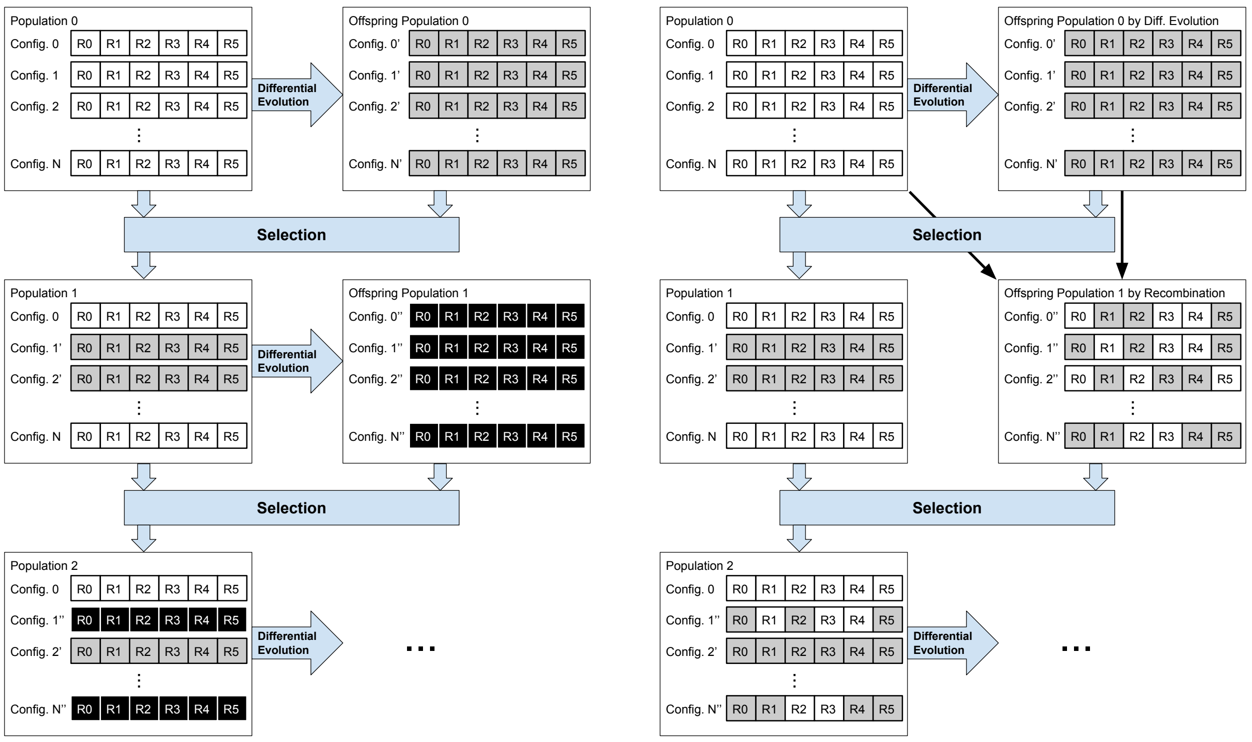
To deal with the problem related to the fourth issue, we developed a novel approach which we call *recombination*. RS-GDE3 generates new configurations by applying an operator called *differential evolution* [145]. For each configuration within the population (we will refer to it as the parent configuration), this operator generates a new program configuration. The new configuration is generated by first calculating the differences between two configurations, randomly selected from the population. The resulting values are then divided by two and added to the parameter values of a third, also randomly selected, configuration. These newly created parameter values replace the values in the parent configuration with a probability of 50% to create a new configuration. This newly created configuration inherits parts of its parameter values form the parent configuration while the other parameter values are generated based on the rest of the population. While this may be beneficial, it also represents a drawback. For example, if the parent configuration k_1 contains the best possible parameter values q_1 for region r_1 and the newly generated configuration k_2 contains the best possible parameter values q_2 for region r_2, it would make sense to compose a third configuration k_3 that uses q_1 in region r_1 and q_2 in region r_2. However this is not possible in RS-GDE3 due to the way the differential evolution operator works. The recombination strategy solves this problem by generating new configurations using a different method every second iteration. This new generation method takes the parent configuration and the newly generated one and swaps several of their parameter values. In particular, the best set of parameter values for individual regions out of the parent and the newly generated configurations are preserved. This approach maximizes the chances of combining parameter values that perform well.

A comparison of a traditional, region-aware auto-tuner to the novel approach using recombination is shown in Figure 7.1. The former, depicted in Figure 7.1a, generates a new offspring population out of the current population using differential evolution in every iteration and selects the best performing configurations from the current population and the offspring population to form the population for the next step, as it is described in [83]. The latter, depicted in Figure 7.1b, uses the differential evolution only in every second iteration. In the other iterations, a new offspring population is generated by recombining the parameter values of the configurations in the current population and the offspring population. The selection of configurations that will form the population of the next step is unaltered compared to the traditional approach. In order to maximize the benefit of the recombination steps, we are using two different kinds of recombination, which are applied in an alternating fashion:

1. The first kind compares each configuration that has been generated during the last iteration of the optimizer with its parent's configuration. Out of those two configurations (offspring and parent), for each region it selects the set of parameter values which give the better values averaged over all objectives and combines them to a new configuration for the entire program.

2. The second kind selects one configuration for each objective from the offspring population as well as their corresponding parents' configurations. From the parameter values of those configurations, it constructs a new configuration for each objective, combining the sets of parameter values of the regions which deliver the best performance for the corresponding objective.

Finally, the pseudo-code of the whole method is given in Algorithm 7.4.



(a) Standard region-aware auto-tuner. In each iteration an offspring population of N elements is generated out of the current population.

(b) Auto-tuner using recombination. In every second iteration, an offspring population is created by recombining the best performing settings for each individual region from the previous population and offspring population.

Figure 7.1: Examples for the evolution of two region-aware auto-tuners over two iterations, using a population of n configurations, tuning a program with six regions (R0 to R5). Both auto-tuners perform $2{\times}n$ evaluations in this example. In each iteration, a new population is created by selecting the best configurations from the population of the previous iteration and the corresponding offspring population.

7.3 Implementation

The presented framework is implemented within the Insieme compiler and runtime system presented in [83], based on branch *inspire_1.3* which is freely available at [2]. The Insieme compiler performs,

Algorithm 7.4 Multi-region auto-tuner using a combination of general evolution and recombination.

```
 1: Generate a population $Pop_{pre}$ of configurations, where regions within the same configuration have
    the same parameter values
 2: set the iteration counter to zero
 3: while iteration counter $\leq$ threshold do
 4:     Generate new offspring population $OPop_{pre}$ using the differential evolution method
 5:     Update $Pop_{pre}$ with the best trade-off solutions from $Pop_{pre} \cup OPop_{pre}$
 6:     Increase the iteration counter
 7: end while
 8: Generate a new population $Pop$ of configurations using solutions in $Pop_{pre}$, replicating the global
    set of parameter values for each region.
 9: while iteration counter $\leq$ maximum number of iterations do
10:     if iteration counter is even then
11:         generate an offspring population $OPop$ using the differential evolution method
12:     else
13:         generate an offspring $OPop$ population using the recombination method
14:     end if
15:     Increase the iteration counter
16:     Update P with the best trade-off solutions from $Pop \cup OPop$
17: end while
18: Return the non dominated solutions (i.e. the Pareto set) of $P$
```

among other tasks, code analysis and code transformations. Insieme uses INSPIRE [81] as intermediate representation for extracting individual regions and applying the required code transformations. The transformed INSPIRE code is then converted back to C. The resulting source code is compiled to binary using the Gnu Gcc Compiler. In the case of transformations such as tiling, the Insieme compiler generates a different version for each tile size to be evaluated in order to obtain the best performance. This means, that a separate version of the code has to be generated and compiled for each configuration that needs to be evaluated.

The Insieme runtime system executes the transformed input code and measures its performance. The measurements are reported to the Insieme compiler which provides them to the auto-tuner. The Insieme runtime system allows to set the number of threads individually for each region and is also responsible for mapping the executed program to OS-level threads. For each OS-level thread used, one *worker* is created and started [159]. The number of workers created for a specific execution is equal to the maximum number of threads used by any region of the given program. Each worker is fixed to a specific CPU core in ascending order. When a region is executed with fewer threads than the number of workers started, the additional workers are set to sleep by the runtime system.

7.3.1 Regions

Our auto-tuner targets parallel programs implemented in C using OpenMP [123] for parallelization. The programs are subdivided into several regions. We define each parallel OpenMP for-loop to be a separate region for several reasons: Besides being parallel, these loops usually contain most of a program's computational work. Furthermore, the implicit synchronization following each parallel OpenMP for-loop is well suited as a point to vary the number of threads, while the number of threads cannot be changed within the body of an OpenMP for-loop. Additionally, the restrictions enforced by OpenMP on parallel for-loops, such as no continue, break or return statements as well as no modification of the iterator variable inside the loop's body, increase the probability that a loop nest starting with a parallel OpenMP for-loop is suitable for tiling.

Loop nests that can be tiled are of special interest to us, as they typically have high optimization potential and consume most resources. Our auto-tuner examines whether a loop nest is suitable for tiling by using the Polyhedral Model [49] integrated in the Insieme Compiler. This analysis determines whether a loop nest is tilable and also provides information about the minimum and maximum tile size for this transformation.

For every region the auto-tuner can tune the number of threads that are used to execute it. Furthermore, for regions which are tilable, the tile size in each dimension is tuned.

7.4 Testing Methodology

To compare different approaches, we use the same objectives as in [83]. For the result K of every auto-tuning run we calculate $|K|$ and $V(K)$. K corresponds to the resulting Pareto set of the auto-tuner while $|K|$ is the number of elements in the Pareto set K. A larger number of elements of the Pareto set is considered superior, as it offers more flexibility to choose a desired solution. $V(K)$ defines the normalized size of the hypervolume covered by the performance measurements of the elements in the Pareto set K [182], i.e. the relative size of a hypervolume formed by all points dominated by the points in K in a normalized hyperrectangle defined by the highest and lowest measurement in each objective. When the function $V(K)$ is used to compare several solutions S_i where $i \in [0, n]$ and $n \in \mathbb{N}^+$, this hyperrectangle is defined by the highest respectively lowest measurement in each objective of the combined Pareto set of all solutions S_i. This means, if all configurations in S_i are dominated by configurations found in S_j with $j \in [0, n]$ and $j \neq i$, the coverage $V(S_i)$ is 0. A configuration k_0 *dominates* another configuration k_1 if k_0 delivers better performance than k_1 in every objective. As the coverage is calculated on a normalized hypervolume, the result of $V(K)$ ranges from 0 to 1 where 1 corresponds to an ideal solution covering the entire hypervolume, i.e. dominating all other solutions. For each approach we report the average population size $\overline{|K|}$ and hypervolume $\overline{V(K)}$ over 14 runs.

The huge search space of the tested programs prevents a comparison of the results of an auto-tuner to the theoretical optimum, as the only way to find the theoretical optimum would imply an exhaustive search over the entire search space.

7.5 Experimental Results

This section presents the performance that we obtained with our approach on some exemplary test cases. The experiments are executed on two different machines. The first machine features two Intel Xeon E5-2690 v2 CPUs in a dual socket layout which are based on the Ivy Bridge-EP architecture with 20 cores in total. The second machine is equipped with four Intel Xeon E5-4650 CPUs, based on the Sandy Bridge-EP architecture and featuring a total of 32 cores. Hyperthreading has been disabled on both architectures. The clock frequency was fixed to the highest base frequency throughout all experiments.

We present the results obtained by different auto-tuners including random, global tuners and region-aware ones. The complete list is:

- Random: It randomly generates 3000 configurations with individual settings for each region.
- RS-GDE3 Global: It uses the RS-GDE3 tuner introduced in [83] to determine values for all tunable parameters. This version resembles is a version of the auto-tuner presented in [83] as described in Section 7.2.2. Every region within the entire program uses the same set of parameter values. Solutions are generated with the differential evolution operator described in[83].
- RS-GDE3 Region: Region-aware version of RS-GDE3 Global, which sets the parameter values for every region individually.
- RS-GDE3 Region GPT: Extends the RS-GDE3 Region using a global pre-tuning phase as described in Section 7.2. A total number of ten iterations are devoted to this phase.
- RS-GDE3 Recombination: Based on RS-GDE3 Region, but new configurations are generated using the recombination method described in Section 7.2 in every second iteration.
- RS-GDE3 Recombination GPT: RS-GDE3 Recombination with a global pre-tuning phase that uses 10 iterations.

All compared RS-GDE3 auto-tuners variations use a population size of 30 and perform 100 iterations, leading to a total of about 3000 executions of the program (as performed also by the Random tuner). Our experiments did not show any significant performance improvements by enlarging the population any further with that budget of iterations.

The effectiveness of these approaches are evaluated on three benchmarks. Two of them, mg and bt, are taken from the NAS parallel benchmarks [115] C/OpenMP implementation by the Omni group [5]. Bt is a block tri-diagonal solver for nonlinear partial differential equations while mg approximates the solution to a three-dimensional discrete Poisson equation using a multi-grid method. For bt we choose problem size w, for mg the problem size b, in order to get reasonable execution times for auto-tuning. The heated-plate benchmark [28] is a stencil-code solving the steady heat equation on a two dimensional, rectangular plate. The matrix size used for this benchmark was set to 384×384 elements. The total number of regions, the number of regions to which tiling can be applied as well as the total number of tunable parameters for each of those programs are listed in Table 7.2. This Table also indicates the search space size when tuning those programs. The number of tunable parameters is the sum of the number of tiling dimensions of all regions plus the number of regions, as we can set the number of threads separately for each region. The search space is the product of the ranges for each of those parameters. Therefore, the size of the search space depends on the target architecture, as a higher number of cores also provides more tuning possibilities. For each tuned program, we include the number of elements in the Pareto set $\overline{|K|}$ of the six auto-tuners is shown in Table 7.3 while the corresponding hypervolumes $\overline{V(K)}$ can be found in Table 7.4. The computed Pareto set of most auto-tuners contains 30 elements, i.e. the entire population. The RS-GDE3 Recombination

GPT auto-tuner delivers the best result in terms of hypervolume in most cases. Only in heated-plate it is slightly outperformed by RS-GDE3 Recombination auto-tuner on the Sandy Bridge-EP architecture. However, the hypervolume values of all RS-GDE3 auto-tuners are very close for that benchmark, most likely due to its relatively small search space. The global pre-tuning phase yields a higher improvement on the Sandy Bridge-EP architecture than on the Ivy Bridge-EP which can be explained by their differing socket numbers: Whereas the Ivy Bridge-EP system has only two sockets, the Sandy Bridge-EP system has four sockets. Changing the number of threads between regions can imply an additional cache coherency overhead when the regions are executed in different number of sockets. This overhead is a consequence of not having shared cache between sockets. Therefore, it is beneficial to start with a configuration that does not change the number of threads between regions, which is achieved by the global pre-tuning phase.

	mg	heated-plate	bt
#Regions	94	10	122
#Tilable Regions	82	4	114
#Tunable Parameters	268	18	453
Search Space Size[1]	10^{590}	10^{33}	10^{897}
Search Space Size[2]	10^{609}	10^{35}	10^{922}

[1] On Ivy Bridge-EP
[2] On Sandy Bridge-EP

Table 7.2: Search space description for the evaluated programs.

	Ivy Bridge-EP			Sandy Bridge-EP		
	mg	heated-plate	bt	mg	heated-plate	bt
Random	15.4	10.6	26.2	26.4	4.1	12.1
RS-GDE3 Global	29.9	29.2	24.9	27.4	29.9	23.7
RS-GDE3 Region	30.0	29.6	29.7	30.0	29.9	23.0
RS-GDE3 Region GPT	30.0	30.0	30.0	30.0	29.9	30.0
RS-GDE3 Recombination	30.0	30.0	30.0	30.0	29.4	30.0
RS-GDE3 Recombination GPT	30.0	30.0	30.0	30.0	29.6	30.0

Table 7.3: Pareto set size $\overline{|K|}$ of several auto-tuner variants for different benchmarks on two different architectures.

The time required for the auto-tuning is dominated by the time needed to compile and execute the program. Therefore, shorter and faster programs can be tuned in less time. The tuning time for our test cases is shown in Table 7.5. The Random auto-tuner exhibits the longest tuning time, as it typically evaluates the configurations with the lowest performance. The fastest auto-tuner over all test cases is the RS-GDE3 Global, because it never experiences any slowdowns from performance penalties caused by region interferences. From the region-aware auto-tuners, those without a global pre-tuning phase are slower than the auto-tuners with a global pre-tuning phase in most cases. The latter converge faster to a population with reasonably fast configurations, which leads to significantly

	Ivy Bridge-EP			Sandy Bridge-EP		
	mg	heated-plate	bt	mg	heated-plate	bt
Random	0.0705	0.0545	0	0	0	0
RS-GDE3 Global	0.9038	0.6497	0.6431	0.8572	0.7953	0.6184
RS-GDE3 Region	0.3348	0.6464	0	0	0.7645	0
RS-GDE3 Region GPT	0.8998	0.6685	0.6196	0.8571	0.7981	0.6034
RS-GDE3 Recombination	0.8429	0.6585	0.6576	0.0640	0.8138	0.5694
RS-GDE3 Recombination GPT	0.9152	0.6869	0.7120	0.8821	0.8262	0.7137

Table 7.4: Hypervolume $\overline{V(K)}$ of several auto-tuner variants for different benchmarks on two different architectures.

lower execution times of the tuned program. Similarly, the region-aware auto-tuners using the Recombination step are faster than their counterparts using only the traditional differential evolution in most cases, as the average execution time of the resulting program versions is shorter.

	Ivy Bridge-EP			Sandy Bridge-EP		
	mg	heated-plate	bt	mg	heated-plate	bt
Random	21836	26453	24162	31848	34751	38631
RS-GDE3 Global	16810	10488	15584	19643	15488	27344
RS-GDE3 Region	16480	13073	19777	33238	20969	43557
RS-GDE3 Region GPT	20565	11468	17251	21344	18037	28696
RS-GDE3 Recombination	19268	11256	17506	30292	14961	32862
RS-GDE3 Recombination GPT	19576	10809	16597	22113	16598	31125

Table 7.5: Tuning time in seconds of several auto-tuner variants for different benchmarks on two different architectures.

In addition to the Pareto set size $|K|$ and hypervolume $V(K)$ we also report the objective values of the best configuration found by the RS-GDE3 Recombination GPT auto-tuner for the three real world codes compared to the non-optimized (without auto-tuning) versions. To calculate the speedup we use the best configuration from the auto-tuner's Pareto set for each individual objective. Typically, this is a different configuration for every objective. We compare these configurations to two non-optimized configurations that do not apply any tiling: the sequential version, using only one thread and the parallel version using all threads available on the corresponding machine. The results are shown in Table 7.6 which demonstrate a superior performance compared to the non-optimized versions, both sequential and parallel, in every objective. As expected, the largest improvement over the sequential version can be achieved in wall time (up to 9.1 fold) while the parallel version is primarily outplayed in resource usage. Especially on the Sandy Bridge-EP architecture with 32 cores, the non-optimized version suffers from the moderate scalability of heated-plate and bt, allowing our auto-tuner to achieve an improvement factor of up to 61.6 in resources usage. These results clearly demonstrate the benefit of our region-aware multi-objective auto-tuner, even if only a single objective is of interest. This is underlined by the achieved speedup of the auto-tuner over the parallel version, which ranges from 1.3

to 7.6 on the tested architectures and programs. If a well-balanced trade-off solution across several objectives is required, the benefit may be even higher, depending on the user's preferences.

	Ivy Bridge-EP						Sandy Bridge-EP					
	mg		heated-plate		bt		mg		heated-plate		bt	
	s[1]	p[2]	s[1]	p[2]	s[1]	p[2]	s[1]	p[2]	s[1]	p[2]	s[1]	p[2]
wall time	6.7	1.3	9.1	3.8	3.9	3.1	3.6	2.2	7.6	4.2	2.3	7.6
energy	3.0	2.4	7.3	4.4	2.0	3.6	2.2	5.8	4.5	7.3	1.6	10.5
resource usage	1.2	4.8	2.3	19.4	1.3	21.7	1.3	25.0	2.7	47.1	1.3	61.6

[1] Improvement over non-optimized sequential version.
[2] Improvement over non-optimized parallel version.

Table 7.6: Improvement over non-optimized versions i.e. not tiled with a constant number of threads, in each individual metric achieved by the RS-GDE3 Recombination GPT auto-tuner.

7.6 Discussion

The results presented in the previous section demonstrate that the best version of our region-aware auto-tuner, the RS-GDE3 Recombination GPT, outperforms a global auto-tuner based on RS-GDE3 (RS-GDE3 Global).

When comparing the Pareto sets generated by these two tuners, we observe that some configurations in the Pareto set of the RS-GDE3 Global auto-tuner are dominated by others in the Pareto set of the RS-GDE3 Recombination GPT. This means that there are configurations computed by the global tuner which are worse in all the considered objectives than some of the configurations computed by the region-aware tuner.

Next, we analyze the configurations computed by these two algorithms in order understand how region-aware tuners exploit different parameter values in different regions of the same program. To this end, we compare the obtained results on two of the considered applications, the bt and the heated-plate. For these comparisons, we pick a configuration computed by each of these two tuners and observe the parameter values within each region.

For the first comparison, the two configurations are taken from the Pareto set generated by the two auto-tuners on the Sandy Bridge-EP architecture. We refer to the configuration computed by the region-aware tuner as k_r, and to the configuration computed by the global tuner as k_g. The first thing to note is that the tiling values used for different regions in k_r show a high variation. For example, while region 40 is tiled using the values $\{368,13,1\}$, the region 88 uses the tiling parameters $\{1,1,2\}$. The tile sizes used by k_g are $\{1,8,2\}$ for all the regions within the program. All regions in k_g are executed with eight threads, which corresponds to the maximum number of cores on one socket on our Sandy Bridge-EP architecture. Also in k_r, the maximum number of threads used is eight, meaning that some of the regions are also executed using eight cores. However, many regions in k_r are executed with fewer threads, where any number between one and eight is used at least once. This results in the objective values presented Table 7.7. While in terms of wall time and energy consumption both configurations are similar, the resource usage of k_g is 20% higher than for k_r. This means, k_r is as fast and consumes as little energy as k_g using less resources. This is possible because the region-aware tuner found a configuration which executes regions that do scale well up to eight threads with such number of threads; at the same time, regions that do not benefit from being executed on eight cores are executed with fewer threads. Such a degree of adaption is not possible with any auto-tuner that uses the same parameter values for every region in the entire program.

	${k_g}^1$	${k_r}^2$	Improvement
wall time (ms)	2076	2074	1.00
energy consumption (J)	148	147	1.00
resource usage (ms)	16611	13838	1.20

[1] Taken out of the Pareto set generated with the RS-GDE3 Global auto-tuner.
[2] Taken out of the Pareto set generated with the RS-GDE3 Recombination GPT auto-tuner.

Table 7.7: Performance of two individual configurations in bt on the Sandy Bridge-EP architecture.

For the second comparison, we present the performance figures for two configurations for the heated-plate benchmark on the Ivy Bridge-EP architecture. Again we label k_r the configuration found by the region-aware tuner and k_g the configuration found by the global tuner. In this case, for k_g we choose the configuration with the least resources usage from the Pareto set. While k_r is not the configuration leading to the lowest resource usage in its Pareto set, it still dominates k_g. Obviously, k_g uses only one thread for every region in order to minimize the resource usage. The tile sizes used by this configuration are {1,214}. In contrast to that, k_r uses two threads to execute the biggest region of heated-plate, i.e. region 8. The increased number of threads also requires a different tile size in order to perform well; in this case it uses {31,251}. All other regions, which account for more than 1% of the total wall time, use very similar parameter values as the one used in k_g: they are executed using only one thread, and the tile size in the first dimension is equal to 1, while the tile size in the second dimension varies from 233 to 251. This indicates, that a tile size of 1 in the first dimension is beneficial when regions are executed sequentially, while a higher number of threads benefits from larger tile sizes. The combination of a custom tile size and higher number of threads for region 8 of heated-plate results in a significantly lower wall time as shown in Table 7.8. As that region does scale well, k_r has also a slight advantage over k_g in both, resource usage and energy consumption, despite the increased number of cores used. Additionally, as indicated in the table, the rather different parameter values for region 8 cause a 25% drop in wall time, compared to k_g. As in the comparison before, these performance figures can only be achieved using different parameter values for the individual regions of the program.

7.7 Related Work

In the literature we find several frameworks for software auto-tuning, for example self-tuning libraries like ATLAS [176], OSKI [169], SPIRAL [129] or FFTW [53], or other auto-tuning frameworks including Active Harmony [151], Sequoia [48], PetaBricks [10, 9], Patus [35], and OpenTuner [11].

In the past, most auto-tuners focused on the optimization of the wall time of programs. However, recent work shows an inarguable attention to optimize applications regarding several objectives.

	k_g[1]	k_r[2]	Improvement
wall time (ms)	2323	1749	1.33
energy consumption (J)	65	63	1.04
resource usage (ms)	2323	2289	1.02

[1] Taken out of the Pareto set generated with the RS-GDE3 Global auto-tuner.

[2] Taken out of the Pareto set generated with the RS-GDE3 Recombination GPT auto-tuner.

Table 7.8: Performance of two individual configurations in heated-plate on the Ivy Bridge-EP architecture.

Besides wall time, energy consumption entailed by a program's execution is becoming a popular objective [100, 131, 162, 132, 142, 126, 42, 102, 65]. Resource usage [83, 65], compilation time, or the size of the executable binary [73, 107, 55] also received attention in related work. Most of these works fail to capture the trade-off between these objectives and reduce them to a single one. Only a few works focus on computing and analyzing trade-off between several conflicting objectives [52, 83, 16].

All the aforementioned works applied the same tuning options to the whole program. Approaches that individually tune code regions and examine their inter-relationships with respect to single or multiple objectives are rare. A major issue is the definition of code regions for programs. In [98] program functions are considered to be the regions to optimize. In MPI programs regions are often defined as the code between pairs of communication directives in [102]. In [41] regions are obtained from applications Regions within the same cluster are tuned using the same parameters or code transformations. The Periscope tool of the AutoTune project [113] tunes regions regarding any function measuring properties of that function (run-time, energy consumed, etc.). Although different objectives can be tuned, they are not considered simultaneously. Furthermore, Periscope tunes regions individually without considering side effects among regions. In contrast to the framework presented in this work, Periscope does not describe a methodology to identify regions within a program.

Some related work splits programs into several regions but optimize them in isolation [98, 41]. However, the authors of [102] show that regions within a program impact each others execution time behavior. They demonstrate that when regions are executed with their local optima set of parameter values, a non-negligible penalty may be paid as a result of changing hardware settings across adjacent regions. The same work also discusses the benefits of using the same set of parameter values for every region in the entire program versus a per-region tuning approach, and the need for tuning mechanisms that can find configurations aware of interferences between regions.

7.8 Summary

Most existing auto-tuner focus on a global setting of parameter values which are fixed for the entire program, ignoring the optimization potential by customizing parameter values to individual region's peculiarities. In this chapter, we introduced a novel auto-tuning framework that is based on the Insieme source-to-source compiler an runtime system as well as a new RS-GDE3 auto-tuner variation that aims at solving the challenges of multi-region, multi-objective auto-tuning. The challenges introduced by the huge search space, region dependencies and conflicting objectives are tackled by adding a pre-tuning phase to the region-aware auto-tuner which tunes the program using the same parameter values for all regions, and an intermediate evolutionary step for the RS-GDE3 auto-tuner, generating new configurations by recombining the parameter values generated in previous steps.

Experiments have shown that our new approach is more effective in tuning three different programs on two different parallel computers than non-region-aware global auto-tuning. We outperform a non-region-aware RS-GDE3 auto-tuner in hypervolume $V(K)$ by up to 15%. Furthermore, we demonstrated that our approach reaches up to 7.6, 10.5 and 61.6 fold improvements in wall time, energy consumption and resource usage respectively, over the non-optimized parallel version.

Chapter 8

Applications

As my doctorate is part of the Doktoratskolleg Plus Computational Interdisciplinary Modelling (DK+CIM) [166], I had the opportunity to apply my research to two scientific domain applications in two different fields: One in the area of biology and one in the area of astrophysics. The following sections present the results of these works.

8.1 SAMPO: An Agent-based Mosquito Point Model in OpenCL

This section describes an OpenCL implementation of an existing agent-based model, simulating populations of the Anopheles gambiae *mosquito, one of the most important vectors of malaria in Africa. Discussed are the methods and techniques used to overcome the design challenges, which arise when transitioning from an object-oriented program to an efficient OpenCL implementation. In particular, the parallelism inside the program has been maximized, dynamic divergent branching was reduced, and the number of data transfers between the OpenCL host and device has been minimized. Even though our implementation was designed for this specific use case, the approach can be generalized to other contexts, as most agent-based point models would benefit from the same basic design decisions that we took for our implementation. The work arose from a collaboration with the Center for Research Computing at the University of Notre Dame, Indiana, USA. While Gregory Davis provided the knowledge to model the mosquitoes' live cycle using an agent based simulation, it was my responsibility to design and implement a high performance OpenCL version of it. The results of this collaboration are published in [94].*

Malaria is a vector-borne illness caused by parasites that are transmitted from human to human through an intermediate organism, mosquitoes. Because malaria cannot be transmitted between humans in the absence of these vectors, malaria control and eradication strategies have primarily relied on interventions that directly target the vector, such as insecticide-treated nets, larvicides, and indoor residual spraying. To maximize the effectiveness of campaigns involving these types of interventions, knowledge of population-level malaria transmission dynamics must be understood. Formal modeling and simulation of mosquito populations is an attractive technique for researchers and malaria control managers to understand malaria transmission and assess hypothetical strategies for deploying limited intervention resources. Agent-based modeling and simulation (ABMS) is becoming a popular approach used in this field [78, 66, 181] because each member of the population (i.e. each mosquito) is modeled individually with simple rules, typically derived from laboratory and field research, dictating their behavior. By simulating large populations of these agents and their interaction with each other and their environment, system-level properties emerge, which can be used to better understand the dynamics of malaria transmission.

To facilitate the use of ABMS across many fields of interest, a diverse group frameworks and tools have been developed. For example, FLAME (Flexible Large-scale Agent Modeling Environment) [37] is a generic agent-based modeling system where agents are formalized as Stream X-Machines specified in XML markup and compiled into executable code through a template system alleviating the need to develop custom code for common agent-based modeling tasks. Other libraries such as Repast [4] and MASON (Multi-Agent Simulator of Networks) [106] provide common ABMS foundation objects that can be sub-classed and extended to implement model-specific behavior. Beyond these generic frameworks, there are tools specifically for modeling populations of disease vectors such as DTK (Disease Transmission Kernel) [44] and AGiLESim [181].

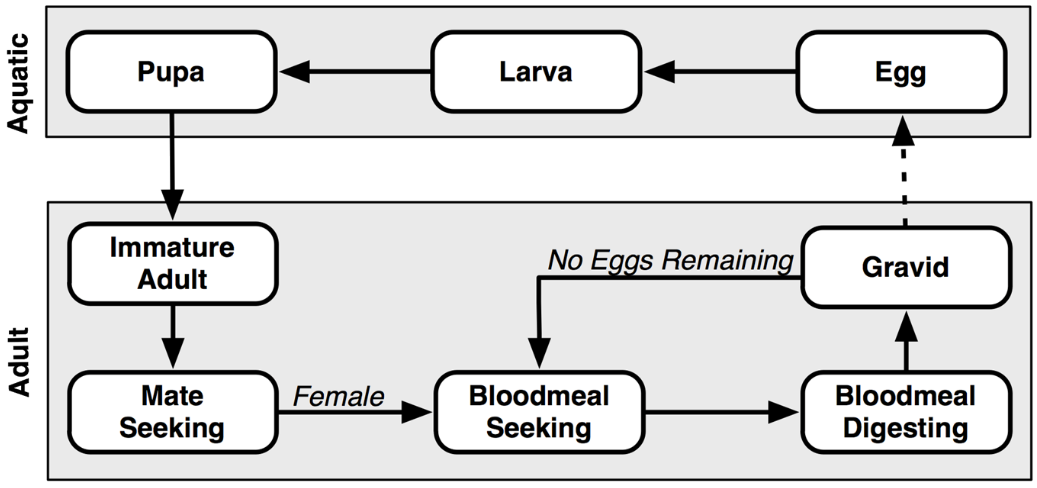


Figure 8.1: Mosquito life cycle

Agent-based simulations can be compute-intensive, especially when there is a need to simulate large numbers of agents to approximate a given population. Furthermore, since the outcomes of these simulations typically depend on stochastic factors, the simulations are often rerun several times to more accurately characterize the model results, extending computing demands even more. Importantly, the agent-based approach is also inherently parallel as the behavior of one agent is not generally directly dependent on the behavior of another. This is reflected in the fact that each of the aforementioned frameworks provide mechanisms for parallelizing the simulations they implement using either a distributed architecture with compute nodes communicating using MPI or taking advantage of multi-core CPUs using OpenMP and/or multithreading. However, none of these frameworks takes advantage of GPU-based computing to automatically scale with the available resources of the machine the simulation is executing on except for FLAME GPU [134]. This extension of FLAME is implemented in CUDA so that it runs on NVIDIA GPUs, but is not compatible with GPUs from other vendors.

In the present work we developed SAMPO (Scalable Agent-based Mosquito POint model), a derivative model of the Java version of AGiLESim, implemented using OpenCL [89] where each mosquito agent follows the life-cycle shown in Figure 8.1. As explained in [181], the model is well-suited for simulating the evolution of mosquito populations. We chose OpenCL because it supports a large variety of processing units as described in Section 3.1. Whereas HPC resources are expensive and not necessarily accessible to users of agent-based simulations, GPUs and multi-core CPUs are widely available even in workstations.

8.1.1 Implementation

Our simulation is implemented in *OpenCL*, using the LibWater library [59] to reduce the implementation effort. It consists of twelve kernel functions which are called in every iteration of the simulation. In general, it is beneficial for OpenCL programs, to have as many independent threads running at a time as possible. In order to achieve that, we create one thread for each agent for most of our kernels. The control flow of our implementation, showing the discrete operations performed during the simulation, can be seen in Figure 8.2.

Although the program may run on many different processing units, it is optimized for GPUs with dedicated DRAM that are connected to the host CPU via PCIe. For such architectures, the connection bandwidth between the host and the device is often a bottleneck. Therefore, this implementation aims at minimizing the communication between host and device. All agents are generated by the device and will never be copied to host memory. The only data that has to be copied to host memory are the snapshots of the environment (containing the number of mosquitoes in each state as well as the number of total and infective bites that occurred) after each iteration. The only data that has to be written to the device are the seeds for the random number generation (see Section 8.1.1 for more details). The following paragraphs describe the most important design decisions that we made during our implementation. For more detailed information, we refer to the source code which is publicly available online [92].

Data Layout

The mapping of data to memory has a big impact on the performance, especially in GPU-based architectures. As demonstrated in Chapter 6, the data layout is crucial in order to gain a high performance on GPUs. This has a number of important implications for design decisions in porting AGiLESim to OpenCL and the most important of these are described below.

Agent Representation Each agent (i.e. mosquito) in our simulation consists of twelve fields, listed in Table 8.1. The most natural way to store the agents of this simulation would be an array of structures where each structure represents one agent. This solution however, leads to a lot of unnecessary load and store operations (when the data of an entire agent is loaded while only few fields of the structure are required) and/or non-coalesced memory accesses [167] (when only a single field of the agent is loaded). Therefore, the transformation of the array of structures into a structure of arrays could be considered. However, since structures containing pointers are not supported in OpenCL, this would lead to a separate array for each field of the agent structure, resulting in a large number of arguments for each kernel. Maintenance and readability of the code would heavily suffer from that transformation. Moreover, if memory accesses occur in a non-coalesced fashion, it can be beneficial to load a single struct instead of multiple scalars to minimize non-coalesced memory accesses.

In order to exploit the advantages of both previously mentioned techniques, we use a hybrid approach for our data layout. The agent structure is split into three smaller ones, called A, B and C in Table 8.1. The layout of the structures is chosen in a way so that each kernel function that has to load one of those structures uses as much of its fields as possible.

Name	Type	Struct
Available Eggs	UINT	A
Gonotrophic Cycle Length	UINT	A
Human Bloodmeal Count	UINT	A
Delay to State Transition	REAL	A
Number of Egg Batches	UINT	A
Ovi Positioning Attempts	UINT	A
Is Dead	BOOL	B
Current State	ENUM	B
Age in Hours	REAL	C
Hours in State	REAL	C
Is Female	BOOL	C
Cumulative Sporgonic Development	REAL	C

Table 8.1: Data structure representing an agent, i.e. one mosquito. The agent's fields are distributed over three different structures, the structure of each field is indicated in column *Struct*.

Agent Storing In order to maximize the performance, all arrays have a fixed size and are allocated at the start of the simulation. For arrays of constant size as well as for arrays whose size depends on the simulation length (i.e. the number of iterations) the size can be calculated accurately at the beginning of the simulation. For arrays whose size depends on the number of active agents, it is more difficult, since the maximum number of agents that will be reached during the simulation is not known at the beginning. In our program, the user has to estimate the maximum number of agents that can be reached during the simulation. This number will then be used to allocate memory for the agents. In our experiments we figured out, that 10× the maximum *carrying capacity* [56] of the environment is a well-suited estimation. The total memory consumption on the OpenCL host and device is shown in Table 8.2.

Since our implementation aims at minimizing the communication between host and device, the agents are generated and stored only on the device. They will never be transferred to the host during the entire simulation. The only data that will be transferred to the host are the statistics that are of interest to the user of the simulation as described in Section 8.1.1.

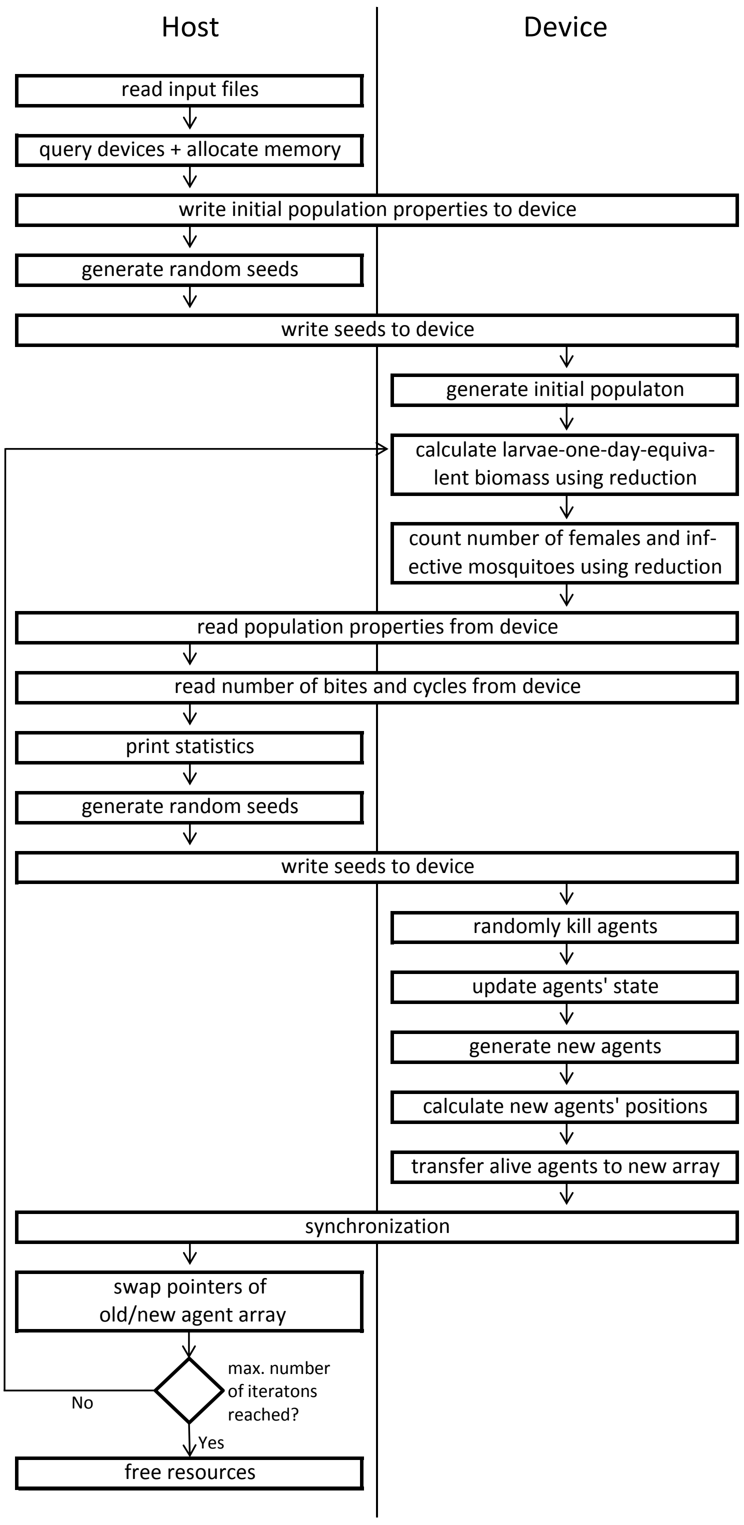

Host
Device
read input files
query devices + allocate memory
write initial population properties to device
generate random seeds
write seeds to device
generate initial populaton
calculate larvae-one-day-equivalent biomass using reduction
count number of females and infective mosquitoes using reduction
read population properties from device
read number of bites and cycles from device
print statistics
generate random seeds
write seeds to device
randomly kill agents
update agents' state
generate new agents
calculate new agents' positions
transfer alive agents to new array
synchronization
swap pointers of old/new agent array
max. number of iteratons reached?
No
Yes
free resources

Figure 8.2: Control flow of SAMPO. All tasks executed on the host are of constant time complexity with regard to the population size. Thus, the execution time of simulations with larger populations is dominated by the tasks executed on the device.

	Size in Bytes
host[1]	$240 + 24 * \#iterations$[2]
device[1]	$240 + 116 * \#agents$[3]
host to device[4]	64
device to host[4]	100

[1] total amount of allocated memory, constant during the entire simulation
[2] number of time-steps in the simulation
[3] maximum number of agents specified as described in Paragraph 8.1.1
[4] data transferred in each iteration

Table 8.2: Memory used and transferred during the simulation

Divergent dynamic branching minimization

In OpenCL, threads are organized in a two-level hierarchy. The entirety of threads is subdivided into work groups, which consist of several work items. The size of the work groups is defined by the `local_work_size`, as described in Section 3.1. To maximize the performance on GPUs, all threads in one work group should follow the same path in the control flow, i.e. should execute the same code. When branches cause threads inside a work group to execute different code, it leads to *divergent dynamic branching*. Since our main target architecture for this program are GPUs, minimizing divergent dynamic branching is crucial to obtain a good performance as described in Chapter 4.

In our implementation, consecutive agents in memory are mapped to consecutive threads. Therefore, agents who are likely to execute the same code should be packed together in memory in order to minimize divergent dynamic branching. To achieve this, our implementation blocks agents that are in the same state together. At the end of each block, padding to the next multiple of the `local_work_size` is added in order to avoid agents belonging to two different states inside a single work group. While this approach is effective in reducing the number of divergent branches to a minimum, it requires a reordering of the agents after each iteration. During this reordering, dead agents are removed from the array, agents that changed their state are moved to the block of their new state and the entire array is packed in order to avoid empty slots within the blocks of states.

Doing this reordering in parallel requires duplicate arrays that hold agent information. In each iteration, all living agents are moved from the currently used array to the other one, doing the previously described reordering at the same time. Duplicating the agent arrays means, that the memory requirement on the device almost doubles. While this is reducing the maximal population size that can be handled by a device, this is the only way we found to make the simulation efficient on modern GPUs. Since this copy is done in parallel with one separate thread for each agent, the copying thread does not know if the surrounding agents have died or advanced to another state. Because there is no efficient way to determine the new index of its agent during the copy, we calculate the new index of each agent before the copy operation using several prefix sums.

The indices are calculated separately for each mosquito state. To do so, we generate a temporal *data array* for each state. It has the same size as the agent array and is filled with 1 on the position of alive agents of the corresponding state and 0 on all other ones. The result of a prefix sum on this array will be the new index for each agent in that state, relative to the beginning of the state's block. Looking at the mosquito development cycle depicted in Figure 8.1, it is obvious, that agents of each state can only be found in a restricted area in the agent array. For example, all agents who are in larval state in the next generation, must be in either egg or larval state in the current iteration. Therefore, the prefix sum can be restricted to that area. To avoid having a separate prefix sum for each state, we use a *segmented prefix sum*.

The segmented prefix sum differs from a "standard" prefix sum as it restarts counting at 0 at the beginning of each segment. In our application we create one segment for each state. Each state's segment is set to the range where agents in that state can occur, as described in the previous paragraph. Those ranges overlap, as shown in Figure 8.3. Most segments span two state blocks, the current one and the previous one in the mosquito development cycle. The segment for the egg state covers only its own block, since there is no previous state. New eggs are simply put at the beginning of the new array. The segment of the blood meal seeking state covers four blocks. Other than mate seeking mosquitoes that develop to this state after finding a mate and blood meal seeking mosquitoes, also gravid mosquitoes go to blood meal seeking state after all their eggs have been laid. The segment also covers blood meal digesting mosquitoes although no agent in this block can become blood meal seeking in the next iteration. However, it is added to the segment, since all segments have to be contiguous in the used prefix sum implementation. Due to the overlapping of the segments, the new index of all agents cannot be calculated using a single segmented prefix sum, but we have to perform three of them. Figure 8.3 elucidates, which sates are handled by each of the three prefix sum.

To calculate the new indices of the agents using the segmented prefix sum, two temporary arrays for each of the three prefix sums have to be created: The first is the *data array* consisting of 1 and 0 as described earlier. The data arrays of the states covered by the same prefix sum can easily be merged, since their segments do not overlap. The second is a *flag array*, determining the single segments. The result of the prefix sum holds the new index of each agent, relative to the start of its state's block. The start of the block can easily be determined by summing up the number of agents in all preceding states, which is equal to the last element of the corresponding state's prefix sum +1. On the GPU we use the parallel prefix sum implementation of Bolt [7]. When the program is executed on a CPU, we use a sequential implementation of the prefix sum, since it delivers higher performance on that kind of processing units. While the sequential implementation of the prefix sum uses only a single kernel, the calculation of the prefix sum in parallel consists of four kernel invocations: one is generating the previously described temporal arrays, while the other three, taken from [7], are performing the actual prefix sums.

Random number generation

Random number generation is a crucial component of many agent-based simulations. Many parameters of our implementation (and similarly in AGiLESim) are influenced by randomness (e.g. mortality rate, delay to develop for some states, number of eggs generated, etc.). This means that many random numbers are needed in each iteration. However, GPUs do not have an available built-in pseudo-random number generator like CPUs, and OpenCL does not provide any means of generating random numbers on the device. Since this implementation aims at minimizing the communication between host and device, a common approach for random number generation involving generating all random numbers on the host CPU and copying all of them to the device is not an option. Therefore,

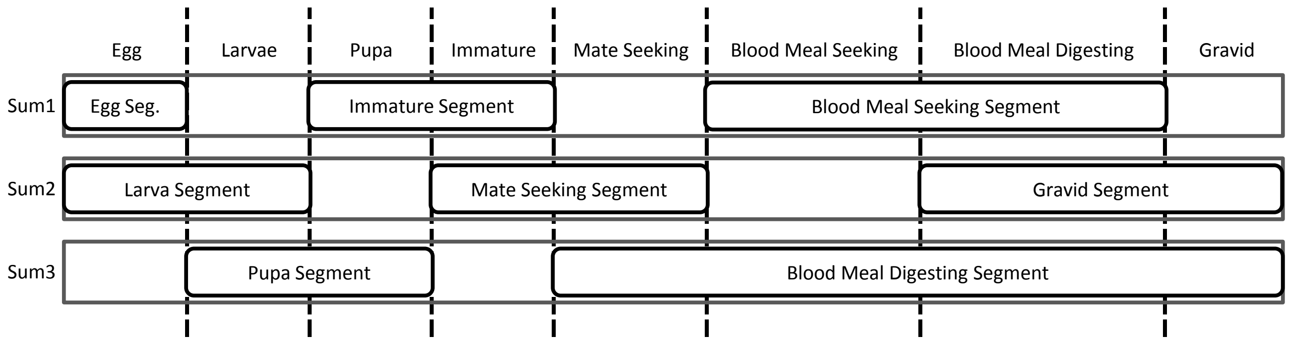


Figure 8.3: Segments for the three prefix sum calculations. Each prefix sum calculations considers all agents from the start of the first segment to the end of the last one.

we use the a hybrid Tausworthe [153, 101]/LCG [128] generator as described in [116]. This algorithm uses four integer random numbers as seeds to generate a single floating point random number. Each of the four seeds is used as an argument for either the Tausworthe or the LCG algorithm with varying parameters. The four resulting floating point numbers are then combined to form a single random number. The main advantage of using four seeds instead of one is the increased period length, equal to approximately 2^{121} for the used implementation.

The hybrid Tausworthe/LCG generator produces uniformly distributed random numbers. However, for some variables in the simulation, we need random numbers following a Gaussian distribution (e.g. the number of eggs generated by a mosquito). To produce normally distributed numbers we use the Box-Muller [23] transform which takes two independent, uniformly distributed random numbers as an input and returns a single random number with a Gaussian distribution. The implementation proposed in [116] produces two normally-distributed random numbers at a time; however, we generate only one because a second random value is not useful at the point of generation.

The approach in [116] also proposes a separate set of seeds for each thread, where the current results of the Tausworthe/LCG steps are used as a seed for the next one. This requires a considerable amount of memory (equal to the number of agents) to store those seeds in addition to one read and one write operation to global memory every time a random number is generated. To avoid this overhead, we use an approach, similar to the *One-PRNG-per-kernel-call-per-thread* approach described in [127]: all threads use the same seed, which is mangled using the global id of the thread. Instead of incorporating the simulation time step in the mangling function, we generate a new set of seeds in every iteration on the host CPU, and transfer them to the device. Doing so, allows us to replace the hashing function to mangle the seed used in [127] with a single addition or multiplication operation on the individual seeds. In comparison to the approach proposed in [116], this approach allows SAMPO to handle larger populations due to the reduced memory overhead. Furthermore, our approach does not require write access to the device memory and all threads read the seeds from the same memory location, which means the read operations on the seed can be cached on most modern GPU architectures.

In our implementation, each thread needs to generate up to four random numbers per iteration. Therefore, the host writes fours sets of seeds to the device in each iteration, which each thread combines with its global id to generate four independent random numbers. Using the previously

	Random Seeds Generation and Transfer (s)
AMD Opteron 6168	0.34
Intel Xeon X5650	0.29
AMD Radeon HD5870	4.59
AMD FirePro S9000	1.70
NVIDIA GeForce GTX 480	0.19
NVIDIA Tesla k20c	0.24

Table 8.3: Time required to generate the random seeds and transfer them to the device.

described method, the number of seeds generated and transferred from the host to the device in each iteration is constant and independent of the number of agents in the simulation. Due to the small amount of data to be transferred, the time needed for the data transfer can be neglected on most the architectures used in our tests as shown in Table 8.3. A considerable overhead could be measured only on the AMD Radeon HD5870; however its impact is constant and does not scale with agent population size.

Simulation Output

The simulation output consists many measures of the state of the simulation at each time step including: number of mosquitoes in each development state, an age-adjusted larval biomass (a measure of how saturated the aquatic environment is - affecting both larval mortality and the number of eggs laid by a given mosquito), the number of female adult mosquitoes and how many of them are infective, the number of mosquito bites that have occurred (both infective and non-infective), the number of gonotrophic cycles[1] that have been completed, and the sum duration for all completed gonotrophic cycles.

To determine the number of agents in each state, the starting and end index of each state-block (as described in Paragraph 8.1.1) is copied to the host. The host will then use those values to calculate the actual numbers.

To calculate the number of bites as well as the number of gonotrophic cycles and the sum of their length we use *atomic operations* [89]. This means that there is a global counter for each of those values which is increased by each thread that handles the corresponding action (e.g. a bite occurred or a mosquito laid all is eggs). Although atomic operations are a potential performance bottleneck, especially on GPUs, in this simulation their effect is negligible, since they are not invoked that frequently. Figure 8.4 shows the impact of atomic operations on the execution time, comparing the run-time of the kernel updating the agent's state, with and without atomic operations. By omitting the atomic operations, some of results generated are no longer accurate, but the condition of the the simulation itself is not affected. The comparison clearly shows that the impact of the atomic operations on the execution time is minimal.

There are three numbers in the statistics which require performing a reduction over a large number

[1]A gonotrophic cycle is completed when a gravid mosquito placed all its eggs and goes back into blood meal seeking state

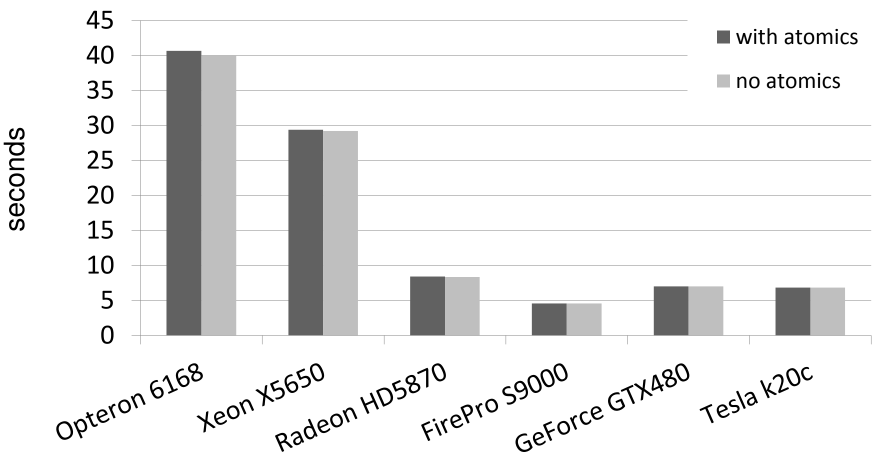


Figure 8.4: Execution time of the agent-state updating kernel of different processing units with and without atomic operations for statistical counters. Times are taken from a one year simulation of a population with maximal 1.4 million agents.

of agents: The age-adjusted larval biomass, which requires a reduction over all larvae, the number of adult female/male, and potentially infective mosquitoes, which require a reduction over all adult mosquitoes. Since these numbers are affected by all agents in the corresponding states, summing then up using atomic operations would not be an efficient solution. Therefore, these numbers are calculated at the beginning of each iteration using a *parallel reduction in local memory*, based on the two-stage implementation in [8]. Since the counting of female and potentially infective mosquitoes is performed on the same data, we use only one parallel reduction to do both of them at once. The parallel reductions are executed in two steps:

1. Reducing the input array to a number of `local_work_size` elements with a kernel function using a total of `local_work_size`^{2} threads.
2. Using `local_work_size` threads to calculate the final result, based on the results of step 1.

This two-step approach is the only feasible way to distribute the reduction to more than `local_work_size` threads since OpenCL does not provide the possibility to synchronize over several work groups.

8.1.2 Correctness

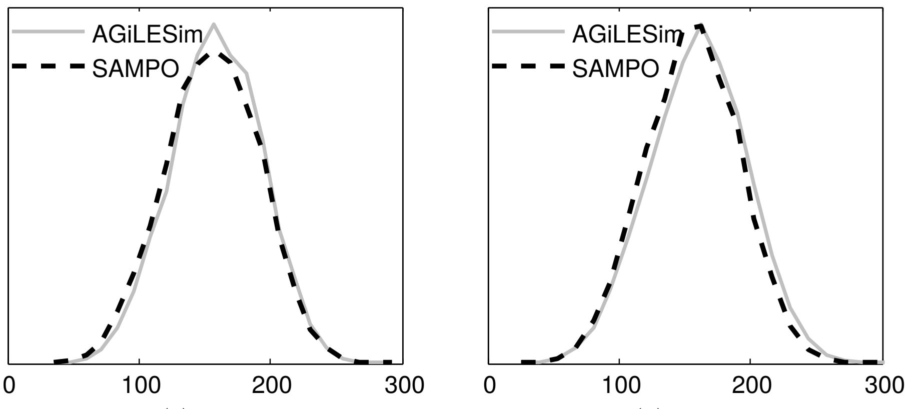
To verify the correctness of our implementation we compared the generation of random numbers and the structure of the agent population – number of adult mosquitoes and their age structure (see [56]) – to these metrics from a similarly configured Java version of AGiLESim.

Random number generator

To verify our approach to generate random numbers, described in Section 8.1.1, we compare the number of eggs produced by each mosquito in a one year simulation with a maximum of approximately 40,000 agents. For the mosquitoes that we are simulating, the number of eggs is calculated based on

Algorithm 8.5 Formula calculating the number of eggs produced for a single mosquito developing from blood meal digesting into gravid state. *numEggBatches* is the count how often a specific mosquito already spawned new eggs.

```
function GenerateEggs(int:numEggBatches)
    $eggBatchSize \leftarrow 170$
    $stdDev \leftarrow 30$
    $nEggs \leftarrow \lfloor \text{RandNormal}(eggBatchSize, stdDev) \rfloor$
    if $eggs < 0$ then
        $eggs \leftarrow 0$
    end if
    $eggs \leftarrow \text{RoundToNearest}(eggs \cdot 0.8^{numEggBatches})$
    return $eggs$
end function
```



(a) 5 runs (b) 50 runs

Figure 8.5: Histogram of the number of eggs generated per mosquito for SAMPO and AGiLESim. The left picture shows the histogram for five runs, the right picture for 50 runs. Each run simulates one year with a maximal population size of approximately 40,000 mosquitoes.

a normal distributed random number as shown in Algorithm 8.5. To spawn these numbers, SAMPO uses the Box-Muller transformation described in Section 8.1.1, while the used version of AGiLESim uses the random number generator provided by RepastJ [117]. Figure 8.5 shows the histograms of the number of eggs created by the individual mosquitoes. It can be seen, that the histogram curves are very similar, especially with a high number of runs.

Population comparison

In order to show that SAMPO produces correct populations we perform two validity checks on the produced population of adult mosquitoes. The first one compares the average number of adult mosquitoes on each day of a one-year-long simulation to the ones produced by the Java version of AGiLESim. Figure 8.6 shows how the number of adult mosquitoes evolves during the simulation.

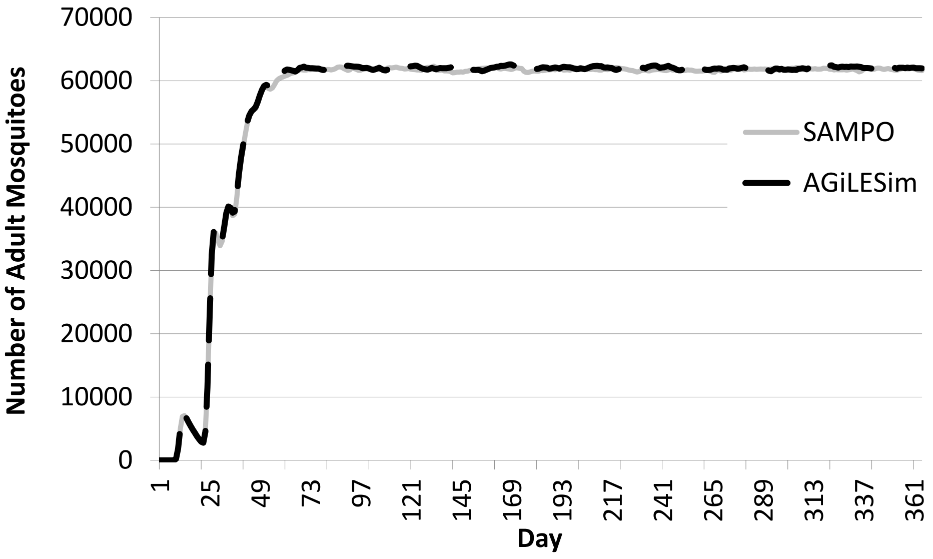


Figure 8.6: Average number of adult agents on each day of a one-year-long simulation, created with SAMPO and AGiLESim, respectively. The maximum population size for this experiment was approximately 1.4 million agents, the environment temperature was a constant 30°C.

The picture clearly elucidates that the populations generated by the two different implementations can be considered as equal.

The second validation compares the ages of all adult agents at the end of a one-year-long simulation with the hypothetical age structure calculated using the formula presented in [56]. Figure 8.7 shows a histogram of the adult mosquitoes' age distribution as well as the curve calculated based on the daily mortality rate of agents, as described in [56]. Also in this case, the values produced by SAMPO match the expected ones, thus we consider it as correct.

8.1.3 Performance

The main goal of implementing this mosquito simulation in OpenCL was to reduce the simulation time using modern accelerator hardware such as GPUs. In the following paragraphs we will demonstrate the performance of our implementation in comparison to the Java version of AGiLESim [56] as well as its scalability with increasing population sizes on various processors. The population size is controlled by adapting the initial number of eggs put into the system, as well as the carrying capacity. For all experiments, the carrying capacity is constant throughout the entire simulation and set to 5× the initial number of eggs. We chose a simulation length of one year with a resolution of one hour, leading to 8760 simulation steps. The temperature was constant at 30°C.

Figure 8.8 shows the execution times using different population sizes of the AGiLESim compared to SAMPO. The execution times of the Java implementation of AGiLESim were measured on a dual socket Intel Xeon X5650 CPU (see Table 8.4 for more details). It can clearly be seen, that SAMPO's OpenCL implementation scales much better with increasing problem sizes than the Java version. While the smallest problem size with a maximal population size of approximately 40 thousand

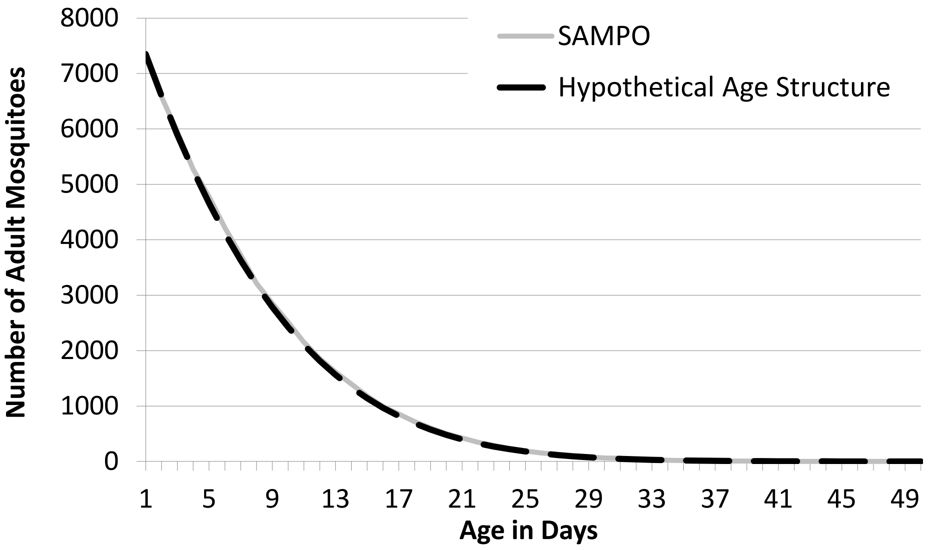


Figure 8.7: Histogram showing the age in days of the adult mosquitoes at the end of a one-year-long simulation as well as the hypothetical age structure calculated using the formula presented in [56]. The maximum population size for this experiment was approximately 1.4 million agents, the environment temperature was a constant 30°C.

	AMD Opteron 6168	Intel Xeon X5650	AMD Radeon HD5870	AMD FirePro S9000	NVIDIA GeForce GTX 480	NVIDIA Tesla k20c
Type	CPU	CPU	GPU	GPU	GPU	GPU
# Chips	2	2	1	1	1	1
Frequency (MHz)	1900	2670	850	900	1401	706
Compute Units	24	24	20	28	15	13
# Parallel Ops	96	48	1600	1792	480	2496
FLOPS (SP)	365	256	2720	3225	1345	3524
Memory (GB)	32	24	1	6	1.5	5
Memory BW (GB/s)	83	62	153	264	177	208

Table 8.4: Used hardware

agents is almost equally fast in both implementations, for the biggest tested problem size, with a maximal population size of approximately 5.5 million agents, the OpenCL version outperforms the Java implementation by a factor of 46. There are several reasons for that:

- The AGiLESim uses double precision for all floating point numbers in the simulation. SAMPO uses only single precision. However, as shown in Section 8.1.2, this has no negative effect on the result of the simulation.
- While the Java version of AGiLESim is sequential, the OpenCL implementation is parallel and uses all cores available in the system.

- The Java version of AGiLESim has a considerably lower constant overhead than our implementation using OpenCL. Executing all 8760 simulation steps without any actual calculation (but including all necessary data transfers for the OpenCL implementation), the Java version takes 3.5 seconds, while the OpenCL implementation needs 24.2 seconds to execute this task.
- Java has an automated memory management. The memory allocation is hidden from the programmer. Occasionally, a garbage collector is invoked by the Java VM, which frees all unused memory. The bigger the population, the more often the garbage collector has to be invoked in order to limit the maximum memory requirements. Since one invocation of the garbage collector consumes linear time in relation to the population size and it is invoked more often with increasing population sizes, this process consumes a considerable amount of time of the Java version when simulating large populations.
- Many agents in AGiLESim are stored in Java `ArrayLists`. Dead agents are eliminated from those lists using the `remove` operation. Since `ArrayLists` internally store data in an array, this operation requires moving all elements positioned after the eliminated element. This results in an average of $\frac{n}{2}$ data movements for each remove operation, leading to above linear time complexity.

	Compilation Time	
	Uncached (s)	Cached[1](s)
AMD Opteron 6168	1.10	N/A
Intel Xeon X5650	2.03	N/A
AMD Radeon HD5870	5.23	N/A
AMD FirePro S9000	1.85	N/A
NVIDIA GeForce GTX 480	5.83	0.14
NVIDIA Tesla k20c	9.93	0.14

[1] The NVIDIA OpenCL run-time compiler automatically caches compiled kernels in the user's home directory. If a cached kernel is executed, it can be directly loaded form the cache without recompilation resulting in much less overhead.

Table 8.5: OpenCL run-time-compilation time for the used processing units.

We analyze the performance and scalability of the OpenCL implementation on the processors described in Table 8.4, including CPUs and GPUs. Figure 8.8 shows the performance of those processors with varying population sizes. The numbers shown in Figure 8.8 represent the execution time of the simulation only, excluding the time needed for the OpenCL run-time compilation. The compilation times for the various devices can be found in Table 8.5.

The performance measurements clearly show that GPUs suit this simulation better than CPUs. A NVIDIA Tesla k20c achieves a speedup of 6-7 and 8-11 compared to the tested Intel and AMD dual socket CPUs, respectively. The CPUs and the NVIDIA GPUs scale almost linear with the population size of the simulation. On the AMD GPUs, however, the execution time raises less than linear with increasing problem sizes. The downside of the AMD GPUs is that they show relative long execution times for small population sizes. According to our measurements, just invoking all

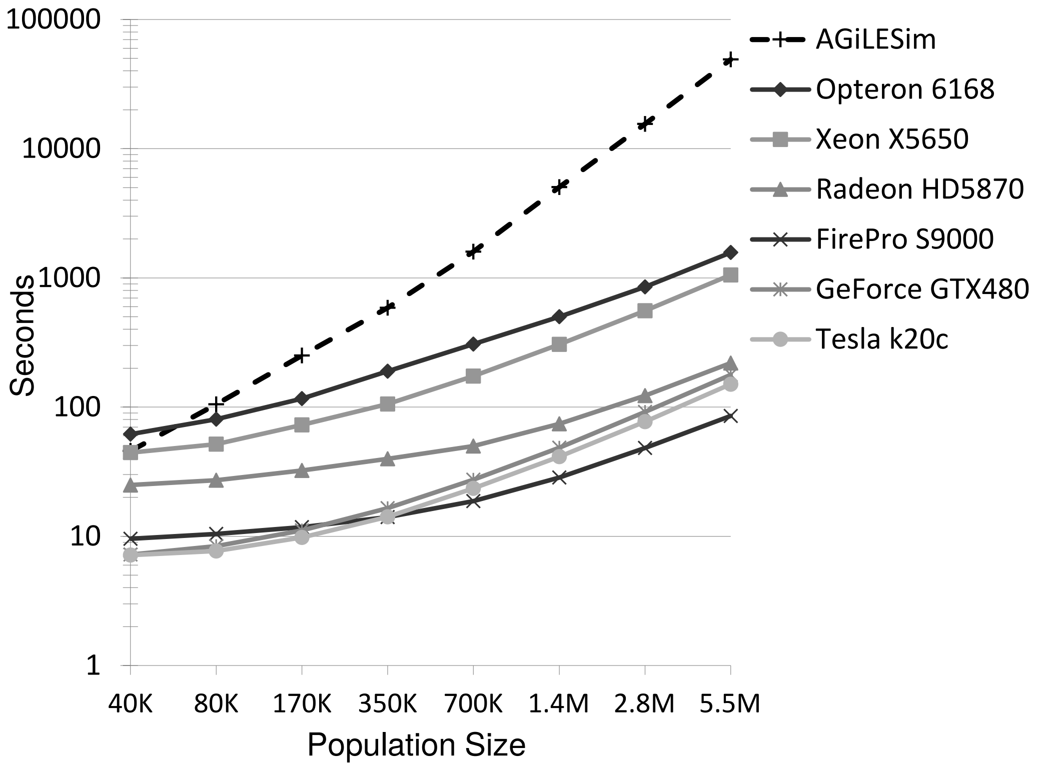


Figure 8.8: Execution times with varying population sizes for the Java implementation used in AGiLESim [56] and our OpenCL implementation on various processing units. The execution times of AGiLESim were measured on an Intel Xeon X5650.

the kernels and data transfers needed for the simulation without doing any actual calculations takes about 17 and 9 seconds on the Radeon HD5870 and FirePro S9000, respectively. In contrast, both NVIDIA GPUs need only 5 seconds for this task. The high overhead on the AMD GPUs seems to be related to the high kernel invocation overhead on such hardware as already pointed out by the microbenchmarks presented in Chapter 4. The inefficient use of resources on the AMD GPUs is also underlined by the low GPU usage. Depending on the population size, we observed a GPU usage of 35-95% and 38-94% on the Radeon HD5870 and FirePro S9000, respectively. On the Tesla k20c the GPU utilization ranges from 80 to 99% while NVIDIA does not provide any tool to remotely observe the GPU utilization on the GeForce parts. The overall performance of the NVIDIA Tesla k20c is disappointing. Despite the report in [119] we observed a maximal speedup of 1.19 over the previous generation part for our simulation. Furthermore, when simulating large populations, the Tesla k20c is outpaced by the FirePro S9000, despite its bigger overhead and lower GPU usage.

8.1.4 Summary

In this work we implemented SAMPO, a parallel OpenCL version of an existing agent based point model for simulating populations of the *Anopheles gambiae* mosquito. We described some of the more important challenges that arise when converting an ABMS to the parallel architecture necessary for

OpenCL and how we addressed them. Furthermore, we demonstrated correctness of our implementation by comparing the output to the one generated by AGiLESim. The theoretical age structure of the population is verified similarly to the verification applied for the original AGiLESim model [56].

The main goal of this work was to investigate the use of OpenCL as a means of minimizing simulation time for large scale ABMS executions. Our implementation is very effective for larger population sizes, achieving a speedup of up to 46 over the Java implementation of AGiLESim when running on the same multi-core CPU. In some respects this same-CPU comparison illustrates the overhead of using non-native code (such as Java programs) for simulation. Furthermore, we might expect the OpenCL version, running on the CPU, to have similar performance characteristics to a highly-optimized C version. Running the simulation on modern GPUs, we observed that the performance improved approximately 12 times compared to the same code running on the CPU, and a total speedup of 576 over the original Java implementation. These findings indicate that OpenCL is an attractive environment for building compute-intensive agent-based simulations.

In the future, we expect that the functionality of our simulation will be extended, incorporating features to simulate the effects of various vector-targeted interventions like insecticide-treated mosquito nets and indoor residual spraying (IRS) in order to observe their impact on the malaria transmission cycle. SAMPO was designed from the beginning to allow this kind of interventions. Furthermore, additional mosquito species could be added to our simulation. To support that, our OpenCL implementation encapsulates the species specific behavior a single header file, which can easily be exchanged with one characterizing another species.

8.2 KD-tree based N-body simulations

This chapter introduces a novel method to effectively parallelize N-body simulations for GPUs. Our method is based on an efficient, three-phase, parallel Kd-tree building algorithm and a novel volume-mass heuristic to reduce the simulation time.The contribution is a joint effort of members of the DPS group, Institute of computer science and the Astrophysics Group at the University of Innsbruck as part of the Doktoratskolleg Plus Computational Interdisciplinary Modelling (DK+CIM) [166]. Dominik Steinhauser was providing the background knowledge about astrophysics simulations and their current state of the art as well as implementing the checks of quality and correctness. My responsibilities were mainly the parallel implementation of the simulation in OpenCL and optimization of the program as well as running the experiments. The results of the research performed during this collaboration are published in [96].

In history, understanding the motion of celestial objects under their mutual gravitational attraction, motivated to search for a solution to the N-body problem. Newtonian attraction forces between each pair of bodies lead to their acceleration and hence collective motion. The goal is to predict future positions and velocities for all bodies (often called particles in this context) starting from a given initial state. The corresponding differential equations to this initial value problem can be solved analytically only for $N \leq 3$. Larger simulations can only be calculated numerically.

The challenge in N-body codes is to find those mutual gravitational attraction forces. The simplest way to evaluate the force acting on a single particle is by summing the contribution from all the other particles, called *direct summation* approach. However, this rather brute-force method is of order $\mathcal{O}(N^2)$ and is hence only a viable option for small problems.

Particle-mesh codes, as described in [99] are more efficient than direct summation. However, close particle interactions are not well modeled. Hybrid approaches such as P^3M [70] overcome this problem.

When calculating the gravitational force contribution of a reasonably distant group of close bodies on a single particle, this group can be approximated with a single, more massive proxy-body, in order to reduce the computational effort. To add information about the distribution of the particles inside this proxy-body, higher order moments of the gravitational potential's multipole expansion of the particle group can be used. This observation is exploited in the so-called tree code approach to the N-body problem [18]. Tree codes make use of space-partitioning data structures to recursively divide up the simulation domain in sub-volumes. While Barnes&Hut [18] used octrees, we adopt Kd-trees in our work: Each volume is split in two sub-volumes according to a splitting plane. The leaves of the tree contain just one single particle each. Thus, for each node the potential in terms of a multipole expansion is calculated. When calculating the force contribution for a single particle, a tree traversal is done. In case a tree node, representing a part of the simulation domain, is reasonably remote from this particle, the approximate potential in this node can be used to calculate the force contribution of all particles contained within this node. Hence, the subtree of this node does not need to be considered anymore. The *cell opening criterion* [18] defines whether the tree walk can be stopped at the current node, using this node as proxy body, or the descent is continued further in the tree to calculate the force more accurately.

Being a very computational intensive application, N-body simulations have been historically interesting for high performance computing. Recently, even GPUs have been exploited for this task.

However, while *direct summation* approaches are quite easy to be run on GPUs, more advanced hierarchical methods are very challenging. The use of intricate data structures, their traversal and in particular building them is a task which makes it hard to exploit the massive parallelism offered by such hardware. Three factors are critical for N-body simulations based on hierarchical methods on GPUs: 1) to run the whole algorithm on the GPU in order to avoid expensive communication bottlenecks due to CPU calculation of intermediate steps; 2) a fast traversal algorithm for the hierarchical data structure; 3) a fast building algorithm for such a data structure.

In this section, we introduce a novel method to accurately and efficiently calculate the gravitational forces, the computationally most expensive part of N-body simulations, on the GPU, needed for N-body simulations in each timestep. Our contributions are:

- a novel hierarchical method for calculating gravitational forces based on a Kd-tree on a GPU;
- a probabilistic approach based on volume-mass heuristic (VMH) to efficiently group particles in a Kd-tree and drastically improve the efficiency of the Kd-tree traversal;
- a parallel approach to efficiently build the Kd-tree on the GPU: by using a three phase building algorithm, we maximize the thread utilization of the GPU during the building in both top- and bottom-part of the Kd-tree;

8.2.1 Parallel Kd-tree building

Our Kd-tree building algorithm is designed to perform well on modern GPUs, exposing a large amount of parallel operations. This means that it has to expose a large amount of parallel operations. The program was implemented in OpenCL. Our highly parallel implementation is inspired by the algorithm presented in [180] and consists of three phases:

- Large node phase
- Small node phase
- Kd-tree output phase

The large node phase takes place at the beginning of the tree construction, where only few nodes, containing many particles, exist. To increase the degree of parallelism, both, inter- and intra-node parallelism are exploited during this phase. In the small node phase, many nodes are handled at a time. Therefore, it is better to avoid the additional synchronization overhead introduced by the intra-node parallelism and rely on inter node parallelism only in this phase.

A pseudo code representation of our implementation is shown in Algorithm 8.6. It shows, that the implementation is split into four loops. The first loop represents the so called large node phase, the second loop the small node phase while the last two loops perform the up pass and the down pass to sort the tree nodes in depth first ordering. All iterations of these four main loops in our implementation have to be executed sequentially. Therefore, these loops cannot be used to exploit parallelism. However, as explained in the following paragraphs, there are several possibilities for parallelization inside those loops.

Algorithm 8.6 Kd-tree construction

```
function buildKdTree(particles:list)
    nodelist ← new list()
    activelist ← new list()
    nextlist ← new list()
    smalllist ← new list()
    rootnode ← new node(particles)
    rootnode.offset ← 0
    nodelist.add(rootnode)
    activelist.add(rootnode)
    while !activelist.empty() do                                        ▷ large node phase
        processLargeNodes(nodelist, activelist, nextlist, smalllist, particles)
        activelist ← nextlist
    end while
    activelist ← smalllist
    while !activelist.empty() do                                        ▷ small node phase
        processSmallNodes(nodelist, activelist, nextlist, particles)
        activelist ← nextlist
    end while
    treeHeight ← max level of all nodes in nodelist
    for level ← treeHeight to 0 do
        upPass(nodelist, particles, level)
    end for
    tree ← new list()
    for level ← 0 to treeHeight do
        downPass(nodelist, tree, level)
    end for
end function
```

Large node phase In the large node phase, all large nodes are split in two child nodes in the middle of their longest dimension. Their particles are distributed to the children depending on their position. This step is repeated until no more large nodes are left. A large node is defined as a node containing at least 256 particles. In this phase, the inter node parallelism is maximized, e.g. by reductions in local memory and parallel prefix scans which are both known to perform well on GPUs [116]. While the reductions in local memory are used to accelerate the bounding box calculation, parallel prefix scans are needed to calculate the position of particles in the particle array in parallel after a node is split. The application of the aforementioned techniques introduces several global synchronizations due to data dependencies. However, the overhead introduced by additional synchronization is outweighed by the increase of parallelism. Furthermore, in this phase the node splitting decision is designed not to be affected by the number of particles inside the node in order to scale to bigger data-sets. Algorithm 8.7 depicts the large node phase implementation. It is composed of six parallel loops, each of which is implemented as a separate OpenCL kernel function.

Distributing the particles of a parent node to its two child nodes is the most time consuming part of the large node phase, since it requires rearranging of the particles of the parent node. When building a Kd-tree, this can be done only after selecting the splitting point, since the particles have to be partitioned according to the splitting point along the splitting dimension. The particles don't

Algorithm 8.7 Large Node Phase

```
 1: function processLargeNodes(nodelist:list, activelist:list, nextlist:list, smalllist:list, particles:list)
 2:     chunklist ← new list()
 3:     ▷ group particles to chunks
 4:     for all node in activelist in parallel do
 5:         Group all particles in node into fix sized chunks and store them in chunklist
 6:     end for
 7:     ▷ calculate per-chunk bounding box
 8:     for all c in chunklist in parallel do
 9:         Compute bounding box for each chunk c
10:     end for
11:     ▷ calculate per-node bounding box
12:     for all node in activelist in parallel do
13:         Compute bounding box for each node node using the bounding boxes of the chunks
14:     end for
15:     ▷ split large nodes
16:     for all node in activelist in parallel do
17:         set node.splittingPoint to the spatial median along the longest dimension
18:         Split node node at node.splittingPoint
19:         Store generated child nodes in nextlist
20:     end for
21:     nodelist.add(nextlist)                                  ▷ add all new nodes to nodelist
22:     ▷ sort particles to children
23:     for all node in activelist in parallel do
24:         for all particle in node.particles do
25:             if particle.pos[splitDim] ¡ node.splittingPoint then
26:                 node.leftChild.particles.append(particle)
27:             else
28:                 node.rightChild.particles.append(particle)
29:             end if
30:         end for
31:     end for
32:     ▷ small node filtering
33:     for all node in nextlist in parallel do
34:         if node is small node then
35:             smalllist.add(node)
36:             nextlist.remove(node)
37:         end if
38:     end for
39: end function
```

have to be sorted, but all particles belonging to the left child have to be at the beginning of the parent's node particle sub-array, while all the particles belonging to the right child have to be moved to the end of that array. This can be done in linear time in each timestep. When executing our implementation on a CPU, one OpenCL work item is started for each active node which assigns the particles to the child nodes in a sequential loop. This approach works well for CPUs. However, it does not expose enough parallelism to reach a good performance on GPUs, since there are not many active nodes in this phase. Therefore, we use a parallel prefix scan to determine for each particle its index in the particle list of the left and right child, respectively. Using the result of the prefix scan, the particles can be inserted into the particle lists of the two child nodes in parallel.

Small node phase When no more large nodes are left, the program enters the small node phase. In this phase we aim at reducing the synchronization overhead (it needs only one synchronization at the end of each iteration) by starting only one single thread per active node. Increasing the parallelism any further would not improve the performance, since the number of active nodes is very high in most iterations. In order to improve the quality of the tree, this phase uses a splitting strategy based on VMH as described in Section 8.2.2. In our environment, each particle inside a node introduces one splitting candidate (along the node's longest dimension). The VMH cost has to be evaluated on each splitting candidate, which makes this strategy infeasible for large nodes. After the split, the particles of the parent node are assigned to the two child nodes, depending on their position. Each node is split according to the candidate giving minimal VMH, until the leaf nodes, containing only one single particle, are reached. The implementation of this phase is shown in Algorithm 8.8. It is composed of one parallel loop which is mapped to an OpenCL kernel function. Due to the typically high number of active nodes during this phase, the splitting of nodes into chunks is not necessary. Starting one OpenCL work item for each active node is sufficient to keep all processing elements of a GPU busy.

Kd-tree output phase During the first two phases, new nodes are added to the nodelist in the same order as they are created, which means, they are not sorted. In order to enable an efficient tree walk, the nodes are ordered in a depth-first manner, which is done in the last phase of our tree construction. To sort the nodes of the Kd-tree, two passes have to be performed: First a bottom-up pass which calculates the center of mass and the mass of each node (which corresponds to the proxy-body in the node or the potential's monopole moment) as well as the size of the subtree underneath it. The size of the subtree is important in order to calculate the actual position of a node in the final tree. The implementation of this pass is described in Algorithm 8.9. The second pass is building the final tree top down. The root is written at the beginning of the array. For each node at position i, the left child will be written to position $i+1$ and the right child to position $i+1+sizeof(leftChild)$. Sorting the nodes in that way, a linear traversal of the node array is equal to a depth-first traversal of the tree. A detailed description of this pass can be found in Algorithm 8.10.

8.2.2 Volume-Mass Heuristic (VMH)

When analyzing the requirements of an optimal Kd-tree for an N-body simulation, we determined that they are very similar to the requirements for ray-tracing. In both cases, it is not really important that the tree is balanced, but that the average path length of the walks through the trees are minimized. However, there is a slight difference between those two applications. In ray-tracing each ray walks from the root to a leaf, deciding at each node if it should advance to the left or the right child of it. Therefore, the SAH heuristic tries to even the probability of taking the left or right path at each node. In contrast to that, for the N-body simulation, the path of each particle is highly divergent, since it always advances to both children of a node, unless in the cases when no further descent is needed on that node (i.e. the cell opening criterion is not fulfilled). Therefore we want a tree where,

Algorithm 8.8 Small Node Phase

```
1: function processSmallNodes(nodelist:list, activelist:list, nextlist:list, particles:list)
2:     ▷ split small nodes
3:     for all node in activelist in parallel do
4:         ▷ calculate VMH
5:         VMH ← $\infty$
6:         for all potential splitting points sp of node do
7:             VMH_sp ← calcVMH(node, sp)
8:             if VMH > VMH_sp then
9:                 VMH ← VMH_sp
10:                node.splittingPoint ← sp
11:            end if
12:        end for
13:        Split node at node.splittingPoint
14:        Store generated child nodes in nextlist
15:        nodelist.add(nextlist)                                ▷ add all new nodes to nodelist
16:        ▷ sort particles to child nodes
17:        for all particle in node.particles do
18:            if particle.pos[splitDim] ¡ node.splittingPoint then
19:                node.leftChild.particles.append(particle)
20:            else
21:                node.rightChild.particles.append(particle)
22:            end if
23:        end for
24:        ▷ Leaf node filtering
25:        for all node in nextlist do
26:            if node.particles.size = 1 then
27:                nextlist.remove(node)
28:            end if
29:        end for
30:    end for
31: end function
```

on each node the probability to stop the decent at the left child is equal to the probability to stop it on the right child.

Due to this similarity of requirements on the tree, we used a variation of the SAH to determine the splitting point of the nodes in our tree (at least for the small nodes, see Section 8.2.1). In our case, the heuristic is ported to 3D and the surface area is replaced by the mass of the corresponding node. This leads to the following equation:

$$VMH(x) = Vol_l(x) \cdot Mass_l(x) + Vol_r(x) \cdot Mass_r(x)$$

where $Vol_l(x)$ and $Mass_l(x)$ correspond to the volume and mass of the potential left child of the node, splitting the node at an axis aligned plane crossing the parent node at position x in the splitting dimension. $Vol_r(x)$ and $Mass_r(x)$ denote the volume and mass of the node's right child when split at position x. $VMH(x)$ is the volume-mass heuristic cost for the splitting position x. The cost is evaluated once for each particle inside the node, whereas the position of the particle in the splitting

Algorithm 8.9 Up pass

```
1: function upPass(nodelist:list, particles:list, position:int)
2:     for all node in nodelist at level position in parallel do
3:         if node.isLeaf then
4:             node.size ← 1
5:             node.mass ← node.particles[0].mass
6:             node.centerOfMass ← node.particles[0].pos
7:             node.l ← 0
8:         else
9:             node.size ← node.leftChild.size + node.rightChild.size + 1
10:            node.mass ← node.leftChild.mass + node.rightChild.mass
11:            node.centerOfMass ← (node.leftChild.centerOfMass
12: ·node.leftChild.mass + node.rightChild.centerOfMass
13: ·node.rightChild.mass) / node.mass
14:            node.l ← maximum side length of node.boundingBox
15:        end if
16:    end for
17: end function
```

Algorithm 8.10 Down pass

```
1: function downPass(nodelist:list, tree:list, position:int)
2:     for all node in nodelist at level position in parallel do
3:         if !node.isLeaf then
4:             node.leftChild.offset ← node.offset + 1
5:             node.rightChild.offset ← node.offset + 1 + node.leftChild.size
6:         end if
7:         tree[node.offset] ← node
8:     end for
9: end function
```

dimension determines the splitting position x. The node is split at the position x which minimizes the VMH cost.

8.2.3 Force calculation with Kd-trees

As described in the previous section, trees can be used to efficiently reduce the computational effort to solve the N-body problem numerically. In our implementation we are using a Kd-tree since they proved to be very efficient in other fields like e.g. ray tracing [180].

Gravitational force calculation The main idea of tree algorithms is to reduce the computational cost of the force computation on a single particle by using a hierarchical multipole expansion. All particles in the simulation domain are hierarchically grouped into cells, the tree nodes for which the multipole expansion is calculated. For the force contribution of distant particles, a larger grouping of particles, namely a node in a higher level of the tree, can be used. Using the cell opening criterion, it can be decided whether a group of particles can be used or the tree needs to be traversed further. This approach allows to compute the force on a single particle with approximately $\log(\#nodes)$

interactions.

Unlike other implementations which are using quadrupole (e.g. Bonsai [20]) or even higher moments (e.g. Gasoline [170]), we follow the approach of GADGET-2 and only use monopole moments with the advantage that less memory is required, as just the total mass in a node and the center-of-mass coordinates need to be stored. Furthermore, the computational effort is lower while constructing the tree as higher moments do not need to be calculated and the monopole moments can be calculated conveniently during tree construction (see Section 8.2.1). However, using monopole moments lowers the force accuracy. Still, the force accuracy can be controlled by the cell opening criterion e.g. by using a smaller cell opening angle. Depending obviously on the problem to be solved and on the implementation, opening more cells is still a small trade-off compared to computing higher order moments during tree construction.

Cell opening criterion In our implementation, as we are using monopole moments for tree nodes, we apply the same strategy used in GADGET-2 [140], and use their *optimal* cell opening criterion. A cell (i.e. a node in the Kd-tree) is accepted if

$$\frac{G \cdot Mass}{d^2}\left(\frac{len}{d}\right)^2 \leq \alpha \cdot |\mathbf{acc}|$$

evaluates to true, with G being the gravitational constant, $Mass$ the total mass in the node, d the distance of the particle under consideration to the center-of-mass of the node, and len the largest side-length of the axis aligned bounding box around all nodes inside the corresponding node. Finally, *acc* is the acceleration of the particle from the last timestep and α a *tolerance parameter*, used to control the force accuracy. However, in some cases this criterion is fulfilled also if the actual particle is located within a considered node which would lead to large force errors. To prevent against this, we additionally require the particle to lie sufficiently outside the bounding box of a node to be accepted. For more details we refer to [140].

Parallel force calculation using a Kd-tree

After the tree has been built, the actual force on each particle can be calculated by walking through the tree in a depth first manner. For each particle an OpenCL work item is started, walking the tree beginning from the root. On each node, the cell opening criterion is evaluated. If it is fulfilled, the walk will advance to both child nodes of the current node. If not, the force acting on the particle is calculated, using the current node as a proxy for all particles within the node. The pseudocode for our tree walk is shown in Algorithm 8.11. Although the tree walk is highly divergent, it can be implemented as a single loop, due to the depth-first ordering of the nodes.

8.2.4 Time Integration

To carry out full N-body simulations, we implement a time-centered leapfrog integration scheme (e.g. [18, 130]) with constant timesteps. Positions of the particles are advanced at full timesteps (drift) while new velocities are calculated at halfsteps (kick),

Algorithm 8.11 Force calculation for each particle using the previously constructed Kd-tree.

```
1: function TreeWalk(particles:list, tree:list)
2:     for all particle in particles in parallel do
3:         for currentNode ← 0 to tree.size do
4:             node ← tree[currentNode]
5:             if node.isLeaf or !openCell(particle, node) then
6:                 particle.force ← particle.force+ calcForce(particle, node)
7:                 currentNode ← currentNode + node.size          ▷ skip entire subtree of current node
8:             else
9:                 currentNode ← currentNode + 1                  ▷ continue depth-first walk
10:            end if
11:        end for
12:    end for
13: end function
```

$$\begin{aligned}\mathbf{x}_{i+1} =& \mathbf{x}_i + \mathbf{v}_{i+\frac{1}{2}}\,\Delta t\\ \mathbf{v}_{i+\frac{1}{2}} =& \mathbf{v}_{i-\frac{1}{2}} + \mathbf{acc}_i\,\Delta t\end{aligned}$$

with x_i being the position and v_i the velocity of a particle at time i and Δt being the timestep. At each full timestep, the acceleration acc_i is calculated using the Kd-tree implementation presented in the previous sections. Dynamic tree updates are used to prevent rebuilding the tree in each timestep: after calculating the new positions of the particles, the center of mass and bounding box of each tree node are updated. This update is performed by propagating the updated positions/bounding boxes bottom up the Kd-tree in a single pass. The tree is rebuilt when the computational cost (measured in numbers of interactions per particle) exceeds the initial value (when the tree was rebuilt the last time) by 20%. Initially, $v_{i-\frac{1}{2}}$ is calculated by kicking the system of particles by half a timestep.

Being of $\mathcal{O}(n)$, the time needed for the time integration is negligible with respect to the tree building and force calculation.

8.2.5 Results and Evaluation

In this section we evaluate the result of our implementation in comparison to the state of the art simulation codes in terms of accuracy and execution speed.

Accuracy

In order to demonstrate the accuracy and quality of our implementation, we observe and compare the energy conservation of our implementation to other N-body codes. As comparison we use the widely spread GADGET-2 code, also because we use the same monopole and cell opening criterion, and to Bonsai, the state of the art N-body code for GPUs.

For our tests we are using a particle distribution according to a Hernquist density profile [71], an analytical model to describe dark-matter halos, spherical galaxies and bulges. For accuracy evaluation,

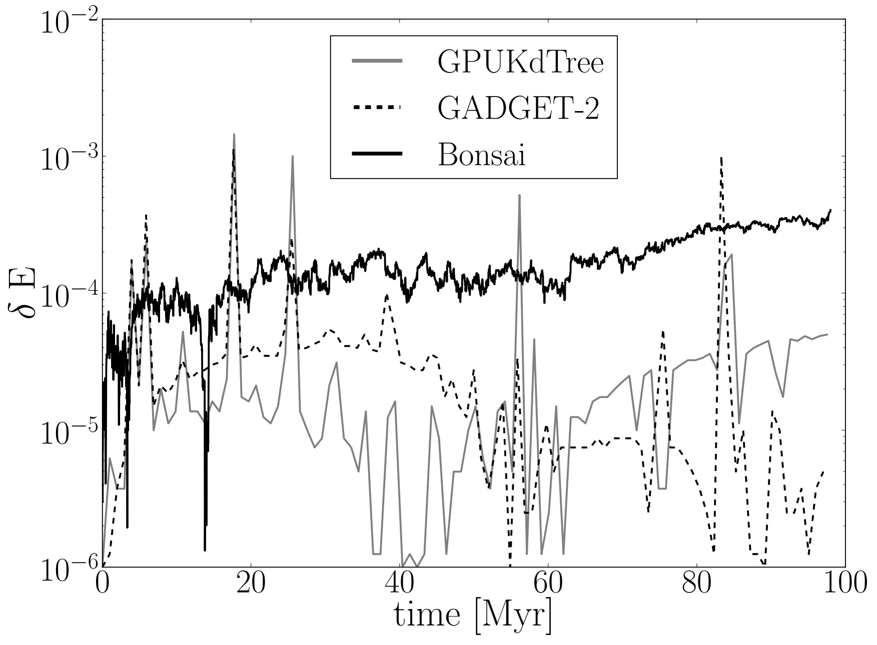


Figure 8.9: Relative energy error δE throughout the simulation

we use 250,000 particles with a total mass of $1.14 \times 10^{12}\mathrm{M}_{\odot}$. In Figure 8.9 we plot the relative energy error

$$\delta E = \frac{Energy_0 - Energy_i}{Energy_0}$$

with $Energy_0$ being the total energy (kinetic plus potential energy of the particle distribution) at the beginning of the simulation and $Energy_i$ the total energy at simulation time i. For GPUKdTree and Bonsai we chose a fixed timestep of $0.003\,Myr$. In GADGET-2 we set this value as the maximum allowed timestep in order to prevent the usage of the individual timestepping (differently sized timestep for each particle depending on the current acceleration acting on the particle) for a fair comparison between all codes. The results show that our GPUKdTree implementation provides a small energy error throughout the whole simulation, comparable to GADGET-2. Bonsai however shows a somewhat higher but at the same time also more constant error. Both, GPUKdTree and GADGET-2 show more scatter in the error with some spikes in the distribution having a higher maximum error than Bonsai.

Performance

We evaluate the performance of our implementation on various CPUs and GPUs from different vendors, listed in Table 8.6. Our OpenCL implementation was designed to run on any device that supports OpenCL. For performance reasons we use a dedicated algorithm to sort bodies during the large node phase for GPUs and CPUs. NVIDIA GPUs could not run our OpenCL code correctly,

	Intel Xeon X5650	AMD Radeon HD5870	AMD Radeon HD7950	NVIDIA GeForce GTX 480	NVIDIA Tesla k20c
# Chips	2	1	1	1	1
Frequency (MHz)	2670	850	860	1401	706
Compute Units	24	20	28	15	13
# Parallel Ops	48	1600	1792	480	2496
FLOPS (SP)	256	2720	3082	1345	3524
Memory (GB)	24	1	3	1.5	5
Memory BW (GB/s)	62	153	240	177	208

Table 8.6: Used hardware

N. Particles	250k	500k	1M	2M
Xeon X5650	881	1795	3640	7278
GeForce GTX480	158	290	595	1202
Tesla k20c	167	293	586	1195
Radeon HD5870	262	381	675	-
Radeon HD7950	152	219	380	698
GADGET-2 (X5650)	50	90	180	370
Bonsai (GTX480)	24	43	83	167

Table 8.7: Tree building times in ms

giving wrong results without any error message. However, since we used LibWater [59] to implement our program, it could easily be ported to CUDA without any changes in our code. The CUDA version works flawlessly on the NVIDIA GPUs.

To evaluate the performance, we are using datasets containing different number of particles, all using a Hernquist density profile as used in our accuracy experiments described in the previous paragraph, with 250,000 to 2,000,000 particles. We also compare the performance of our implementation with the one achieved with GADGET-2 [140] and Bonsai [20]. GADGET-2 contains no implementation for GPUs and can only be executed on CPUs. For our experiments we use the same dual socket Intel Xeon X5650 system with a total of twelve cores that we used to evaluate the performance of our implementation on CPUs. Bonsai is implemented in CUDA and is therefore limited to NVIDIA GPUs. Furthermore, the version of Bonsai which is available online did not work on our Tesla k20c GPU. On this hardware, the program crashed due to a CUDA driver error. Hence, we could evaluate Bonsai's performance only on a NVIDIA GeForce GTX480.

For a fair comparison, we set the accuracy parameters for each implementation to achieve an error below 0.4% for 99% of the particles. This results in an α of 0.001 and 0.0025 for GPUKdTree and GADGET-2, respectively. For Bonsai, Θ is set to 1.0.

Tree building Table 8.7 shows the time needed for tree building on different hardware with different data sizes. The numbers show, that our tree building algorithm fits the GPU architecture quite well. All GPUs show a speedup between 3.3 and 10.4 over the tested CPU. It is noteworthy, that the NVIDIA GPUs are more effective for smaller datasets, while the AMD GPUs scale better with the problem size. The relative bad performance of the AMD GPUs on small problem sizes is related to the very high number of kernels that have to be called during the tree building (see Section 8.2.1) in correlation with their high kernel invocation overhead, as mentioned in Chapter 4. The simulation with two million particles could not be run on the AMD Radeon HD5870 due to its restriction to the maximum size of a single buffer. It is interesting to note, that the NVIDIA GeForce GTX480 shows almost the same performance as the much newer NVIDIA Tesla k20c, although the latter one has a much higher peak performance (1.3 vs. 3.5 TFLOPs).

The times given for GADGET-2 and Bonsai include the sorting of the particles and the building of the tree, since they can construct the tree only on pre-sorted particles. Building the octree used in both GADGET-2 and Bonsai is much faster than building the Kd-tree used in our approach. This is caused by the rearranging of the particles. To build an octree, the domain is decomposed using a Peano-Hilbert curve [141]. At the beginning of the tree building, the particles are sorted according to this domain composition. By doing so, the particles do not have to be rearranged during the rest of the tree building. When building a Kd-tree, on the other hand, the particles have to be rearranged in each iteration of the tree building step, which takes a significant amount of time.

Tree Walk As explained in Section 8.2.3, the force on each particle is calculated by walking through the previously built tree. The performance for the tree walk is given in Table 8.8. Also, the tree walk is faster on all tested GPUs than on the tested CPU. The speedup varies between 1.9 and 6.3 depending on the GPU and data size. The AMD GPUs are suited better for the tree walk than both NVIDIA GPUs. Even the old AMD Radeon HD5870 is able to outperform both NVIDIA GPUs. During the tree walk, the large kernel invocation overhead of the AMD GPUs plays a minor role, since the tree walk of all particles consists of only one single kernel call. Using a AMD Radeon HD7950, our implementation can reach a throughput of 3 Mparticles/s.

The measurements clearly show, that our implementation is much faster than GADGET-2, mainly due to the efficient use of the massively parallel GPUs. Also, using the same CPU, the tree walk of our implementation is approximately twice as fast as in GADGET-2. However, GADGET-2 lacks an OpenMp implementation and is handicapped by overhead due to the MPI library in these tests. Bonsai shows a very high performance in our test case. However, this high performance comes at the cost of a worse energy conservation, as depicted in Figure 8.9, which shows the energy error δE equal to $\frac{Energy_0 - Energy_t}{Energy_0}$.

8.2.6 Related Work

The following paragraphs give an overview of some related work in several relevant fields for this section.

N-body simulations Historically, many researchers from both computer science and astrophysics have developed parallel N-body simulations on supercomputers. Warren and Salmon [173] designed

N. Particles	250k	500k	1M	2M
Xeon X5650	456	966	1996	4145
GeForce GTX480	236	476	934	1844
Tesla k20c	204	405	801	1588
Radeon HD5870	155	287	572	-
Radeon HD7950	85	169	332	651
GADGET-2 (X5650)	909	1940	4160	8580
Bonsai (GTX480)	40	81	163	325

Table 8.8: Force calculation using a previously constructed tree times in ms.

one of the first parallel implementation of the Barnes&Hut algorithm. The authors of [143] propose a parallel implementation of an N-body code using a Kd-tree structure.

A very widespread code used in astrophysics, mainly for cosmological simulations and simulations on galaxy scales, is the treePM code GADGET [141, 140]. This code implements a combination of a particle-mesh and a Barnes&Hut tree code, massively parallelized for distributed memory machines using MPI.

Nyland et al. [120] implemented a *direct summation*, brute-force technique. They improved their code by means of loop unrolling and by manually prefetching a certain number of body descriptions on the GPU. Elsen et. al [46] created a similar solution using the BrookGPU programming language [25].

The Gravity Pipe (GRAPE) [75] designated a very efficient hardware implementation of Newtonian pair-wise force calculations between particles in a self-gravitating N-body system. GRAPE-6 has been the first computer breaking the petaflops barrier.

Instead of direct summation, smarter approaches to attack the N-body problem use hierarchical algorithms and three dimensional space partitioning strategies. Hamada et al. [67] implemented a parallel, hierarchical N-body simulation which efficiently calculates the $\mathcal{O}(N \log N)$ tree code and $\mathcal{O}(N)$ fast multipole method (FMM) on multiple GPUs. Using this fast N-body solver, they performed a gravitational N-body simulation using 1,608,044,129 particles and, in addition, the vortex particle simulation of homogeneous isotropic turbulence using 16,777,216 particles. The tree code was used for the gravitational simulation, while the FMM was used for the vortex particle simulation. Hamada and Nitadori reached 190 TFlops [68] on DEGIMA, a cluster of 576 GPUs interconnected by InfiniBand, using their tree code.

The parallel cosmological simulator ChaNGa [80] is a hierarchical N-body gravity solver written in CHARM++, which has been run on the NCSA Lincoln GPU cluster.

2HOT [174] is a N-body simulation code based on a parallel hashed octree algorithm. It is designed to run efficiently on up to 256k processors with an efficiency of 0.86. 2HOT also includes a CUDA, as well as an OpenCL version for the gravitational interaction functions.

Bédorf et al. [20] implemented Bonsai[2], a sparse octree gravitational N-body code that runs entirely on the GPU, reaching up to 4 Mptcl/s on a Tesla 2075 and up to 17.6 Mptcl/s on a Tesla k20c for the tree walk and force computation. In contrast to most other tree codes, Bonsai traverses

[2]Version 2, http://castle.strw.leidenuniv.nl/software/bonsai-gpu-tree-code.html

the tree breadth-first to calculate the force acting on each particle.

GPU data structures The recent rise of general-purpose GPU computing has given rise to a number of methods for efficiently constructing spatial data structures and hierarchies, such as bounding volume hierarchies (BVHs), octrees, and Kd-trees. Karras [87] maximizes the parallelism with an in-place algorithm for constructing binary radix trees, which have been used as a building block for other types of trees. Feltmann et al. [50], in the context of ray tracing, show an optimized BVH building which improves the traversal of shadow rays. Li et al. [103] implemented a simple method for finding k approximate nearest neighbors (ANNs) on the GPU by exploiting a shifted sorting algorithm that provides a more GPU-friendly basis for ANN searching than the more well-known Kd-tree algorithm.

Probabilistic Heuristics for Hierarchy Building An approach to improve the traversal time of hierarchical data structures is to use a probabilistic heuristic while building it: the higher the probability of a node to be accessed, the higher it will be placed in the hierarchy. Similar heuristics have been largely used in the context of ray tracing, known as SAH (Surface Area Heuristics). Wald et al. [171] introduced an algorithm to build SAH-based Kd-trees in $\mathcal{O}(N \log N)$. Other approaches using SAH have been used for BVH (bounding volume hierarchy) and other hierarchical approaches [172]. Zhou et al. [180], in particular, presented an algorithm for constructing Kd-trees on GPUs. They achieve real-time performance by exploiting the streaming architecture of modern GPUs at all stages of Kd-tree construction. They develop a special strategy for large nodes at upper tree levels to further exploit the fine-grained parallelism of GPUs. Our work applies a similar method to build a tree for N-body simulations.

8.2.7 Summary

We have observed, that octrees for N-body simulations can be built very fast, on both GPUs and CPUs, when the particles are pre-sorted according to a Peano-Hilbert curve. Constructing a Kd-tree takes more time, mainly due to the rearranging of the particles in every timestep. By using GPUs, the Kd-tree construction can be accelerated up to 10 fold over the execution time on CPUs. The tree building time of GPUKdTree scales linearly with the number of particles in the simulation.

Our implementation of the tree walk is noticeably faster than the one of GADGET-2, when using the same hardware. It also shows better scalability than GADGET-2 with increasing problem sizes. Even more, executing the treewalk of our implementation on GPUs gives another speedup of up to 6 times over the CPU. We achieve a throughput of up to 3 Mparticles/s which is the highest performance reached on an AMD GPU that we are aware of. However, Bonsai achieves an even higher performance using NVIDIA GPUs. This shows, that Bonsai's breadth-first tree walk fits the GPU architecture better than our implementation, performing a depth-first walk. However, Bonsai also shows a worse energy conservation.

Chapter 9

Conclusion

Implementing high performance programs for parallel, heterogeneous hardware is a very time intensive and error-prone task. New tools and techniques are needed to automate as many parts of the optimization process of parallel programs as possible and thereby raise the programmers' productivity. In this thesis we demonstrated that several optimization tasks can be automated using the Insieme source-to-source compiler and runtime system infrastructure. In order to achieve this goal we applied techniques from different fields, including various machine learning algorithms, graph theory, the polyhedral model, iterative compilation and general differential algorithms.

9.1 Contributions

The implementation work performed in the course of this thesis added OpenCL support to the Insieme compiler frontend, as described in [82], and extended the auto-tuning capabilities of the Insieme compiler and runtime system. The version of Insieme applying the aforementioned contributions is publicly available at [2]. By using the Insieme compiler and runtime system we can show how parts of the optimization process for parallel and heterogeneous programs can be automated and thereby address the problems listed in Section 1.1:

Characterization of Heterogeneous Processing Units We developed uCLbench, a system that can analyze the performance characteristics of OpenCL devices to support programmers in optimizing their programs for their OpenCL devices. uCLbench has been published in [161] and proved to be useful to analyze the diversities of OpenCL devices, which cause low performance portability, demonstrating the need of an automatic optimization process.

Heterogeneous Task Partitioning We introduced a system that can automatically distribute OpenCL kernels over a given set of devices using an a-priori generated model based on machine learning. We demonstrated that not only the set of available devices and the executed program, but also the input size influences the performance of different workload distributions. We showed the effectiveness of machine learning based models in finding fast workload distributions [58, 95].

Data Layout Optimization In [93] we highlighted the importance of the data layout for the performance of programs executed on GPUs and presented a novel approach to deduce an optimized data layout for pairs of programs and GPUs using a two-step approach: The first is step based on the Kernel Data Layout Graph, the second step uses a decision tree. In combination, these two steps are effective at optimizing OpenCL programs for GPUs.

Multi-Objective Optimization of Parallel Programs To approach this problem we analyzed the trade-offs arising from a multi-objective environment with conflicting objectives. To find configurations that form a Pareto set, i.e. configurations that optimize the trade-off between different objectives, we developed a system that can automatically and effectively approximate the Pareto set for OpenMP programs in a multi-objective environment using iterative compilation with an auto-tuner based on general differential evolution with extensions.

Additionally, we demonstrated the advantages of massively parallel implementations for GPUs in OpenCL for two exemplary scientific domain applications, that arose from collaborations with other research institutes. The first one, published in [94] shows how OpenCL and modern GPUs can be used to accelerate the simulation of large mosquito populations. The publication arose from a collaboration with the Center for Research Computing at the University of Notre Dame, Indiana, USA. The second one, published in [96], shows how GPUs, OpenCL and techniques from computer vision can be applied to astrophysics simulations. It was developed as part of the Doktoratskolleg Plus Computational Interdisciplinary Modelling (DK+CIM) [166] in collaboration with the Astrophysics Group at the University of Innsbruck.

9.2 Future Work

The Insieme compiler and runtime system, which forms the base of most research presented in this thesis, is continuously developed and extended by the Distributed and Parallel Systems group at the University of Innsbruck. This ongoing development will allow the exploration of new research topics as additional features are added to the framework.

The task partitioning system presented in Chapter 5 could be extended to support systems with multiple, heterogeneous compute nodes and distributed memory, with the aid of a framework like LibWater [59]. The task partitioning system could also be expanded to optimize additional objectives, such as energy consumption or resource usage. Furthermore, the capabilities to accurately analyze and efficiently distribute OpenCL programs with multiple kernel functions on heterogeneous systems could be added. Another valid extension would be the support of multiple, device-optimized kernel function versions to further increase the performance of the program. A good candidate to create such device-optimized kernel function versions is the Insieme compiler.

The data layout transformation system presented in Chapter 6 is currently designed and tested only for GPUs. However, it could be adapted also for CPUs and accelerators. Additional benchmarks would be needed to see if the presented two step approach is suited also for CPUs and accelerators or if it should be adapted to support such processing units.

The most obvious extension of our auto-tuner presented in Chapter 7 would be the addition of more code transformations. Since the Insieme compiler is continuously extended, the number of available optimizations will increase over time. While adding additional transformations will further increase the search space, we are confident that the presented algorithm would still deliver solid performance. Additionally, the approach could be extended by adding other tunable parameters such as varying loop scheduling policies [160] or controlling the CPU's frequency [64]. Another useful extension of the auto-tuner would be the support of OpenCL programs as well as programs that can be distributed over multiple compute nodes.

List of Symbols

Notation	**Description**	**Page List**
A	The set of all task partitionings for a given set of devices	58, 63, 66
$Array$	A collection of an arbitrary number of elements	11
E	A set edges, connecting statements in S	10
$Energy$	The total amount of energy (kinetic plus potential energy)	128
F	The set of all available features	59
G	Set of features selected by the Greedy Feature Selection algorithm	59, 126
H	The set of training programs to generate a machine learning model	63
K	A Pareto set, consisting of several Pareto optimal program configurations.	95
$Mass$	The mass of a node	124, 126
Myr	A million years	128
P	The set of all programs	10
P_p	The set of all parallel programs	10
S	A set of statements	10
$Struct$	A collection of tuples consisting of a name and a data field	11
T	Set of all types	11
T_s	Set of all scalar types	11
$Tuple$	A collection of data fields with varying types	11
$V(K)$	The hypervolume of the Pareto set K.	95, 98, 102
VMH	Volume Mass Heuristic	xi, 120, 123–125
Vol	The volume of a node	124

Notation	**Description**	**Page List**
X	The set {0,10,20,30,40,50,60,70,80,90,100}	63
Δt	The size of the time step	127
Θ	The cell opening angle parameter	129
α	The tolerance parameter for the cell opening criterion	126, 129
δE	The relative energy error	128, 130, 142
γ	The RBF parameter for a SVM	66
$\mathrm{M}_\odot$	A million solar masses	128
a	A task partitioning over several devices	58
acc	The acceleration of a particle	126, 127
b	Achieved performance in percentage of optimal performance	64
c	Regulaization parameter of the SVM	66
d	The distance between the center of mass of two nodes	126
f	A feature	59
g	A feature selected by the Greedy Feature Selection algorithm	59
h	A training program for the machine learning model	63
k	A configuration for a program, consisting of a set of parameter values.	92, 95, 99–101
len	The largest side length of an axis aligned bounding box	126
m	An arbitrary natural number	79
mse	The mean squared error	59
n	An arbitrary natural number	58, 67, 79, 93, 95
p	A program $\in P$	10
q	A set of parameter values for a single code region.	92
r	A code region	91, 92
t	A type in T	11
t_{actual}	Achieved execution time	64
t_{best}	Optimal execution time	64
ts	A natural number representing the tile size of a loop or data structure.	79

Notation	**Description**	**Page List**
v	The velocity of a particle	127
x	The position of a particle	127
$\|K\|$	The number of program configurations in the Pareto set K.	95, 98
ANN	Artificial neural network	54, 60, 62, 66–69, 71
SVM	Support vector machine	54, 59, 60, 62, 66, 67, 69–71

List of Figures

List of Tables

List of Definitions

List of Algorithms

Bibliography

[1] Insieme Compiler Runtime Framework. `http://insieme-compiler.org/`.

[2] Insieme Source Code Repository. `https://github.com/insieme/insieme/tree/inspire_1.3`.

[3] OpenACC Application Program Interface. `http://openacc.org/`, 2012.

[4] Repast - recursive porus agent simulation toolkit, October 2013.

[5] Omni OpenMP Compiler website. `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/download/download-benchmarks.html`, 2016.

[6] Top 500 Supercomputer. `http://www.top500.org/lists/2016/06/`, 2016.

[7] Advanced Micro Devices Inc. Bolt. `http://hsa-libraries.github.io/Bolt/html/index.html`, 2013.

[8] Advanced Micro Devices Inc. OpenCL Optimization Case Study: Simple Reductions . `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/`, 2013.

[9] Saman P. Amarasinghe. Petabricks: a language and compiler based on autotuning. In *HiPEAC*, page 3, 2011.

[10] J. Ansel, C. Chan, Y.L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[11] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[12] Ryo Aoki, Shuichi Oikawa, Takashi Nakamura, and Satoshi Miki. Hybrid OpenCL: enhancing OpenCL for distributed processing. In *ISPA*, pages 149–154, 2011.

[13] Apple Inc. Clang/LLVM. `http://clang.llvm.org/`, 2012.

[14] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[15] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par*, pages 863–874, 2009.

[16] Prasanna Balaprakash, Ananta Tiwari, and Stefan Wild. Multi-objective optimization of hpc kernels for performance, power, and energy. In *4th International Workshop on Performance Modeling, Benchmarking, and Simulation of HPC Systems (PMBS12)*, 2013.

[17] A. Barak and A. Shilo. The Virtual OpenCL (VCL) Cluster Platform. In *Proc. Intel European Research & Innovation Conference*, page 196, 2011.

[18] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[19] K. E. Batcher. Sorting networks and their applications. In *Proc. of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS'68 (Spring), pages 307–314, New York, NY, 1968. ACM.

[20] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *J. Comput. Physics*, 231(7):2825–2839, 2012.

[21] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *The Journal of Political Economy*, 81, 1973.

[22] Blaise Barney. Message Passing Interface (MPI). `https://computing.llnl.gov/tutorials/mpi/`, 2016.

[23] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29:610–611, 1958.

[24] I. Buck, K. Fatahalian, and P. Hanrahan. GPUBench, 2004.

[25] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[26] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguad, and J. Labarta. Productive programming of GPU clusters with ompss. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 557–568, May 2012.

[27] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *Proc. of 1st Europ. Workshop on OpenMP*, pages 99–105, 1999.

[28] John Burkardt. HEATED_PLATE_OPENMP. `http://people.sc.fsu.edu/~jburkardt/c_src/heated_plate_openmp/heated_plate_openmp.html`, March 2016.

[29] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium*, 0:44–54, 2009.

[30] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[31] Shuai Che, Jiayuan Meng, and Kevin Skadron. Dymaxion++: a Directive-Based API to Optimize Data Layout and Memory Mapping for Heterogeneous Systems. AsHes'14, 2014.

[32] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *SC'11*, pages 13:1–13:11, New York, NY, 2011. ACM.

[33] Dehao Chen, Wenguang Chen, and Weimin Zheng. CUDA-Zero: a framework for porting shared memory GPU applications to multi-GPUs. *SCIENCE CHINA Information Sciences*, 55(3):663–676, 2012.

[34] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI'99*, pages 1–12, New York, NY, 1999. ACM.

[35] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[36] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[37] Simon Coakley, Marian Gheorghe, Mike Holcombe, Shawn Chin, David Worth, and Chris Greenough. Exploitation of high performance computing in the flame agent-based simulation framework. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, HPCC '12, pages 538–545, Washington, DC, USA, 2012. IEEE Computer Society.

[38] C.C. Coello, D.A. Van Veldhuizen, and G.B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems.* Genetic Algorithms and Evolutionary Computation. Springer US, 2013.

[39] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU '10: Proc.*, pages 63–74, New York, NY, USA, 2010. ACM.

[40] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU*, pages 63–74, 2010.

[41] Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. Fine-grained benchmark subsetting for system selection. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 132:132–132:142, New York, NY, USA, 2014. ACM.

[42] Yong Dong, Juan Chen, Xuejun Yang, et al. Energy-oriented OpenMP parallel loop scheduling. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on.* IEEE, 2008.

[43] J. J. Durillo, A. J. Nebro, C. A. C. Coello, J. Garcia-Nieto, F. Luna, and E. Alba. A study of multiobjective metaheuristics when solving parameter scalable problems. *IEEE Transactions on Evolutionary Computation*, 14(4):618–635, Aug 2010.

[44] Eckhoff PA. A malaria transmission-directed model of mosquito life cycle and ecology. *Malaria Journal*, 2011;12:303. doi: 10.1186/1475-2875-10-303, 2011.

[45] Vinoth Krishnan Elangovan, Rosa. M. Badia, and Eduard Ayguadé. *Scalability and Parallel Execution of OmpSs-OpenCL Tasks on Heterogeneous CPU-GPU Environment*, pages 141–155. Springer International Publishing, Cham, 2014.

[46] Erich Elsen, Mike Houston, Vaidyanathan Vishal, Eric Darve, Pat Hanrahan, and Vijay S. Pande. Poster reception - n-body simulation on GPUs. In *SC*, page 188, 2006.

[47] Ethem, Alpaydin. *Introduction to Machine Learning.* The MIT Press, Cambridge, MA, USA, 2004.

[48] K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D.R. Horn, L. Leem, J.Y. Park, M. Ren, A. Aiken, W.J. Dally, et al. Sequoia: programming the memory hierarchy. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 4–4. IEEE, 2006.

[49] Paul Feautrier and Christian Lengauer. *Encyclopedia of Parallel Computing*, chapter Polyhedron Model, pages 1581–1592. Springer, 2011.

[50] Nicolas Feltman, Minjae Lee, and Kayvon Fatahalian. Srdh: Specializing bvh construction and traversal order using representative shadow ray sets. In *High Performance Graphics*, pages 49–55, 2012.

[51] J.A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150. ACM, 1983.

[52] Vincent Freeh and David Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Proc. of the 10th ACM SIGPLAN PPoPP.* ACM, 2005.

[53] M. Frigo. A fast fourier transform compiler. In *Acm Sigplan Notices*, volume 34, pages 169–180. ACM, 1999.

[54] W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO 40.*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[55] Grigori Fursin, Abdul Memon, Cristophe Gillon, and Anton Lokhmotov. Collective mind, part ii: Towards performance- and cost-aware software engineering as a natural science. In *18th International Workshop on Compilers for Parallel Computing (CPC15)*, 2015.

[56] James E. Gentile, Gregory J. Davis, and Samuel S. C. Rund. Verifying agent-based models with steady-state analysis. *Computational and Mathematical Organization Theory*, 18:404–418, 2012.

[57] Anirban Ghose, Soumyajit Dey, Pabitra Mitra, and Mainak Chaudhuri. Divergence aware automated partitioning of OpenCL workloads. In *Proceedings of the 9th India Software Engineering Conference*, ISEC '16, pages 131–135, New York, NY, USA, 2016. ACM.

[58] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 281–282, 2013.

[59] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. LibWater: heterogeneous distributed computing made easy. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 161–172, New York, NY, USA, 2013. ACM.

[60] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, , and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *InPar*, 2012.

[61] Chris Gregg and Kim M. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, pages 134–144, 2011.

[62] Dominik Grewe and Michael F.P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *CC*, 2011.

[63] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. *OpenCL Task Partitioning in the Presence of GPU Contention*, pages 87–101. Springer International Publishing, Cham, 2014.

[64] Philipp Gschwandtner, Charalampos Chalios, Dimitrios S. Nikolopoulos, Hans Vandierendonck, and Thomas Fahringer. On the potential of significance-driven execution for energy-aware hpc. *Computer Science - Research and Development*, 30(2):197–206, 2015.

[65] Philipp Gschwandtner, Juan J. Durillo, and Thomas Fahringer. *Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage*, pages 87–98. Springer International Publishing, Cham, 2014.

[66] Weidong Gu and Robert J. Novak. Agent-based modelling of mosquito foraging behaviour for malaria control. In *Transactions of the Royal Society of Tropical Medicine and Hygiene*, volume 103, pages 1105–1112. Oxford University Press, 2009.

[67] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[68] Tsuyoshi Hamada and Keigo Nitadori. 190 tflops astrophysical n-body simulation on a cluster of GPUs. In *SC*, pages 1–9, 2010.

[69] Julia Handl, Simon C. Lovell, and Joshua Knowles. Multiobjectivization by decomposition of scalar cost functions. In *Proceedings of the International Conference on Parallel Problem Solving from Nature, 2008*, 2008.

[70] J. Harnois-Déraps, U.-L. Pen, I. T. Iliev, H. Merz, J. D. Emberson, and V. Desjacques. High-performance P^3M N-body code: CUBEP^{3}M. *MNRAS*, 436:540–559, November 2013.

[71] Lars Hernquist. An analytical model for spherical galaxies and bulges. *The Astrophysical Journal*, 356:359, June 1990.

[72] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: writing parallel program portable between CPU and GPU. In *PACT*, pages 217–226, 2010.

[73] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proc. of the 6th Intl. Symposium on Code generation and optimization.* ACM, 2008.

[74] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. Practical structure layout optimization and advice. In *CGO'06*, pages 233–244, Washington, DC, 2006. IEEE Computer Society.

[75] Piet Hut, Jeffrey M. Arnold, Junichiro Makino, Stephen L.W. McMillan, and Thomas L. Sterling. Grape-6: A petaflops prototype. In *proceedings of the 1997 Petaflops Algorithms Workshop (PAL'97)*, 1997.

[76] Institut für Neuroinformatik, Ruhr-University Bochum. Shark Machine Learning Library. `http://shark-project.sourceforge.net/`, 2012.

[77] MPI Intel. Benchmarks: Users Guide and Methodology Description. *Intel GmbH, Germany*, 2004.

[78] Carlos Isidoro, Nuno Fachada, Fbio Barata, and Agostinho Rosa. Agent-Based Model of Aedes aegypti Population Dynamics. In Lus Seabra Lopes, Nuno Lau, Pedro Mariano, and Lus Mateus Rocha, editors, *Progress in Artificial Intelligence, 14th Portuguese Conference on Artificial Intelligence, EPIA 2009, Aveiro, Portugal, October 12-15, 2009. Proceedings*, volume 5816 of *Lecture Notes in Computer Science*, pages 53–64. Springer, 2009.

[79] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP'95*, pages 179–188, New York, NY, 1995. ACM.

[80] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on GPU clusters. In *SC*, pages 1–11, 2010.

[81] Herbert Jordan. *Insieme: A Compiler Infrastructure for Parallel Programs.* PhD thesis, University of Innsbruck, 8 2014.

[82] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. INSPIRE: the insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pages 7–17, 2013.

[83] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[84] Ma Kai, Li Xue, Chen Wei, Zhang Chi, and Wang Xiaorui. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *ICPP*, 2012.

[85] M. Kandemir, A Choudhary, J. Ramanujam, and P. Banerjee. A framework for interprocedural locality optimization using both loop and data layout transformations. In *Proc. of the Int. Conference on Parallel Processing*, pages 95–102, 1999.

[86] S. Kanur, W. Lund, L. Tsiopoulos, and J. Lilius. Determining a device crossover point in CPU/GPU systems for streaming applications. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1417–1421, Dec 2015.

[87] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and kd trees. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics*, pages 33–37. The Eurographics Association, 2012.

[88] Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee. *Multicore Processors and Systems.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[89] Khronos OpenCL Working Group. The OpenCL 1.2 specification. `http://www.khronos.org/opencl`, 2015.

[90] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. OpenCL as a unified programming model for heterogeneous CPU/GPU clusters. In *PPoPP*, pages 299–300, 2012.

[91] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *ICS*, pages 341–352, 2012.

[92] Klaus Kofler. SAMPO source code repository. `https://github.com/klois/SAMPO`.

[93] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic data layout optimizations for GPUs. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 263–274, 2015.

[94] Klaus Kofler, Gregory J. Davis, and Sandra Gesing. SAMPO: an agent-based mosquito point model in OpenCL. In *Proceedings of the Agent-Directed Simulation Symposium, part of the 2014 Spring Simulation Multiconference, SpringSim '14, Tampa, FL, USA, April 13-16, 2014*, page 5, 2014.

[95] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*, pages 149–160, 2013.

[96] Klaus Kofler, Dominik Steinhauser, Biagio Cosenza, Ivan Grasso, Sabine Schindler, and Thomas Fahringer. Kd-tree based n-body simulations with volume-mass heuristic on the GPU. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 1256–1265, 2014.

[97] Joseph B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. of the American Mathematical Society*, 7(1):48–50, 1956.

[98] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. Archit. Code Optim.*, 6(1):1:1–1:36, April 2009.

[99] P. E. Kyziropoulos, C. K. Filelis-Papadopoulos, and George A. Gravvanis. N-body simulation based on the particle mesh method using multigrid schemes. In *FedCSIS*, pages 471–478, 2013.

[100] James H. Laros, III, Kevin T. Pedretti, Suzanne M. Kelly, Wei Shu, and Courtenay T. Vaughan. Energy based performance tuning for large scale high performance computing systems. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 6:1–6:10, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[101] P. L'Ecuyer. Maximally Equidistributed Combined Tausworthe Generators. *Mathematics of Computation*, 65:203–213, 1996.

[102] Dong Li, Bronis R. de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk W. Cameron. Strategies for energy-efficient resource management of hybrid programming models. *IEEE Trans. Parallel Distrib. Syst.*, 24(1):144–157, 2013.

[103] Shengren Li, Lance Simons, Jagadeesh Bhaskar Pakaravoor, Fatemeh Abbasinejad, John D. Owens, and Nina Amenta. kann on the GPU with shifted sorting. In *High Performance Graphics*, pages 39–47, 2012.

[104] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Y. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS*, pages 287–296, 2008.

[105] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, pages 45–55, 2009.

[106] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, July 2005.

[107] Lianjie Luo, Yang Chen, Chengyong Wu, Shun Long, and Grigori Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. *arXiv preprint arXiv:1407.4075*, 2014.

[108] Robert F. Lyerly. *Automatic Scheduling of Compute Kernels Across Heterogeneous Architectures.* PhD thesis, Virginia Polytechnic Institute and State University, 5 2014.

[109] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Comp. Soc. Tech. Comm. on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[110] R. Membarth, F. Hannig, J. Teich, M. Krner, and W. Eckert. Generating device-specific GPU code for local operators in medical imaging. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 569–581, May 2012.

[111] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. *Mastering Software Variant Explosion for GPU Accelerators*, pages 123–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[112] I. D. Mironescu and L. Vinan. Coloured petri net modelling of task scheduling on a heterogeneous computational node. In *2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 323–330, Sept 2014.

[113] Technische Universität München. AutoTune, Automatic Online tuning. `http://www.autotune-project.eu/`, 2016.

[114] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda. *Software Automatic Tuning (From Concepts to State-of-the-Art Results).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2010.

[115] NASA. The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB/`, 2016.

[116] Hubert Nguyen. *GPU Gems 3.* Addison-Wesley Professional, first edition, 2007.

[117] M. J. North, N. T. Collier, and R. J. Vos. Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation*, 16(1):1–25, 2006.

[118] NVIDIA Corporation. CUDA Programming Model. `https://developer.nvidia.com/cuda-toolkit`, 2012.

[119] NVIDIA Corporation. What do K20 users think? `http://www.nvidia.com/docs/IO/122634/K20-testimonial.pdf`, 2013.

[120] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31. 2007.

[121] M. F. P. O'Boyle and P. M. W. Knijnenburg. Efficient parallelization using combined loop and data transformations. In *PACT'99*, pages 283–, Washington, DC, 1999. IEEE Computer Society.

[122] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[123] OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2015.

[124] Alexandros Panagiotidis, Daniel Kauker, Filip Sadlo, and Thomas Ertl. Distributed computation and large-scale visualization in heterogeneous compute environments. In *Proceedings of the 11th International Symposium on Parallel and Distributed Computing*, 2012.

[125] Karl Pearson. On lines and planes of closest fit to a system of points in space. In *Philosophical Magazine, Series 6, vol. 2, no. 11*, pages 557–572, 1901.

[126] Joshua Peraza, Ananta Tiwari, Michael Laurenzano, et al. PMaC's green queue: A framework for selecting energy optimal DVFS configurations in large scale MPI applications. *Submission to CCPE Special Issue on Analysis of Performance and Power for Highly Parallel Systems*, 2012.

[127] Carolyn L. Phillips, Joshua A. Anderson, and Sharon C. Glotzer. Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices. *Journal of Computational Physics*, 230(19):7191 – 7201, 2011.

[128] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[129] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[130] T. Quinn, N. Katz, J. Stadel, and G. Lake. Time stepping N-body simulations. *ArXiv Astrophysics e-prints*, October 1997.

[131] Mohammed Rahman, Louis-Noël Pouchet, and P. Sadayappan. Neural network assisted tile size selection. In *International Workshop on Automatic Performance Tuning (IWAPT'2010)*, Berkeley, CA, June 2010. Springer Verlag.

[132] Shah Rahman, Jichi Guo, Akshatha Bhat, et al. Studying the impact of application-level optimizations on the power consumption of multi-core architectures. In *Proc. of the 9th conference on Computing Frontiers.* ACM, 2012.

[133] Easwaran Raman, Robert Hundt, and Sandya Mannarswamy. Structure layout optimization for multithreaded programs. In *CGO'07*, pages 271–282, Washington, DC, 2007. IEEE Computer Society.

[134] P. Richmond, S. Coakley, and D. Romano. Cellular level agent based modelling on the graphics processing unit. In *High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on*, pages 43–50, 2009.

[135] Rodinia. LavaMD, November 2014.

[136] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL'02*, pages 140–153, New York, NY, 2002. ACM.

[137] RULEQUEST RESEARCH. Data mining tools see5 and c5.0, October 2014.

[138] Kamal Sharma, Ian Karlin, Jeff Keasler, James R. McGraw, and Vivek Sarkar. User-specified and automatic data layout selection for portable performance. Technical report, Lawrence Livermore National Laboratory, 2013.

[139] Fadi N. Sibai. Performance analysis and workload characterization of the 3dmark05 benchmark on modern parallel computer platforms. *ACM SIGARCH Computer Architecture News*, 35(3):44–52, 2007.

[140] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, December 2005.

[141] Volker Springel, Naoki Yoshida, and Simon D.M. White. GADGET: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6(2):79–117, April 2001.

[142] Robert Springer, David Lowenthal, Barry Rountree, et al. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. In *Proc. of the eleventh ACM SIGPLAN PPoPP*. ACM, 2006.

[143] J. G. Stadel. *Cosmological N-body simulations and their analysis.* PhD thesis, University Of Washington, 2001.

[144] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[145] Rainer Storn and Kenneth Price. Differential evolution &ndash; a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, December 1997.

[146] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Li-Wen Chang, Nasser Anssari, Geng (Daniel) Liu, Wen mei W. Hwu, and Nady Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *IEEE Computer*, 45(8):26–32, 2012.

[147] Magnus Strengert, Christoph Muller, Carsten Dachsbacher, and Thomas Ertl. Cudasa: Compute unified device and systems architecture. In *EGPGV*, pages 49–56, 2008.

[148] Robert Strzodka. Data layout optimization for multi-valued containers in OpenCL. *J. Parallel Distrib. Comput.*, 72(9):1073–1082, 2012.

[149] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli. 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, 2012.

[150] I-Jui Sung, Nasser Anssari, John A. Stratton, and Wen mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. *International Journal of Parallel Programming*, 40(1):4–24, 2012.

[151] C. Tapus, I.H. Chung, and J.K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 44–44. IEEE, 2002.

[152] A. Tarakji and N. O. Salscheider. Runtime behavior comparison of modern accelerators and coprocessors. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 97–108, May 2014.

[153] R. C. Tausworthe. Random Numbers Generated by Linear Recurrence Modulo Two. *Mathematics of Computation*, 19:201–209, 1965.

[154] TechTarget . integrated circuit (IC). `http://whatis.techtarget.com/definition/integrated-circuit-IC`, 2005.

[155] TechTarget. Arithmetic-logic unit. `http://whatis.techtarget.com/definition/arithmetic-logic-unit-ALU`, 2005.

[156] TechTarget. program counter. `http://whatis.techtarget.com/definition/program-counter`, 2012.

[157] TechTarget. operating system (OS). `http://whatis.techtarget.com/definition/operating-system-OS`, 2014.

[158] The Khronos Group Inc. Khronos group. `https://www.khronos.org/`. Accesses: 08.02.2015.

[159] Peter Thoman. *Insieme-RS: A Compiler-supported Parallel Runtime System.* PhD thesis, University of Innsbruck, 7 2013.

[160] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. *Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach*, pages 88–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[161] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic OpenCL device characterization: guiding optimized kernel design. In *Euro-Par*, pages 438–452, 2011.

[162] Ananta Tiwari, Michael Laurenzano, Laura Carrington, et al. Auto-tuning for energy usage in scientific applications. In *Euro-Par 2011: Parallel Processing Workshops.* Springer, 2012.

[163] J. Torrellas, M.S. Lam, and J.L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[164] Ehsan Totoni, Mert Dikmen, and María Jesús Garzarán. Easy, fast, and energy-efficient object detection on heterogeneous on-chip architectures. *ACM Trans. Archit. Code Optim.*, 10(4):45:1–45:25, December 2013.

[165] Ying-Yu Tseng, Yu-Hao Huang, Bo-Cheng Charles Lai, and Jiun-Liang Lin. Automatic Data Layout Transformation for Heterogeneous Many-Core Systems. In Ching-Hsien Hsu, Xuanhua Shi, and Valentina Salapura, editors, *11th IFIP International Conference on Network and Parallel Computing (NPC)*, volume LNCS-8707 of *Network and Parallel Computing*, pages 208–219, Ilan, Taiwan, September 2014. Springer. Part 2: Parallel and Multi-Core Technologies.

[166] University of Innsbruck. Doctoral Programme Computational Interdisciplinary Modelling (DK CIM). `https://www.uibk.ac.at/dk-cim/`, 2016.

[167] Jan Verschelde. Memory Coalescing Techniques. `http://homepages.math.uic.edu/$\sim$jan/mcs572/memory_coalescing.pdf`, 2012.

[168] V. Volkov and James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *SC '08: Proc.*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[169] R. Vuduc, J.W. Demmel, and K.A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[170] J. W. Wadsley, J. Stadel, and T. Quinn. Gasoline: a flexible, parallel implementation of TreeSPH. *New astronomy*, 9:137–158, February 2004.

[171] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[172] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In Dieter Schmalstieg and Jiří Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, September 2007.

[173] M. S. Warren and J. K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing '92, pages 570–576, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[174] Michael S. Warren. 2hot: An improved parallel hashed oct-tree n-body algorithm for cosmological simulation. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 72:1–72:12, New York, NY, USA, 2013. ACM.

[175] Nicolas Weber and Michael Goesele. Auto-tuning complex array layouts for GPUs. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV14. EG.

[176] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[177] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246, 2010.

[178] P. y. Li, Q. h. Zhang, R. c. Zhao, and H. n. Yu. Data layout transformation for structure vectorization on simd architectures. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–7, June 2015.

[179] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI'04*, pages 255–266, New York, NY, 2004. ACM.

[180] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.

[181] Ying Zhou, S. M. Niaz Arifin, James Gentile, Steven J. Kurtz, Gregory J. Davis, and Barbara A. Wendelberger. An agent-based model of the anopheles gambiae mosquito life cycle. In *Proceedings of the 2010 Summer Computer Simulation Conference*, SCSC '10, pages 201–208, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[182] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *Trans. Evol. Comp*, 3(4):257–271, November 1999.