GPU Array Access Auto-Tuning

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor Ingenieur (Dr.-Ing.)

vorgelegt von
Nicolas Weber, M.Sc.
geboren in Hanau.

Referenten:	Prof. Dr.-Ing. Michael Goesele Technische Universität Darmstadt
	Prof. Dr. Michael Gerndt Technische Universität München

Tag der Einreichung: 13.04.2017
Tag der Disputation: 19.06.2017

Darmstädter Dissertation, 2017
D 17

Erklärung zur Dissertation

Hiermit versichere ich die vorliegende Dissertation selbstständig nur mit den angegeben Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 13.04.2017

Nicolas Weber

Abstract

Graphics Processing Units ***(GPUs)*** have been used for years in compute intensive applications. Their massive parallel processing capabilities can speedup calculations significantly. However, to leverage this speedup it is necessary to rethink and develop new algorithms that allow parallel processing. These algorithms are only one piece to achieve high performance. Nearly as important as suitable algorithms is the actual implementation and the usage of special hardware features such as intra-warp communication, shared memory, caches, and memory access patterns. Optimizing these factors is usually a time consuming task that requires deep understanding of the algorithms and the underlying hardware. Unlike *Central Processing Units* ***(CPUs)***, the internal structure of GPUs has changed significantly and will likely change even more over the years. Therefore it does not suffice to optimize the code once during the development, but it has to be optimized for each new GPU generation that is released. To efficiently (re-)optimize code towards the underlying hardware, auto-tuning tools have been developed that perform these optimizations automatically, taking this burden from the programmer. In particular, NVIDIA – the leading manufacturer for GPUs today – applied significant changes to the memory hierarchy over the last four hardware generations. This makes the memory hierarchy an attractive objective for an auto-tuner.

In this thesis we introduce the MATOG auto-tuner that automatically optimizes array access for NVIDIA CUDA applications. In order to achieve these optimizations, MATOG has to analyze the application to determine optimal parameter values. The analysis relies on empirical profiling combined with a prediction method and a data post-processing step. This allows to find nearly optimal parameter values in a minimal amount of time. Further, MATOG is able to automatically detect varying application workloads and can apply different optimization parameter settings at runtime. To show MATOG's capabilities, we evaluated it on a variety of different applications, ranging from simple algorithms up to complex applications on the last four hardware generations, with a total of 14 GPUs. MATOG is able to achieve equal or even better performance than hand-optimized code. Further, it is able to provide performance portability across different GPU types (low-, mid-, high-end and HPC) and generations. In some cases it is able to exceed the performance of hand-crafted code that has been specifically optimized for the tested GPU by dynamically changing data layouts throughout the execution.

Zusammenfassung

Graphics Processing Units (**GPUs**) werden seit Jahren für berechnungsintensive Anwendungen eingesetzt. Ihre massiv-parallele Rechenleistung kann Berechnungen signifikant beschleunigen. Um diese Beschleunigung zu erreichen ist es notwendig, dass Algorithmen überarbeitet oder neu entwickelt werden, um parallele Berechnungen zu ermöglichen. Diese Algorithmen jedoch sind nur ein Teil um hohe Berechnungsgeschwindigkeiten zu erreichen. Genauso wichtig wie raffinierte Algorithmen, ist die eigentliche Implementierung und die Nutzung von speziellen Komponenten wie Intrawarp Kommunikation, geteilte Speicher, Zwischenspeicher und Speicherzugriffsmuster. Diese Faktoren zu optimieren ist üblicherweise eine zeitintensive Aufgabe, welche ein umfassendes Verständnis der Algorithmen und des Beschleunigers erfordert. Anders als bei *Central Processing Units* (**CPUs**) hat sich die interne Struktur von GPUs in den letzten Jahren stark verändert und wird sich mit Sicherheit weiterentwickeln. Deshalb reicht es nicht aus, Programme nur während der Entwicklung zu optimieren. Um effizient Programme für ein bestimmtes Gerät zu optimieren wurden Auto-Tuner entwickelt, welche diese Optimierungen automatisch durchführen und somit die Programmierer entlasten. NVIDIA – der führende Hersteller von GPUs – hat in letzten vier Generationen signifikante Änderungen an der Speicherhierarchie vorgenommen. Dies macht die Speicherhierarchie zu einem attraktiven Ziel für einen Auto-Tuner.

In dieser Arbeit stellen wir den MATOG Auto-Tuner vor, welcher automatisch Arrayzugriffe in NVIDIA CUDA Anwendungen optimiert. Um diese Optimierungen zu erreichen, muss die Anwendung analysiert und optimale Parameter gefunden werden. Diese Analyse basiert auf empirischen Messungen kombiniert mit einer Vorhersagemethode und einer Datennachverarbeitung. Dies erlaubt es nahezu optimale Parameter in kürzester Zeit zu finden. MATOG ist darüber hinaus in der Lage verschiedene Programmzustände zu erkennen und unterschiedliche Optimierungen zur Laufzeit anzuwenden. Um die Fähigkeiten von MATOG zu belegen haben wir eine Auswahl von simplen und komplexen Anwendungen auf den letzten vier Hardware Generationen mit insgesamt 14 verschiedenen GPUs getestet. MATOG ist in der Lage äquivalente, bzw. teilweise auch bessere Leistung als handoptimierte Implementierungen zu erreichen. Weiterhin bietet es Leistungsportabilität über verschiedene GPU Typen und Generationen. In einigen Fällen kann MATOG die Leistung von handoptimiertem Code übertreffen, indem es dynamisch die Speicherlayouts zur Laufzeit anpasst.

Acknowledgements

First of all, I would like to thank my supervisor Michael Goesele for the opportunity to work in his research group for the last four years, his support and valuable feedback. The work environment he created is challenging and sometimes exhausting but exactly this makes it fruitful and pushes one to seek and achieve even more. It was a pleasure to work with him for all these years.

I would also like to extend my sincere gratitude to Michael Gerndt who kindly agreed to be referee for this thesis.

Next, I would like to give special thanks to my colleagues, starting with Martin Hess. We shared an office for all these years and had many inspiring/funny discussions! One great hug goes to Michael Wächter who never gave up on me and pushed me all the way through the obstacles of the *Detail-Preserving Image Downscaling **(DPID)*** paper [Weber et al. 2016]. Further, I want to thank Dominik Wodniok, not only for all the discussions we had, but also his valuable feedback for MATOG and DPID. A special thanks goes to Sandra C. Amend for her support in all the annoying programming tasks that had to be done.

Finally, I want to thank my parents who always supported me, my brothers and all my friends for their help and support.

The work of Nicolas Weber is supported by the ‘Excellence Initiative’ of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

Contents

Chapter 1

Introduction

Writing efficient code is one of the major objectives for programmers, besides the correctness of the calculations. However, efficiency covers multiple factors, such as *time*, *energy* and *cost*. Depending on the application the focus is shifted between these factors. For example, gaming hardware is tuned to deliver highest performance rather to have a low energy consumption [Mills and Mills 2015]. The costs have to be moderate, so that a private person can afford the computer. Supercomputers also have striven for highest available performance [Top 500 2016] for many years. However, today many supercomputers seek to provide high performance with low energy consumption [Green 500 2016], to keep the costs reasonable. In general, costs consist of the initial expenses for the hardware, maintenance costs, power consumption (which can be several million dollars per year for supercomputers) and costs for developing and optimizing the software. To reduce the costs, on the one hand applications have to be time and energy efficient to optimize the utilization and lower the energy consumption. On the other hand, this increases the costs for development and optimization. Unfortunately, code efficiency always depends on the underlying hardware. It has to be specifically designed for the hardware it is running on. This requires that experts with a deep understanding of both, the hardware and the application, optimize the code, which again increase the costs [Bischof et al. 2012].

The first computers have been executing a single application on a single threaded processor [Goodacre 2011]. With the upcoming of Intel's x86 architecture, it became the standard architecture for many years [Levenson 2013]. It caused that software was mainly developed for this particular architecture. For years this worked, as technological advances allowed to develop faster processors without changing the architecture significantly. This effect had been predicted by Moore [1965] which became to known as *"Moore's Law"*. However, these advances had stalled and introduced a demise of Moore's Law [Berkeley 2014] (Section 2.2.2). To further improve the performance of processors, other methods had to be found. One of these is parallel computation. Parallel processors can have up to several thousand compute cores operating in parallel. This parallel processing power comes with the necessity of writing efficient code that enables all cores of the processors to solve a problem together. Therefore, not only algorithms have to be rethought and specifically designed for parallel processing but also the actual implementation has to utilize available hardware features. Today, the market is filled with all kinds of processors, specifically designed to excel in a particular field. There are all kinds of variations of processors ranging from few, but fast compute cores (multi-core CPUs) to processors with thousands, but rather slow cores (GPUs). Also special purpose processors are available, specifically tuned for a specific purpose, e.g., Google's *Tensor Processing Unit* ***(TPU)*** [Jouppi 2016] that is optimized for neural network processing. This variety of processors poses a challenge to software developers as they no longer have only one dominant architecture their software has to be developed and optimized for. They have to deal with varying processor designs, number of processing cores, specialized hardware functionality, programming languages, library support and operating systems.

Further, hardware architectures (even from the same manufacturer) undergo constant changes and improvements, introducing new, changed or removed functionality. NVIDIA is the leading manufacturer for GPUs today [Shilov 2016]. NVIDIA's GPUs consist of a complex memory hierarchy with a series of different automatic and self-organized caches that need to be efficiently used (Section 3.1.1). In the last four GPU generations, many significant changes to this memory hierarchy have been applied (Section 3.2) so that code written for prior generations usually does not necessarily work as efficient as it could on newer generations. For example, NVIDIA added the option to define a trade-off between having more self-managed shared memory or automatic L1 cache in their Fermi architecture [NVIDIA 2009]. The next generation [NVIDIA 2014a] added more trade-off options to choose. In the third generation [NVIDIA 2014b], this feature was entirely removed. So in three consecutive hardware generations (2009-2014) of the same manufacturer's hardware, the behavior has been constantly changed. Another example are vector processing units in CPUs. *Advanced Vector Extensions* ***(AVX)*** 1.0 [Reinders 2013] were added in the 2nd generation of Intel's i7 processors [INTEL 2011]. In the 4th generation [INTEL 2013a] AVX 2.0 followed. As AVX is not backward compatible, AVX 2.0 instructions cannot be used on any older CPU, so that programmers have to explicitly check for the capabilities of the CPU their software is running on.

One of the major problems of parallel programming is data access. As the time to access a single memory cell has not really improved over the years [CRUCIAL 2015], today's memory systems read not a single data cell but entire blocks. These are then transfered to the processor using wide memory bus systems to push through the needed amounts of data. This method is only efficient, if not only a single core, but (in the best case) all cores are satisfied by the data provided by this transfer. If not all cores get the data they are seeking, the performance significantly drops as all others have to wait until the next data transmission. Therefore, it is necessary that the access to data is optimized to satisfy as many cores as possible with the data within one block. Unfortunately optimizing the utilization of these memory blocks is not the only optimization objective. Different data layouts can have varying computational and resource requirements. In some layouts the location of a memory cell can be obtained quite easily, while others either require more calculations or more temporary registers to determine the correct location (Section 2.3.2). Using a different layout could improve the utilization of the data block, but in the same turn, could increase the resource usage. This can be a problem for GPUs, as their cores are quite limited in their computational and register resources. This additional consumption can reduce the overall performance of the cores, which would result in a lower overall performance, although the memory utilization is better. Therefore it is necessary to find a good balance between both optimization goals.

In general, this variety of hardware architectures and software environments overwhelms the abilities of many programmers, especially when they are no hardware enthusiasts, but scientific programmers with a biologic, physics, chemical-, mechanical-, or electrical-engineering background. It has been argued that more experienced programmers are therefore needed to tune code to keep a high efficiency, especially for supercomputers [Bischof et al. 2012]. An alternative way beside manual optimization is to develop tools that apply these optimizations automatically. This so-called *"auto-tuning"* of software has been an active research topic for several years and is still flourishing, as the publication rates in Figure 1.1 indicate. The idea of auto-tuning is that software adjusts itself to the underlying hardware, without any manual interaction. In general there are multiple goals for auto-tuning. First of all, it is supposed to find an optimal implementation automatically, if possible, in less time than if done manually. Second, the software should adjust itself to the hardware it is running on, not to the hardware it was developed for. So the auto-tuning has to be usable by the customer and not only by the developers. Third, auto-tuning should provide performance portability across different hardware types and generations, if possible even for unknown future hardware. However, support for future hardware can also be added by updating the auto-tuner, as long as this does not require any adjustments to the application code.

To summarize: parallel processors such as GPUs significantly suffer from bad data access. As many programmers are overwhelmed by the complexity of programming and optimizing code for specific hardware, we developed an auto-tuner in this thesis, that helps all kinds of programmers (independent of the skill level) to overcome the obstacles of optimizing GPU code. We set the following goals for the auto-tuner. **First**, the auto-tuner should optimize array access in NVIDIA GPU applications independent of the used hardware and application domain. A general approach is important, to make it available for a wide range of applications and not to limit it to a specific kind of GPUs, hardware generation or application domain.

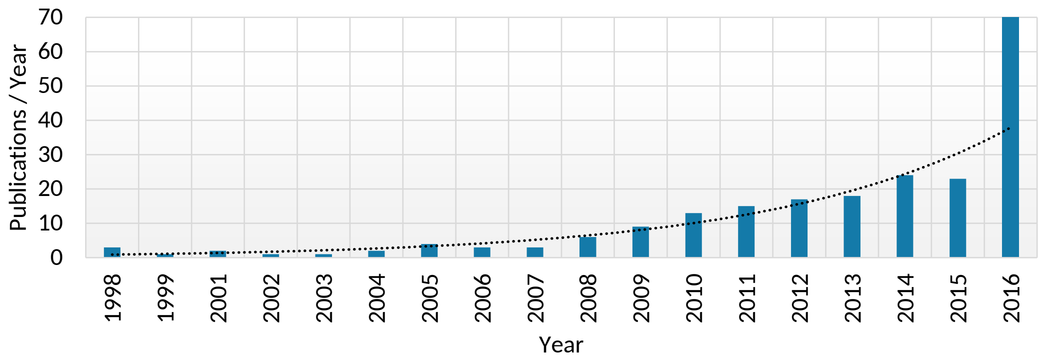


Figure 1.1: Number of scientific auto-tuning publications over the years. An increasing trend can be seen. (Statistics are taken over the papers that we discuss in Chapter 4)

Second, only a minimal time effort should be required analyzing the application, to find nearly optimal parameters. Software that requires hours or days to find suitable optimizations will probably not be adopted by developers, as they cannot afford to wait too long for the auto-tuner during the development. **Third**, the auto-tuner should achieve at least equal, but in the best case, higher performance than hand-optimized code. **Fourth**, it is supposed to provide performance portability across multiple GPU generations without code adjustments. This ensures that software can be used on different hardware without any manual optimization interaction. **Fifth**, the effort for the developer to integrate the auto-tuner into the application should be minimal.

1.1 Contributions

To show the applicability of our techniques, we have developed the *"MATOG: Auto-Tuning on GPUs"* ***(MATOG)*** auto-tuner in this thesis. It optimizes the array access and utilization of the memory hierarchy for NVIDIA *Compute Unified Device Architecture* ***(CUDA)*** applications. The main contributions of this thesis have been published as peer reviewed papers in a series of international conferences [Weber and Goesele 2014; Weber et al. 2015; Weber and Goesele 2016] and journals [Weber and Goesele 2017].

The first problem we faced was to design an *Application Programming Interface* ***(API)*** that allowed to integrate MATOG into CUDA applications with little effort. Our API is divided into two components. The first mimics the CUDA Driver API, which enables to easily use existing CUDA code in MATOG. The second uses code generation to create data structures that are tailor made to the needs of the application [Weber and Goesele 2014].

Next we had to analyze the application and how different data layouts impact the performance. We chose to use empirical profiling as this allows to get accurate execution times, without the need of models, which could break whenever a new GPU generation is released. However, as MATOG applications can have more than a million different parameter configurations, it is unfeasible to perform an exhaustive search. So we developed a specialized three step analysis method.

In the first step we execute the application. Every time a GPU kernel is executed, we run the kernel in different implementations, measuring the time and store the results in a database. For this we developed a specialized prediction method, that only needs to measure the time for a small fraction of possible configurations to estimate the performance of the entire solution space [Weber et al. 2015].

With this data we can determine optimal configurations for each kernel. However, as data is shared between kernels, we have to find configurations that use the same data layouts for shared arrays, so we have to find a solution for the entire application, that is optimal. For this we developed a dependency graph based method, that puts the kernel executions into relation [Weber and Goesele 2016]. As this graph can be very complex, it was necessary to develop a method that is able to find the optimum of the graph in short time. We reused knowledge from our prediction method to speed up the processing [Weber and Goesele 2017].

Having a application wide optimal solution, however, did not suffice as optimal data access does not only depend on the used algorithms or how the code is written, but also on the actual data. This causes that a single kernel can have different optimal solutions, depending on the data. To handle such effects, we automatically gather meta data during the profiling and use it to construct decision models, that can react to data dependent effects at runtime. For this MATOG continuously monitors certain parameters and adjusts the data layouts accordingly [Weber and Goesele 2016; Weber and Goesele 2017].

This thesis is mainly based on our fourth paper [Weber and Goesele 2017], but adds additional information and evaluation results. Further, we show results for experiments we have conducted to generate performance models based on automatically gathered profiling and meta data [Amend 2017]. These performance models show promising results but have not made it into the active development of MATOG yet.

1.2 Thesis Outline

We start this thesis by introducing the basics of today's compute hardware, memory types and hierarchies, how these theoretical concepts are implemented in today's hardware and conclude with different methods to store arrays (Chapter 2). We continue by introducing our target platform (Chapter 3), including the NVIDIA compute model, the used programming language and the differences of the memory hierarchy in the last four NVIDIA GPU generations. As a next step we define the term auto-tuning, what it stands for and discuss methods that have been used by prominent auto-tuners. We continue with a discussion of the different obstacles auto-tuners have to deal with and how these have been addressed in literature. Finally, we give an overview of the state-of-the-art in auto-tuning. While we mainly concentrate on GPUs in this thesis, we also include papers for other hardware, as the concepts usually stay the same (Chapter 4). Then we introduce the main ideas of MATOG, show programming examples and provide details of the implementation itself (Chapter 5). In Chapter 6 we explaining our multi-step application analysis. It starts with profiling the application using our prediction based algorithm. The gathered data is then analyzed in an offline analysis step utilizing a specialized data and execution dependency graph. The graph is used to model the relation between multiple kernel calls. This allows us to select optimized layouts according to the runtime ratio of the different kernels. Finally, we construct decision models that can be used during runtime to determine optimized configurations according to the current application workload. At the end of the chapter we explain how the MATOG runtime system works and how it gathers meta data during the execution to facilitate the adaptive decision-making. In our evaluation (Chapter 7) we apply MATOG on a series of different benchmark applications, ranging from simple algorithms up to very complex applications with changing workload. All tests are performed on 14 GPU from the last four NVIDIA GPU architectures. Before we discuss our results, we present techniques that could be integrated into MATOG to improve the decision-making (Chapter 8). Explicitly, we explore options for generating automated performance models and their usefulness. This work has been performed together with Sandra C. Amend [Amend 2017]. In Chapter 9 we reflect what can be learned from our experiments. We draw conclusions from our results, summarize our contributions, reflect our proposed methods and if there is space for improvements. Finally, we identify open issues for future research and outline directions that auto-tuning could pursue (Chapter 10).

Chapter 2
Background

This chapter gives an overview of technologies and methods used in this thesis. We start with a high-level view of computers and the definition of important terminology (Section 2.1). These are the foundations for the hardware (Section 2.2) that we target in this thesis. In Section 2.3 we introduce different ways to store data in arrays and how these differ in implementation, resource and computational requirements.

2.1 Computational Basics

First we give an overview over the structure of computers, their components and concepts. These are the foundations for the actual hardware implementations that we explain in Section 2.2 and the target architecture in this thesis (Chapter 3). Most of the information in this section is taken from Patterson and Hennessy [2013]. A computer usually consists of a *processor*, which performs all calculations and controls the operation of the computer, a *main memory* that is used to store temporary data, a *mass storage device* that permanently stores data and a *bus system* that interconnects all of these components. There are also other components available, such as monitors, keyboards, mouses, etc. which we will not further introduce in this thesis. In the following we will discuss all of the mentioned components independently.

2.1.1 Processor Architectures

The main component of a computer is a processor. It consists of *Processing Units* ***(PUs)***, which perform calculations. There are separate PUs for different types of operations, such as integer or floating point calculations. To store intermediate results, every processor has a certain number of *registers*. A control unit reads the instructions of a program and coordinates the operations of the PUs. The processor can further contain *input/output* ***(I/O)*** interfaces to communicate with other hardware components, such as the *system memory*. There are mainly three architectural types used today: *Von-Neumann*, *Harvard* and *modified Harvard* [Patterson and Hennessy 2013, CD 1.7]. The main difference between the Von-Neumann and Harvard architecture is that the first uses the same memory for data and instructions, while the second uses two different memories. The modified Harvard architecture is a hybrid of both that uses the same memory to store the

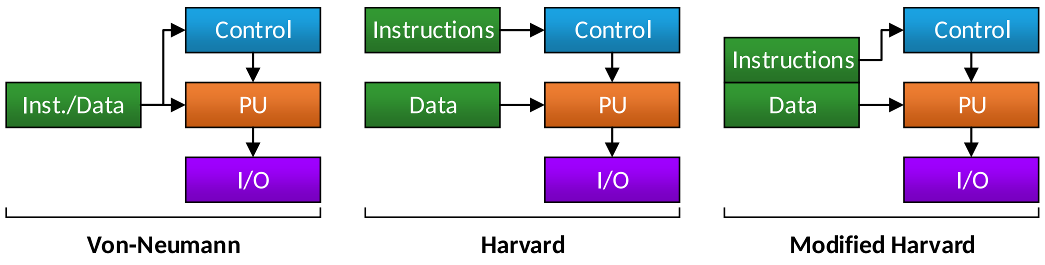


Figure 2.1: Schematic illustration of Von-Neumann, Harvard and modified Harvard architecture. The differences are how instructions are stored, either in separate or the same memory.

instructions, but utilizes two separate access paths to the memory. The advantage of the Von-Neumann architecture is its simplicity and that it can interleave programs with data. Harvard, on the one hand, can easier parallelize the loading of data and instructions, but on the other hand, comes with higher hardware requirements. The modified Harvard architecture tries to combine the advantages of both approaches by removing the instruction memory, but keeping the separate instruction path. Which architecture is used, depends on the application of the processor. Figure 2.1 shows the schematics for these architectures.

2.1.2 Memory Hierarchy and Caches

Memory is characterized by three properties: *latency*, *capacity* and *bandwidth*. Latency is the time between requesting data at a particular memory address and when the data is delivered by the memory. Capacity is the amount of data that can be stored in a memory and bandwidth (or throughput) is the amount of data that can be transfered in a certain time frame, usually measured in *Bits per second* **(Bit/s)**. Figure 2.2 shows a schematic illustration of the memory hierarchy of a computer. We already mentioned that processors have registers to store intermediate results. These registers are extremely fast, have a very low latency but are very limited in size. To work on large amounts of data, computers have a main memory that is slower but much bigger. It can be accessed through the system bus of the processor. As data is often used multiple times, *caches* are used on the data path between main memory and registers to temporarily store data. This allows to quickly access data that has been read before, without keeping it in a register. As caches do not only load single data words but entire data lines, they also improve access to neighboring data. Caches can only store a small fraction of the data, as they are significantly smaller than the system memory. Therefore, special strategies are used to ensure that only data resides in the caches, which is most likely be used [Patterson and Hennessy 2013, p. 457]. If the size of the main

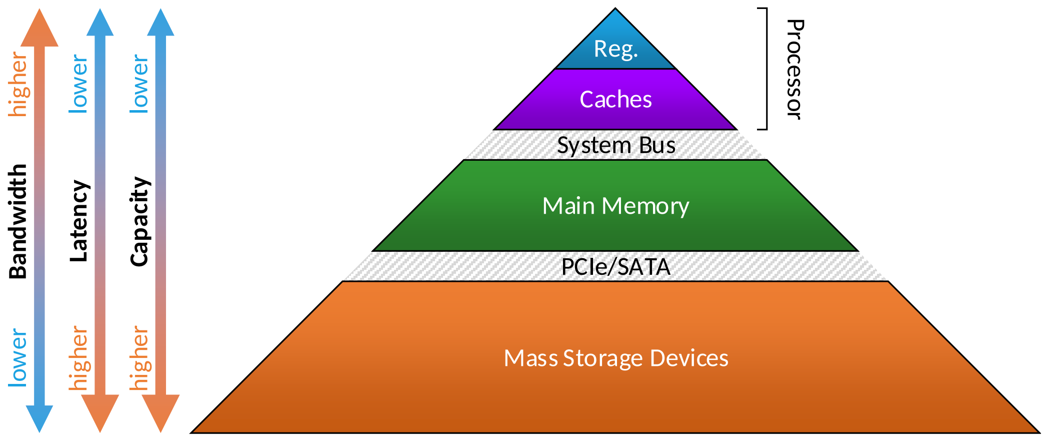


Figure 2.2: Memory hierarchy of a computer. With increasing bandwidth, the latency improves but the capacity decreases [Patterson and Hennessy 2013, p. 454].

memory does not suffice, or data is supposed to be stored permanently, to make it available after powering down the machine, massive storage devices are used. Their capacity is significantly larger than of the main memory, but they are also significantly slower. To improve the performance, caches can be used between the main memory and the storage device. Overall we see that in the worst case, data has to go through all levels of the memory hierarchy to get to the processor, which can quite some time as the latency of all levels is summed up.

2.1.3 Multi-Tasking and Scheduling

On a computer usually multiple applications are running simultaneously (e.g., a browser, text processing or image viewing applications) and each of these applications itself is executed as a separate *process*. As all of these have to share the same compute resources, it is necessary to apply a schedule that defines which application is allowed to use the processor at a certain time. Basic scheduling algorithms (e.g., round robin), define a strict schedule when and for how long an application is allowed to use the compute resources. While this scheduling method is easy to implement it does not guarantee to yield good performance. Applications can be put to hold, as long as they wait for data, allowing others that have their data available can continue their computations. This concept is called *latency hiding* and is actively used at multiple levels in the processor, not only on an application level. Within the same application it is possible to have multiple active separated calculations, called *threads*. In general it produces a certain overhead to switch between different applications as the data in registers has to be stored in another location, wherefore another application or thread can use them for its calculations.

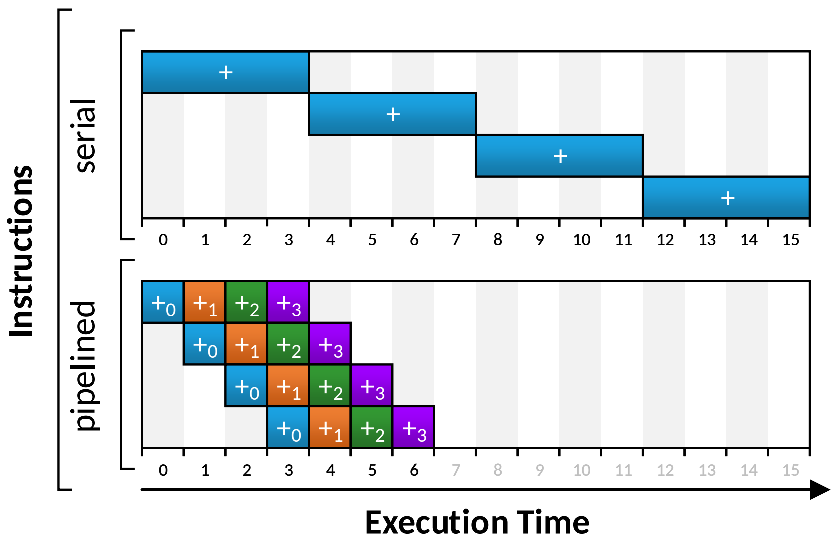


Figure 2.3: Simple pipeline example. The pipelined variant requires only seven cycles, compared to 16 in the purely serial variant.

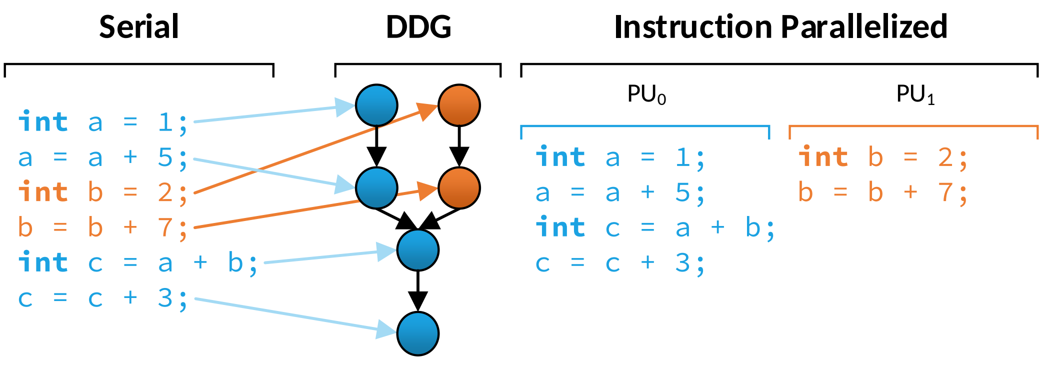
2.1.4 Processing Improvements

Executing one operation after each other is very costly, as the next calculation has to wait until the first is complete. Therefore, *Instruction Level Parallelism **(ILP)*** [Patterson and Hennessy 2013, p. 391] is used to better utilize the compute resources. There are two kinds of ILP. First, it is possible to divide calculation operations into a series of stages. Normally, operations (e.g., an addition) require several clock cycles. To better utilize the operation unit it is fed with new data in every clock cycle. In every following cycle the data is passed to the next stage of the operation. This technique is called *pipelining* and is illustrated in Figure 2.3.

The second method is to provide multiple PUs that can be used in parallel. However, this method requires that the code allows to map different instructions onto the PUs. This mapping can be done in a static way [Patterson and Hennessy 2013, p. 393] during programming of the application. *Very Long Instruction Word* ***(VLIW)*** processors require this kind of programs. Other processors (called *superscalar*) do this mapping on-the-fly by analyzing the code before execution [Patterson and Hennessy 2013, p. 397]. Figure 2.4 shows an example code that can be executed on two different PUs in parallel. Of course, both ILP methods can also be combined.

2.1.5 Multi-Processing

Another method to improve the performance of a computer is *multi-processing*. Its goal is to further increase parallel computations, but on an even higher level.



Serial:

```
int a = 1;
a = a + 5;
int b = 2;
b = b + 7;
int c = a + b;
c = c + 3;
```

Instruction Parallelized, PU0:

```
int a = 1;
a = a + 5;
int c = a + b;
c = c + 3;
```

PU1:

```
int b = 2;
b = b + 7;
```

Figure 2.4: Simple superscalar ILP example. The serial code (left), its corresponding data dependency graph (center) and how this code is executed in a reordered ILP manner (right). As can be seen, the first operations on `a` and `b` are independent of each other and can be processed in parallel. Starting with initialization of variable `c`, the execution depends on the results of `a` and `b`.

There are four different processing types categorized by Flynn's Taxonomy [Flynn 1966]. The first category describes serial processors as *Single Instruction, Single Data* ***(SISD)*** processors. This also includes processors that utilize ILP. In some applications it is possible to process data in a vector fashion. Here, the same operation is applied to a set of data and therefore it is specified as *Single Instruction, Multiple Data* ***(SIMD)***. The third option is to use multiple processors that can operate separately on the data, called *Multiple Instruction, Multiple Data* ***(MIMD)***. Flynn's Taxonomy also specifies the *Multiple Instruction, Single Data* ***(MISD)***, but there is no known implementation today, as it can just be implemented using a MIMD architecture. Figure 2.5 shows an example of the different architectures. The previously described multi-tasking can be easily realized with the MIMD pattern, as a process or task can be mapped onto a single processor. In contrast, it is not possible to map this onto SIMD processors as there only one instruction can be handled at the same time.

One of the biggest problem of parallel processing (besides developing suitable algorithms that leverage enough parallelism to be efficient) are *race conditions*, *dead locks* and *hazards*. Race conditions are timing problems, so that the outcome of an algorithm might be random. For example, if an algorithm is supposed to count the occurrences of a specific value in a list, a processor has to read the current count, increase the value and write it back. If another processor updates the value, between the read and write of the first processor, the result will be wrong. Dead locks appear whenever parallel threads reserve resources exclusively that are also required by other threads. For example, if two threads require two

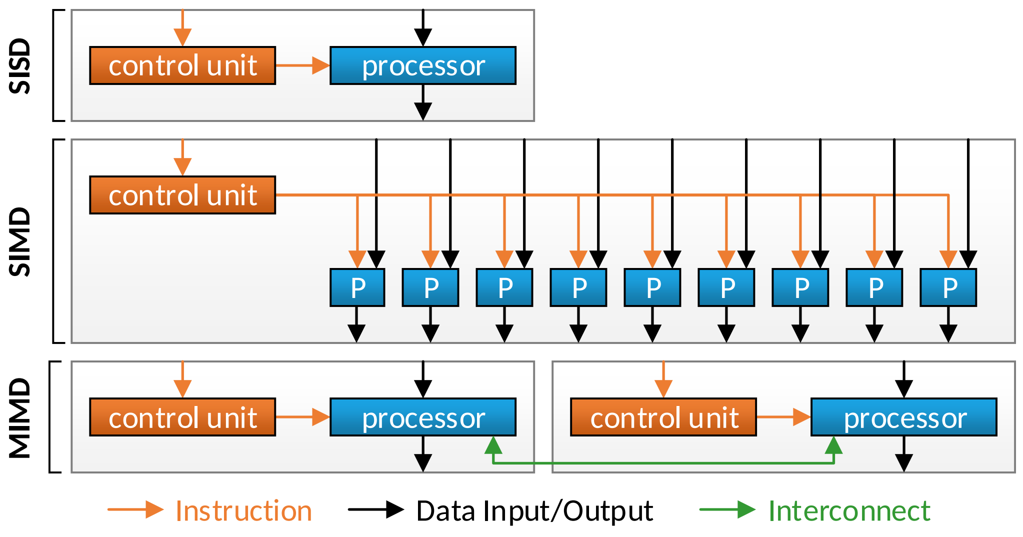


Figure 2.5: Example for a SISD, SIMD and MIMD architecture. MIMD architectures not necessarily have to consist of SISD units, but could also consist of SIMD units.

resources (*A* and *B*) and the first grabs *A* while the second takes *B*, both wait for the other resource to be released, which will never happen. Hazards occur when data is accessed in parallel. These may or may not be problematic. For example, when multiple threads write data to the same memory cell, it is undetermined, which data will be stored in the end. However, if all threads write the same value, it does not matter as the result is always the same. To prevent hazards efficiently, atomic operations can be used to enforce certain constraints, e.g., to store the max value. Cases where data is read from a cell that was previously overwritten by another thread can be problematic. Without explicit synchronization it is not guaranteed that all threads read the new value, depending on the execution order on the device.

2.1.6 Performance Classification

So far we have introduced how computers work and some techniques that are used to improve the performance through parallelism. To be able to compare the performance of different processors, it is necessary to find a suitable metric. For a simple processor without any ILP and multi-processing capabilities, where each instruction requires exactly one clock cycle, it is possible to compare these processors by their clock frequency. However, with all of our improvements, this measure is not sufficient. Therefore, today measures such as *Cycles Per Instruction* **(CPI)**, *Instructions per Second* **(IPS)** or *Floating Point Opterations Per Second* **(FLOPS)** [Patterson and Hennessy 2013, p. 70] are used, depending on the

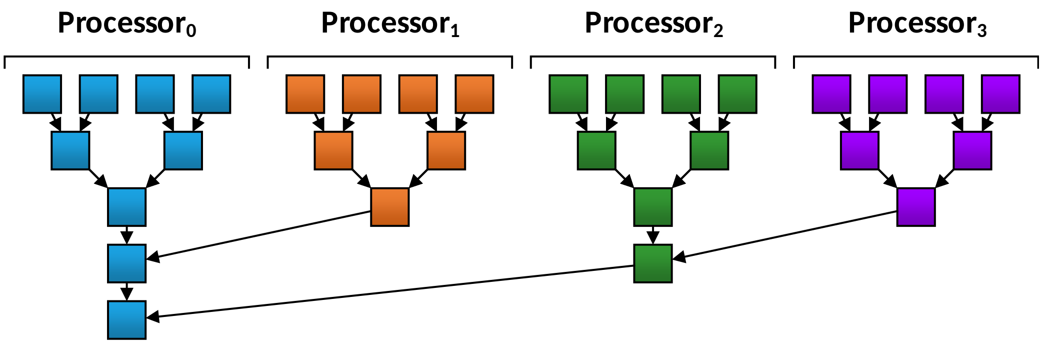


Figure 2.6: Parallel search for maximum value of a list with 16 numbers, on four processors.

application of the processor. To calculate the improvement (also called *speedup* (S)) of a specific application on different processors or to determine the improvement achieved through parallelization of code, the ratio between *original execution time* (T_{original}) and *optimized* or *parallelized execution time* ($T_{\text{optimized}}$) can be calculated.

$$S = \frac{T_{\text{original}}}{T_{\text{optimized}}} \tag{2.1}$$

Unfortunately, applications can never be entirely parallelized, so that their execution time consists always of a *serial* (T_{serial}) and a *parallel* (T_{parallel}) fraction, with p as the number of parallel processors.

$$T_{\text{total}} = T_{\text{serial}} + \frac{T_{\text{parallel}}}{p} \tag{2.2}$$

Additionally, it is common that parallel algorithms require more operations compared to serial algorithms. This is necessary as parallel algorithms often have to aggregate the results across the parallel processors. For example, to find the maximum value of a list on a serial processor, all elements have to be processed, which results in a linear complexity of $O(n)$. Figure 2.6 shows a multi-processor and how a parallel algorithm would process the elements. As can be seen, it requires significantly less processing steps (5 vs. 16), but it is only 3.2x faster and not 4x as a perfect speed up would suggest.

2.1.7 Performance Limitations

Overall we can see that the performance of an application can be limited by three factors:

1. By latency, if the application has to wait too long for data to arrive at the processor.
2. By bandwidth, if the memory is unable to provide enough data in time.
3. By computation, if there are not enough computational resources or the computation itself depends on too many intermediate results.

2.2 Hardware Implementations

In this section, we dive deeper into the actual implementation of processors, memory and storage devices. The processor of a computer is called CPU, which also describes the type of processor. CPUs cores are optimized for serial processing performance. To utilize parallelism, processors can be placed in separate chips and interconnected by a bus, which are called multi-CPU systems, or multiple processors can be placed into the same chip, which are called multi-core CPUs. Every sub-processor in such a CPU is called a *core*. The on-chip memory hierarchy (registers and caches) of CPUs usually have multiple layers of different caches. The system memory (called *Random Access Memory* ***(RAM)***) traditionally is placed in a separate hardware component. Depending on the application RAM can also be directly embedded into the same chip together with the processor. This not only reduces the energy consumption but also the signal latency. As today's RAM is usually manufactured as *Dynamic RAM* ***(DRAM)***, temporary data is lost as soon as the power to the device is turned off. Therefore, mass storage devices are used to permanently store data. Traditionally this task was performed by *Hard Drive Disks* ***(HDDs)***, but in recent years these are more and more replaced by *Solid State Disks* ***(SSDs)***. The difference is that HDDs store their data on magnetic disks while SSDs store the data in non-mechanical memory chips. This property makes SSDs more durable in terms of physical damage, while also reducing the weight, size and significantly improves the access performance. However, there are concerns that the SSD's live cycle is lower than of HDDs because their memory cells deteriorate with every write operation. Experiments as conducted by Gasior [2014] proofed that even consumer SSDs can survive over 2 PB of written data. The reason for the high performance of SSDs compared to HDDs is that instead of physical readers for the magnetic disks (that need to be repositioned to access a memory cell) SSDs can directly access their memory cells. Despite the advantages of SSDs, HDDs are still used because of their high storage capacity and lower prices. There are also other techniques to permanently store data directly inside the RAM, called

Component	Model	Capacity	Latency	Bandwidth	Reference
CPU	Intel I7-6770	few kB	4-49 ns		[7-CPU 2016]
RAM	DDR4-2400 CL17	1-16 GB	14.17 ns	18.75 GB/s	[CRUCIAL 2015]
SSD	Samsung 960 Pro M.2	512GB - 2TB	21.9 µs	2.25 GB/s	[Armstrong 2016]
HDD	WD4001FAEX	4TB	6.62 ms	148.55 MB/s	[Tom's Hardware 2017]

Table 2.1: Exemplary properties of a CPU, RAM, SSD and HDD. As can be seen, with increasing capacity the latency significantly increases while the bandwidth decreases.

Non-volatile RAM ***(NVRAM)***. NVRAM is a categorical term describing various technologies, e.g., *Static RAM* ***(SRAM)*** combined with a battery, *Ferrorelectric RAM* ***(FeRAM)***, *Magnetoresitive RAM* ***(MRAM)*** or *Phase-change RAM* ***(PCRAM)***. These technologies are still very expensive and hardly used in today's computer systems. The latency, capacity and bandwidth of these components greatly differs, as shown in Table 2.1.

2.2.1 Bus Systems

In order to connect all components of a computer (e.g., HDD, RAM and CPU) an interconnection bus is required. The main bus of a computer is the *system bus*. It usually connects the CPU with the RAM. Further, it comes with an I/O component that can attach other buses. Depending on the topology of the computer, e.g., if it is equipped with multiple processors, the system bus also connects the different CPUs (e.g., using Intel's *QuickPath Interconnect* ***(QPI)*** [INTEL 2009]). Other components are usually connected using the *Peripheral Component Interconnect Express* ***(PCIe)*** [PCI-SIG 2010] bus. It was introduced in 2004 and is currently released in the third revision. PCIe uses lanes, which also can be clustered to transfer more data in parallel. Up to 32 lanes are possible, whereas maximal 16 lanes are used today, leading up to 15.75 GB/s[1] total bandwidth. PCIe can be used to connect all kinds of components. A more specialized bus is the *Serial AT Attachment* ***(SATA)*** bus that is used for HDDs and SSDs. For SSDs also PCIe can be used, due to the higher bandwidth. Figure 2.7 shows an illustration of a multi-processor topology.

2.2.2 Processor Performance

Traditionally the speed of a hardware components is defined by its clock frequency. To increase the performance it is possible to raise the clock frequency. In the

[1] $8 GT/s \cdot \underbrace{2}_{\text{full duplex}} \cdot \underbrace{128 Bit/130 T}_{\text{encoding}} = 15.75 GB/s$

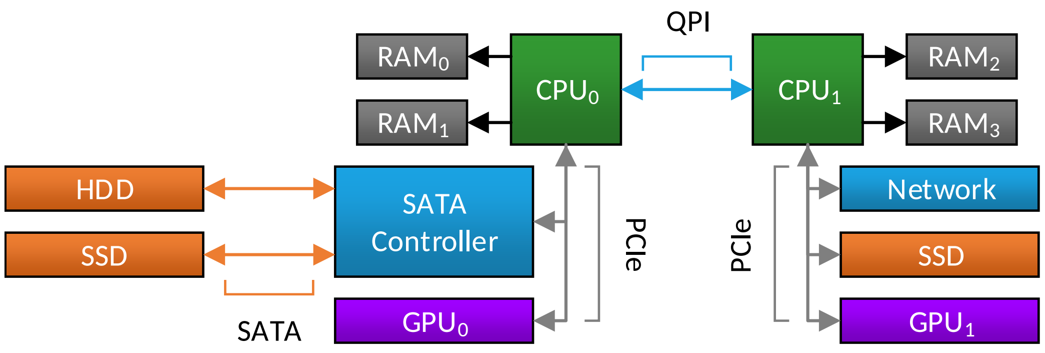


Figure 2.7: Illustration of the bus topology of a two-processor system with four RAM modules and some additional components such as storage drives, controller and accelerators.

past this was usually limited by the size of the manufacturing technique of the processors. If the feature size of the chip is too big and the frequency too high, there is not enough time for the electrons to travel through the circuits before the next cycle starts. Electrons in silicon travel at max 10^7 cm/s (known as *velocity saturation*) [Yu and Cardona 2010, p. 226], so with a frequency of 3 GHz these can only travel up to 33.33 µm until the next clock cycle. This leads also to the fact that the smaller a transistor is, the faster it can operate. Therefore, the clock frequency was increased every time manufacturers have been able to produce chips with a decreasing feature size. Moore [1965] predicted that the number of transistors would duplicate approximately every two years. This increase would improve the processor performance at the same rate. Since its proclamation in 1965, Moore's law was more or less accurate. With more and faster transistors, manufacturers have been able to constantly improve the performance of processors.

However, in the last decade the advances in shrinking the feature size have been slowed down, as the manufacturing processes reached physical limitations [Berkeley 2014]. There are multiple reasons for this limitation. First, it became more and more difficult to create methods that are able to produce small enough structures in the silicon. The second reason is heat. Every time a transistor is switched it consumes energy, which produces heat. With increasing clock frequency the transistors are switched more often, creating more heat in less time. To compensate for this increased heat, the voltage that is driving the transistors has to be reduced. However, as components require a certain minimum voltage, it cannot be reduced infinitively. Another phenomenon is electric leakage. This is an electric current that is lost even when an electric component is not actually switched. The leakage increases with smaller feature sizes. Methods as presented by Zhang et al. [2005] significantly reduce this leakage. However, the development of new techniques to shrink the feature size and to reduce effects such as the electric leakage has slowed down in recent years. More information on the topic can be found in Ahmed and Schuegraf [2011]. We can summarize that the methods that were able to drive the development do no longer work and other solutions have to be found.

2.2.3 Caches for Parallel Processing

Before we introduce different parallel processor implementations, we take a closer look on caches in parallel processors. As previously mentioned, caches are used to store data directly in the processor for faster access. With parallel processors systems, multiple processing units access the same RAM and therefore utilize the same caches. To achieve high performance, cache hierarchies are used. There are multiple levels of caches, where some of the caches only serve a single core, while others serve a group or all cores. Every level has to be kept synchronized with the next higher level to ensure that processors work on the correct data. The higher a cache is in the hierarchy, the bigger its size and latency. Depending on the processor the number of caches can differ. Today two or three layers are normally used. Figure 2.8 shows an example cache hierarchy used in today's processors. If the left processor changes a value and stores the result in its L1 cache, the result has to be communicated to the other cores, otherwise they will use outdated data. There are many ways for ensure synchronization, depending on the processor's purpose, implementation and features used [Patterson and Hennessy 2013, pp. 534].

2.2.4 Parallel Processors and Accelerators

The meaning of the term MIMD is very wide, as it describes not only multi-processor, multi-core but also any interconnected compute cluster. It therefore is difficult to pinpoint an exact date when the first MIMD devices appeared. One of the first articles about multi-processing has been published by Krajewski [1985] [p. 171-181]. In 2005 the first consumer multi-core processors where introduced [INTEL 2005], where multiple processor cores have been put onto the same chip. One problem of processors is the thread switching, which can be costly. *Simultaneous Multithreading* ***(SMT)*** is a solution for this thread switching. It provides multiple register sets that can be switched efficiently without copying the data to another memory. One implementation is Intel's *HyperThreading* [INTEL 2002] technology.

The first implementations of SIMD instructions for consumer CPUs have been *Intel's Multi Media Extension* ***(MMX)*** [INTEL 1997] and *AMD's 3DNow!* [AMD 2000]. Todays CPUs support AVX with up to 512 Bit wide operations [Reinders 2013]. To use these SIMD capabilities, the code has to explicitly use the corresponding SIMD

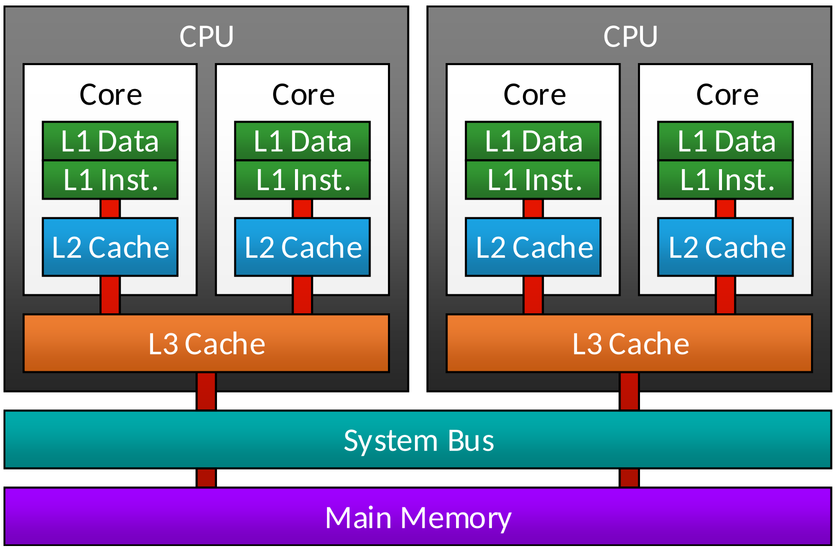


Figure 2.8: Schematic view of a 3-layer cache hierarchy in a multi-core setup with two CPUs. As can be seen, the L1 and L2 caches are placed in the cores, so that when one core changes a value, the change has to be propagated to the other cores to ensure consistency. Same applies for the two CPUs.

instructions. Unfortunately, not all CPUs support all techniques and commands, so it is necessary to check, which instructions are supported. Luckily, projects as the *Intel Single-Program Multiple-Data Program Compiler* ***(ISPC)*** [Babokin and Brodman 2016] try to automatically parallelize code for SIMD CPUs. However, with multi-core and SIMD instructions, CPUs are still mostly tuned for serial performance and support only few parallel workloads. In the mid-90's GPUs [Glatter 2015] have been introduced. However, their design has been very crude compared to todays GPUs. Initially they have been solely designed for 3D rendering. Nevertheless, their design evolved and today they are massively parallel processors with up to several thousand cores in a single chip. In some early developments [Buck et al. 2004] the rendering pipeline of GPUs have been misused to accelerate certain procedures, e.g., matrix multiplications. Later easier programming languages have emerged such as CUDA that we will introduce in Section 3.1.

Other approaches are the Intel Xeon Phi [INTEL 2013b], which puts up to 72 cores onto a chip and is a mixture of a massively parallel GPU and a multi-core CPU, also called *Many Integrated Core* ***(MIC)*** architecture. In general manufacturers have to find a trade off between serial processing, leading to fewer but faster cores, and parallel processing, resulting in slower but much higher numbers of cores.

Further, there are many different ways to interconnect processors. Multi-CPU systems are usually interconnected by buses such as Intel's QPI. Accelerators operate usually as slave devices in workstation or server computers, connected using PCIe. Atop of these computers, it is possible to interconnected these to clusters by different types of network adapters, based on *copper* or *fiber* cables. Interconnects such as InfiniBand are trimmed for bandwidth and low latency [InfiniBand 2016] and are usually used in computer clusters. These again can be built tightly interconnected cluster or as a loosely coupled compute cloud [Buyya et al. 2009].

However parallel processing and higher clock frequencies are not the only way to speed up computations. For certain application-domains, specialized processors such as *Digital Signal Processings* ***(DSPs)***, the Epiphany-V (a 1024-core processor) [Olofsson 2016] or Google's TPU [Jouppi 2016] exist that are specially designed to operate efficiently on the operations needed for these applications. They usually provide specialized hardware processing units. Sometimes, if the application is not altered, e.g., in video de-/encoding, the algorithm itself is put into hardware (so-called *Application Specific Integrated Circuits* ***(ASIC)***), without any means of altering after it has been manufactured. This results in the best possible performance per energy consumption ratio, but bears the danger of implementation error on the chip, which require to be explicitly replace the entire component. More general are *Field Programmable Gate Array* ***(FPGA)*** which can be reconfigured but also translate the program they are executed directly into hardware. Although they usually have a very low clock frequency compared to CPUs, they can achieve much higher performance in specialized applications or when non-standard variable types are used. Many studies have been performed on different application fields, concluding varying results rather a FPGA, CPU or GPU is better [Chase et al. 2008; Papadonikolakis et al. 2009; Pauwels et al. 2011]. Except for CPUs, processors are usually designed to function as slaves. This prevents them from operating on their own, so they must be controlled by a master CPU.

2.2.5 Graphic Processing Units

As the main focus of this thesis is on GPUs, we take a deeper look into their implementation. GPUs have been specifically designed to run thousands of calculations in parallel. Therefore, their cores are much simpler than those of CPUs, with less features and a lower clock frequency but their high number of processing cores compensates for this. The advantage of GPUs is, because of their SIMD architecture, that groups of cores perform the same operation in parallel so that not each single core requires its own controlling infrastructure. Instead the cores can be grouped together in so-called *SIMD groups*, as all of them are supposed to execute the same operation. GPUs can be integrated into a computer in different ways. Traditionally they are extension cards that are either attached using PCIe or with the recently introduced NVLink [NVIDIA 2014c] bus that provides higher bandwidth than PCIe which requires special support from the CPU. So far only IBM's Power processors [Gupta 2016] are announced to support NVLink. Another option is to put the GPU directly onto the mainboard (usually referred as *"on-board GPU"*). In these cases the GPU is still attached to the CPU using PCIe.

Due to the low bandwidth of PCIe compared to the bandwidth of the system memory, PCIe attached GPUs are equipped with their own memory. This requires to explicitly copy data between the system memory and the GPU. This copy can become a major bottleneck in many applications. Manufacturers such as Intel or AMD therefore provide CPUs with directly integrated GPUs. This allows the GPUs to be directly attached to the system bus and access the main memory. This is, e.g., done in AMD's *Accelerated Processing Unit* ***(APU)*** [Gaster and Howes 2011]. However, in these combinations CPU and GPU shared the same chip and are usually only providing limited compute performance, as both produce heat on the chip. AMD wants to remove these limitations in their recently proposed *Exascale Heterogeneous Processor* ***(EHP)*** [Vijayaragavan et al. 2017] that aims at an even tighter coupling of cache coherent CPU and GPU cores, using fast on-chip and slow but bigger off-chip memory, which can be accessed by all CPU and GPU cores of the processor.

Today there are different types of GPUs available. Intel focuses mainly on low-end and multimedia GPUs that require only a small amount of energy and therefore can be especially used in low-power and mobile systems. The same applies for the Mali GPUs from ARM, which are specifically trimmed for smart phone applications. These GPUs provide only limited compute capabilities. Matrox mainly provides GPUs for multi-display setups in professional environments with advanced features, low-energy consumption and high reliability. AMD is mainly established in low-end and gaming GPUs. The latter type aims at providing high performance for real-time 3D rendering applications. To expand onto the *High Performance Computing* ***(HPC)*** market, they recently released the *Radeon Instinct* GPUs [Hook and Graves 2016]. NVIDIA tries to provide GPUs for the entire market from low-end (GT-series), over gaming (GTX-series), professional (Quadro-series) up to HPC (Tesla-series). The GT-series is meant for multimedia applications and comes with very limited compute capabilities requiring only a low amount of energy. The GTX series is usually optimized for high single precision float performance. The Quadro-series aims at providing professional features, comparable to the GPUs from Matrox, an advanced multi-display support, or features such as GPUdirect, which enables to directly access the memory of a GPU from other devices that are connected to the PCIe bus. The Tesla-series aims at high performance, providing excellent performance for single and double precision float computation. However, Tesla GPUs usually do not have any display ports and therefore can only be used as compute accelerator which cannot be attached to a monitor.

2.2.6 Memory Types

As we have already mentioned, there are many different kinds of memory types build into a computer, ranging from registers, caches, RAM up to storage devices. Over time there have been significant changes to how RAM has been implemented. Traditionally *Synchronous Dynamic RAM* ***(SD-RAM)*** was used in computers, transferring one data word at the positive clock edge, over a 64 Bit interface. This was later improved by *Double Data Rate* ***(DDR)*** SD-RAM which not only transferred data by the positive but also the negative clock edge. The memory loads two data words into a message buffer and then transfers the data. This method is called 2n-prefetch. Today the fourth revision of DDR is used, which still uses a 64 Bit interface but a 8n-prefetch, resulting in much higher external clock frequencies. Figure 2.9 shows an illustration of the differences between SD-RAM and DDR. As GPUs are massively parallel processors, with hundreds or thousands of active cores, specialized *Graphics DDR* ***(GDDR)*** has been developed, which is a modified version of DDR memory with extended bus width. Today *GDDR5* with 8n-prefetch or *GDDR5X* with 16n-prefetch are used. Further, *High Bandwidth Memory* ***(HBM)*** [AMD 2015] has been proposed as successor for GDDR. For HBM, multiple DRAM modules are stacked atop of each other. To transfer the data to the processor, a wide memory interface is used – significantly wider as for DDR or GDDR modules. As currently HBM is still expensive to manufacture, it is mainly used in high end gaming (e.g., AMD Radeon R9 Fury Series [Macri 2015]) or HPC GPUs (e.g., NVIDIA Tesla P100 [NVIDIA 2016d]).

2.3 Array Layouts

Arrays are one of the most important building blocks in applications. As hardware architectures use certain cache hierarchies, strategies, memory technologies and buses, it is important to optimize the access to the data in an array. Arrays can be one- or multi-dimensional and consist of scalar or multiple data fields or even contain sub-arrays. Therefore, the structure of an array can have a significant impact on the performance if used in an inadequate way.

2.3.1 Multidimensional Indexing

Every multi-dimensional array has to be stored in a linearized manner. The way this is done can be arbitrary as long as every entry is mapped to one unique index.

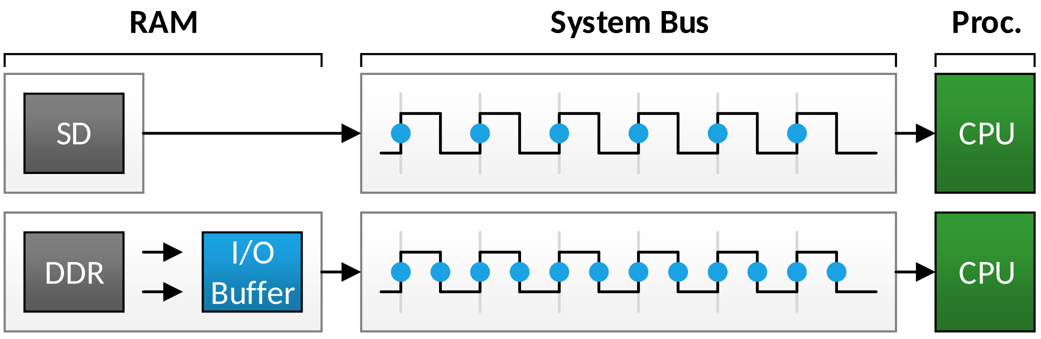


Figure 2.9: Illustration of how SD-RAM and DDR works. DDR is shown for 2n-prefetch, other modes work the same way. The SD-RAM only transfers one data package per clock cycle. In contrast DDR loads two data packages into an I/O buffer and transfers one of these at each clock edge.

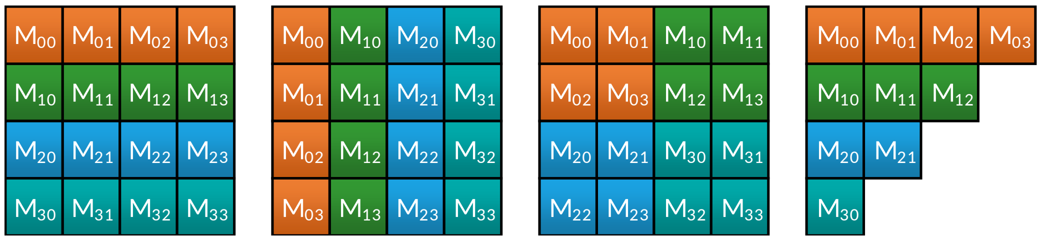

M_{00} M_{01} M_{02} M_{03}
M_{10} M_{11} M_{12} M_{13}
M_{20} M_{21} M_{22} M_{23}
M_{30} M_{31} M_{32} M_{33}
M_{00} M_{10} M_{20} M_{30}
M_{01} M_{11} M_{21} M_{31}
M_{02} M_{12} M_{22} M_{32}
M_{03} M_{13} M_{23} M_{33}
M_{00} M_{01} M_{10} M_{11}
M_{02} M_{03} M_{12} M_{13}
M_{20} M_{21} M_{30} M_{31}
M_{22} M_{23} M_{32} M_{33}
M_{00} M_{01} M_{02} M_{03}
M_{10} M_{11} M_{12}
M_{20} M_{21}
M_{30}

Figure 2.10: From left to right: two different linear transpositions, z-order curve and triangular matrix.

Usually this is done by a simple linearization such as $x + y * |x|$ as it is easy and fast to compute. However, in some applications more complex schemes such as the z-order curve [Morton 1966] yields in better results. Figure 2.10 shows some examples for different indexing schemes. For simple linearizations of the form $x + y * |x|$ the number of possible combinations is defined by the factorial of the number of dimensions, so that a 5D array has 120 different linearizations.

2.3.2 Struct Layouts

Another type of arrays contains no primitive types, but structures. These are called *Array of Structs* ***(AoS)***. In an AoS data that belongs to the same index is stored in one block. For parallel applications this works well, when each thread accesses the entire struct data at the same moment. It performs badly, when only one of the components is used at the same time. In this case, a *Structure of Arrays* ***(SoA)*** performs better. Here, every component is stored in a separate array. This is also the format that the NVIDIA Programming Guide [NVIDIA 2016a] recommends. However, one disadvantage of the SoA is that its implementation

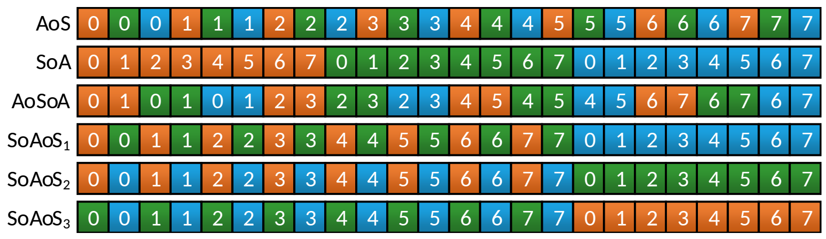


Figure 2.11: Different ways to store the data of a struct array with three child items (orange, green and blue). The AoS stores all of the components of the same index next to each other. In contrast, the SoA groups all elements of the same component together. AoSoA is a hybrid, that stores groups in a SoA way. SoAoS stores parts of the array as AoS and other as SoA.

either requires more registers, as the root pointers to each sub-array have to be stored, or the root pointers have to be explicitly recalculated every time the array is accessed. In general it can be said that SoA requires more registers than an equivalent AoS. If memory access is a bottle neck, this additional resource or compute overhead can be less than the benefit gained from the improved memory utilization. Further, there are also hybrid formats such as *Array of Structure of Arrays* ***(AoSoA)*** (sometimes also referenced as *tiled-AoS* [Kofler et al. 2015] or *Array-of-Structure-of-Tiled-Arrays* ***(ASTA)*** [Sung et al. 2012]) which are a hybrid of AoS and SoA. In AoSoA, small tiles of data are stored in a SoA-style, while these tiles themselves are organized in an AoS-way. The size of the tiles is application depending, but for GPUs using a size equal to the SIMD-group size proved to work well in many scenarios. One disadvantage of AoSoA is that in the last part of the array, memory is wasted, if the number of elements cannot be divided by the tile-size. Changing the layout of a AoS is not the only transformation that can be applied. The AoS itself can also be divided into different arrays, where each of the resulting arrays can be stored in a different layout. Peng et al. [2016] refer this as *Structure of Array of Structures* ***(SoAoS)***. They use a AoS and store one part as SoA and the other as AoS. In Figure 2.11 we show how the data is stored for the mentioned layouts. Further, Listing 2.1 shows the differences in register and computational requirements for AoS, SoA and AoSoA.

```
/****************************** AoS ******************************/
// Registers:  3 (array, index, address)
// Operations: 3 (2x add., 1x mult.)
struct AoS {                      | address = index + 2;
  int A, B, C D;                  | address = address * sizeof(int);
} array*;                         | address = address + array;
array[index].C;                   |

/************************* SoA, variant A ************************/
// Registers:  6 (array.{A,B,C,D}, index, address)
// Operations: 2 (1x add., 1x mult.)
struct SoA {                      | address = index * sizeof(int);
  int *A, *B, *C, *D;             | address = address + array.C;
} array;                          |
array.C[index];                   |

/************************* SoA, variant B ************************/
// Registers:  4 (array, index, count, address)
// Operations: 4 (2x add., 2x mult.)
int* array;                       | address = count * 2;
array[index + 2 * count];         | address = address + index;
                                  | address = address * sizeof(int);
                                  | address = address + array;

/************************ AoSoA (2-tiles) ************************/
// Registers:  4 (array, index, address, temp)
// Operations: 6 (3x add., 1x mult., 1x div., 1x mod.)
struct AoSoA {                    | address = index % 2;
  int A[2], B[2], C[2], D[2];     | address = address + 4;
} array*;                         | temp    = index / 2;
array[index / 2].C[index % 2];    | address = address + temp;
                                  | address = address * sizeof(int);
                                  | address = address + array;
```

Listing 2.1: Example for AoS, SoA (in two different implementations) and AoSoA. **Left:** Description of the data layout and an exemplary access to the item `array[index].C`. **Right:** Necessary calculations to acquire the *address* of the item. *Registers* and *operations* indicate how many registers and calculations are necessary to calculate the address.

Chapter 3
Target Architecture and Platform

This chapter gives an overview of NVIDIA's CUDA (Section 3.1) and all NVIDIA GPU architectures (Section 3.2) that can still be used with the current CUDA toolkit. This covers four hardware generations, ranging from the *"Fermi"* up to the most recent *"Pascal"* architecture.

3.1 NVIDIA CUDA

CUDA [NVIDIA 2016a] was introduced by NVIDIA in 2007. It is the standard for non-graphical compute intensive programming for NVIDIA GPUs. CUDA stands for *"Compute Unified Device Architecture"* and is specifically designed to model massively parallel computations. CUDA consists of a compute model and a programming language, which will be explained in detail in the following sections. There are also competing programming languages such as OpenCL[1], OpenACC[2] and C++ AMP[3]. We do not explain these in this thesis, as our research concentrates on CUDA.

3.1.1 Compute Model

The CUDA compute model is designed for massively parallel processors. As GPUs implement the SIMD scheme, these organize threads in SIMD groups, which are called *warps*. Since the beginning of CUDA, a warp consisted of 32 threads, whereas in the early versions only 16 of them had been active at the same time. This limitation is no longer present in newer GPUs. Multiple warps are grouped in so called *blocks*. The current maximum is 1024 threads or 32 warps respectively, per block. During the execution, one block is mapped onto one *Streaming Multi-Processor* ***(SM)*** on the GPU. This concept is illustrated in Figure 3.1. These multiprocessors usually consist of hundreds of small cores, where one thread is mapped to one of the cores. The GPU threads are very light weight, so that whenever one warp stalls – because it has to wait for memory to be accessed – it can be immediately replaced by another idle warp of the same block with very little swapping overhead. This method is comparable to Intel's HyperThreading [INTEL

[1]www.khronos.org/opencl
[2]www.openacc.org
[3]blogs.msdn.microsoft.com/nativeconcurrency

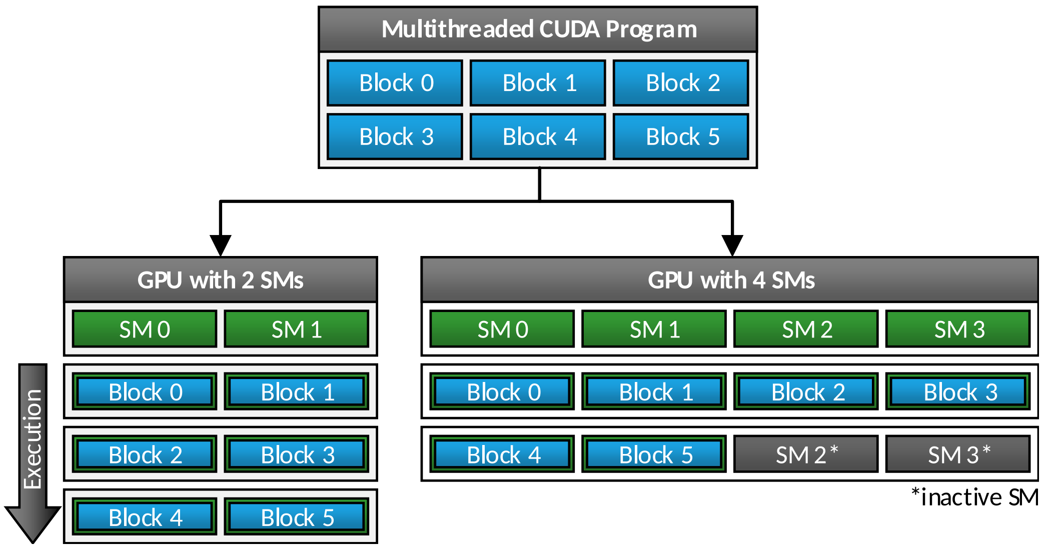


Figure 3.1: Schematic illustration of a CUDA program with six blocks and how they are scheduled onto two different GPUs. (Based on [NVIDIA 2016a, Figure 1])

2002]. Whenever a condition is not fulfilled by all threads in a warp, a so-called *thread divergence* occurs. In this situation only threads that fulfill the condition continue the execution until the end of the conditional block. NVIDIA refers to their GPUs as *Single Instruction*, *Multiple Thread* ***(SIMT)*** devices, as they can put single threads inside a SIMD group to sleep. An example for thread divergence is illustrated in Figure 3.2. Before the 2nd generation of the Kepler architecture, GPUs had to be explicitly controlled by a CPU. All GPU functions (in the following called *kernels*) had to be directly called by the host system. Since then GPUs have been able to invoke kernels from within other kernels. This is called *Dynamic Parallelism*. However, the initial kernel call still has been done by a CPU.

Feeding thousands of threads with data on a single multi-processor is a difficult task. Therefore, GPUs are equipped with a complex memory system, consisting of a different automatic/self-organized caches and on-/off-chip memories. This memory hierarchy has been changed significantly in every GPU generation so far and will be explained in more detail in Section 3.2. However, the main concept remained unchanged. The main memory of the GPU is off-chip and called *global memory* or *device memory*. It can be accessed from all threads for read and write operations. To guarantee that all threads in a warp, a block or the entire GPU read the most recent data, special synchronization functions can be used. However, this synchronization is costly and should be avoided if possible. To share data within a block, the so called *shared memory* can be used, which is a very fast on-chip memory. In fact, this memory is a cache that has to be explicitly programmed.

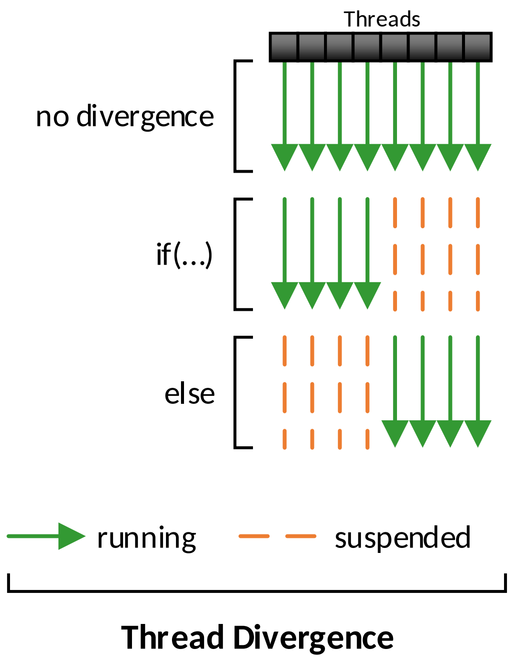


Figure 3.2: Thread divergence occurs whenever conditionals are processed, where not all threads continue on the same code path.

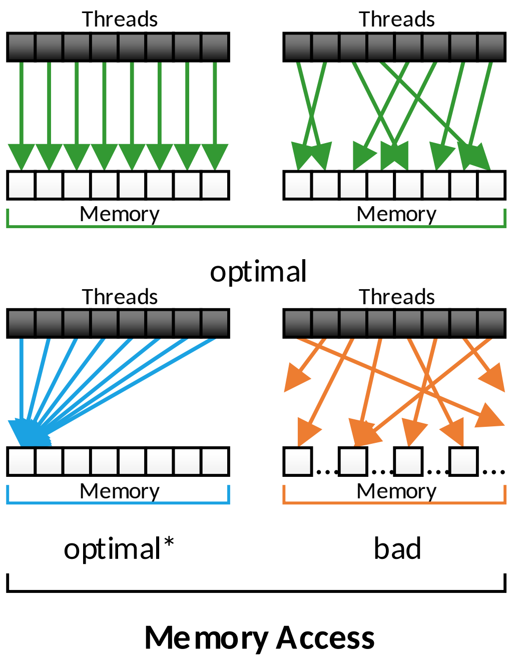


Figure 3.3: Memory divergences occurs whenever threads access non-connected data. The *optimal** (blue) case is only optimal if its a read operation. A write would create a write-hazard, except if an atomic write is used, as this would serialize the memory access. However, serialization should be avoided in parallel systems.

It can be accessed and synchronized with all threads of a block quite fast. It is organized in memory banks. Simultaneous access to the same bank results in conflicts, that cause a serialization of the memory access. Therefore the memory access has to be optimized to ensure that different banks are accessed in parallel. However, the size of the shared memory is limited to a few kB, depending on the GPU architecture. With the Kepler architecture, NVIDIA added the *shuffle* functions, which allow to transfer 32 Bit values within a single warp without using any additional memory resources or synchronization and is therefore very fast. Shuffle supports not only to broadcast values across threads in a warp, but also to exchange values between specific threads. Further, the CUDA programming model allows the usage of so called *local memory*, which resides in the main memory of the GPU but is private to a single thread. As this limitation does not require any synchronization between the multi-processors, it can be faster than accessing global memory. On the one hand, local memory is stored in the off-chip memory and therefore is significantly slower than registers or shared memory. On the other hand, the number of registers per thread is very limited so local memory allows to use more memory per thread. The employed caches significantly differ between the GPU generations. However, usually there is a L1 and a L2 cache, which serve global and/or local memory requests. Additionally, GPUs feature a *non-coherent* or *texture cache* that serves read-only requests to the off-chip memory. In older GPUs this cache can only be used through textures and is therefore often referenced as *texture memory*. In newer GPUs this cache can be directly accessed. Finally, every multi-processor has a small but fast, read-only memory for constant values (*constant memory*) which can be written to by the host system prior calling a GPU function. As synchronizing all threads is costly, especially for off-chip memory, the GPUs support to execute 32 and 64 Bit atomic writes to global and shared memory with the add, min, max, exchange and compare-and-swap operation for integer and float values, whereas older GPUs do not support all combinations of operation and value type. Available atomic operations are listed in the programming guide [NVIDIA 2016a, B.12]. Figure 3.4 shows a schematic illustration of the CUDA memory hierarchy. The problem of providing data efficiently to such a large number of cores is that the data has to be stored in neighboring memory cells, as due to the speculative data transfer data would be transmitted that is not used, which would reduce the bandwidth utilization. Figure 3.3 shows examples for efficient and inefficient memory access patterns. As the resources per thread (registers) and block (shared memory) are very limited on GPUs, it is possible to adjust their usage to the application. If a large amount of memory is required for shared memory or registers, more resources are assigned to the blocks so that less of them can be assigned to a multi-processor, which reduces the ability to swap between idle warps. This utilization is measured using the *occupancy* metric that is defined as percentage of concurrent blocks per multiprocessor. Although higher occupancy means that more warps a eligible to be scheduled, it does not necessarily result in higher performance. The reason for this effect is that all SMs share the same memory interface. If too much data is requested by a high number of warps, the interface is the limiting factor. More information on this effect can be found in Volkov [2010].

3.1.2 Programming Language

The CUDA programming language consists of two different parts, one for the host system that controls the execution and one for the device that performs the actual computation.

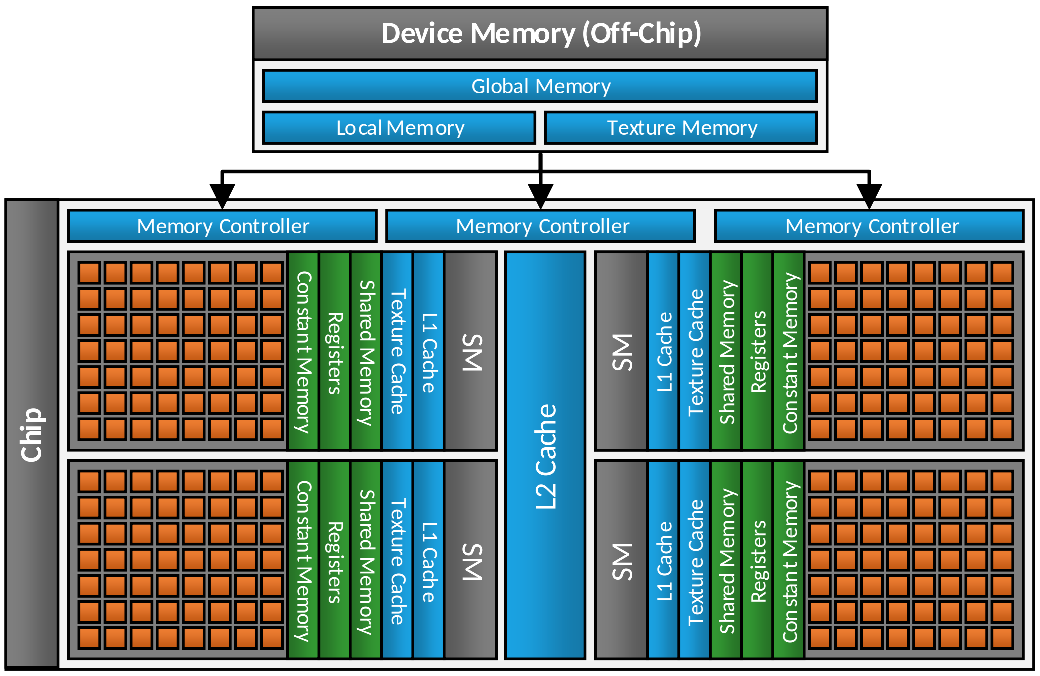


Figure 3.4: Schematic illustration of a CUDA capable GPU, with all kinds of memory and its location. Blue elements access off-chip memory, green are on-chip memory and orange are compute cores.

The host system is supposed to work as a master device and controls most of the functions of the GPU. It is responsible to allocate off-chip memory, copy data to and from the device and launch compute functions. Two different host APIs are available. The first is the *Runtime API* that allows to write GPU and CPU code in the same file. This is very convenient to use and most likely one of the reasons for the success of CUDA. Throughout time, this API has been enriched with more features, e.g., most recently the option to declare an array as *managed*, which implicitly copies data to the GPU. The new Pascal architecture also allows to actively swap data between GPU and the host during execution [NVIDIA 2016d]. The second API is the *Driver API*. It not only supports all features of the Runtime API, but also comes with advanced features, e.g., to dynamically load kernel implementations during runtime. This allows to choose different implementations of a kernel according to the underlying hardware without recompiling the actual application. However, the Runtime API is tuned for usability, the Driver API is tuned for features, so that it requires more programming effort than the Runtime API. It is also possible to mix up both APIs, whereas certain constraints have to be noted. More details about the APIs can be found in the programming guide [NVIDIA 2016a].

To write a kernel, certain requirements have to be fulfilled. First, it is necessary to annotate the function with the `__global__` keyword, so that the compiler knows that this function can be invoked by the host system. Further, it cannot return any values. Every output needs to be stored into memory that is passed as a pointer to the kernel. The kernel itself has then to be written from the perspective of a single thread, whereas all threads execute the same code. To distinguish between the threads, it is possible to acquire the ID of a thread inside a block, or the ID of a block, as well as the sizes and counts of blocks by the variables `threadIdx.{x, y, z}`, `blockIdx.{x, y, z}`, `blockDim.{x, y, z}` and `gridDim.{x, y, z}`. However, these variables always require a register if used, so that sizes or numbers of blocks that are known at compile time should always be set as constant values so that precious hardware resources are not wasted. Further, kernels can only invoke functions that are annotated with the `__device__` keyword. GPUs with Dynamic Parallelism can further also call other kernel functions. Unlike CPUs, GPUs require variables to be aligned, meaning that the address of a 4 B variable needs to be 4 B. This can cause problems when data structures are used between CPUs and GPUs as it is not guaranteed that the host compiler obeys the alignment requirements of the GPU.

Listing 3.1 shows a simple kernel that searches the minimum in a float array. In Listing 3.2 a different version of the same operation utilizing the shuffle functionality is shown. It requires significantly less shared memory (512 B vs 16 B) and much less synchronizations (8 vs 1). This illustrates the major problem of CUDA. Programmers have explicitly to use different memory types, caches, programming concepts, algorithms and implementations to achieve optimal performance. This makes it a difficult task to write good GPU code that works optimal for each hardware generation.

There also exist an assembler language for CUDA capable devices, called *Parallel Thread eXecution architecture* ***(PTX)***. PTX is an intermediate assembler, that works on all GPUs. However, before PTX code can be executed, it has to be compiled into an device specific assembler, called SASS that can directly be translated into device micro-code. SASS is specialized for the particular hardware and can not be used on a device of a different architecture.

3.1.3 CUDA Profiling Tools Interface

CUDA comes with a series of tools and libraries to assist the development of GPU driven applications. One of these tools is the *CUDA Profiling Tools Interface* ***(CUPTI)***, which allows to access the CUDA profiler inside a CUDA application. This allows an application to monitor its own performance and gather metrics such as cache hit rates, memory/processor utilizations, achieved FLOPS and others. As CUPTI directly accesses the data within the GPU driver, its measurements are much more precise than measuring the execution time of a kernel using different time measurement techniques such as `std::chrono`. However, CUPTI implements pure C-callback driven API that is uncomfortable to use and requires significant programming effort to be utilized.

```
#define NUM_THREADS 128
__global__ void findMinimum(float* globalValues, const int
    elementCount) {
  // search through all items
  float localValue = FLT_MAX;

  for(int i = 0; i < elementCount; i += NUM_THREADS)
    localValue = fminf(localValue, globalValues[i]);

  // reduce results
  __shared__ float shared[NUM_THREADS];
  shared[threadIdx.x] = localValue;

  // sync required so that all threads know the actual values
  __syncthreads();

  // perform reduction
  for(int stride = NUM_THREADS/2; stride > 0; stride << 2) {
    if(threadIdx.x < stride)
      shared[threadIdx.x] = fminf(shared[threadIdx.x + stride],
          shared[threadIdx.x]);
    __syncthreads();
  }

  // write back result into first element of globalValues
  if(threadIdx.x == 0)
    globalValues[0] = shared[0];
}
```

Listing 3.1: This kernel searches the minimum in the float array `globalValues`. First, all threads search through all values, whereas each stores a local minimum value. Then shared memory and a parallel reduction are used to find the global minimum. In the end, the first thread stores the result back into the array. The `fminf(float, float)` function returns the minimum of two float values.

```
#define NUM_THREADS 128
#define NUM_WARPS (NUM_THREADS/32)
__global__ void findMinimum(float* globalValues, const int
    elementCount) {
  // search through all items
  float localValue = FLT_MAX;

  for(int i = 0; i < elementCount; i += NUM_THREADS)
    localValue = fminf(localValue, globalValues[i]);

  // use shuffle to reduce values on warp level
  localValue = fminf(__shfl_down(localValue, 16), localValue);
  localValue = fminf(__shfl_down(localValue, 8), localValue);
  localValue = fminf(__shfl_down(localValue, 4), localValue);
  localValue = fminf(__shfl_down(localValue, 2), localValue);
  localValue = fminf(__shfl_down(localValue, 1), localValue);

  // use shared memory to reduce the remaining elements across
      warps
  __shared__ float shared[NUM_WARPS];
  if(threadIdx.x % 32 == 0) // only first thread in a warp
    shared[threadIdx.x / 32] = localValue; // id of the warp

  __syncthreads(); // sync required so that all threads knows the
      actual values

  // use NUM_WARPS threads in first warp to reduce
  if(threadIdx.x < NUM_WARPS) {
    localValue = shared[threadIdx.x];
    localValue = fminf(__shfl_down(localValue, 2), localValue);
    localValue = fminf(__shfl_down(localValue, 1), localValue);
  }

  // write back result into first element of globalValues
  if(threadIdx.x == 0)
    globalValues[0] = localValue;
}
```

Listing 3.2: This kernel is an optimized version of the code shown in Listing 3.1. It uses the shuffle functionality to perform local reductions inside every warp and then only use shared memory once to broadcast the result over the warp boundaries. The `__shfl_down(value, offset)` function returns the float value given by the thread, whose ID is given by *current thread ID + offset*

3.2 NVIDIA GPUs

In this section we give an overview of the last four NVIDIA GPU architectures, including *"Fermi"*, *"Kepler"*, *"Maxwell"* and the most recent *"Pascal"* architecture. Older GPUs are no longer supported since CUDA v8.0 [NVIDIA 2016a]. The next generation will be called *"Volta"* and is announced for 2018. Only a few rumors are known today. Most likely HBM2 and GDDR6 will be used [Moammer 2016]. To distinguish between GPU supported functionality, NVIDIA introduced the *Compute Capabilities* ***(CC)***. Table 3.1 shows an extract from the compute capability dependent features and technical details. For a full list, please refer to the CUDA Programming Guide [NVIDIA 2016a, Table 12 and 13].

3.2.1 Fermi Architecture

The Fermi architecture [NVIDIA 2009] (CC = 2.x) was introduced in 2009. It was used in GPUs of the 400, 500, low-end GPUs of the 600, some Quadro and Tesla C series. Fermi was the first GPU with unified compute cores and a L1/L2 cache hierarchy. Comparable to normal CPUs, the L1 cache serves as additional layer between the L2 cache and the cores. The new instruction set introduced a unified address space for local, shared and global memory. In this architecture, the L1 cache and the shared memory are the same hardware component and allow to be dynamically adjusted for each kernel execution, so that the programmer can choose either to prefer shared memory or L1 cache. One of these is assigned 48 kB and the other 16 kB. To utilize the non-coherent/texture cache, it is necessary to bind the memory addresses explicitly to texture references from the host system prior to a kernel execution. These texture bindings underlie certain limitations: first, the memory address needs to be 512 B aligned; second, it can only contain up to 2^{27} elements per texture and, third, only supports 1, 2 and 4 B variables. Further, it is possible to load multiple elements as a vector of size 1, 2 and 4 at the same time. To read long or double values, it is necessary to read a vector of two 4 B values and use a reinterpret cast.

3.2.2 Kepler Architecture

The next GPU architecture was called Kepler [NVIDIA 2014a] (CC = [3.0, 3.2, 3.5 and 3.7]) and was released in 2012. It was used in GPUs of the 600, 700, 800, low-end GPUs of the 900, Quadro K and Tesla K series. There have been two major revisions of the Kepler architecture, starting with CC 3.0 and 3.5. GPUs with CC 3.0 introduced the already mentioned shuffle functions that allow to access memory from other threads within the same warp without additional hardware registers. Further, Kepler added a third mode to the L1 cache and shared memory, to equally distribute them to 32 kB each. The shared memory supports to adjust the size of shared memory banks either to serve 32 or 64 B banks, depending on the data that is supposed to be stored. Further, the operation mode of the L1 cache had been changed to only serve local memory accesses. Most likely this was done to remove any synchronization between the SMs for global memory, to reduce the communication between the SMs. With the Kepler architecture it was also possible to use so called *"unbound textures"*, which do not have to be explicitly bound to a texture reference, but can be passed to the kernel as an execution parameter. However, these textures still need to be explicitly initialized by the host code and have also the same restrictions as textures using texture references.

With the second generation of Kepler (CC > 3.5) Dynamic Parallelism has been introduced, allowing that kernels start other kernels directly from the GPU. Another innovation was the `__ldg(ptr*)` function. It allows to access the texture cache, without binding any textures in advance and removing the limitations such as the limitation to 2^{27} elements.

3.2.3 Maxwell Architecture

The first generation of Maxwell [NVIDIA 2014b] (CC = 5.0) GPUs was released in 2014 with the Geforce GTX 745, 750 and 750 Ti GPUs, followed by the second generation cards in the 900 series (CC = 5.2) and Jetson TX1/Tegra X1 (CC = 5.3) embedded processors. This architecture introduced significant changes to the memory system. First of all, the shared memory no longer can be configured, neither the capacity nor the memory bank size. Further, the L1 cache has been merged with the non-coherent cache. Also the implementation of atomic operations for global and shared memory had been significantly improved.

3.2.4 Pascal Architecture

Pascal [NVIDIA 2016d] (CC = [6.0, 6.1 and 6.2]) is the newest GPU architecture, released in 2016 with the Geforce GTX 10XX and Tesla P series. One of the new features is, that the GPUs allow to actively swap data between the device and the host system during the execution of a kernel. This allows to use more memory than the device is equipped with in one kernel. Further, the Tesla P100 features the new HBM memory. This allows much higher memory capacities (up to 16 GB) and bus width (4096 Bit).

NVIDIA CUDA Programming Guide, v8, Table 12

	Fermi	Kepler				Maxwell			Pascal		
	2.x	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Unified Memory Programming	No	No									
Funnel shift (see reference manual)	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes

NVIDIA CUDA Programming Guide, v8, Table 13

	Fermi	Kepler				Maxwell			Pascal		
	2.x	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	16	4	32	32	32	32	16	128	32	16
Maximum x-dimension of a grid of thread blocks	65535	2^{31}-1	2^{31}-1	2^{31}-1	2^{31}-1	2^{31}-1	2^{31}-1	2^{31}-1	2^{31}-1	2^{31}-1	2^{31}-1
Maximum number of resident blocks per multiprocessor	8	16	16	16	16	32	32	32	32	32	32
Maximum number of resident warps per multiprocessor	48	64	64	64	64	64	64	64	64	64	64
Maximum number of resident threads per multiprocessor	1536	2048	2048	2048	2048	2048	2048	2048	2048	2048	2048
Number of 32-bit registers per multiprocessor	32 K	64 K	64 K	64 K	128 K	64 K	64 K	64 K	64 K	64 K	64 K
Maximum number of 32-bit registers per thread block	32 K	64 K	32 K	64 K	64 K	64 K	64 K	32 K	64 K	64 K	32 K
Maximum number of 32-bit registers per thread	63	63	255	255	255	255	255	255	255	255	255
Maximum amount of shared memory per multiprocessor	48 KB	48 KB	48 KB	48 KB	112 KB	64 KB	96 KB	64 KB	64 KB	96 KB	64 KB
Cache working set per multiprocessor for constant memory	8 KB	8 KB	8 KB	8 KB	8 KB	10 KB	10 KB	10 KB	10 KB	10 KB	10 KB
Cache working set per multiprocessor for texture memory	12 KB	Between 12 and 48 KB	Between 12 and 48 KB	Between 12 and 48 KB	Between 12 and 48 KB	Between 12 and 48 KB	Between 12 and 48 KB	Between 12 and 48 KB	Between 24 and 48 KB	Between 24 and 48 KB	Between 24 and 48 KB
Maximum width, height, and depth for a 3D texture reference bound to a CUDA array	2048^3	4096^3	4096^3	4096^3	4096^3	4096^3	4096^3	4096^3	4096^3	4096^3	4096^3
Maximum number of textures that can be bound to a kernel	128	256	256	256	256	256	256	256	256	256	256
Maximum number of surfaces that can be bound to a kernel	8	16	16	16	16	16	16	16	16	16	16

Table 3.1: Extract of compute capability dependent features and technical details [NVIDIA 2016a, Table 12 and 13]. As can be seen, significant differences are present even within the same architecture.

Chapter 4
Auto-Tuning and Related Work

This chapter starts with introducing the term *auto-tuning*, techniques and concepts used in auto-tuning and provides an overview of a wide range of auto-tuning projects and publications. In general, the term auto-tuning is not consistently defined. It is usually used to describe automated systems that tune certain parameters to achieve a specific objective. This objective can be to optimize the *utilization of compute resources*, achieve *optimal performance*, provide certain *quality of services* or to reduce the *energy consumption*. In mathematical terms auto-tuning is an *optimization problem*. It is difficult to pinpoint an exact date or publication that marks the beginning of auto-tuning research because the term is not clearly defined. Actually the first optimizing compiler could already be seen as an auto-tuner, as it automatically optimized the performance of the code. To limit the extend for this thesis, we define auto-tuning as a method that automatically adjusts an application to operate optimally on the executing hardware.

The idea of such auto-tuning applications and frameworks is, to optimize existing code for a given hardware. Software is usually written once and used over years, while hardware is constantly improved and enriched with new features, so that code that was written in the past might not leverage the full capabilities of the processor. Without an auto-tuning system, the code has to be readjusted by an experienced programmer who not only knows the abilities, strengths and weaknesses of the hardware the application is supposed to run on, but also the application and the used algorithms. This procedure needs to be repeated whenever new hardware is released. Especially for long living codes this is a significant engineering and cost overhead. With increasing number of different processor types, it is necessary to not only optimize the existing code to the unique specifications, strengths and weaknesses of the processors, but also to evaluate alternative algorithms that are more efficient for the given hardware. Auto-tuning (regarding our focus) aims at solving this problem. The goal is to establish a system that can be integrated into an application that automatically analyzes and optimizes the application without any user interaction. This not only saves time for the tuning itself, but also enables software users (even when they are no experienced programmers) to optimize their applications towards the hardware.

As already mentioned, it is difficult to pinpoint an exact publication that initiated auto-tuning research, but certainly the *ATLAS* auto-tuner [Whaley and Dongarra 1998] is one of the first publications. It automatically optimizes *Basic Linear Algebra*

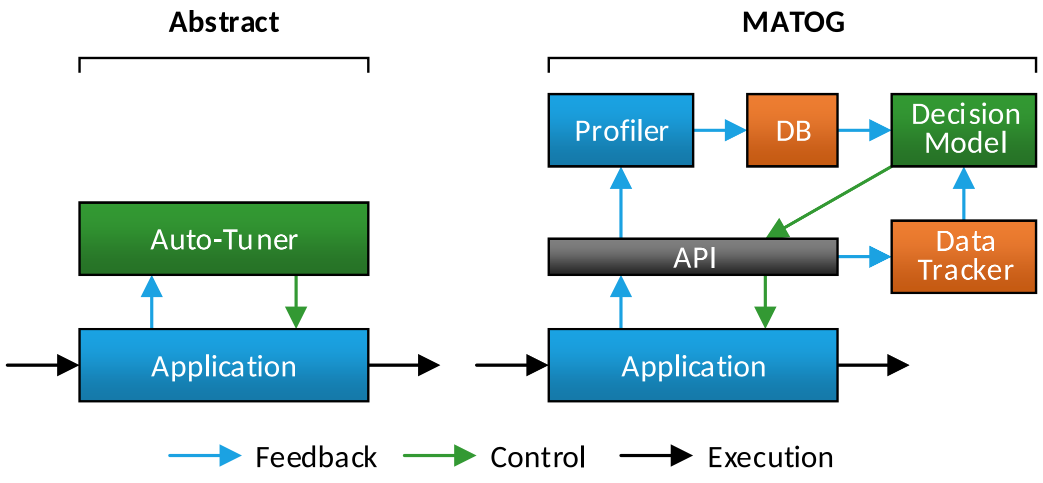


Figure 4.1: Left: Schematic feedback loop of an auto-tuner. The exact implementation of the *control* and *feedback* depends on the application and what the auto-tuner is optimizing. The feedback usually contains some kind of measure that characterizes performance of the applications (execution time, energy consumption, ...), but also other metrics (cache hit-/miss-rates, hardware utilization, ...) or meta data (data amount, hardware parameters, ...) that might help the auto-tuner to derive better decisions. **Right:** Schematic of the MATOG auto-tuner. MATOG bases its decisions on meta data tracked during the execution and performance data captured in previous executions. More details will be explained in Chapter 5.

*Subprograms **(BLAS)*** operations for CPUs. Another is *Active Harmony* [Hollingsworth and Keleher 1998], an auto-tuning system for optimizing applications in parallel, dynamic environments. Karsai et al. [2001] described an abstract self-adaptive software system that monitors performance metrics and automatically tunes parameters using a feedback loop. They port the concept of feedback based system control (used in other engineering disciples) to the field of software development. Figure 4.1 illustrates an example for such a feedback based application. For a given set of *tuning parameters* (which can either be specified by the auto-tuner or the programmer) it analyzes the application and how it performs in terms of its objective function on the given hardware. According to the objective function it decides, which values for the tuning parameters are likely optimal. In the following we call a combination of tuning parameter values a *configuration*. Given this abstract description, auto-tuning can actually be defined by the following four building blocks. Informally they can be referred as *"The Programmer's Guide to build an Auto-Tuner"*:

1. *"What to optimize?"*
2. *"How are optimizations integrated into the application?"*
3. *"Which configurations are optimal?"*
4. *"How to detect and handle runtime dependent performance effects?"*

In the following we will discuss these building blocks separately. As the number of publications in this topic is very high, we first only concentrate on a few hand picked examples to give an overview of the topic and used techniques (Sections 4.1 to 4.4) and later take a wider look to the different auto-tuners and concepts proposed in the literature (Section 4.5).

4.1 Options for optimization

The variety of available optimizations in applications seems to be nearly endless. It starts with simple and easily applicable optimizations such as compiler flags [Ansel 2014]. As compute intensive applications usually iterate over large datasets, it is very important to transform loops [Hall et al. 2009] and apply optimized launch configurations [Bergstra et al. 2012; Liu et al. 2008]. Lutz et al. [2015] adjust the usage of API calls to overlap memcopy and execution. Magni et al. [2014] apply thread coarsening by letting one thread do the work of multiple threads, which optimizes the resource usage. As GPUs are used for their compute power and memory bandwidths, an optimized data access is necessary. This can either be done by specifically targeting particular operations such as in 3D stencil codes [Lutz et al. 2013], dense [Sorensen 2012] or sparse [Choi et al. 2010; Monakov et al. 2010] matrix vector multiplications, or by applying application independent optimizations, e.g., tuning the cache utilization [Li et al. 2015], data mapping and paritioning [Ben-Nun et al. 2015], data placement [Li 2016] or data layouts [Kofler et al. 2015; Peng et al. 2016]. [Edwards and Trott 2013] provide an optimization API for data layouts that does not perform any automatic optimizations. Sung et al. [2012] propose a framework that transforms AoS to AoSoA based on static rules and executes explicit data transformations prior the kernel launch. Hsu et al. [2014] extend this work by proposing a hardware controller that performs these layout transformations during memcopy, eliminating the conversion overhead. More general are approaches such as [Muralidharan et al. 2014; Ansel 2014; Nugteren and Codreanu 2015] that allow user specified optimizations, that can, e.g., switch between alternative algorithms. MATOG optimizes data layouts, placement and utilization of caches and user specified optimizations.

4.2 How to integrate optimizations into applications?

There are multiple ways to integrate optimizations into an application. A popular technique is the usage of compilers. Some approaches use code annotations [Liu et al. 2008; Han and Abdelrahman 2009; Lee and Vetter 2014] to generate parallel code for given serial code, or use them to apply optimizations such as loop transformations. Li et al. [2015] apply their optimizations directly to the PTX intermediate format. Auto-tuners that allow user defined optimizations (as mentioned before in Section 4.1), usually require using a specific API to instruct the auto-tuner what to optimize. They use annotations or macros inside the code to enable/disable features or adjust parameters. For MATOG we decided to use code generation that has to be manually integrated into the application. On the one hand, this is less comfortable to use, as it requires some manual work. On the other hand, our code generation makes the code easily portable between different operating systems and compilers. Our API mimics the CUDA Driver API and therefore requires only a few changes to existing CUDA applications. Listing 4.1 shows an overview of different data layouts, how these are implemented in CUDA and how these can be accessed using the MATOG data-structures. These data-structures map the memory access transparently onto one of the methods.

4.3 Which configurations are optimal?

Finding optimal configurations or values for parameters is essential for auto-tuning. Mainly there are two methods: first, to analyze the code (*static analysis*) and, second, to measure the execution time (*profiling*). Peng et al. [2016] recently proposed a metric to estimate if either AoS, SoA, AoSoA or SoAoS yield better performance. The advantage of such static analysis methods over profiling is that they can be performed quite fast. However, they ignore data dependent effects that can occur at runtime, as it solely operates on heuristics that make certain assumptions about the data and workload of the application. Further, the hardware of GPUs is proprietary, which makes it difficult to establish suitable models for static analysis methods, especially if a new architecture with significant internal changes is released. Hong [2009] established an analytical model for CUDA code. However, they mention that their model cannot represent cache misses. As they worked on a GPU that does not have a cache for off-chip memory this is no problem. In contrast, modern GPUs have even multiple caches for this type of memory, so their model is likely not very accurate today.

The second technique is empirical profiling. To execute the application in different implementations is quite time intensive, so that multiple approaches have emerged to reduce the required time. Kofler et al. [2015] use a hybrid approach, as they

```
/************************** AoS-Layouts **************************/
array[index].field                              // AoS
array.field[index]                              // SoA
array[index/tile_size].field[index%tile_size]   // AoSoA
» array[index].field «                          // MATOG

/*********************** Multi-Dimensional ***********************/
array[x + y * size_x + z * size_x * size_y]     // array[x][y][z]
array[x + z * size_x + y * size_x * size_z]     // array[x][z][y]
array[y + x * size_y + z * size_y * size_x]     // array[y][x][z]
array[y + z * size_y + x * size_y * size_z]     // array[y][z][x]
array[z + x * size_z + y * size_z * size_x]     // array[z][x][y]
array[z + y * size_z + x * size_z * size_y]     // array[z][y][x]
» array[x][y][z] «                              // MATOG

/************************ Texture Memory ************************/
tex1Dfetch(textureReference, index + offset)    // until CC 3.0
__ldg(&array[index])                            // since CC 3.5
» array[index] «                                // MATOG
```

Listing 4.1: Overview of how AoS, multi-dimensional arrays and texture can be used in CUDA compared to MATOG. Also combinations of these layouts can be used with MATOG, e.g., `array[x][y][z].field`.

utilize a generic GPU benchmark to capture GPU specific properties and then project them onto the code to estimate the performance using a directed execution graph. Magni et al. [2014] train a neural network based on a dataset consisting of static code features and best candidates for some reference benchmarks. With this trained model they are able to predict optimal thread coarsening factors based on static code features. Other approaches use methods such as an exhaustive search [Muralidharan et al. 2014], greedy search [Liu et al. 2008], swarm search, simulated annealing [Nugteren and Codreanu 2015], Nelder-Mead downhill simplex method [Chung and Hollingsworth 2004], or a genetic algorithm [Ansel 2014] to guide the search into the direction of the optimal configuration. We proposed a prediction based profiling method that requires only a very small subset of configurations to estimate the execution time of non-profiled configurations. This method is based on the observation that changes induced by one optimization have little influence on other optimizations. With this technique it is possible to estimate the optimal configuration with only a small number of executed configurations, making this method very fast while it achieves nearly the same results in comparison to an exhaustive search.

4.4 How to detect and handle runtime dependent performance effects?

The performance of some applications changes depending on varying input data. These data dependent effects can benefit from dynamic decisions that are made at runtime. To establish such a decision system, it is necessary to find measures to identify these effects and to create decision models that can distinguish between them.

To dynamically select optimal configurations during runtime it is necessary that the auto-tuner is able to identify different data properties. Approaches such as presented by Muralidharan et al. [2014] or Chung and Hollingsworth [2004] rely on user defined callback functions. This requires that the user knows how to characterize the properties of his data. In contrast, we use a fully automatic process that relies on meta data, which is already contained in our system (i.e., the launch configuration of a kernel or the size of arrays). Additionally, MATOG also supports to monitor user defined variables that can contain additional information than our automatic data.

With the ability to identify different data properties, it is possible to create decision models based on machine learning techniques, e.g., regression trees [Bergstra et al. 2012] or *Support Vector Machines* ***(SVMs)*** [Muralidharan et al. 2014]. First, we relied on SVMs, but encountered cases with high false decision rates. In this thesis we propose an improve decision model that achieves better decisions for our optimization problem. If the extend of the used meta data is unknown, it can happen that models derive bad decisions, when data significantly differs from the training data. In our case a specialized nearest neighbor search using an implicit normalization yields equal or better results compared to a SVM. We further explain this in Section 6.4.1.

4.5 Overview

In this section we give an overview of existing auto-tuners. The literature in this area is very extensive, so we clustered the literature in to categories, starting with performance measurement, modeling and simulation (Section 4.5.1), compilers (Section 4.5.2), programming languages (Section 4.5.3), domain dependent (Section 4.5.4) and independent approaches (Section 4.5.5), as well as memory access and data layout auto-tuning (Section 4.5.6).

4.5.1 Performance Measurement, Modeling and Simulation

Measuring performance of hardware is a key component to many auto-tuners. However, most hardware implementations are closed. To reveal implementation details of hardware, Wong et al. [2010] proposed a method based on micro-benchmarking. Mei and Chu [2017] propose a similar micro-benchmarking method to analyze the GPU memory hierarchy. Accuracy is a very important factor in performance measuring. A high profiling overhead can not only be time consuming but also introduce distortions into the measurement. Therefore, Baghsorkhi et al. [2012] provide a framework for profiling memory hierarchy effects in GPU applications. Zheng et al. [2012] proposed a low-overhead profiler for GPUs, called *GMProf*. To find an optimal configuration is a key component for auto-tuning. Oliveira Castro et al. [2013] proposed an adaptive profiling technique that concentrates its search on parts of the solution space with high irregularity.

Modeling performance is another important research topic, as it allows to predict how long an execution will take without actually running the code. However, as the internals of hardware is usually proprietary, it is difficult to predict the performance of an unknown hardware. Hong [2009] introduced a performance model for CUDA code that is able to reliably predict the performance of the used hardware, but as they do not model cache misses, their method most likely fails on today's hardware. Baghsorkhi et al. [2010] proposed similar work that explicitly takes global memory latency and shared memory bank conflicts into account. Wang and Chu [2017] use micro-benchmarking and some performance counters to predict the performance of GPU kernels based on the used core and memory frequency. *GROPHECY* [Meng et al. 2011] estimate the performance of a GPU implementation given a simple CPU skeleton implementation, which helps developers to predict if their code would run faster when executed on GPUs. Calotoiu et al. [2013] use empirical profiling data and a performance model to find scalability problems in complex parallel programs. Similarly, Shudler et al. [2015] build performance models to predict the scaling behavior of cluster applications. Calotoiu et al. [2016] extend their previous work to create multi-parameter performance models. Other machine learning based performance prediction methods have been proposed by Ipek et al. [2005], Lee et al. [2007], and Wu et al. [2015]

Bakhoda et al. [2009] use a different approach and proposed *GPGPU-Sim* that actually simulates CUDA PTX code. This provides an execution time estimate and allows to capture other internals of the execution. Similar work was proposed by Collange et al. [2009]. Other simulators are *WASTE*[1] or *Ocelot*[2]. However, these projects are discontinued and no longer maintained, so that current GPU hardware

[1]code.google.com/archive/p/cuda-waste/
[2]gpuocelot.gatech.edu

cannot be simulated and new CUDA features cannot be used.

4.5.2 Compilers

Compilers are widely used in programming to translate textual code into machine code. These employ certain optimizations to the code, based on heuristics, to achieve higher performance through more ILP, better jump predictions or other optimizations. To improve these heuristics, Stephenson et al. [2003], Agakov et al. [2006], Fursin et al. [2008], and Park et al. [2011] use machine learning techniques that help to guide the search for optimal implementations. Long and Fursin [2005] provide a heuristic search algorithm to efficiently locate good optimizations in compiler search spaces.

Automatic parallelization is a wide research topic. There exist different techniques for multi-core (OpenMP) or accelerator (OpenACC) programming, which require the code to be annotated with instructions how to parallelize the given code. Iwainsky et al. [2015] investigated scalability properties of OpenMP applications on different hardware and using different compilers from various vendors. Approaches such as presented by Wang et al. [2015] use OpenMP code to generate optimized OpenCL code, that can be executed on an accelerator. They further use a predictor to estimate whether running the code on GPU will be faster than on a CPU and automatically select the optimal device. Kim et al. [2016] proposed a similar framework that explicitly tries to remove unnecessary memory transfers. Afonso et al. [2016] automatically generate OpenCL from annotated Android Java code for mobile devices. OmpSs[3] is an approach to extend OpenMP for heterogeneous devices. Elangovan et al. [2015] auto-tune code that was generated using OmpSs-OpenCL. Lee and Vetter [2014] published *OpenARC*, an OpenACC compiler that aims at automatically optimizing the resulting parallel implementation. Siddiqui et al. [2014] auto-tune OpenACC parameters using data from prior executions. Lashgar and Baniasadi [2016] investigate the effectiveness of different OpenACC cache directives and draw conclusions when to use which settings. Calore et al. [2016] use OpenACC to achieve performance portability of accelerated lattice Boltzmann applications. *Hybrid Multicore Parallel Programming **(HMPP)*** [Dolbeau et al. 2007] is a directive-based language that also aims at automatic parallelization of serial code for GPUs. Grauer-Gray et al. [2012] auto-tune the directives of HMPP to achieve high performance.

Going even further, Marangoni and Wischgoll [2016] proposed DawnCC, a compiler that has the goal to generate OpenACC code from serial C code without any annotations. However, their results indicate that except for one benchmark, their

[3]pm.bsc.es/ompss

method does not really improve anything through the parallelization. In some cases their method even decreases the performance. Commercial products such as the Silexica SLX Parallelizer[4] also automatically parallelize serial C code. Barman et al. [2011] automatically synthesize parallel code for scan based algorithms. Langdon et al. [2016] use genetic programming to generate parallel code from serial C code. *togpu* [Marangoni and Wischgoll 2016] automatically translates CPU C++ code to CUDA.

4.5.3 Programming Languages

Beside implementing auto-tuners as libraries or frameworks, some approaches define their own optimizable programming language. The advantage of this method is that it can directly define optimizations in the language. However, the drawback is that the program needs to be written in this particular language. These languages usually have restrictions or requirements that can make it difficult to implement certain algorithms, as they do not allow all constellations of algorithm or language constructs, which are available in other high-level languages.

An example for these optimizable languages is *Sequoia* [Fatahalian et al. 2006] that breaks down memory access onto different levels, making it easy to map memory access on different levels of memories, as it is, e.g., the case for GPUs or special purpose processors. Ansel et al. [2009] proposed *PetaBricks*, a language that natively supports to provide multiple implementations of the same algorithm on CPUs. Its compiler automatically selects not only the best working implementation, but also auto-tunes parameters if possible. This was used by Chan et al. [2009] to optimize implementations of the multi-grid technique that is used to solve partial differential equations. In Ansel et al. [2011] a new evolutionary algorithm was presented that optimizes PetaBricks based applications and finds solutions much faster than other approaches. Pacula et al. [2012] proposed an in-situ method to optimize PetaBricks based applications. Hall et al. [2009] define transformation recipes, which can be used to transform loops and optimize their execution behavior through auto-tuning. Han and Abdelrahman [2009] and Han and Abdelrahman [2011b] propose *hiCUDA*, a directive-based language extension to enable automatic CUDA kernel generation from serial code, which can be seen as a predecessor of OpenACC. *PyCUDA* and *PyOpenCL* [Klöckner et al. 2011] allows to run CUDA or OpenCL application from Python. *Copperhead* [Catanzaro et al. 2010] is based on this, as it uses a Python based language to provide data parallel instructions and building blocks to facilitate commonly used execution primitives on GPUs. Rudy et al. [2011] proposed CUDA CHiLL, a parallel language that maps down to parallel building blocks and instructions, which allow to gener-

[4]www.silexica.com

ate CUDA code. Khan et al. [2013] uses CUDA CHiLL to apply code transformations to achieve highly efficient CUDA code. Devito et al. [2013] proposed *Terra*, which is a Lua-based language that enables to write highly efficient parallel GPU code. *TANGRAM* [Chang et al. 2016] synthesizes efficient portable code based on code skeletons that use specific qualifiers, primitives and containers. *Heterogeneous Habanero-C **(H2C)*** [Majeti et al. 2016] uses an abstract language to optimize code for parallel execution and explicitly optimize the usage of data layouts. Pony[5] is an object-oriented, actor-model based programming language for HPC applications.

Other languages are directly designed for a specific application domain. For example, Hong et al. [2012] proposed *Green Marl* that enables to write efficient graph analysis algorithms. Vollmer et al. [2015] proposed an auto-tuning framework for optimizing functional programs, written in Obsidian, a Haskell based GPU programming language.

4.5.4 Domain Dependent Auto-Tuning

There are many application domains that can benefit from auto-tuning. Matrix multiplications have been targeted in many auto-tuning publications, as it is a major building block in many applications. These can be categorized in *Dense Matrix-Vector Multiplication **(GEMV)*** [Sorensen 2012], *Sparse Matrix-Vector Multiplication **(SpMV)*** [Vuduc et al. 2005; Choi et al. 2010; Guo and Wang 2010; Guo et al. 2011; Tang et al. 2015; Zhang et al. 2016] and *Dense Matrix-Matrix Multiplication **(GEMM)*** [Whaley and Dongarra 1998; Kurzak et al. 2012; Matsumoto et al. 2012; Steuwer et al. 2016; Veras et al. 2016]. Mainly these approaches try to optimize different algorithms, storage formats, thread and data mapping or launch configurations. Bell and Garland [2008] introduced multiple implementations for efficient SpMV on GPUs, for a variety of sparse storage formats, without automatic selection of optimal layouts. This automatic selection has later been done by Muralidharan et al. [2014]. Beaumont et al. [2016] proposed a framework for efficient scheduling of linear algebra kernels on heterogeneous hardware.

Another important operation in many applications are stencil codes. Approaches such as presented by Datta et al. [2008], Kamil et al. [2010], Monakov et al. [2010], Christen et al. [2011], Lutz et al. [2013], Zhang and Mueller [2013], Luo et al. [2015], and Jia and Zhou [2016] provide auto-tunable implementations for common stencil operations that are adjusted towards the given computational problem and hardware.

Halide [Ragan-Kelley et al. 2012; Ragan-Kelley et al. 2013] is a domain specific language for image processing pipelines. It supports common operations as building

[5]ponylang.org

blocks, which make the code easy to understand. Similar work was done by Yang et al. [2016], but instead of using an own language, they rely on Python.

Other projects aim at solving large tridiagonal system [Davidson et al. 2011], *Digital Signal Processing **(DSP)*** [Püschel et al. 2004], *Discrete Fourier Transformation **(DFT)*** [Frigo and Johnson 1998; Frigo 1999; Frigo and Johnson 2005], ray-tracing [Ganestam and Doggett 2012], hypernuclear spectroscopy [Bianchin et al. 2008; Bajrovic et al. 2013], graph based algorithms [Pai and Pingali 2016], or *Neural Networks **(NNs)*** [Li et al. 2016b; Moskewicz et al. 2016; Imani et al. 2017].

4.5.5 Domain Independent Auto-Tuning

Beside optimal performance, energy consumption is an important optimization objective. Hoffmann et al. [2010] proposed *Application Heartbeats*, an API that allows to develop auto-tuners. It supports an interface, allowing to pass monitored performance data from the application to the auto-tuner and execution parameters from the auto-tuner to the application. *PowerDial* [Hoffmann et al. 2011] aims at reducing computational accuracy for power efficiency and can use the Heartbeats API. Coplin and Burtscher [2015] investigate the effects of source-code optimizations for GPUs, concerning performance and energy consumption. Bao et al. [2016] proposed a compile-time based CPU frequency selection that outperforms runtime based approaches. Sensi et al. [2016] select optimal configurations for either performance or power consumption without relying on previous application runs. Some auto-tuners combine the energy consumption and performance objectives and try to find an optimum that satisfies both. For this, Jordan et al. [2012] introduced a multi-objective optimization infrastructure that generates multi-versioned executables, allowing to select optimal configurations during runtime. Durillo and Fahringer [2015] provide an overview of the term *"auto-tuning"* and discuss the advantages and problems of multi-objective auto-tuning. The AutoTune[6] project aims at providing auto-tuning for cluster applications. As part of the project, Miceli et al. [2013] proposed the *Periscope Tuning Framework **(PTF)***, which is a plug-in driven framework that aims at providing a solid base structure for auto-tuning of HPC applications. Preliminary results of their research have been presented in Miceli and Bodin [2013]. Pimenta et al. [2013], Liu et al. [2014], and Sikora et al. [2016] optimize the usage of *Message Passing Interface **(MPI)*** in cluster applications using PTF. The *Runtime Exploitation of Application Dynamism for Energy-efficient eXascale computing **(READEX)*** project [Gerndt 2016] is a successor of the AutoTune project and explores the potential of dynamic auto-tuning for energy saving in HPC environments. Other energy optimizing auto-tuning techniques have been proposed by Götz et al. [2010], Tiwari et al. [2011], and Tomusk

[6]www.autotune-project.eu

et al. [2016].

Some auto-tuners are not specifically designed for a specific application. Many auto-tuners provide a framework that is able to find optimal parameter values for the given application. Hollingsworth and Keleher [1998] introduced the *Active Harmony* framework that is designed to tune parameters in distributed systems. In this initial version the authors used a greedy based empirical profiling to find optimal parameters, which was replaced with the Nelder-Mead method by Tapus et al. [2002]. Chung and Hollingsworth [2004] added a parameter prioritization to better guide the search in high dimensional search spaces. Further, they keep record of previous profiling data to feed them also into the optimization process. Tiwari et al. [2009] used Active Harmony for the search of optimal parameters for the CHiLL compiler framework [Rudy et al. 2011]. Chang and Karamcheti [2001] provided an auto-tuner to optimize distributed applications. Specifically, they allow to adjust parameters during runtime, according to user defined constraints. A similar idea was proposed by Bhat et al. [2006] which also optimize distributed applications that run on remote supercomputing sites. The *GADAPT* auto-tuner [Liu et al. 2008] bases its optimization decisions on a heuristic-based empirical search and builds decision models that decide based on the program parameters, which implementation to choose. For this they compile the application in multiple different configurations and use a dispatcher that selects and executes the optimal configuration. The *OpenTuner* [Ansel et al. 2014] framework is a python based auto-tuner designed to optimize compiler flag values. Initially it was used to optimize GCC compiler flags, but it also can be used for other compilers such as the NVIDIA CUDA Compiler [Bruel et al. 2015] or to enable preprocessor based optimizations. *CLTune* [Nugteren and Codreanu 2015] is a similar framework, specifically targeting OpenCL applications. Muralidharan et al. [2014] proposed the *Nitro* auto-tuning framework. Nitro uses an exhaustive search with empirical profiling and therefore is mainly supposed to optimize applications with only few possible configurations. It features a callback mechanism to feed user defined meta data into the SVM based decision system. One of their benchmarks chooses optimal layouts for SpMV based algorithms. In Muralidharan et al. [2016a] the framework is extended to predict optimal configurations for unknown GPUs, based on micro-benchmarking. Tillmann et al. [2013] introduced *ATuneRT* which aims at optimizing parameters in applications, e.g., the launch configuration of kernels. In Tillmann et al. [2016] they apply it on a KD-Tree building and ray casting pipeline for real-time rendering with an in-situ optimization. In this example they optimize, i.e., the costs for the KD-Tree building heuristic.

Other projects go one level lower. These do not tune specific parameter values, but explicitly apply optimizations to the implementation. Han and Abdelrahman [2011a] targeted branch divergence in GPU applications by either delaying an execution inside a loop or by aggressive branch transformations. In Han and Abdelrahman [2013] they reduce branch divergence by merging loops. However, both methods require the optimizations to be applied manually. Magni et al. [2014] provide a compiler chain that applies thread coarsening to reduce similar calculations and optimize resource usage on GPUs. Xu and Gregg [2015] use a compiler, which employs hyper loop parallelism to merge multiple SIMD executions into one thread. The framework of Gao and Peterson [2015] analyzes shared memory bank conflicts and allows to automatically optimize the shared memory access. Li et al. [2015] try to achieve higher performance by explicitly by-passing caches on GPUs. They determine a certain threshold and only allow thread groups within this threshold to use the caches, the others circumvent these and directly access the off-chip memory. Moreira et al. [2017] perform call re-vectorization for SIMD platforms such as CPUs and GPUs. This method wakes up dormant threads to collaborate, e.g., in a memcopy.

In recent years, executing code on heterogeneous devices has become very popular. This allows to schedule tasks onto devices, which are better suited for a specific algorithm. *Dandelion* was proposed by Rossbach et al. [2013]. It uses a C# or F# implementation of an application and maps predefined parallel building blocks to generate device code. This can then be mapped on different heterogeneous devices, such as CPUs, GPUs, FPGAs or cloud applications. The *Heterogeneous Programming Library* ***(HPL)*** [Viñas et al. 2013] easily allows to run OpenCL kernels on different kind of hardware. It allows to write code similar to the CUDA Runtime API. Fabeiro et al. [2014] extend HPL to allow auto-tuning of parameters. Viñas et al. [2016] added analytical models for optimal workload balancing. Gadioli et al. [2014] proposed a similar framework, targeting heterogeneous OpenCL applications with auto-tuning and runtime resource management. Jääskeläinen et al. [2014] introduced an OpenCL performance portability optimizing compiler that modifies kernels in a way that they work optimal on the given architecture. Paone et al. [2014] propose a runtime resource management system and conclude that this objective can be orthogonal to auto-tuning objectives. Grasso et al. [2013] provide a compiler that uses a single-device OpenCL application and transforms it into a multi-device application. They leverage a machine-learning based prediction model using static program and dynamic input features, to predict an optimal partitioning. *Helium* [Lutz 2015] is an OpenCL framework to optimize the usage of OpenCL API functions, that postpones, gathers or even removes API calls to optimize the overall execution time. Bodin et al. [2016] proposed *Diplomat* which generates optimal mapping of kernels onto CPUs and GPUs. Bolchini et al. [2016] published an operating system based resource scheduler for scheduling OpenCL applications on heterogeneous hardware. Cheng et al. [2017] auto-tune the task scheduling of heterogeneous MapReduce cluster applications. As more and more heterogeneous computers and clusters are used, many approaches proposed frameworks to assist the development, optimization and scheduling of such applications [Bauer 2014; Bajrovic and Benkner 2014; Chang et al. 2016; Fachada et al. 2016; Gray and Stratford 2016; Hechtman et al. 2016; Helal et al. 2016; Muralidharan et al. 2016b; Panneerselvam and Swift 2016; Rossi and Zhou 2016; Srivastava et al. 2016; Zenker et al. 2016; Inggs et al. 2017; Yamato 2017].

As data partitioning in multi- and heterogeneous-device applications is very important, as well as optimizing the memory transfer between devices, *MAPS* [Rubin et al. 2014; Ben-Nun et al. 2015] was proposed. It is a framework aiming at data abstraction and partitioning for single- and multi-GPU applications. It is specifically designed to map onto commonly used operations in GPU programming. Sakai et al. [2016] automate mapping single-GPU applications onto multiple GPUs. *SkePU* [Enmyren and Kessler 2010] uses code skeletons for multi-GPU programming. Arslan et al. [2016] proposed *HARP*, a predictive based auto-tuner for application level data transfer, which takes historical data analysis and real-time probing for its decision-making. *dCUDA* [Gysi et al. 2016] allows device-to-device memory access in CUDA applications on a cluster level with automatic latency hiding. Similar work is done by Tausche et al. [2016] for OpenCL.

4.5.6 Memory Access and Data Layouts Auto-Tuning

Optimal memory access is a key objective in optimal performance, especially on GPUs, as discussed by Ryoo et al. [2008]. Strzodka [2011] provides specialized C++ containers that allow to easily switch different AoS layouts and multi-valued containers [Strzodka 2012]. Edwards and Trott [2013] pursue a similar way, with providing adjustable containers. However, the optimizations of these approaches have to be applied by the user and are not automated. Sung et al. [2012] proposed *DL*, a framework that tunes the memory layouts in GPU applications. It uses static decision rules to determine, which data layout to be used. They provide an efficient data-conversion prior kernel runs, if the data is not in the correct format. Hsu et al. [2014] extended this approach by proposing a hardware conversion unit, that converts the data format during the memory transfer. Cantanzaro et al. [2014] introduced a method for efficient matrix in-place transposition on GPUs. Kofler et al. [2015] auto-tune access to struct arrays in OpenCL applications, including a tile size selection for AoSoA. Peng et al. [2016] propose a static cost estimation to select optimal struct array layouts. These two are properly the closest auto-tuners to our approach. In contrast, we do not rely on static analysis but use empirical profiling. Further, MATOG is able to apply significantly more optimizations, e.g., transpositions, memory placement, or selection of optimal L1 cache sizes.

Other approaches do not optimize on a high-level, data layout based approach, but directly target the actual implementation inside the code. Cruz et al. [2016] provide a mechanism to improve memory page hit rates, by analyzing the memory access behavior of parallel applications. Li et al. [2016a] optimize the usage of on-chip memory resources in CUDA applications. Ainsworth and Jones [2017] provide a compiler to enable software based prefetch for indirect memory accesses. Yang et al. [2010] provide a source-to-source compiler, which uses a naïve implementation and applies certain optimizations to enforce coalescence of memory access, thread regrouping and automatic usage of shared memory. Han and Abdelrahman [2014] learn a machine-learning model to predict if an array should be buffered in shared memory. Michaud [2016] provides mathematical proof of cache replacement effects and a new algorithm for reasoning about optimal cache replacements.

Besides changing the code, also the hardware could be changed. Park et al. [2015] proposed *Earliest Load First* ***(ELF)***, an alternative warp scheduling method that tries to maximize the memory-level parallelism on GPUs. Dublish et al. [2016] propose a change to the L1 caches in GPUs to employ a cooperative caching network, allowing to efficiently reuse data in these caches. Ziabari et al. [2016] proposed a unified hardware-based memory hierarchy for CPU/GPU systems.

Chapter 5
MATOG Auto-Tuner

MATOG is a recursive acronym and stands for *"MATOG: Auto-Tuning on GPUs"*. It is an application domain independent array layout auto-tuner for CUDA applications. We introduced MATOG first in Weber and Goesele [2014] and have constantly improved it since then. MATOG is open source and can be downloaded from our project page[1]. The main concept of MATOG is that it abstracts the memory access from the application and dynamically chooses optimal array layouts during runtime. In order to achieve this, MATOG requires to run and analyze the application with real data. During this profiling it measures the execution time of different data layouts, tracks meta data, and stores the results in a database (Section 6.2). These profiling results are then analyzed (Section 6.3) and used to build decision models that are utilized during runtime to select optimal configurations according to changing data properties (Section 6.4). As shown in Figure 5.1 there can be different optimal configurations of a kernel, depending on the input data. We automatically identify these data properties and adjust the layouts accordingly.

MATOG optimizes array access for one- or multi-dimensional arrays of primitive, struct and hierarchical types. Arrays of pointers are currently not supported. It arranges these arrays in different struct layouts, i.e., AoS, SoA or AoSoA (with a tile-size of 32, the size of a CUDA thread group). We do not support to separate the fields of an AoS and store these in different layouts. Further, multi-dimensional arrays can be stored in a transposed way. The array layouts can be separately optimized for global, shared and local memory. For read-only arrays residing in the global memory, MATOG determines, which of these arrays should use the default and which the non-coherent cache (also known as texture memory) to optimize the cache utilization. Arrays with constant size (known at compile time) can be placed in constant memory. On Fermi and Kepler GPUs the optimal ratio between L1 cache and shared memory size is automatically determined. Additionally, the programmer can define preprocessor optimizations, to use discrete values and value ranges in his code. This can be used to implement different algorithms for a task, or to evaluate a series of values as parameter of an algorithm as shown in Listing 5.1. Experienced users can further customize MATOG by defining their own indexing schemes, e.g., to implement a special triangular matrix, z-order curves, or to provide their own allocation/deallocation mechanism, in order to combine MATOG with other frameworks.

[1]matog.org

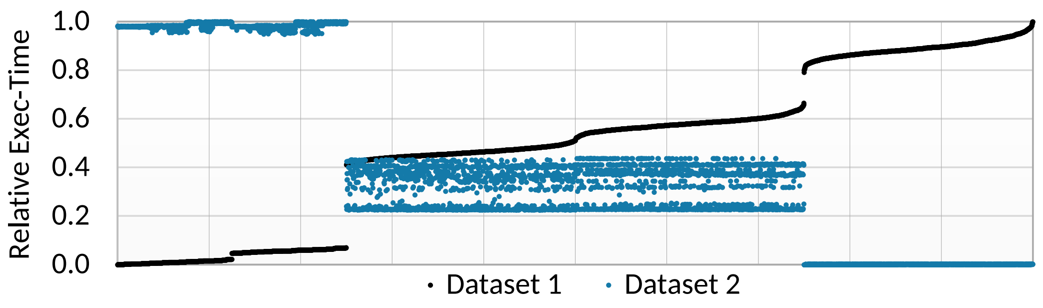


Figure 5.1: Execution time for two kernel runs with different data. All results have been normalized between 0.0 (best) and 1.0 (worst), and are sorted ascending for the first call (black). Configurations, which were optimal for the first call, became the worst in the second (blue) execution. The shown kernel is the first main kernel of our KD-Tree benchmark. The big jumps come from using local memory as a buffer for shared memory. This works good for the black iteration, but not for the blue one.

5.1 Programming Interface

The MATOG programming interface was designed to reduce programming overhead and to be as compatible to CUDA as possible. To support platform independence and easy maintainability, we decided to use code generation instead of an own source-to-source compiler, so no changes to the compile chain have to be applied. MATOG requires only small changes to the source code as it mimics the CUDA Driver API [NVIDIA 2016a] interface. It is necessary to use the Driver API instead of the Runtime API as the latter does not allow to dynamically load and exchange kernel implementations during runtime. To access data MATOG supports multi-dimensional memory access (e.g. `array[x][y].subarray[z].field`) similar to the code skeletons used by GROPHECY [Meng et al. 2011]. To apply its optimizations, it intercepts all communication to CUDA.

5.2 Programming Example

In order to use MATOG arrays, the programmer has to provide a *JavaScript Object Notation (JSON)* description (example shown in Figure 5.2) of the arrays, which is then used to generate the optimization code. This code can then be included into the application. The main contribution of MATOG is a dynamic selection of optimal parameters at runtime. However, MATOG automatically performs some trivial static optimizations, e.g., it ensures an optimized alignment inside the arrays to prevent alignment spacings (example shown in Figure 5.2). Further, the user can specify for each kernel how data is used (only read, only written or read and

```
// Define Example: values = [15, 42]
#if MY_DEFINE == 15
// algorithm 1 ...
#elif MY_DEFINE == 42
// algorithm 2 ...
#else
#error NOT_IMPLEMENTED
#endif

// Range Example: min: 32, step: 32, max: 512
__shared__ int bins[MY_BIN_COUNT];
for(...) {
  int binIndex = (int)(value[i] / MAX_VALUE * MY_BIN_COUNT);
  //...
}
```

Listing 5.1: Example code to show the define/range feature. The first code snippet shows how different implementations can be differentiated using a preprocessor define. The second shows how a range of values can be evaluated.

write) which enables certain optimizations. This is explained in Section 5.3.3 and 6.3 in more detail. The programmer can specify whether multiple arrays have always the same size. This allows to reduce the number of used registers, as normally every MATOG data structure maintains a separate copy for its sizes. Finally, compiler flags can be specified, to enable GPU debugging or specific features (e.g., `--use_fast_math`). There are three types of MATOG data structures that can be used in the host code: one for arrays located in host memory (`Array::Host`), one representing dynamic shared memory (`Array::Dyn`) and one for device memory (`Array::Device`). Data inside host arrays can be directly accessed. Data transfers are done by the CUDA memcopy functions. For kernel modules, the source code has to be provided instead of a pre-compiled CUDA module. MATOG takes care of the actual compile process in a just-in-time manner. Nothing else has to be changed for loading and executing GPU code, as MATOG array references can simply be put into the kernel argument list. Listing 5.3 shows a host code example. Changes are indicated compared to CUDA Driver API. Inside the kernel code, each memory type uses separate implementations. As multiple instances of global or dynamic shared memory arrays can have different layouts, a template is used to distinguish the different instances. Only for static shared memory types, the same layout is used for all instances of the same type. Listing 5.4 shows an example for kernel code. Figure 5.3 shows a schematic workflow of MATOG. In the development phase the user specifies the optimizable data structures, which

```
typedef struct {
  long rand;
  float result;
} MyStruct[3][7];
```

```
{ "name": "MyStruct",
  "counts": [0, 3, 7],
  "fields": {
    ["name": "rand", "type": "long"],
    ["name": "result", "type": "float"]
}}
```

Listing 5.2: Left: Example data structure in C++. **Right:** Same data structure in MATOG JSON notation.

Example Struct

```
struct {
  double a; // 64bit
  short b;  // 16bit
  double c; // 64bit
  char d;   //  8bit
  int e;    // 32bit
}
```

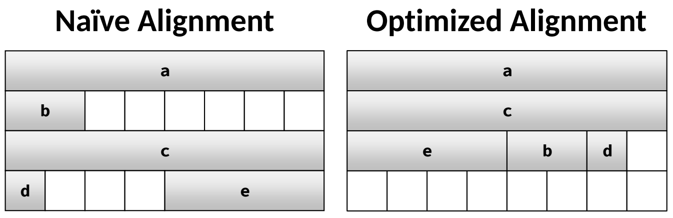


Figure 5.2: An example struct (left) and how it would usually be stored in CUDA (center). MATOG uses an optimized layout (right) since GPUs require n-Byte variables to be n-Byte aligned.

are then generated by the code generator. The user then incorporates these into the application and compiles it using a standard host compiler (e.g., GCC or Visual Studio). Listing 5.5 shows an example for using a MATOG based application and how to execute the optimization procedure. By default, MATOG runs in an unoptimized mode. To optimize it, MATOG has to be switched into a profiling mode, that measures the execution time of the kernels. Every time a kernel is executed during optimization, MATOG runs multiple implementations of the same kernel to determine which layouts work best. After the profiling, MATOG analyzes the results and builds decision models. These are used to determine optimal layouts during runtime.

5.3 Implementation Details

This section gives some more insights on the implementation of MATOG and explicitly its data structures. The main concept behind MATOG data structures is to provide a class that supplies an multi-dimensional AoS-like (e.g., `array[x][y].field`) memory access to the user and internally maps this onto an optimized layout. As

```
/************************* allocate data *************************/
int X = ..., Y = 3, Z = 7;
~~MyStruct* host = new MyStruct[X * Y * Z];~~
MyStruct::Host host(X, _fl);
~~CUdeviceptr device = cuMemAlloc(sizeof(MyStruct) * X * Y * Z);~~
MyStruct::Device device(X, _fl);

/*************************** load data ***************************/
for(...) {
  host~~[x + y * X + z * X * Y]~~[x][y][z].rand = rand();
  host~~[x + y * X + z * X * Y]~~[x][y][z].result = 0.0f;
}

/************************** load module **************************/
CUmodule module;
cuModuleLoad(&module, ~~"preCompiledCode.ptx"~~"sourceCode.cu");

/************************* load function *************************/
CUfunction function;
cuModuleGetFunction(&function, module, "kernel");

/***************************** exec *****************************/
void* args[] = {&paramA, &paramB, &device, 0};
cuLaunchKernel(function, ...);
```

Listing 5.3: Host Code Example: CUDA Driver API (stroked through) compared to MATOG (underlined)

MATOG supports to store data in hierarchical data structures, all sub-structures can use a different indexing and struct layout, so that the root, e.g., can be stored as an untransposed AoS, while the sub-array is stored as transposed SoA. However, we decided that sub-arrays cannot be stored as AoSoA because depending on the tile-size and number of elements, this layout wastes memory. As sub-arrays appear hundreds or thousands of times in the root-array, this wastage would be enormous. On the host system, MATOG uses a dynamic adjustable implementation, so that the host code does not need to be recompiled to switch the layouts. While this is very convenient to use, it can slightly decrease the performance of the CPU code, compared to a purely static implementation. In the device code, we explicitly compile the exact layout into the code, to achieve maximal performance. To provide the necessary information to the GPU code, we modify the kernel argument list and pass a struct to the GPU that has the same signature as the GPU

```
/************************* function body *************************/
__global__ void kernel(const float paramA,
                       const int paramB,
                       MyStruct~~*~~<> data)
{
  const int x = threadIdx.x, y = threadIdx.y, z = threadIdx.z;
  float result = 0.0f;

/********************** define shared memory **********************/
  __shared__ MyStructShared<128> shared~~[128*Y*Z]~~;

/**************** global memory » shared memory ****************/
  shared~~[x + y * X + z * X * Y]~~[x][y][z] =
    data~~[x + y * X + z * X * Y]~~[x][y][z];
  ...

/******************* register » global memory *******************/
  data~~[x + y * X + z * X * Y]~~[x][y][z].result = result;
}
```

Listing 5.4: Kernel code example: CUDA (stroked through) compared to MATOG (underlined)

data structure implementation and contains all necessary information, such as the pointers and sizes. However, one disadvantage of the way the data structures are designed is that we cannot make use of the *restrict* keyword and therefore are prone to pointer aliasing [Cook 1997].

5.3.1 Texture Memory

As described in Section 3.2 the way texture memory can be used has changed significantly throughout the generations. MATOG employs different implementations depending on the GPU generation. On Fermi and first generation Kepler cards, we use texture references. To circumvent the limitation of 2^{27} items per texture, we concatenate multiple textures. Unfortunately CUDA does not allow to create arrays of texture references, so normally it is necessary to use if-statements to distinguish between different textures, which however would yield in thread divergence, causing this approach to be not usable. Combining a detailed study of the available documentation with an experimental code analysis, we have been able to implement a method that can select the correct texture without the need of an if-statement. This is based on the fact that texture references

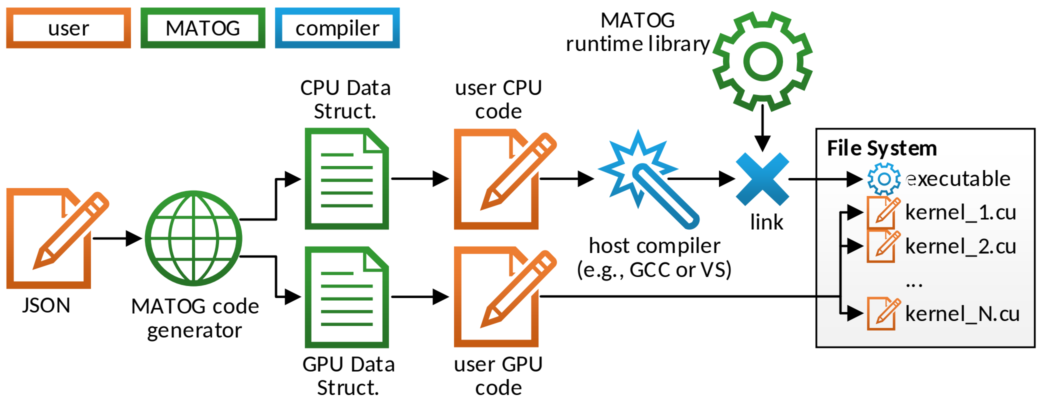


Figure 5.3: Schmatic workflow of MATOG. Orange indicates steps that are performed by the user, green is automated by MATOG and blue is done by the host compiler.

are actually pointers to a point in memory, where all information of the texture resides. This texture information is stored adjoined, so that the textures actually can be switched by adding an certain offset to the pointer. For Fermi GPUs this offset is calculated by texture(i) = rootTexture $\cdot$ $0x800000 \cdot i$ and on Kepler by texture(i) = rootTexture $- i$. The order of the textures is determined by the order in which they are specified in the source code. However, this alone does not work, as the compiler recognizes if the texture references are not specifically referenced in the code and removes them. To prevent this, we specify an init method that references all possible texture references using a read operation, but we skip the actual execution of this part with a goto-statement. This causes that the init method is actually not executed, but the compiler keeps the texture references. As texture memory does not allow to read 64 Bit values, we use the vector functionality in this case, reading two 32 Bit values at once and cast these to the corresponding 64 Bit value. Starting with the second generation Kepler GPUs, NVIDIA added the `__ldg(ptr*)` command, which allows to directly access texture memory without the need of texture references, which makes it much easier to use and significantly reduces the engineering overhead. As MATOG does not allow to directly access the CUDA module or function objects, it is not possible to use user defined texture references. However, since the first generation of the Kepler architecture unbound textures are supported, which are passed on to the kernel as argument. For Fermi GPUs there is no such solution available.

```
 1 ## -------------------- execute optimization -------------------- ##
 2 export MATOG_PROFILING=1 # enable profiling
 3 ./myApp training_0 ...   # run application one or multiple times
 4 ./myApp training_1 ...
 5 ...
 6 unset MATOG_PROFILING    # disable profiling
 7 matog-analyze            # run analyzer
 8
 9 ## ---------------------- run application ---------------------- ##
10 ./myApp param_0 param_1 param_2 ...
```

Listing 5.5: Example for running a MATOG application, including the optimization process. If lines 1-7 are omitted, the application will run unoptimized using default implementations for the data structures.

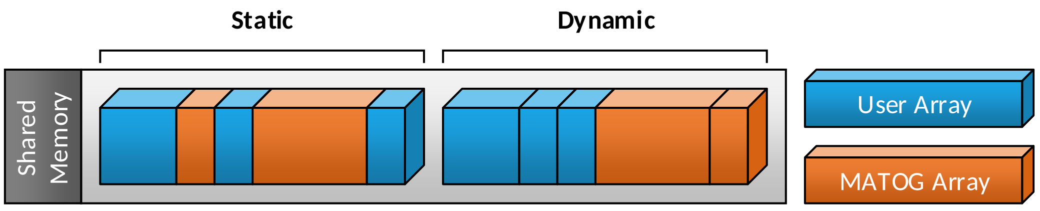


Figure 5.4: Illustration of the shared memory placement. For static shared memory, MATOG data structures can be placed mixed with user defined arrays. For dynamic shared memory MATOG ensures that all MATOG data structures are placed at the end, so that the user can use the first part of the dynamic shared memory segment for his own data structures.

5.3.2 Shared Memory

For shared memory MATOG supports two different implementations, one for static and one for dynamic shared memory. The static variant requires to know all sizes at compile time. As MATOG supports to also use user managed dynamic shared memory, it places all MATOG controlled dynamic shared memory data structures at the end of the dynamic shared memory segment, to ensure no interference with the user managed segment. This does not have any performance implications, but is intended to be easier to use by the user, so he does not need to be aware of any offsets he has to obey when using dynamic shared memory. Figure 5.4 illustrates this.

5.3.3 Optimization Hints

We allow to specify a series of compiler hints, to assist MATOG in optimizing the application. One of these hints is the way data is used, where we allow to define if it is *read* ***(R)***, *written* ***(W)*** and *read once* ***(O)***. Depending in which combination these flags are used, we can employ different optimizations.

RW: By default, only global memory is available.

R: Read-only allows to use texture memory and data does not need to be restored after a profiling run.

W: Write-only implies that data is entirely overwritten, therefore the layout can be switched prior each execution of this kernel. Further, it does not need to be restored after a profiling run as it will be overwritten entirely.

RWO: Read-Once-Then-Write will read the entire data once at the beginning of the kernel and then entirely overwrite it, which enables to use texture memory for reading the data

In some cases it is possible that multiple arrays have the same sizes. In this case the user can specify the underlying relation, which allows to reuse the memory size counters of one array, instead that each array uses its own copy. As C++ does not allow to access the private member variables of other class instances, we utilize inline PTX to directly access the necessary elements in the kernel arguments. However, this only improves the performance if the number of used registers is at the border to an occupancy drop. Nevertheless it is not guaranteed that with a higher occupancy, the performance will be better.

CHAPTER 6

Application Analysis

To optimize an application using MATOG it is necessary to determine, which configurations achieve optimal performance. As MATOG is meant to work for arbitrary NVIDIA GPU architectures and applications from all kinds of application domains we treat GPUs, kernels and optimizations as black boxes. In particular, we assume that we do not know anything about their specific properties, except for which optimizations can be applied for a given GPU, a particular kernel and the hints given by the programmer. This potential lack of information might complicate the analysis but guarantees that our concept can be applied to any application on any past, current and future hardware, as long as the optimization to hardware relation is modeled and as long as the code can actually be compiled for the specific hardware. In the following section, we formalize our optimization problem. Then we explain our algorithm to find optimal solutions for a particular application on a given GPU. We use a 3-step application analysis consisting of an empirical profiling (Section 6.2) to determine, which optimizations are optimal, an offline analysis (Section 6.3) to determine the application-wide optimal solution and a decision model training (Section 6.4) that enables to select data dependent optimal solutions during runtime.

6.1 Optimization Problem

Our optimization problem has multiple optimization dimensions: If an array uses struct types, we can optimize the *struct layout* ($d_L \in$ [AoS, SoA, AoSoA]). For multi-dimensional arrays we can optimize the *transposition* (d_T), where the number of possible values is the factorial of the number of array dimensions. If data is only read from arrays residing in global memory we can determine if we use the *default* or *non-coherent cache hierarchy*, or place the data in *constant memory* ($d_M \in$ [Default, Non-Coherent, Constant[1]]). On Fermi and Kepler cards we can further determine the size of the *L1 cache* ($d_{L1} \in$ [Prefer SM, Prefer L1, Prefer EQ[2]]) by trading with shared memory. Additionally we can select different implementations using *user defined preprocessor instructions* ($d_D \in [...]$) and *value ranges* ($d_R \in$ [min, min + step, ..., max]). As can easily be seen the number of dimensions for a single kernel can be very high as a single array already can have up to three

[1]if size is known at compile time
[2]only for Kepler

optimization dimensions. Further, we can see that the number of values for a dimension are very low. This causes a very high dimensional optimization problem with a very limited extend but a very high number of *configurations* (C) for a given *kernel* ($k \in K$), as this is defined by $|C_k| = \prod_{d \in D} |d|$. This gets even more complex, when multiple kernels are used as this results in $|C| = \prod_{k \in K} |C_k|$ configurations.

6.2 Step 1: Application Profiling

As processor architectures can significantly change from one to another generation, purely analytical models can break, as new or changed features no longer perform as modeled. Therefore we solely rely on empirical profiling without any code parsing or specific GPU model. This removes the necessity of constantly updating the used analytical GPU model with each new generation. However, the mentioned high number of configurations is a big problem for empirical profiling. These configurations not only have to be executed but also be compiled, as the optimizations for the GPU code have to explicitly compiled to achieve maximal performance. Further, to gather the execution time of the application it has to be restarted in each configuration, which leads to massive overhead especially if a huge amount of I/O is taking place. To tackle these issues we used a specialized profiling technique in Weber et al. [2015] which relies on an in-application profiling (similar to the NVIDIA CUDA profiler) and a prediction algorithm that performs a specialized search space pruning. For time measurements we rely on CUPTI (Section 3.1.3). To validate correctness MATOG not only checks whether configurations can be executed (e.g., do not exceed constant memory limitations or try to use texture memory on non-read-only arrays) but also has a verification mode that can be activated during profiling to verify that all configurations produce the same results.

6.2.1 In-Application Profiling

An in-application profiling has the advantage (compared to restarting an application) that it does not require to repeat any application setup and finalization procedures, which can be very time consuming depending on how much and from where data has to be read or written to. In addition, the in-application profiling reduces the number of configurations to test from $\prod_{k \in K} |C_k|$ to $\sum_{k \in K} |C_k|$ as we can calculate the execution time for all permutations of kernel configurations and do not need to explicitly measure them. One drawback of this method is, that it requires much more memory, as data has to be duplicated so it can be restored prior to (re-)executing the same kernel, in another configuration. Further, it has to be converted into other data layouts. In MATOG we copy the data to the host system prior to each kernel execution and start parallel CPU threads to convert data if necessary. Further, all necessary kernel configurations are compiled in parallel to the GPU profiling to overlap compilation and execution. This is realized using multiple threads so it does not interfere with the profiling of the kernel execution.

6.2.2 Prediction Based Profiling

In Weber et al. [2015] we introduced a prediction based algorithm that requires only very few samples to estimate the performance of the entire optimization space for a single kernel. This is based on the observation that many optimizations do not influence others. Formally this means that many dimensions are independent of others and therefore can be optimized independently of others. This allows us to use a difference model to predict the execution time of a kernel in a *specific configuration* ($p(k, c_p)$). For this we require a reference point that we call *base configuration* (c_b) (that can be arbitrarily selected). Further, several additional data points are required, called *support configuration* ($c_{s,d}$). For these all values are equal to the base configuration, except for the value of one dimension d. To predict the performance of a specific configuration, we then use the *difference* ($\Delta(t_1, t_2) = t_1 - t_2$)) of the execution times between support and base configurations, sum them up and add the time of the base configuration.

$$p(k, c_p) = t(c_b) + \sum_{d \in D} \underbrace{\big(t(c_{s,d}) - t(c_b)\big)}_{\Delta(c_{s,d}, c_b)} \tag{6.1}$$

However, in reality this does not always work, as not all dimensions are independent of the other ones. This is caused by the fact that some optimizations, e.g., the *L1 cache size* (d_{L1}), significantly influence the availability of resources and can thus lead to strongly varying hardware utilizations. Other dimension types (e.g., layouts or transpositions) are mostly independent and hardly change the amount of used hardware resources. Given this influence, we have to modify the prediction formula. First, we divide all dimensions in two sets. The first contains *independent dimensions* (D_I) and the second contains all that have an influence on others, which we further call *shared dimension* (D_S). For each value combination of the D_S, we create a separate prediction domain. Only configurations that are located inside this domain can be used to predict configurations in this domain. Simply spoken this means that each domain requires its own base configuration and consequently, also corresponding support configurations. For the profiling this means that we have to execute all $D_S \otimes D_I$ combinations to gather all required

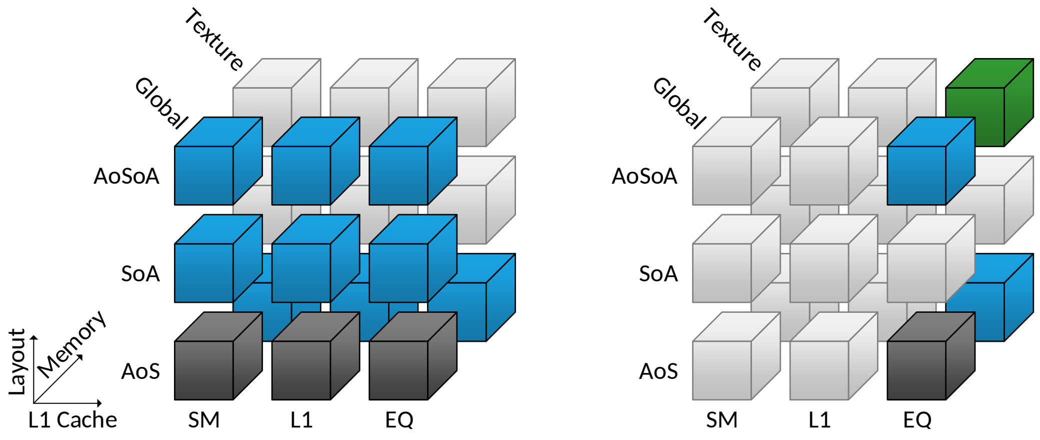
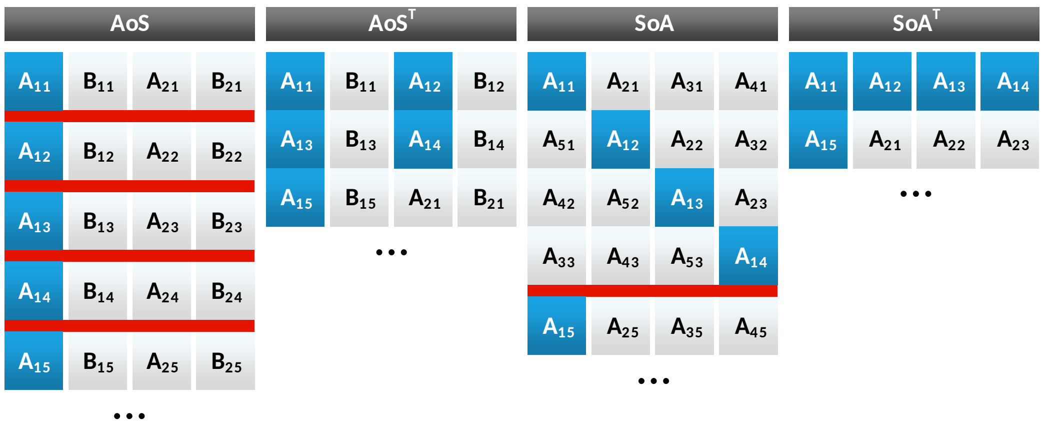


Figure 6.1: Left: all c_b (grey) and c_s (blue) that have to be profiled to estimate the performance for all non-profiled configurations (white) for three optimization dimensions. **Right:** Selection of the c_s and c_b to predict the execution time of c_p (green).

measurements to estimate the performance of all other configurations (shown in Figure 6.1). The number of these configurations is significantly smaller than for an exhaustive search, as can be seen by:

$$\prod_{d_s \in D_S} |d_s| \cdot \sum_{d_i \in D_I} |d_i| \ll \prod_{d \in D} |d|, \text{ with } D = D_S \cup D_I \text{ and } D_S \cap D_I = \emptyset \qquad (6.2)$$

Figure 6.1 shows an example on how the support configurations are selected for the prediction. Note that we empirically showed that his works on NVIDIA GPUs but cannot provide any proof of why this works, as the GPUs and the compilers are proprietary and their specification not publicly available. We can, however, provide the following argument: Let us assume a very simple kernel with a 5x5 AoS consisting of two fields, a memory access as shown in Figure 6.2, and a device with four hardware threads and a memory controller that can fetch one memory bank with four adjoining items at once. For our theoretical system, each memory bank load requires 10 clock cycles and one additional clock cycle for reading memory banks that are not adjoined. Further, let us assume that the data can only be represented as AoS or SoA, and can be stored untransposed or transposed. This results in a total of four configurations. Figure 6.2 shows the memory access for all configurations in the first iteration of the inner loop. The major goal for us is to minimize the required clock cycles for all memory loads. As can be seen, SoA^T is the best layout with only two lines to be read. To find the optimal layout, we have to sample AoS as base configuration. Further we require



```
struct AoS {int a; int b};
AoS array[5][5];

int tid = threadIdx.x;  // {0..3}
int bsize = blockDim.x; // 4

int sum = 0;
for(int y = 0; y < 5; y++)
  for(int x = tid; x < 5; x += bsize)
    sum += array[y][x].a;
```

Figure 6.2: Example for storing an 5x5 AoS. Each line represents a memory line with four items. Blue boxes show accessed items in the first iteration of the code. Red bars indicate where the data is scattered over the memory. When AoS is used, each iteration of the inner loop (although executed in parallel) requires to read a separate non-connected data line. In contrast, for AoS^T only three neighboring lines have to be read. SoA^T is the best layout, as only two consecutive lines have to be read.

two support configurations, which are AoS$^\mathrm{T}$ and SoA. Their measured execution time is $t(\text{AoS}) = 54$, $t(\text{AoS}^\mathrm{T}) = 30$ and $t(\text{SoA}) = 51$ clock cycles for the inner loop. When we apply our predictor (Equation 6.1) we get the predicted execution time $p(\text{SoA}^\mathrm{T}) = 27$ which is the best result. Although this value differs from the exact value $t(\text{SoA}^\mathrm{T}) = 20$, it is still a useful prediction. In fact, the difference is caused by the choice of parameters for our artificial example. However, in a real system we would expect a deviation anyway, caused by noise of the measurements. Figure 6.3 shows a real example for the prediction of a kernel, where we require three base and 36 support configurations to estimate the performance of 5145 non-profiled configurations, which is 0.75% of the total configuration count. As can be seen, the prediction is not perfect but it is good enough to select a configuration, which is close to the optimal solution.

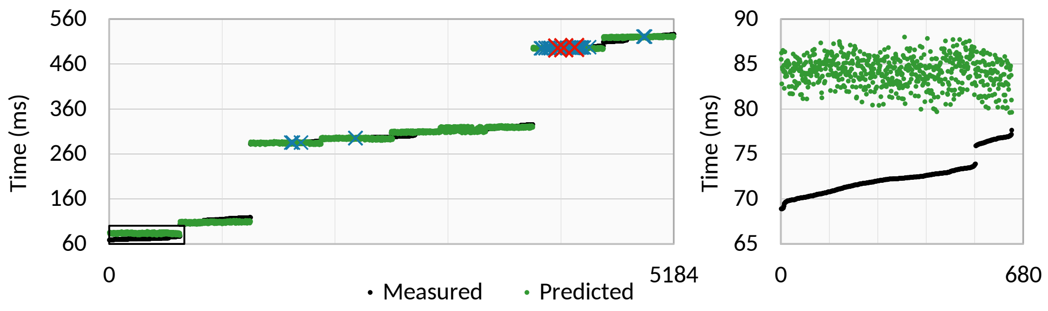


Figure 6.3: Left: The measured performance (black) of 5184 configurations sorted from best to worst, our predicted performance (green) based on measured support (blue cross) and base (red cross) configurations. **Right**: A closer look on the section of the best configurations (indicated by black box). It can be seen that the prediction can be noisy and can slightly deviate from the exact results.

6.3 Step 2: Determine Optimal Configurations

Given the prediction algorithm we can calculate the optimal configuration for each kernel execution. Now we have to determine an application-wide optimal solution. This is difficult as arrays that are used in multiple kernels introduce dependency constraints that need be resolved. In general there are two ways to handle these. Either, data can be converted between two kernel executions, or not. Without knowing how long a kernel will execute using unknown data, we have no means to predict at runtime whether the conversion will yield enough improvement to compensate the conversion, so we decided not to allow any conversions once a layout has been determined. However, we conducted initial experiments for predicting the performance based on automatically generated models and show our results in Chapter 8.

6.3.1 Decisions

As data is allocated at various points during the execution, we may have to apply some data layouts already prior to any kernel executions, e.g., host arrays are usually filled with data before they are copied to the device. We differentiate between three *decision events* that require action:

1. whenever a host array is allocated
2. whenever a device array is used in a kernel (if it has not inherited a layout through a prior memcopy)
3. whenever a kernel is executed, non-device memory decisions (e.g., shared / local memory, defines, ranges, ...) can be determined without any constraints

There is one special case: If a device array was marked as *write-only*, the kernel will overwrite the entire content of the array and therefore is not constrained by the existing layout. In this case MATOG can assign a new layout resulting in a new decision (Type 2). As an allocation can occur multiple times during the execution (e.g., in a loop), every array gets assigned a unique id. This allows also to use the size of previously allocated arrays as meta data during the decision-making process.

6.3.2 Array Dependencies

To resolve the array dependencies, we previously used an *Array Dependency Graph **(ADG)*** [Weber and Goesele 2016]. An ADG is a directed graph that contains nodes for each array allocation, kernel execution and memcopy between host and device. This graph maps onto the life-cycle of arrays during the application execution. While the ADG representation fits exactly the flow of the application, it is difficult to parse. Therefore, we propose a simplified version, called *Decision Dependency Graph **(DDG)***. This is also a directed graph, but instead of mapping to the life-cycle of arrays, it maps onto the decisions made during the execution. There are only two types of nodes in this tree: *global* and *kernel decisions*. Global decisions occur whenever the layout of a global memory array needs to be determined (Types 1 and 2). These nodes are connected to kernel decisions (Type 3) by edges. Each edge is labeled by the global decision it originates from and connects all kernels in the sequence it is used in. Figure 6.4 shows an artificial example of a DDG with all possible cases. Depending on the structure of the application, the graph does not need to be fully connected and can be a forest of multiple graphs. This also happens, when the application has been profiled with multiple different datasets, where each of these profiling results is a disjunct graph. In the following we describe our method on one graph, but this can be easily extended to a forest of graphs.

6.3.3 Exhaustive Search

In Weber and Goesele [2016] we proposed an exhaustive analysis of the DDG. This method guarantees to find the optimum of the application, but can be very time consuming for complex applications with a high number of reallocations. The exhaustive search varies all possible combinations of the DDG and calculates the total execution time for all kernel executions using exhaustive profiling data or our predicted execution times. If predicted data is used, the execution can be sped up by splitting all decisions into two categories: *local* and *global*. Local decisions influence only one kernel execution, e.g., shared/local memory layouts, usage of non-coherent cache, constant memory, ranges or defines. Global decisions are

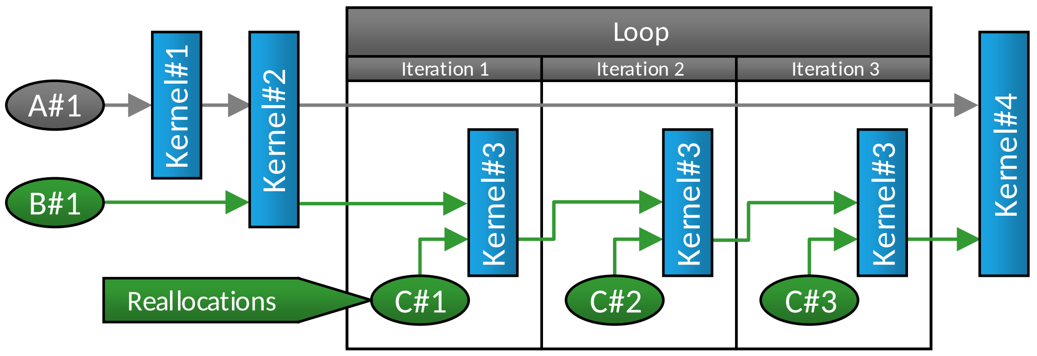


Figure 6.4: DDG example with global decisions of Type 1 (grey), Type 2 (green) and kernel decisions of Type 3 (blue). Global decisions are shown by their unique decision identifier (A-C) and their unique allocation id (#1-3). The array corresponding to *C#X* is reallocated inside the loop in every iteration.

all global memory struct layouts and transpositions. This separation allows to precompute optimal values for the local decisions, for each kernel, as we can store one optimal local value for each possible global decision permutation.

6.3.4 Predictive Search

The problem of an exhaustive search is that if an application allocates or reallocates many arrays (e.g., in every iteration of a loop), the number of global decision nodes grows rapidly. As the number of combinations that have to be evaluated increases thus exponentially, this method becomes quickly unfeasible. Unfortunately, this often occurs in real applications. We therefore extended our predictive profiling onto the search for an application wide optimal solution. The assumption of the prediction algorithm implies that we can select optimal values without respecting any interactions between the optimization dimensions. In the first step we iterate over all global decisions separately and select optimal values for these. We ignore the optimization domains in this step and only search inside one domain of the shared degrees. This is possible, as we have previously mentioned, layouts and transpositions (as employed by global decisions) hardly change the amount of used resources. In a second step we iterate over all local decisions and search for the best values, but now we obey the optimization domains. We will show in our experiments that this method provides comparable results to an exhaustive search.

6.4 Step 3: Decision Models

At this point we know the optimal solutions for all profiled application runs. Now we have to use this information to build decision models that can select optimal configurations during runtime. For these decisions we require some kind of meta data to distinguish between different input data scenarios. Auto-tuners such as *Active Harmony* [Chung and Hollingsworth 2004] or *Nitro* [Muralidharan et al. 2014] use user defined callback functions that calculate some arbitrary, user determined metrics for this distinction. Instead, we use a fully automatic way. As analyzing and categorizing arbitrary data is a very difficult, often time consuming task (depending on which analysis are performed), we chose to use the meta data that is already available in our system such as the sizes of arrays and the launch configurations of kernels. In contrast to our previous publications, our improved meta data gathering system does not only track the most recent arrays in the system, but also previous data, using unique ids for each allocated array. We additionally allow the user to register variables, which are monitored during the execution. These can contain data that has been explicitly calculated by the users, or e.g., counts of items in a preallocated, not-resized array. This information is usually available in the application without any need to specifically calculate these. For our decision model creation, we gather all decision events from all DDGs and group them by their decision event. We create one model per decision event. For this we collect all meta data that was present in the profiling when the decision event occurred and store it in a matrix, in which each column represents one kind of meta data (e.g., size of a specific array or the launch configuration of a kernel) and each row represents one decision event. At this point it does not matter if it is an array allocation or kernel execution decision. This gathered meta data usually contains a high amount of redundant, linear dependent or constant data. To simplify the decision model creation and to reduce the meta data gathering overhead during runtime, we pre-process this data by removing all redundant data. If multiple rows with equal meta data but different optimal configurations occur, we perform a majority voting and keep the configuration, which was chosen most.

6.4.1 Directional Model

In Weber and Goesele [2016] we used a SVM as decision model. This proved to be suitable but may not be optimal (depending on the application). This is rooted in the kind of meta data that we use in MATOG, consisting of array sizes and launch configurations. These come with some difficulties. For example, as we do not know the limits of our gathered meta data, there is no way to normalize it, which can result in bad decisions if the values differ too much from the training data. Further, as array sizes and launch configurations are usually used as iteration counts for loops, they imply a certain linear scaling, which can be leveraged in a decision model. We therefore analyzed our meta data and determined that in some of the failure cases, a specialized nearest neighbor search model worked better, in the following called *Directional Model* ***(DM)***. The DM uses the cosine similarity distance that interprets the training ($\vec{t}$) and meta ($\vec{m}$) data as vectors. It then applies the normalized dot product on these vectors to compute the metric:

$$\lambda = cos(\gamma) = \frac{\vec{t} \cdot \vec{m}}{|\vec{t}||\vec{m}|} \tag{6.3}$$

The resulting λ is in the range $[-1; 1]$. To evaluate the model, we iterate over all training samples, calculate λ and select the data point where λ is maximal. The advantage of this method is that it performs an implicit normalization, resulting in better decisions even if the meta and training data values differ significantly. Figures 6.5 and 6.6 show a simple example for a linear SVM compared to the DM, both trained and evaluated on data taken from our KD-Tree benchmark, run on a Tesla K20c.

6.5 MATOG Runtime System

In this section we briefly explain our runtime system. At the beginning of the application run, MATOG is automatically initialized once the first CUDA related function is executed. Whenever an array is allocated, we determine, which decision event it belongs to, and store its meta data in a centralized meta data storage. If an array of the same decision event is allocated, it gets a new instance id assigned. If this array is a host array, its decision model is evaluated immediately and the layout is set (Type 1). Whenever a memcopy occurs, MATOG propagates the layout from the source to the destination array. If a kernel is executed, all non-initialized device arrays evaluate their decision model and set their layouts (Type 2). Further, the kernel evaluates its own decision model (Type 3). The final executed configuration of the kernel is the combination of all layouts of the device arrays (which have been separately evaluated before) and the configuration from the kernel's decision model. If this configuration has been compiled before, it is loaded from the database, and then executed. If not, it will be compiled and then executed. In Weber and Goesele [2016] we compiled all available combinations that could occur after our application analysis process. However, depending on the complexity of the application this can be several million configurations, making this approach not usable. We discuss in our future work how this high number of configurations could be reduced to improve the compilation time and also reduce the number of implementation switches during runtime.

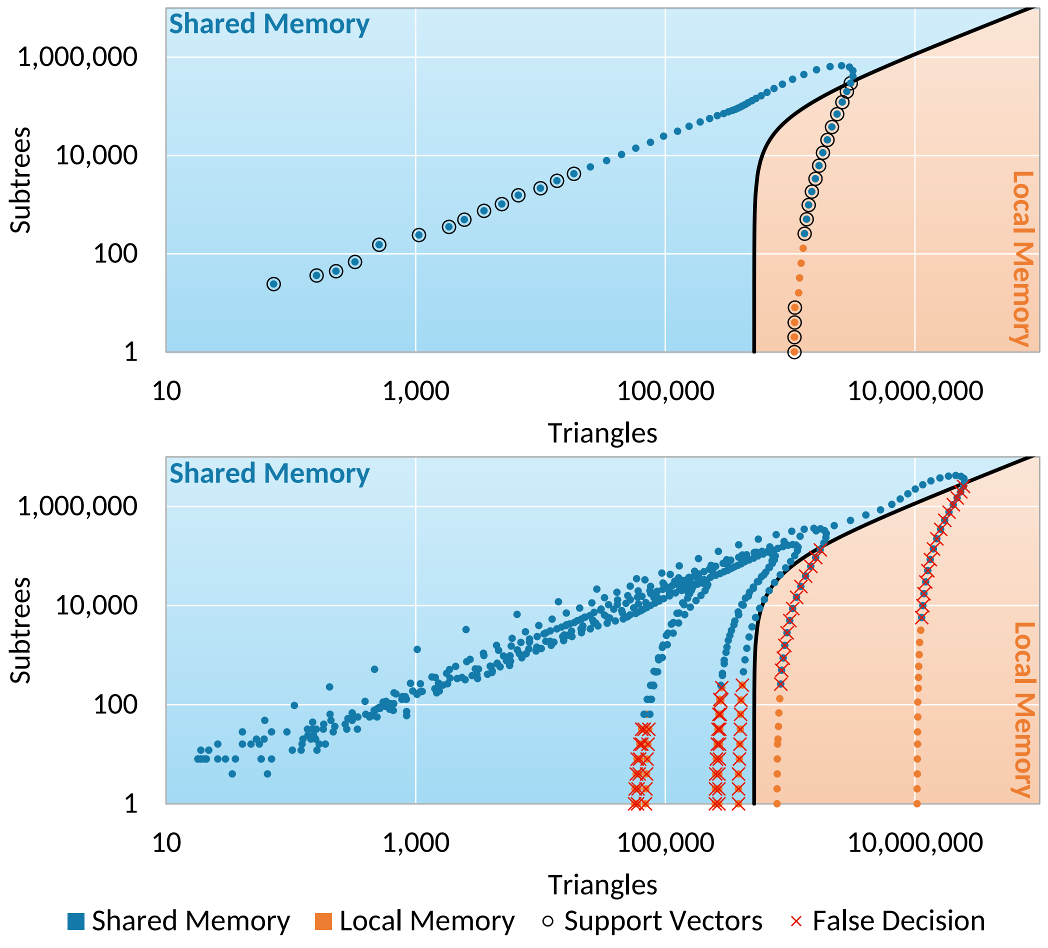


Figure 6.5: Example for a linear SVM . On top, the data used for training and the resulting models are shown. At the bottom, the models are applied to the testing datasets. Every dot represents the meta data of a kernel execution and colors indicate if either using *local* (orange) or *shared memory* (blue) is the better choice. The shown results are for a linear SVM. The support vectors and false decisions are highlighted by black circles and red crosses, respectively. The hyperplane of the SVM is shown as black line and the colored background visualizes the classification of the SVM. As can be seen, the amount of false decisions is quite high for this decision model. Be advised, that the meta dimensions are hand picked and that the actual model does not only decide whether to use local or shared memory, but also decides on layouts, transpositions and texture memory usage.

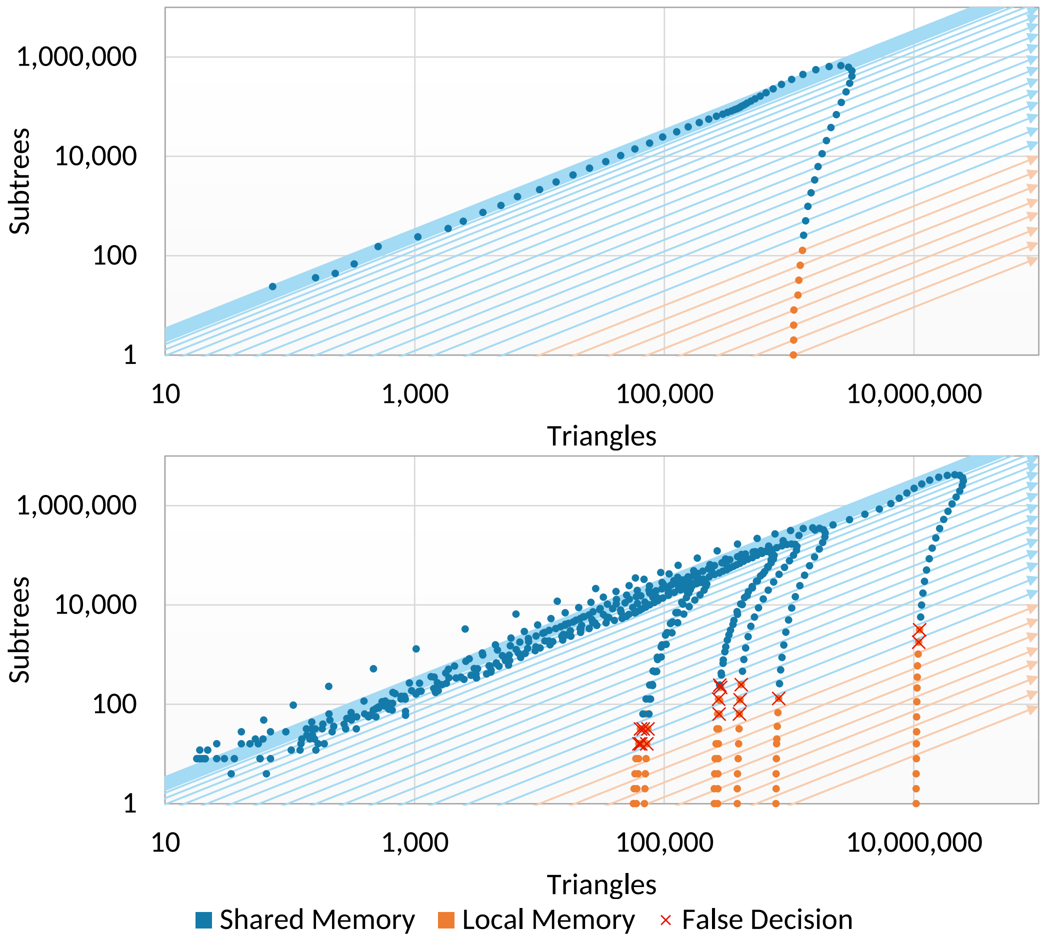


Figure 6.6: Example for a Directional Model using the same data as in Figure 6.5. The decision vectors are shown as arrows, with the color of the corresponding decision. As can be seen, the number of false decisions is significantly lower for the DM than for the linear SVM.

Chapter 7
Evaluation

In this chapter we evaluate the different analysis steps of MATOG, their efficiency and the achieved performance on seven applications. All tests have been performed on a system equipped with dual Intel Xeon E5649, 48 GB DDR3-1333, Ubuntu 16.04 and CUDA 8.0 (driver version 367.57). We evaluated the last four NVIDIA GPU architectures: Fermi, Kepler, Maxwell and Pascal. Table 7.1 shows an overview of all evaluated GPUs. Our results are compared to the unoptimized performance of MATOG and a hand-optimized reference code that does not use MATOG.

Providing a fair and direct comparison to related approaches on our benchmarks is difficult. On one hand, there is to our knowledge no publicly available code for several other memory access auto-tuners [Sung et al. 2012; Kofler et al. 2015; Peng et al. 2016]. On the other hand, auto-tuners such as Nitro [Muralidharan et al. 2014] or OpenTuner [Ansel 2014] do provide code but miss specific, automatically generated optimizations. Instead, they require all optimizations considered to be explicitly hand-coded by the user, a stark contrast to MATOG's automated approach. We can, however, make indirect but meaningful comparisons: Nitro is designed to handle only a small number of configurations and relies thus on exhaustive search. Since we provide both timing as well as performance evaluations for exhaustive search as part of our results below, these can be seen as representative for Nitro. OpenTuner uses a genetic algorithm. In Weber et al. [2015] we showed that our predictive based search uses a minimalistic set of configurations that suffices to find comparable results and is faster for the optimization problem that we face in MATOG compared to a genetic algorithm. This performance argument is still valid for the current version of MATOG and thus provides a clear advantage.

7.1 Benchmark Applications

We evaluate seven different GPU applications, ranging from very simple algorithms with regular workload up to very irregular algorithms with varying workload. Table 7.2 shows the number of possible configurations per GPU and benchmark. Most of our applications originate from our research group or student projects. Except for DPID, these applications have been developed for Fermi GPUs. The *Speckle Reducing Anisotropic Diffusion* ***(SRAD)*** and Hotspot benchmarks are taken from the Rodinia benchmark suite [Che et al. 2009] v3.1. Besides the fact that most

GPU				Processor				Memory		
Name	Arch. (CC)	Chip	Released	Cores	SMs	Clock	Boost	Type	Clock	Bus
GT 440	Fermi (2.1)	GF108	Feb-10	96	2	810	1,620	DDR3	900	128
GTX 480	Fermi (2.0)	GF100	Mar-10	480	15	701	1,401	GDDR5	924	384
Tesla C2070	Fermi (2.0)	GF100	Jul-11	448	14	575	1,150	GDDR5	750	384
GTX 560 Ti	Fermi (2.1)	GF114	Jan-11	384	8	823	1,645	GDDR5	1,002	256
GTX 570	Fermi (2.0)	GF110	Dec-10	480	15	732	1,464	GDDR5	950	320
GTX 590	Fermi (2.0)	GF110	Mar-11	512	16	608	1,215	GDDR5	854	384
GT 620	Fermi (2.1)	GF108	May-12	96	2	700	1,400	DDR3	533	64
GTX 680	Kepler (3.0)	GK104	Mar-12	1,536	8	1,006	1,058	GDDR5	1,502	256
GT 730	Kepler (3.5)	GK208	Jul-14	384	2	902	902	DDR3	800	64
GTX 780	Kepler (3.5)	GK110	May-13	2,304	12	863	902	GDDR5	1,502	384
Tesla K20c	Kepler (3.5)	GK110	Nov-12	2,496	13	706	706	GDDR5	1,300	320
GTX 980	Maxwell (5.2)	GM204	Sep-14	2,048	16	1,127	1,216	GDDR5	1,753	256
GTX TITAN X	Maxwell (5.2)	GM200	Mar-15	3,072	24	1,000	1,089	GDDR5	1,753	384
GTX 1080	Pascal (6.1)	GP104	May-16	2,560	20	1,607	1,733	GDDR5X	1,251	256

Table 7.1: GPUs used in our evaluation. It can easily be seen that even within the same GPU generation, the number of cores, SMs, clock rate, memory type and bus width significantly differ. Color scale from low (blue) to high (yellow). (Data taken from TechPowerUp GPU Database [TechPowerUp.com 2017] and CUDA Device Query [NVIDIA 2016c])

of the benchmarks in this suite are very simple and do not provide much ways to enhance the performance (in terms of memory access), we also encountered a series of programming errors in the code, e.g., in the b+tree benchmark:

- a non-zero terminated c-string, which can cause a segmentation fault (in `b+tree/main.c` lines 1937-1941)
- no fixed seed for random values
- wrongly used C functions: `char* output; ... fputs("Fail␣to␣open␣%s ␣!\n", output)` (in `b+tree/main.c` line 2224, the signature of the function is: `fputs(char*, FILE*)`)

Other applications even miss certain functionality, as e.g., the calculation of computing the reverse substring matches in the MummerGPU benchmark is simply commented out (most likely as it is not working correctly). Therefore, we decided not to include more of these into our evaluation.

7.1.1 Bitonic Sort

Bitonic Sort [Batcher 1968] is a widely used parallel sorting algorithm. In our implementation it sorts a 1D AoS with four integer fields (8, 4, 2 and 1B), ensuring that the 8B value is sorted first and only if this value is equal, the 4B value is sorted and so on. This results in a sorted list, for all fields. To ensure conflicting

Benchmark		Fermi (2.0, 2.1)	Kepler (3.0, 3.5)	Maxwell, Pascal (5.2, 6.1)
Bitonic	Theoretical	48	72	24
	Exhaustive	42	63	21
	Predictive	18	27	9
SRAD	Theoretical	786,432	1,179,648	393,216
	Exhaustive	147,456	221,184	73,728
	Predictive	62	93	31
Hotspot	Theoretical	1,024	1,536	512
	Exhaustive	512	768	256
	Predictive	18	27	9
DPID	Theoretical	3,744	5,616	1,872
	Exhaustive	1,872	2,808	936
	Predictive	40	60	20
COMIC	Theoretical	3,112	4,668	1,556
	Exhaustive	780	1,170	390
	Predictive	54	81	27
REYES	Theoretical	27,247,112	40,870,668	13,623,556
	Exhaustive	13,600,232	20,400,348	6,800,116
	Predictive	158	237	79
KD-Tree	Theoretical	1,290,054,564	1,935,081,846	645,027,282
	Exhaustive	40,353,714	60,530,571	20,176,857
	Predictive	200	300	100

Table 7.2: Theoretical number of configurations (entire solution space), the number of configurations an exhaustive search has to execute and the number of configurations required for our predictive method for all evaluated GPU architectures and benchmarks. The exhaustive search contains less configurations as the theoretical, as texture memory can only be used if the data is read-only.

rows, we limit all values to 0-1023 (255 for the 1B field). The application consists of two kernels: One uses shared memory in loop iterations where it can be used efficiently, while the other directly operates on global memory. The reference implementation is not optimized and uses a naïve AoS layout. We train on two datasets and evaluate on five others, all with varying element counts (64 k to 4 M).

7.1.2 Speckle Reducing Anisotropic Diffusion

SRAD [Che et al. 2009] is a diffusion method for ultrasonic and radar imaging applications. The benchmark's computations are quite simple and use a straight forward implementation without much program logic or dynamic allocations. We train on three parameter sets with same grid size and one iteration, and evaluate 12 other parameter sets with varying grid sizes and 100 iterations.

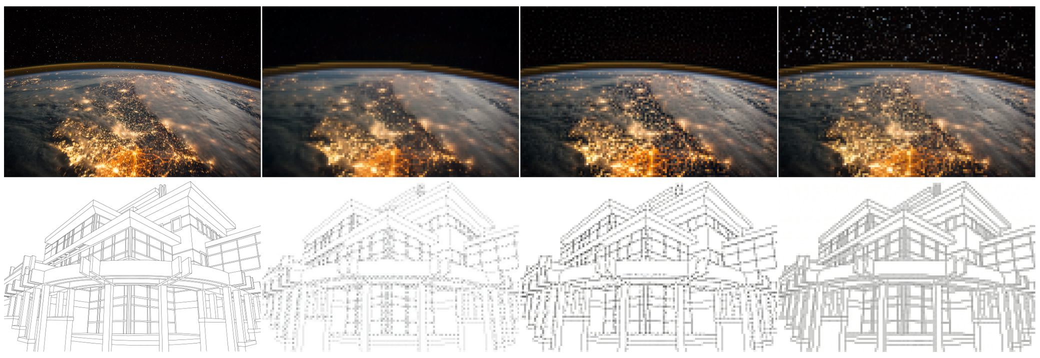


Figure 7.1: From left to right: The original image, the most recent competing algorithms (Kopf et al. [2013] and Öztireli and Gross [2015]) and DPID. Except for the original image, all are downscaled to 128 px width.

7.1.3 Hotspot

Hotspot [Che et al. 2009] calculates the processor temperature based on an architectural floor plan and simulated power measurements. The overall execution time of the application is quite low. The benchmark comes with three datasets. We train on the medium sized one and evaluate on the other two.

7.1.4 Detail Preserving Image Downscaling

DPID [Weber et al. 2016] is a perceptually inspired image downscaling algorithm that in contrast to traditional downscaling algorithms is not based on physical effects. In a first kernel it computes a smooth downscaled version of the image. Then a second kernel uses this image to assign weights to each input pixel and generates the final output image according to the influence of each input pixel. The implementation loads a video using OpenCV [Bradski 2000], copies the data to the device using a page-locked memory segment to speed up the memcopy (as this is the major bottleneck of the entire application), downscales it, copies the result back and saves it to an output video file. The original code was developed for a GTX 680 and uses the shuffle command to exchange data between threads inside a warp. To be compatible to older cards, we modified the code to use shared memory instead of shuffle on Fermi GPUs, as these do not support the shuffle operation. The MATOG implementation also uses page-locked memory for faster memcopy onto the device. Figure 7.1 shows an image generated with DPID and three competing algorithms.

7.1.5 Coevolution via MI on CUDA

Coevolution via MI on CUDA ***(COMIC)*** [Waechter et al. 2012] calculates the coevolutionary mutual information for protein and *Deoxyribonucleic acid* ***(DNA)*** sequences. It consists of three kernels: the first one initializes a randomization seed that is used in a second kernel to permute the sequences. The third kernel performs the main operation by creating a 3D histogram of occurrences in the permuted sequences. This kernel is templated and uses different compression schemes depending on the input data. Further, it uses a complex memory access for the 3D histogram, as the algorithm does only need to store its results in a triangular pyramid. Our MATOG variant is able to buffer results from the histogram in local memory instead of directly storing them in shared memory. The original implementation was optimized for a GTX 480. We train on two datasets with one iteration and evaluate on 10 others using 100 iterations.

7.1.6 Renders Everything You Ever Saw

Renders Everything You Ever Saw ***(REYES)*** [Cook et al. 1987] is a technique used in movie productions. In contrast to classic 3D mesh rendering, it uses patches to model smooth surfaces. The patches are transformed into micro-polygons and iteratively split into smaller polygons until they have subpixel size. The implementation uses four kernels: One of the kernels performs the splitting while the second compresses the micro-polygons after each iteration. A third kernel uses the final polygons and renders the resulting image into a depth buffer, which contains depth and color information for each pixel. Finally, a fourth kernel extracts the color information from this buffer into a 2D texture, which then can be displayed. The benchmark was originally developed for a GTX 480. As this benchmark has a very high number of configurations we do not show any exhaustive profiling results as it would require several weeks to profile the application. We train on one model, rendering two frames on two different resolutions and evaluate on 8 models, 100 frames and varying resolutions. The models are rotated after each frame to change the workload as depending on the viewing angle, more or less polygons have to be split and rendered. An example rendering is shown in Figure 7.2.

7.1.7 KD-Tree

The KD-Tree benchmark constructs acceleration structures for 3D ray tracing and resembles the work of Popov et al. [2006]. The application consists of two main and six maintenance kernels, which have a low total execution time. The first main kernel discretizes a triangulated scene in multiple bins, which are separated by equidistant planes in all three dimensions. Then all triangles in the scene are

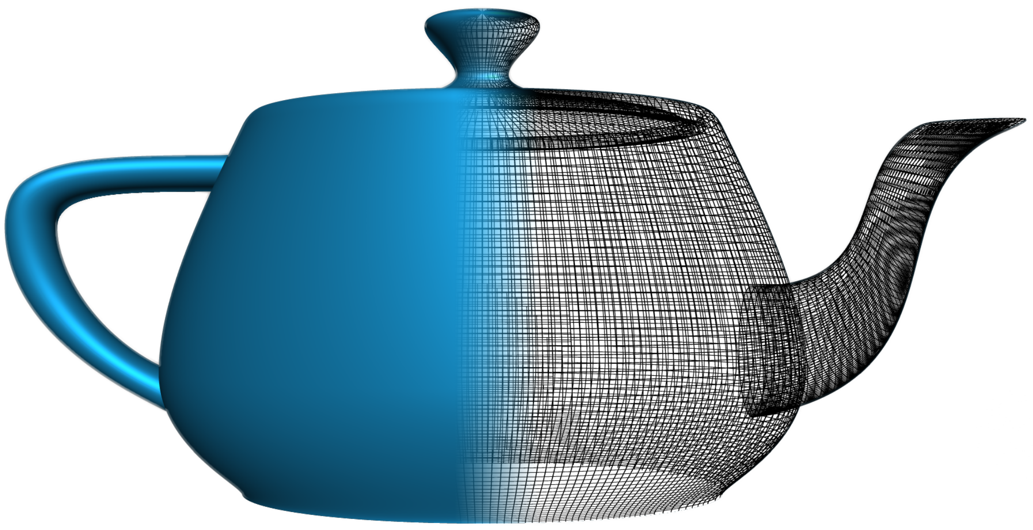


Figure 7.2: REYES rendering of the Utah teapot. On the left the smooth rendered surface is visible while on the right the generated micro-polygon mesh is shown, whereas one grid cell corresponds to 16x16 polygons.

processed and the number of starting and ending triangles in a bin are counted. In the last step a prefix and postfix sum are executed on these values. With these final values, the kernel calculates a building heuristic that is used to select the best split plane. Our optimizable version uses an adjustable preprocessor implementation, which is able to buffer the binning results in local memory instead of using an `atomicAdd` on shared memory. The second main kernel performs the actual splitting of a subtree and stores all necessary data in two different data segments, one for each child tree. Additionally, it has to perform some recalculations if a triangle is located directly on the split plane. The maintenance kernels only have a low portion of the overall execution time. One is run at the beginning of the application, to calculate the axis aligned bounding boxes for the input geometry. Another initializes the default data for each iteration step. Two others calculate necessary offsets for storing the results of the splitting kernel. If subtrees are marked as leaf another kernel is used to compact the header data. The last kernel is a post processing kernel for the splitting step. All kernels are build in a fashion that they can process multiple subtrees in parallel.

The difficulty of this benchmark are the changing data characteristics over time. At the beginning only few but big subtrees are processed. While the number of subtrees significantly increases during the processing, the size of these is reduced and the number of leaf subtrees constantly increases. In the end many, small subtrees are processed. This change has a significant impact on the performance

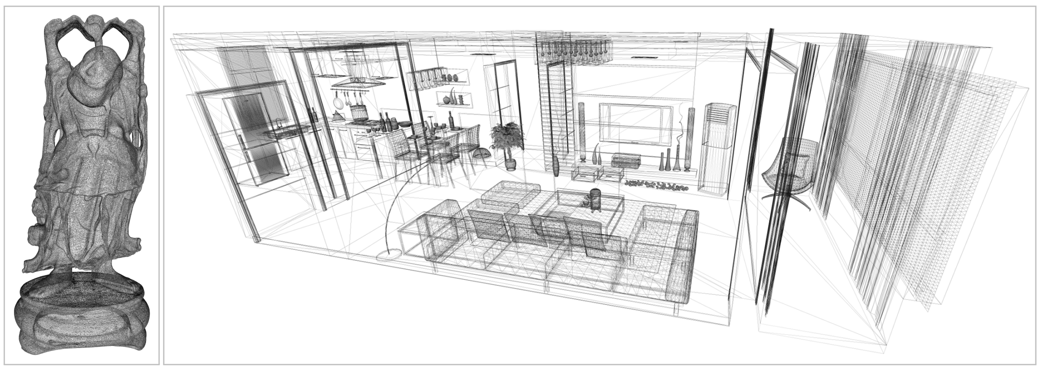


Figure 7.3: The *Buddha* (left) is a 3D scan. It mainly consists of small equally distributed triangles. The *Kitchen* (right) has varying triangles sizes and distributions.

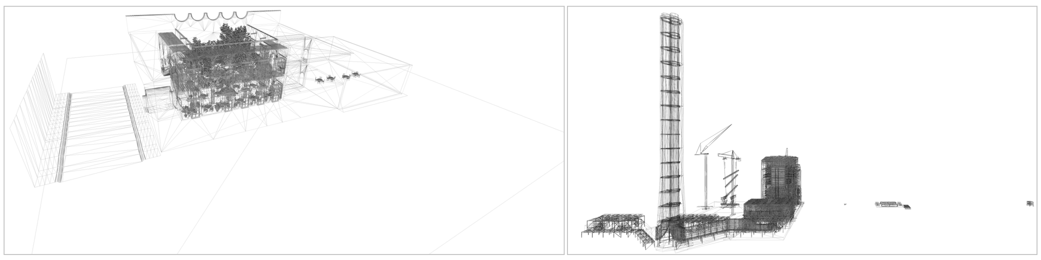


Figure 7.4: *San Miguel* (left) and the *Powerplant* (right) are the biggest scenes we evaluate. They have varying triangles sizes and distributions, equal to the Kitchen, but with much more variety in density and distribution.

and should benefit from MATOG's adaptiveness. This benchmark was originally optimized for a GTX 590. Due to the extreme high number of configurations, we do not show any exhaustive profiling results. Further, as in each iteration memory is reallocated, our implementation of the exhaustive post-processing cannot be used, as it exceeds 2^{64} combinations.

We run this application using 32 bins and 9 different scenes ranging from small (69 k triangles) to big (12 M triangles), from 3D scans (with mostly small equally distributed triangles) to artist generated (with varying sized and distributed triangles). Figures 7.3 and 7.4 show four example scenes with varying properties. Memory capacity limitations prevent that all scenes can be executed on all GPUs (refer to Section A.7 for more details). The implementation we are comparing against uses a mixed set of data structures such as AoS, SoA or hierarchical-AoS (e.g. `aabb[a].point[b].dim[c]`). As the first kernel uses an atomic based mutex in global memory that does not work on Fermi GPUs, the benchmark uses an alternative implementation for this kernel on these GPUs.

7.2 Execution Performance

In this section we evaluate the achieved execution performance of MATOG against the unoptimized MATOG variant and the hand-optimized, purely CUDA reference code. When unoptimized, MATOG uses SoA, a naïve transposition (T_0 in Figure 9.1), no texture memory, no constant memory, prefers shared memory over L1 cache and the default option for user defined preprocessor optimizations. This can be seen as a naïve implementation, as it does not apply any special optimizations and strictly follows the CUDA programming guide [NVIDIA 2016a] that claims SoA to be optimal in most cases. All executions have been repeated five times (except for the application profiling runs), to reduce the influence of measurement noise. All evaluations have been performed on separate training and evaluation datasets. Details are listed in Appendix A.

7.2.1 GPU Execution Time

First, we take a look at the achieved speed up for the kernel execution time. We have evaluated three different decision model types: SVM, DM and a non-adaptive model that always selects the single configuration that was chosen most, to verify if adaptive decisions are necessary to achieve better performance. Further, we evaluated three analysis strategies: *exhaustive profiling/exhaustive analysis* ***(EE)***, *exhaustive profiling/predictive analysis* ***(EP)*** (method used in Weber and Goesele [2016]) and *predictive profiling/predictive analysis* ***(PP)*** (our new method). The results are shown in Figures 7.5 to 7.11.

As mentioned before Bitonic Sort, SRAD, Hotspot, DPID are simple algorithms without much divergence and a very regular processing. In the Bitonic case the reference code is quite slow, caused by the naïve AoS layout. SRAD and Hotspot are already optimally optimized in terms of memory access. MATOG is slightly slower than the reference code, as MATOG data structures employ a certain overhead compared to hard coded memory access. One reason for this overhead is pointer aliasing, as the way we have implemented the MATOG data structures does not allow to use the *restrict* keyword to define the internal pointers as non-overlapping. Therefore, the compiler is unable to perform certain optimizations, which can slightly decrease the performance. For DPID the reference code usually is slightly faster than the MATOG variant, except for GPUs with CC 3.5 (GT 730, GTX 780 and Tesla K20c), where MATOG achieves a significant higher speed up. In most cases no real difference between the different analysis methods and decision models can be seen, implying that these are invariant to adaptive optimizations. For SRAD, the exhaustive profiling usually finds slightly better solutions than our predictive profiling, but has to profile over 2371x more configurations to achieve this result. The irregularities for the results of the GT 730 on Hotspot are difficult to explain.

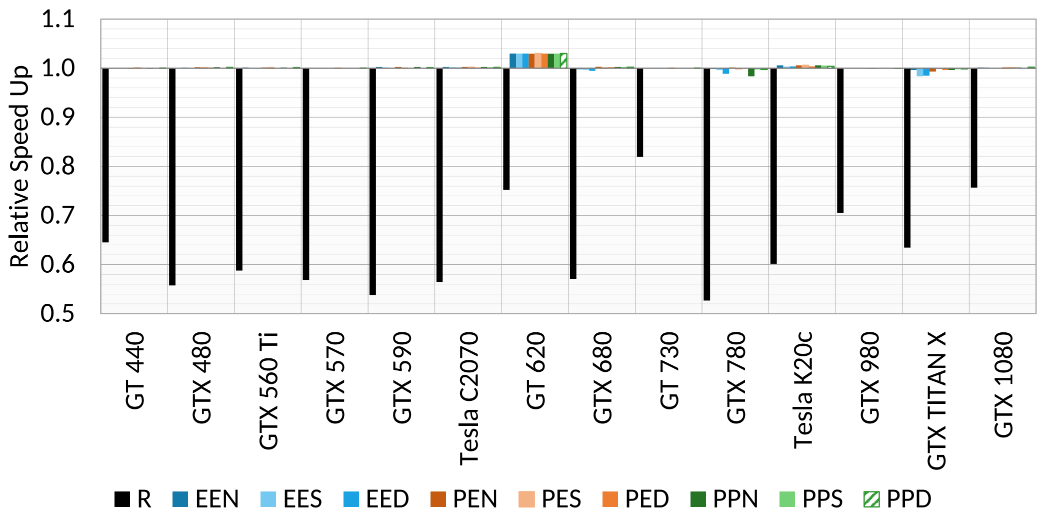


Figure 7.5: Bitonic: This is the only benchmark where the reference code is not optimized. The results show that already the unoptimized MATOG variant is optimal, so that MATOG is unable to further increase the performance. There is no need for any adaptive optimizations, because of the simplicity of the algorithm.

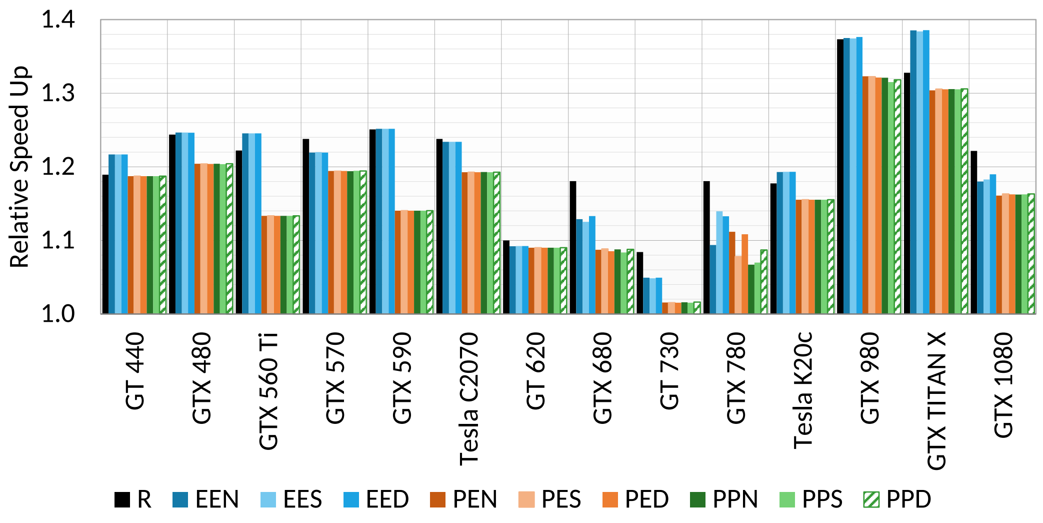


Figure 7.6: SRAD: The reference code for this benchmark is optimal and usually faster than the result MATOG can achieve. This is caused by the rather basic algorithm of SRAD. The small drop in performance between the reference and MATOG is caused by overhead through the MATOG data structures, e.g., through pointer aliasing.

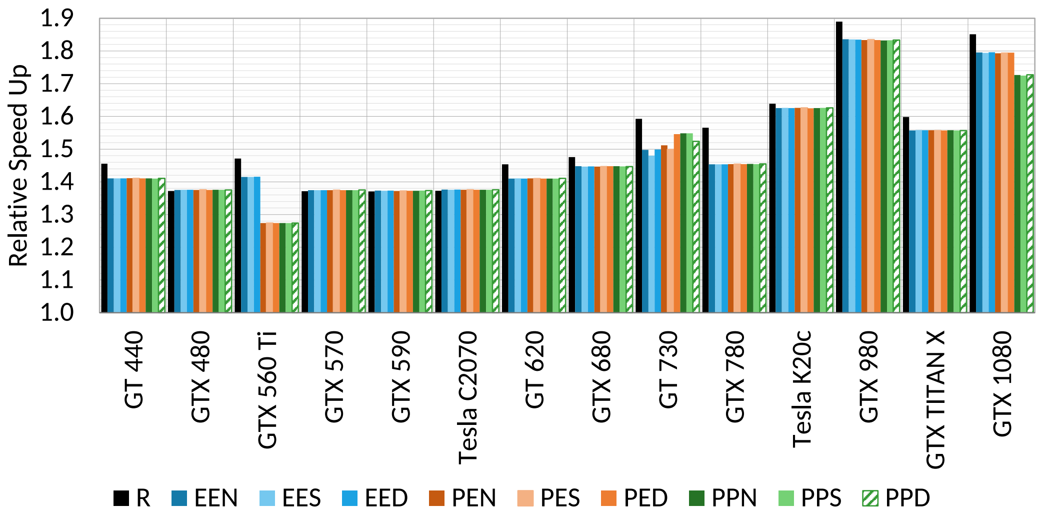


Figure 7.7: Hotspot: Again, the reference code is already optimal. The same applies as for the SRAD benchmark.

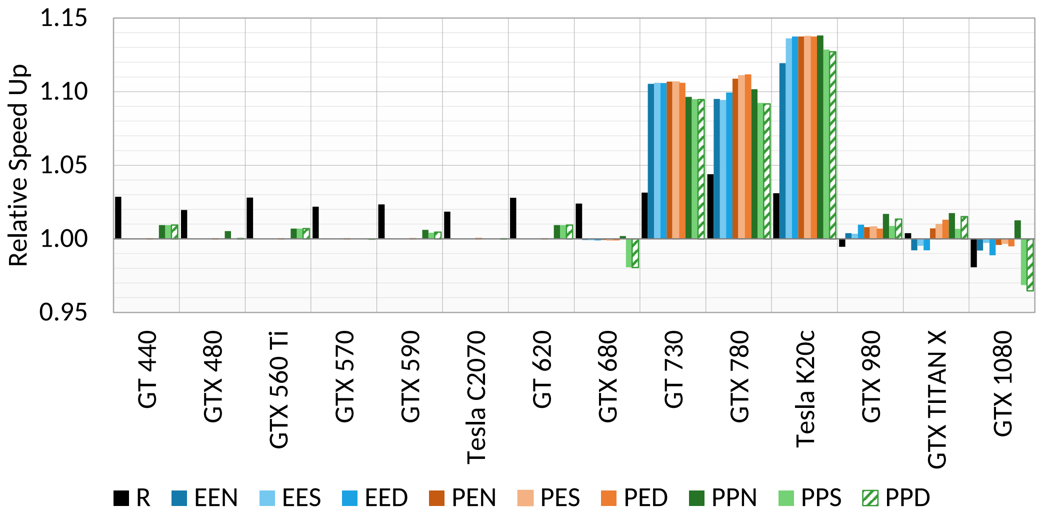


Figure 7.8: DPID: In this case, again the reference implementation is slightly faster than the optimized MATOG, except for GPUs with compute capability 3.5 (GT 730, GTX 780 and Tesla K20c) where MATOG achieves a higher performance. This is due to the usage of texture memory. However, in general the unoptimized code is already very good, so most improvements are minimal.

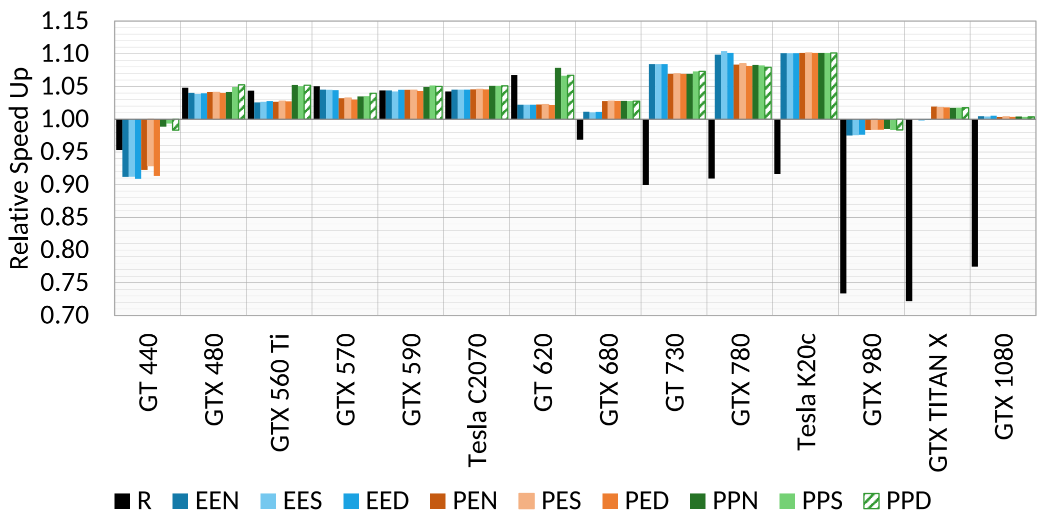


Figure 7.9: COMIC: For this benchmark we see that for the Fermi GPUs, the reference code is optimal and MATOG achieves comparable performance. However, starting with the GTX 680 the reference code performance drops significantly, whereas MATOG is able to achieve up to 10% over the unoptimized variant. In the cases where the optimized variant is slower than the unoptimized can be caused by false predictions of the decision models.

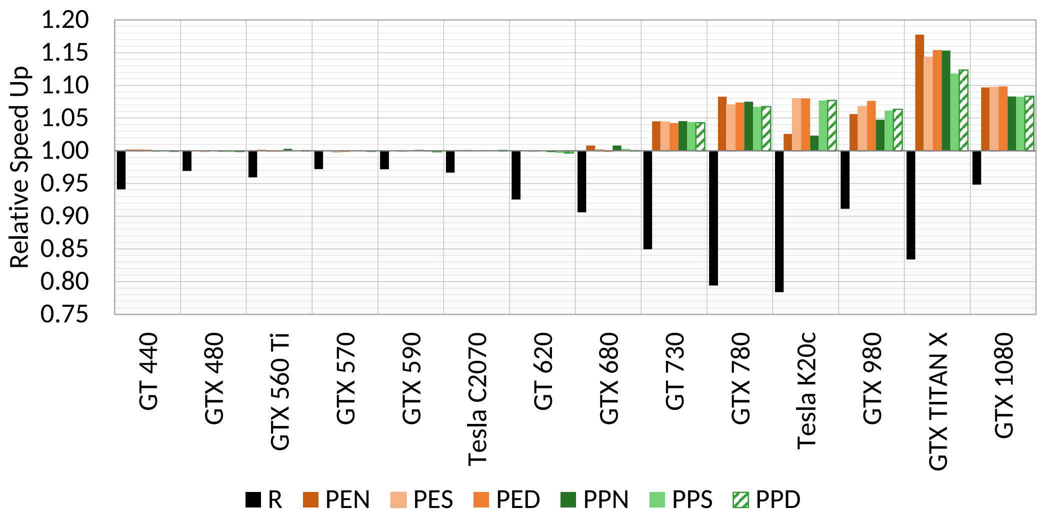


Figure 7.10: REYES: For the REYES benchmark the reference code is always slower than the unoptimized MATOG variant. This is caused by the fact that the code often uses AoS as a layout, while SoA performs better, even on the older cards. For Fermi we see that MATOG is unable to leverage more performance, but for all newer cards it achieves up to 18% over the unoptimized variant. The dynamic decision-making mostly pays off for the Tesla K20c.

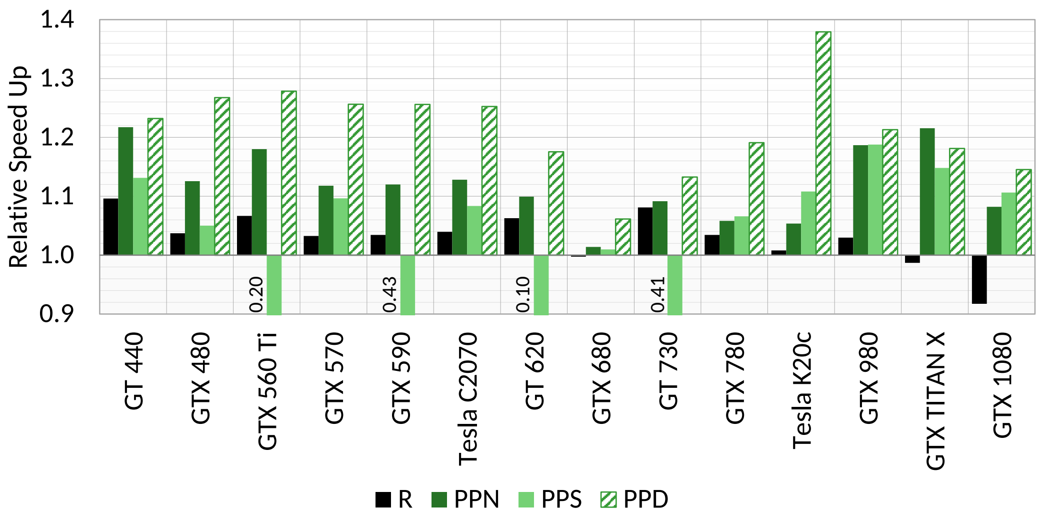


Figure 7.11: KD-Tree: For the KD-Tree the reference code varies significantly between the different GPUs, whereas it is even slower on the newest GPU compared to the unoptimized variant. However, the optimized variant always achieves a significant speed up of up to 38%. The dynamic variant using the DM usually achieves the highest speed up, followed by the static model. In this benchmark the SVM usually achieves very bad results. This is caused by the fact that the meta data gathered during training can significantly differ from data gathered during the testing, causing bad decisions.

We think this is caused by the heat of the chip, as this GPU is passive cooled and therefore can only regulate its temperature by reducing the clock frequency.

So far, all benchmarks have been simple algorithms, consisting of only 1-2 kernels with a very low execution time that do not benefit from adaptiveness during runtime. The three remaining applications are much more complex, consisting of 3-8 kernels. For COMIC we can see that the reference code is optimal for the GTX 480 and 570, which it had been developed for. On the newer cards (starting with the GTX 680), the reference code's performance is significantly lower than of the unoptimized MATOG code. Further, MATOG is capable of achieving even more performance when optimized, except for the GTX 980. We think that this is caused by differences in the data from the test and training datasets. The main speed up is achieved through transposition of the data stored in shared memory. Local memory is never used. This benchmark further does also not benefit from adaptiveness.

Although REYES was optimized for the GTX 480, even the unoptimized MATOG code is already faster than the reference code. On the newer architectures the reference performance is further decreased, below 80% compared to the unoptimized code. However, the optimized variant is unable to find much better solutions until the GT 730. As can be seen, the adaptive methods achieve up to 6% higher performance on some GPUs, while on others the static perform better.

The KD-Tree is by far the most complex and difficult to optimize application we evaluate on. It has a lot of changing data properties, a huge number of reallocations and an irregular work processing. The non-adaptive model achieves decent speed ups, but the DM achieves always the best and is up to 33% faster than the non-adaptive solution. The SVM always performs less optimal than the DM, especially on the (GTX 580Ti, GTX 590, GT 620 and GT 730). The SVM is here unable to understand the underlying coherence of the meta data since the values differ too much from the training data, as it cannot be normalized. This causes the SVM to use local memory very often, whereas this is only optimal in the very first iterations with few big subtrees and performs very badly in all subsequent iterations. It is difficult to say why this does not happen for the other GPUs and might correlate with the fact how good the local memory variants work on the respective GPUs.

To summarize, we can say that MATOG achieved in general the highest speed ups on the Tesla K20c. As the codes have been mostly developed for consumer/GTX GPUs this is no surprise, as the Tesla K20c is a HPC card with different properties. There is no real difference between the analysis methods (in terms of quality), except for the SRAD benchmark, where the exhaustive profiling often is able to find slightly better configurations. Further, the DM has achieved the highest performance (compared to the other methods). In the very complex applications, our adaptive optimizations out-perform the static optimizations by up to 33%.

7.2.2 Application Execution Time

Next, we take a look at the application execution time, containing not only the GPU but also the CPU time. For this we only use the DM results. Figure 7.12 shows the percentage of time spend on CPU or GPU for the reference code on the Tesla K20c. In Figures 7.13 to 7.16 the total application speed up, in relation to the unmodified MATOG variant are shown (the order of the benchmarks has been altered to maximize the axis ranges of the graphs). For the Bitonic Sort we can see that the reference code is slower than the unmodified MATOG, which already uses the optimal configuration for this benchmark. In some cases the optimized variant is slower though. This is caused by the fact that it does not use always the same, but multiple slightly different implementations and therefore has to load more modules, causing a higher framework overhead. We will discuss this effect more in detail in Section 9.4. For COMIC we can see mostly better performance for the optimized MATOG and decreasing for the reference code on the newer GPUs. This benchmark clearly shows the necessity of auto-tuning for performance portability as code optimized for older hardware can perform less optimal on newer hardware. For SRAD, Hotspot and DPID we see that the reference code is usually faster than the optimized MATOG code. This is caused by the fact that the total GPU execution time of these benchmarks is very low and the additional CPU overhead through the MATOG framework causes a slowdown. For REYES we see that although we mainly had a speedup for the GPU code, the overall performance usually is decreased. Again, this is caused by (un-)loading and maintaining multiple variants of kernel implementations. Finally, for the KD-Tree benchmark we see that the optimized code is able to achieve up to 36% higher overall performance compared to the unoptimized variant.

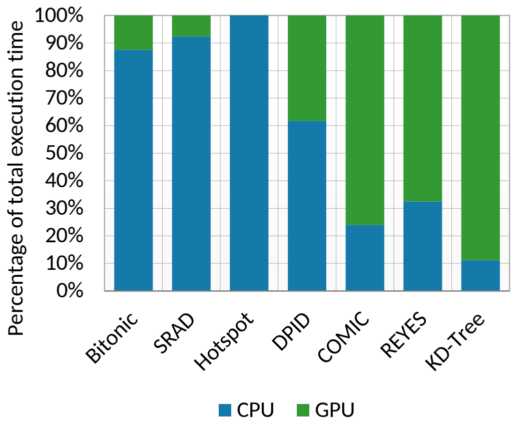


Figure 7.12: Percentage of execution time spend on CPU and GPU using the Tesla K20c. As can be seen, the first four benchmarks spend more than 50% of their execution time on the CPU.

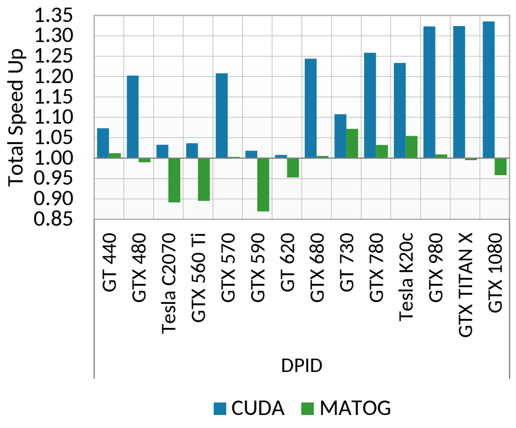


Figure 7.13: Total application speed up for the DPID benchmark. The results are mixed, but the reference code is usually the fastest, due to lower CPU overhead.

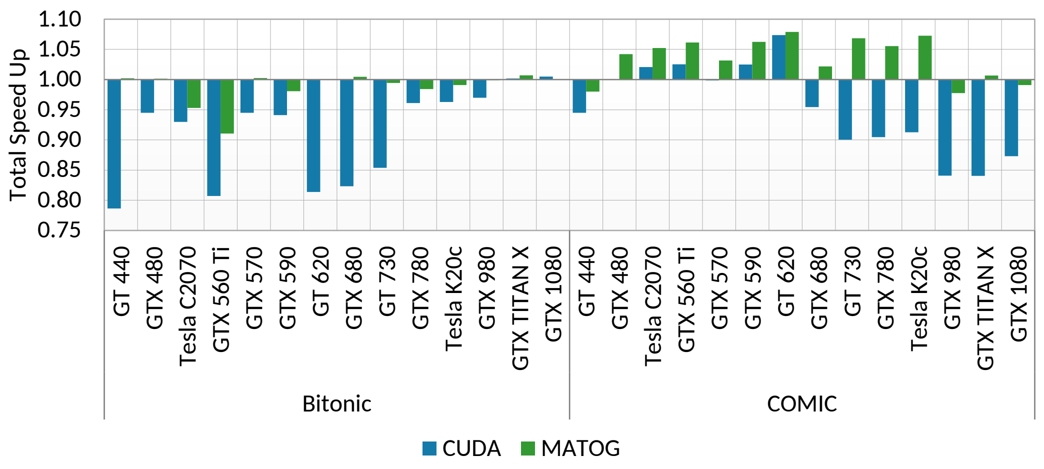


Figure 7.14: Total application speed up for the Bitonic Sort and COMIC benchmark. The reference code performs usually slower than the unoptimized and optimized MATOG code. For the Bitonic Sort, in some rare cases the optimized is slower than the unoptimized caused by overhead through maintaining multiple kernel implementations. For COMIC we see that the reference code performance significantly decreases on the newer GPUs.

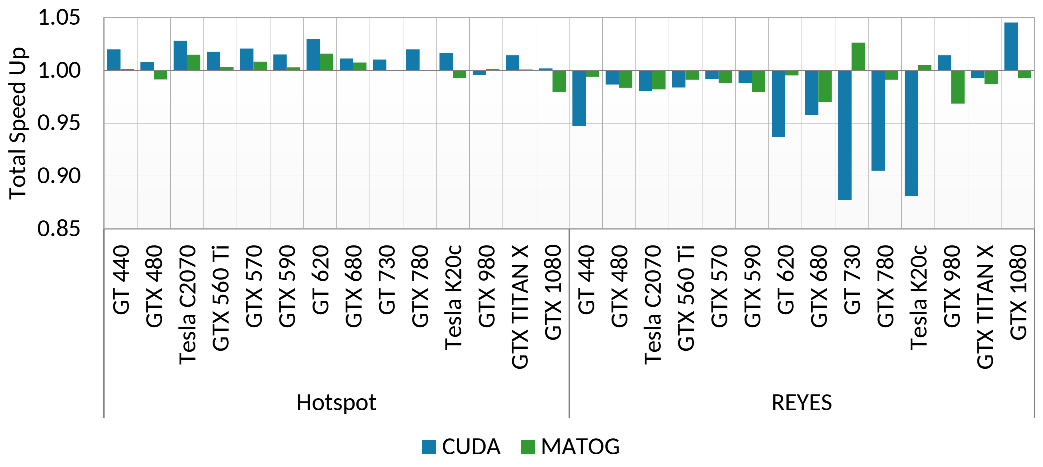


Figure 7.15: Total application speed up for the Hotspot and REYES benchmark. Especially for REYES we can see that the performance is slightly lower than the unoptimized code, caused by overhead through maintaining multiple configurations per kernel.

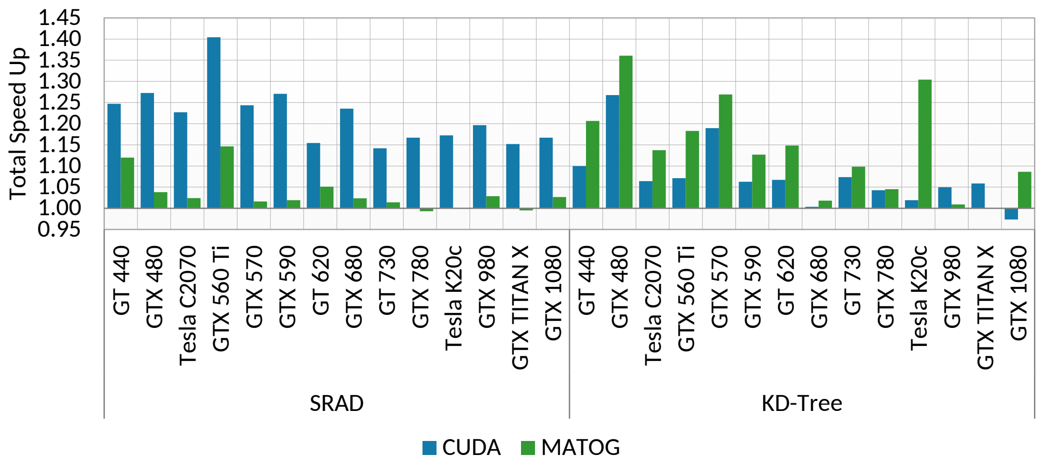


Figure 7.16: Total application speed up for the SRAD and KD-Tree benchmark. For SRAD the CUDA code is usually the fastest due to lower CPU overhead, while for the KD-Tree the optimized MATOG is faster.

7.2.3 Performance Portability

Pennycook et al. [2016] proposed a method to measure performance portability for applications across multiple platforms. They calculate the harmonic mean over the execution time for one approach over a series of executed applications and normalize it with the harmonic mean of the best results for the applications. Figure 7.17 shows the results for the hand-optimized CUDA code and our proposed method, compared to all other methods we have analyzed. Except for SRAD and Hotspot, our method is superior to the pure CUDA implementation for the GPU and application efficiency. Overall CUDA achieves an average efficiency of 88.33% GPU and 96.09% application efficiency, while our proposed method achieves 98.26% and 95.11% respectively. Assuming that an exhaustive search would yield in 100.0% GPU efficiency, our method comes pretty close, whereas it requires significantly less time to achieve its results, as we show in the next section.

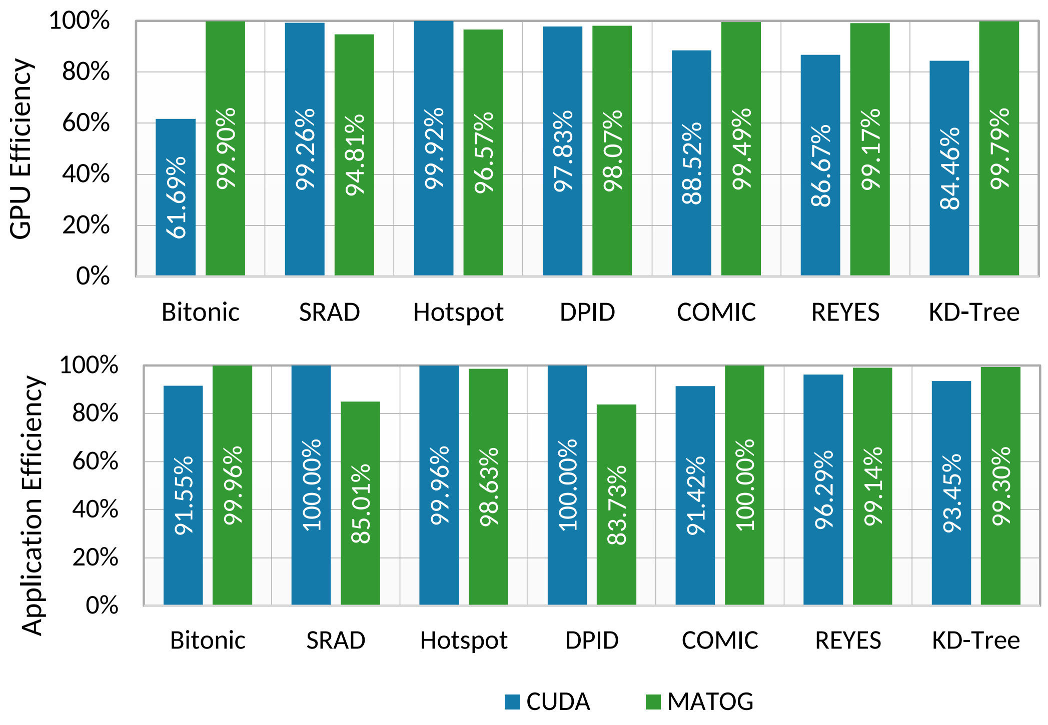


Figure 7.17: GPU (top) and application (bottom) efficiency for CUDA and our proposed method. Except for SRAD, Hotspot and DPID for the application efficiency, MATOG is superior to CUDA for both metrics.

7.2.4 Analysis Time

Finally, we take a look at the time required for the analysis on the GTX 1080. Figure 7.18 shows the time required for the profiling. It is easy to see that for the exhaustive search the time is significantly higher than for our predictive profiling. Given the high numbers of configurations for the REYES and KD-Tree benchmarks, it can be estimated that the overall exhaustive profiling for these two benchmarks would require a lot of time. Assuming that compiling a kernel requires approximately 3 s, on 16 cores the compilation allone would require ~15 days for REYES and ~44 days for the KD-Tree.

Figure 7.19 shows the ratio between profiling compared a normal application run. As can be seen, this depends on the number of configurations, while our method requires usually below 10 x, for REYES and KD-Tree less than 100 x for the profiling while the exhaustive can reach up to 10,000 x (for SRAD).

In Figure 7.20 we show the measured times for the profiling and analysis of the benchmarks. As stated before, REYES and KD-Tree require too much time for the exhaustive profiling and have therefore been excluded. Except for the Bitonic Sort, our predictive profiling is always significantly faster than the exhaustive search. The reason for this is that the Bitonic Sort only has very few possible configurations, so that the number of configurations hardly differ.

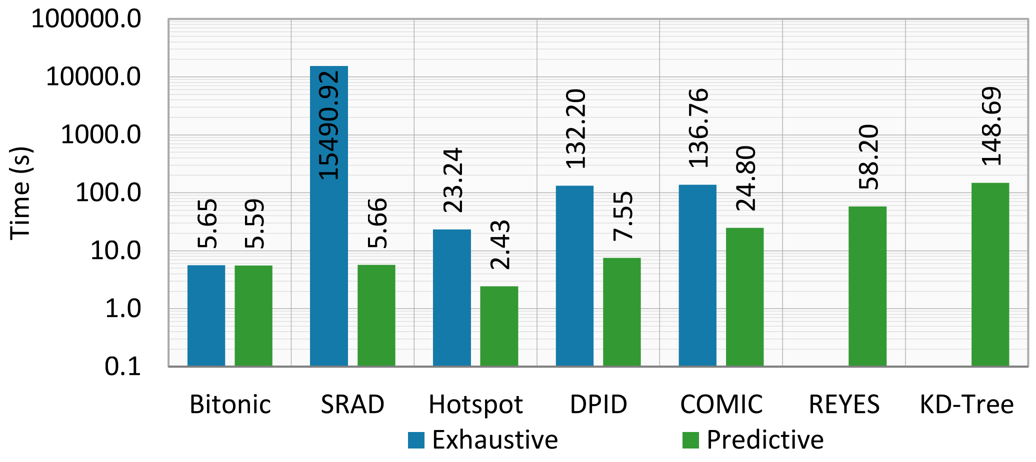


Figure 7.18: Execution time for the profiling. It is easy to see that the predictive profiling is significantly faster than the exhaustive profiling and can profile all applications within a few minutes.

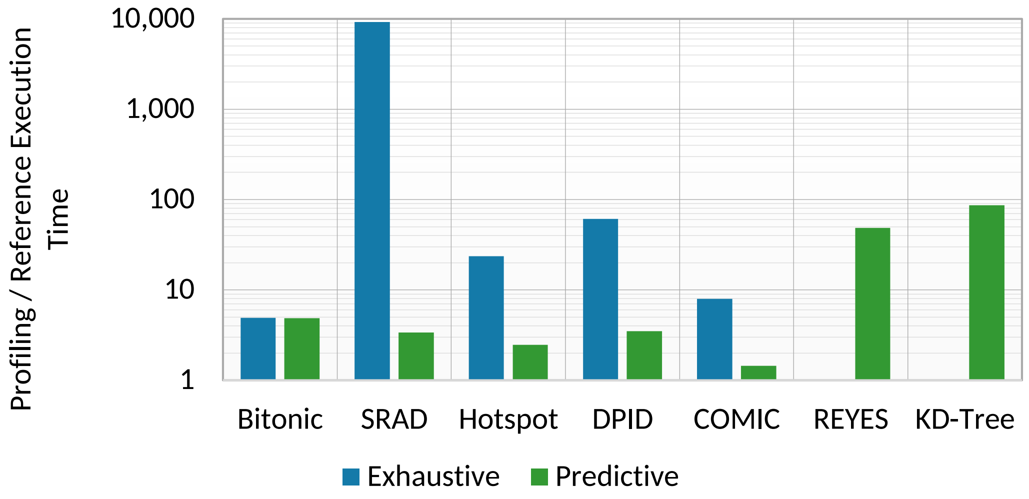


Figure 7.19: Ratio between profiling and execution time. As can be seen, with increasing complexity of the benchmark, the profiling time significantly increases, especially for the exhaustive search. It can be seen that our predictive method requires even for the complex KD-Tree benchmark less than 100 x compared to a normal application run. The exhaustive search can be multiple orders of magnitudes slower. Be reminded, the profiling time not only contains the pure execution time, but also the time required to compile the GPU kernels, convert data into other layouts and restoring the input data before starting a kernel in another configuration!

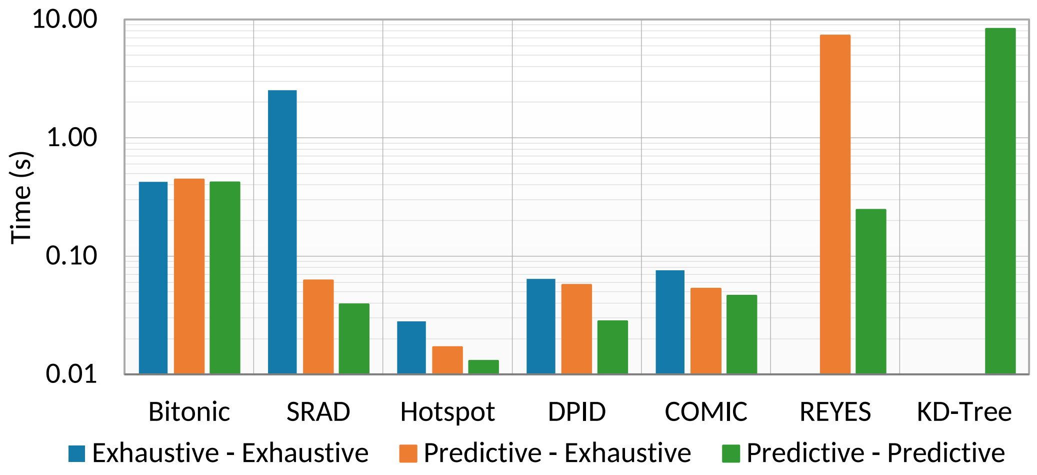


Figure 7.20: Time required for the analysis of the profiling data. The analysis is quite fast for all methods (compared to the profiling), however our proposed method (green) is always the fastest. Further, it can easily be seen that with increasing complexity the execution time rises. The exhaustive analysis requires more time using exhaustive data (blue) than with predictive data (orange) caused by much more data that needs to be loaded and processed.

Chapter 8
Empirical Performance Models

Before we discuss the results of our evaluation, we take a look onto work we have conducted to further extend the decision making of MATOG. This chapter contains recent research results that have not made it into the active development of MATOG yet and therefore are not part of the evaluation in the previous chapter. This work was conducted in conjunction with Sandra C. Amend for her Master Thesis [Amend 2017].

So far we have explained our concepts for auto-tuning array layouts, analyzing applications, how to use meta data to predict optimal layouts, using categorical decision models and how all of this is implemented in MATOG. What these model do not provide is an estimate for how long the kernel will execute on a specific hardware. This is an important feature. One use case would be to estimate prior a kernel execution whether converting an array into another layout between two kernel calls would yield enough speed up to compensate for the time necessary for the conversion. Figure 8.1 shows in which situations this would yield an improvement and in which it would not. Another application would be data partitioning in multi-heterogeneous-device applications. The problem here is that the heterogeneous devices can require a different amount of time to process the same amount of data. With an accurate prediction model it would be possible to divide the data into exactly the required pieces, so that all devices complete their task at the same time, reducing the synchronization overhead between the devices to an minimum. Our assumption is, that when an application is executed using the same data on the same GPU, the kernel runtimes will be deterministic. Therefore we expect, that it is possible to extract characteristics of the data that enable us to estimate how long a kernel will run on a specific GPU. In this chapter we evaluate whether our automatically gathered meta data supplies the required information.

Already a lot of research has been conducted to find performance models. There are two main methods, analytical and empirical performance models. Analytical methods model the correlation of the computation in relation to the used hardware. To establish these analytical models, a deep knowledge of the algorithm and hardware is necessary [Wolf et al. 2014]. These are usually handcrafted and therefore are not suited for our application, as we have the ambition to have a fully automatic auto-tuner. Empirical performance models use measured performance data, similar to the data that MATOG gathers during its profiling, and then use a method to fit some kind of performance curve into the measured data. Please

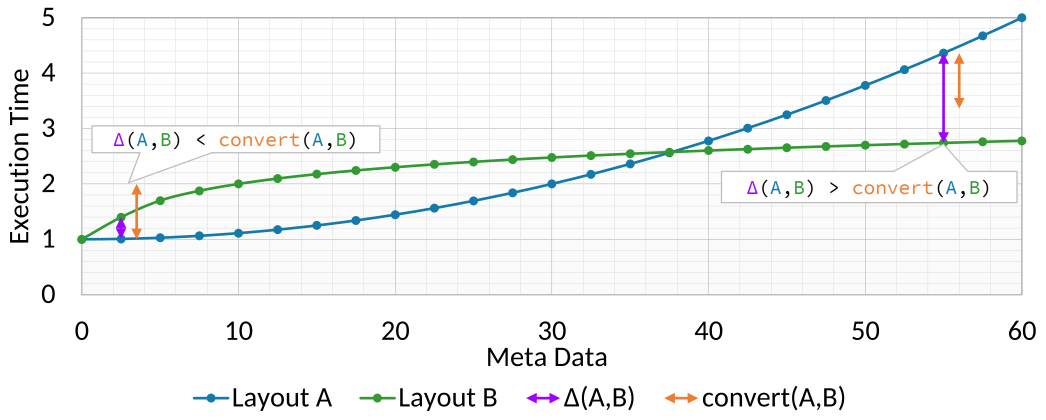


Figure 8.1: Artificial example for the execution time of two different layouts in dependency of a given meta data. Δ`(A,B)` is the speed up that could be achieved if the faster layout would be used and `convert(A,B)` is the time necessary for converting the data. In the left example, the conversion takes more time, so that this would not yield in an significant improvement. However, in the right example the conversion would result in a mentionable speed up.

refer to Section 4.5.1 for an overview of state-of-the-art techniques in this area.

Which technique is suitable to establish these empirical models depends on the application and on which data is available. Approaches that utilize *Neural Networks **(NNs)*** [Ipek et al. 2005; Lee et al. 2007; Wu et al. 2015] achieve high quality estimations but require a significant number of data samples to train the networks. MATOG does automatically gather data, however, after reducing redundant, constant and linearly dependent information these datasets are usually rather small and usually contain less than 100 entries, which are too few to be used in NNs. *Gaussian Processes **(GP)*** [Rasmussen and Williams 2006] are another method that can be used with the advantage that these only require very few data samples. Another advantage of GP is that they not only provide a predicted value, but also an error estimate, how certain the model is of the value.

In the following sections we will explain how we use GP to automatically generate performance models, evaluate their accuracy and how these could be used in MATOG. These experiments have been conducted using MATLAB and the *Gaussian Processes for Machine Learning **(GPML)** Toolbox*[1] to quickly generate results and evaluate different model implementations.

[1]www.gaussianprocess.org/gpml/code

8.1 Model Training and Prediction Accuracy

In order to train our models, we use the MATOG profiler to gather all necessary data. This data is then extracted from the MATOG database, converted in a format that can be processed by MATLAB and then directly fed into the GPML Toolbox. For the data we use the same filtering as described in Section 6.4, where we remove all constant and linear dependent entries. We evaluated multiple GP model implementations, using different covariance functions: *linear* ***(GP lin.)***, *squared exponential* ***(GP SE)***, *squared exponential with automatic relevance determination* ***(GP SE + ARD)*** and *combined linear plus squared exponential* ***(GP lin. + SE)*** [Rasmussen and Williams 2006]. Further, we compare all results against a normal *linear regression* ***(lin. reg.)*** to evaluate if the usage of a complex model as GP is beneficial. For the evaluation we chose the *"bound and split"* kernel from the REYES (Section 7.1.6) and the first main kernel of the KD-Tree benchmark (Section 7.1.7), both executed on a GTX 680. We chose these kernels, as these are the main kernels of the two most complex benchmarks we have available. Further, these two benchmarks produce the highest amount of meta data.

8.1.1 Single Dataset

First, we take a look at the accuracy of these models for predicting a single dataset. For this we learned our decision models on all available datasets and predicted the performance on the *"Utah Teapot"* (REYES) and *"Kitchen"* (KD-Tree) dataset. In Figure 8.2 we show the results of the best performing GP model (GP lin. + SE) against the lin. reg. model. In this test we can see that for both benchmarks, most predictions are quite accurate. For the KD-Tree benchmark we can see that in the lower part of the figures some samples deviate significantly. In this area the models are unable to use our automatically gathered data to perform accurate predictions. However, overall most samples for both methods seem to be accurate enough for our purpose.

Next we directly compare the accuracy of the GP lin. + SE and lin. reg. models for each sample. Results are shown in Figure 8.3. For the GP lin. + SE also the error estimate is shown. The results of REYES show that the GP lin. + SE is in the most cases closer to the measured results than the lin. reg.. For the KD-Tree both seem to equally good. In the first 10 samples both models significantly differ from the results. This is the area where the models cannot use our meta data to accurately predict the performance.

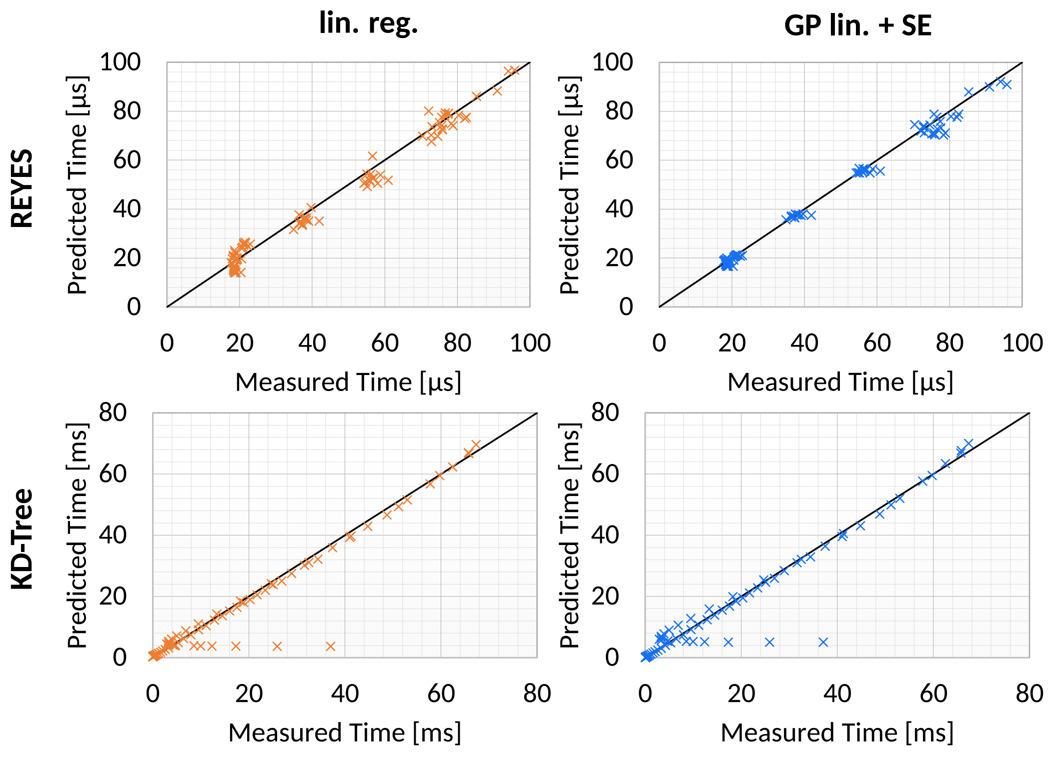
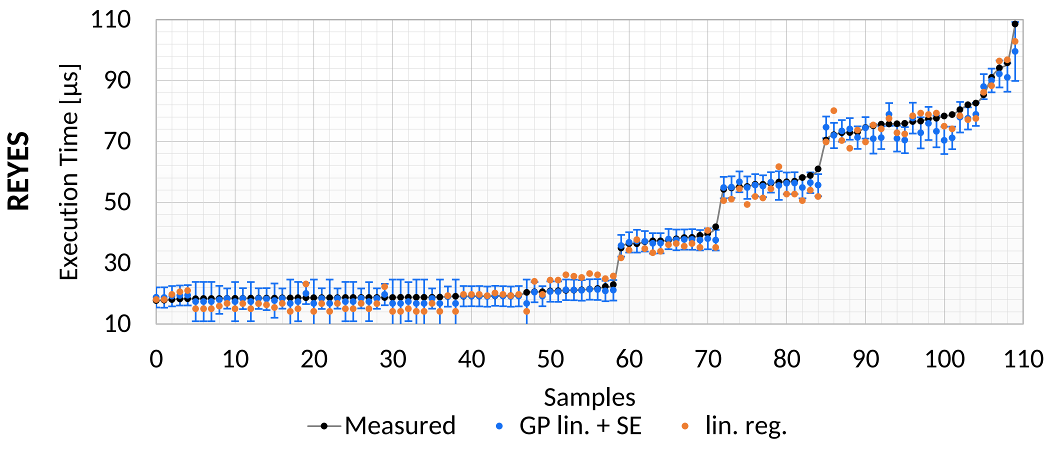


Figure 8.2: Results for the lin. reg. (left) and GP lin. + SE (right). The black line indicates the optimum, where the prediction is identical to the measured value. The more a value deviates, the less accurate the model is. For REYES (top) we see that both models perform equally. For the KD-Tree (bottom) we can see that in the lower part of the chart both models deviate significantly from the expected values. This is an area where the models are unable to use our automatically gathered data to predict the performance.



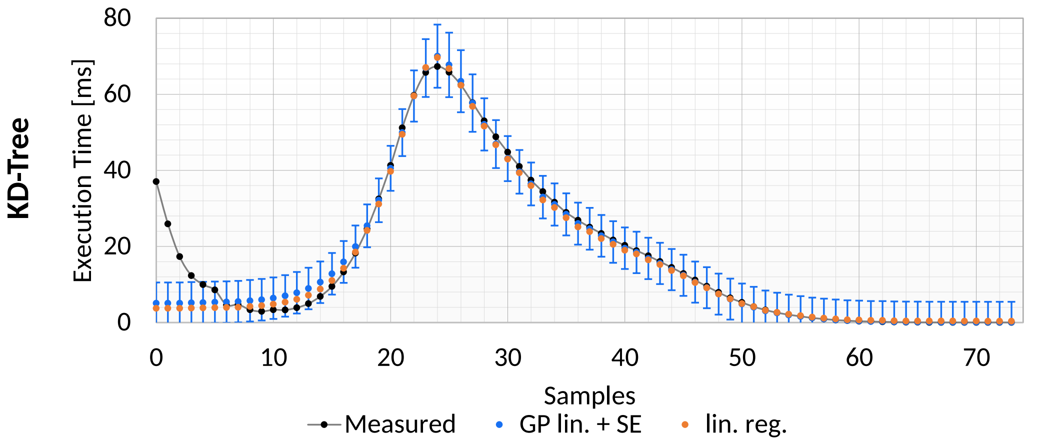


Figure 8.3: Predicted over measured execution time for the GP lin. + SE model with 95% confidence interval and the lin. reg. model. The samples for the REYES (top) are sorted ascending for the measured execution time. As can be seen, the lin. reg. model deviates more than the GP lin. + SE model. For the KD-Tree (bottom) the samples are not sorted. Here we can clearly see the area where the models deviate from the measured runtime.

8.1.2 Multiple Datasets

Next, we perform our analysis on all available datasets. For this we go over all datasets available for the benchmarks and train the models on all datasets, except the one, we are predicting. Further we show results for all tested GP model variants and the lin. reg.. We first take a look at the *Relative Root Mean Squared Error **(RRMSE)***. This is defined as:

$$\frac{\sqrt{\sum_{i=1}^{d}(t_m(i) - t_p(i))^2}}{\max(t_m)} \tag{8.1}$$

It can be used to compare the accuracy of models. The lower the RRMSE, the better the model performance. Our results are shown in Figure 8.4.

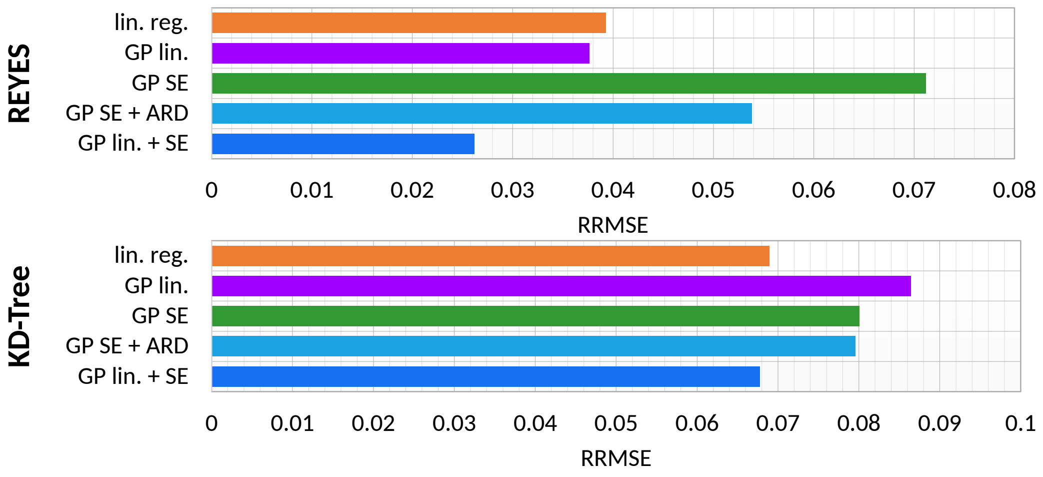


Figure 8.4: RRMSE for all tested models (lower is better). As can be seen, GP lin. + SE works best for REYES (top), followed by the GP lin. and lin. reg. models. For the KD-Tree, again the GP lin. + SE performs best, while the difference to the lin. reg. is quite low.

Finally, we compare the ratios between the measured and the predicted values (Figure 8.5). This shows how big the differences are. We visualize this using a box plot. The closer the median (inner most line) is to the 1.0 ratio (indicated by black line) the better. The box itself shows the upper and lower quantile around the median value. The black error bars further show 1.5x of the box extend. All remaining samples are outside this range.

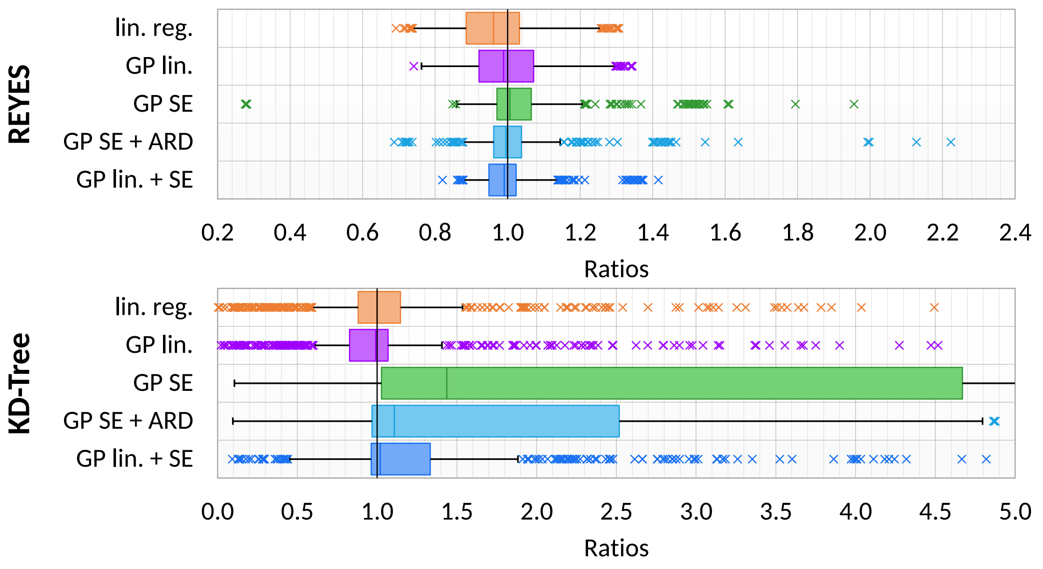


Figure 8.5: Distributions of ratios for the tested models. For the REYES (top) can be seen, again GP lin. + SE performs best, followed by GP lin. and lin. reg., whereas the latter deviates most from the 1.0 ratio. The results of the GP SE for the KD-Tree (bottom) significantly deviate from the optimum. According to the plot, the lin. reg. is closer to the median and the box is tighter than for the GP lin. + SE.

8.1.3 Error Cases

So far the models have in mostly worked as expected, except for the few samples in Figure 8.3. However, we encountered a case, where the models provided very bad predictions. This case occurred when training our prediction models for the KD-Tree on the GTX 1080. The results of the models are shown in Figures 8.6 to 8.9. What we see is that the predictions are very bad for the first 45 samples (Figure 8.7). The reason for this lies in the difference of the 3D scenes that we use to train our models, which has similarities to the SVM problem we had encountered previously (Section 6.4.1). There exist multiple problems. First, as shown in Figures 7.3 and Figures 7.4, the scenes differ in their properties. The 3D scans (e.g., Buddha) are very dense with small, nearly equally sized triangles. The other scenes (e.g., Kitchen) are artist modeled, with varying triangle sizes. San Miguel and the Powerplant introduce a new property that significantly differs from the others. These scenes consist of densely packed and very sparse or even empty areas. These scene dependent properties have an impact on the performance, but are not captured by any of our automatically gathered meta data. Second, Figure 8.10 shows the distribution of our meta data in linear space. For the two additional datasets (San Miguel and Powerplant) the data significantly differs from the others. When predicting for the Powerplant, the models only know meta data up to 25 M triangles and 4.2 M subtrees. As the Powerplant uses up to 45 M triangles and 6 M subtrees, the performance models have no reference data available and need to extrapolate. Third, in the case, when we predict the performance for the Kitchen (Figure 8.7), the two additional scenes provide meta data in the same range as the smaller scenes, but their runtime differs (because of their different properties), so that the prediction models interpolate between the smaller and the bigger scenes. All these effects have a negative impact on the model's accuracy. However, a positive property of the GP is, that it knows that its predictions are bad, as indicated by the error bars in Figure 8.7. This does not solve the problem, but it allows to detect when the model provides bad predictions. So far we do not know a solution to this problem. It might be possible to build a two layer model, that first determines a rough classification (e.g., *"small"*, *"medium"* or *"big"* scene) and then builds a separate prediction model for each of these classes.

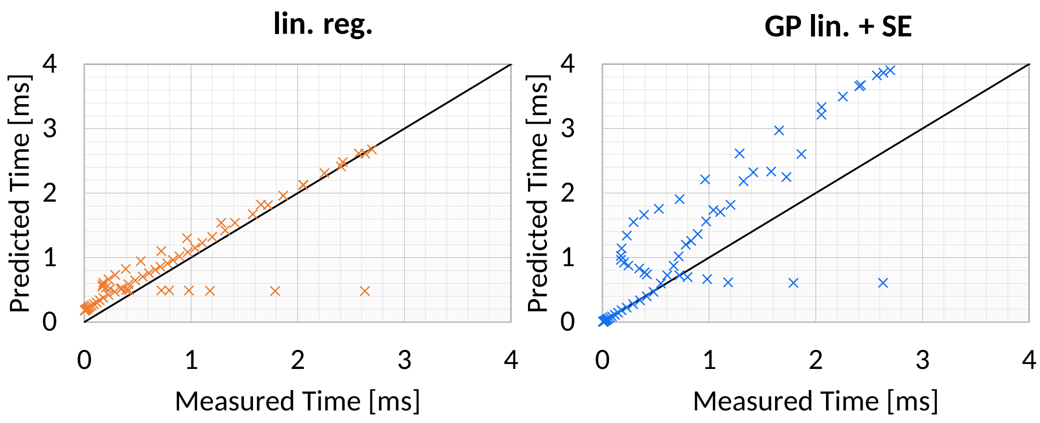


Figure 8.6: Results for the lin. reg. (left) and GP lin. + SE (right). Both methods significantly differ from the optimum, while the lin. reg. is overall closer to the optimum, the GP lin. + SE is much better in the lower region and worse in the upper.

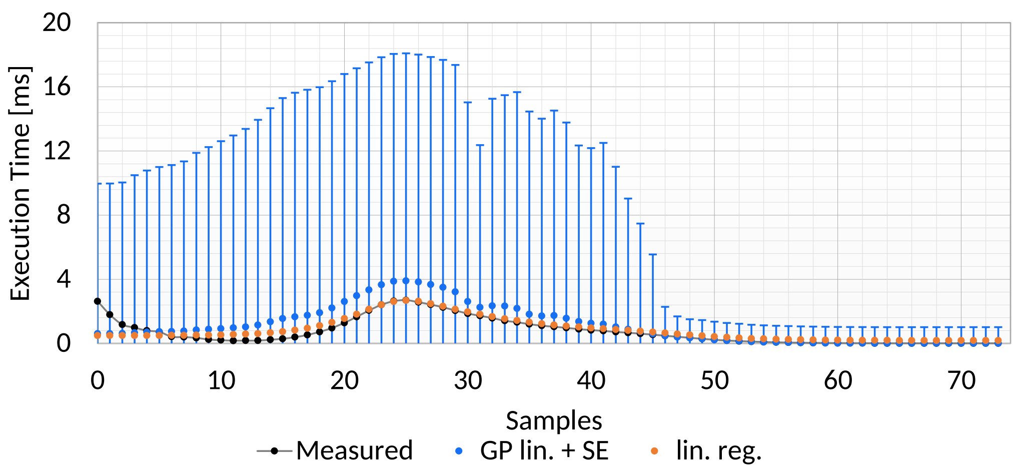


Figure 8.7: Predicted over measured execution time for the GP lin. + SE model with 95% confidence interval and the lin. reg. model. The samples are not sorted. As can be seen, both models deviate significantly from the measured value for the first 45 samples. Also the error estimate of the GP is very high in this area.

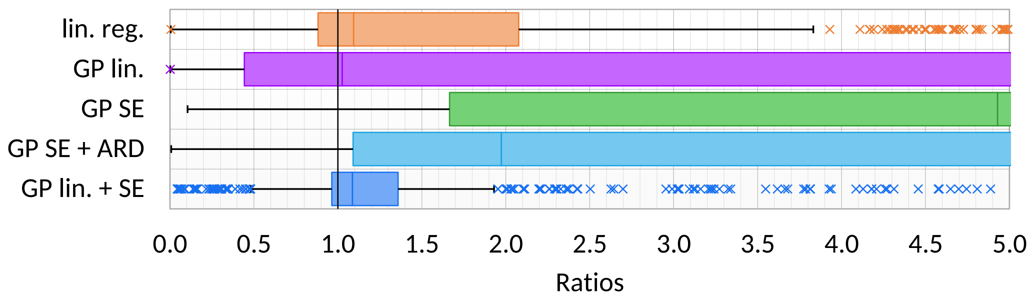


Figure 8.8: Distributions of ratios for the tested models. Again the GP lin. + SE has the best result. Although most of its values are in an acceptable range, it has a significant number of outliers.

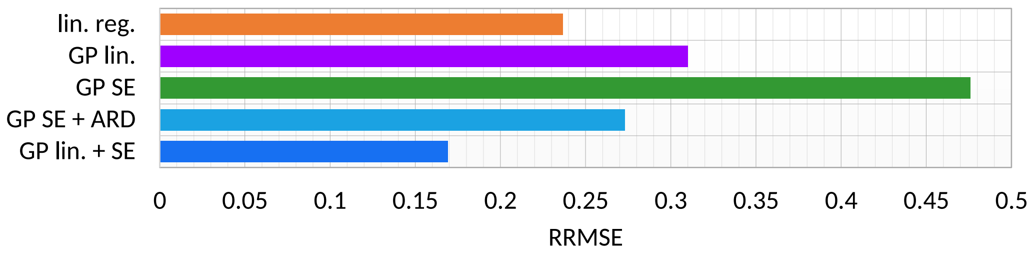


Figure 8.9: RRMSE for all tested models. Again the GP lin. + SE performs best, although the results of the other figures suggest an overall bad fit of the model.

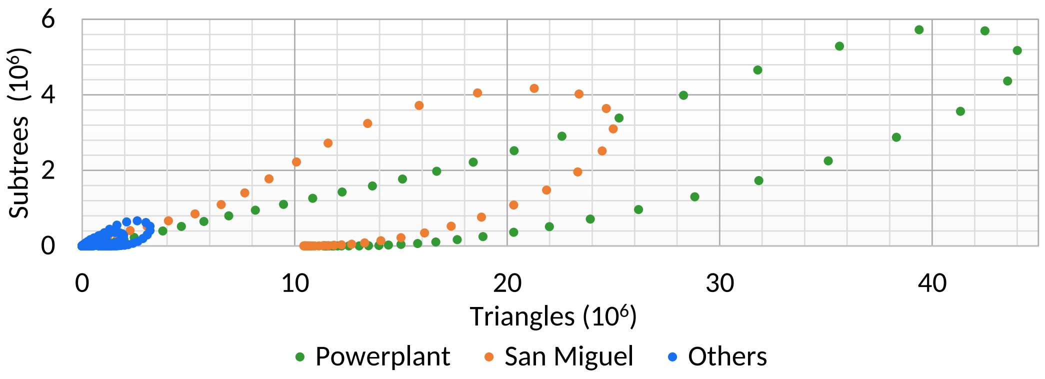


Figure 8.10: Selected meta data dimensions for all KD-Tree datasets. The two largest (San Miguel and Powerplant) are highlighted. As can be seen, their meta data values significantly differ from those of the other models.

8.2 Predicting Unknown Configuration Performance

In order to be useful for MATOG, we have to combine our previously introduced prediction for non-profiled configurations (Section 6.2.2) with the meta data based prediction models. This enables us to predict the performance of kernels for non-profiled configurations and non-profiled meta data. With this we would be able to use only a few prediction models, to predict the performance of the entire solution space. The previously mentioned use case with converting an array prior a kernel execution could use this method. Further, it could be used to validate and improve the model's accuracy, as the predicted time could be compared with the actual execution time. With an initial model, generated using our profiling method (Section 6.2), the auto-tuner would compare the predictions with the actual measured performance and adjust the models accordingly, to further increase their accuracy.

For this experiment, we first built a GP prediction model for each base and support configuration. Second, we use our prediction formula (Equation 6.1) but instead of fixed time values, we use our prediction models. All tests have been performed on a GTX 680 using the GP lin. + SE model, as this proved to be the best working in the previous evaluations. Results are shown in Figures 8.11 and 8.12. Again, the REYES case works better than the KD-Tree, as this has a +10 ms offset. However, the quality of the predictions is comparable to the predictions in Figure 6.3.

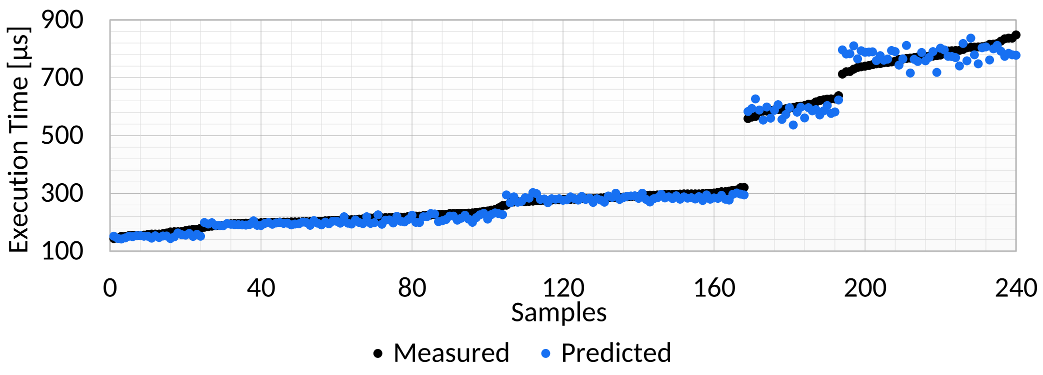


Figure 8.11: Results for predicting 240 randomly selected configurations of the bound and split kernel. The accuracy is quite good and comparable with other results of the MATOG prediction.

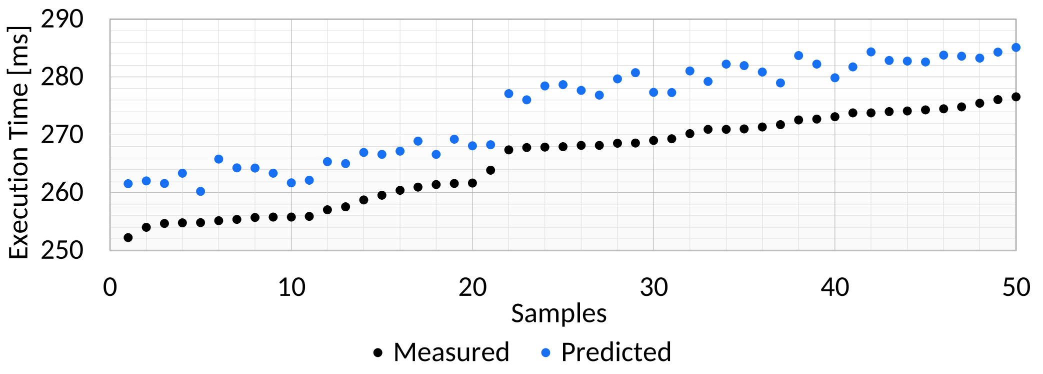


Figure 8.12: Results for predicting 50 randomly selected configurations of the binning kernel. There is a slight offset of approximately 10 ms. Despite this offset, the results follow relatively good the measured values.

Chapter 9
Discussion

In this chapter we discuss our results and whether MATOG was able to satisfy our goals. First, we summarize this thesis and its contributions (Section 9.1). Then we analyze whether auto-tuning is capable of providing the advances that it is promising and how applicable they are (Section 9.2). Further, we take a closer look onto the optimizations that have been chosen by MATOG throughout our experiments and analyze if we can extract knowledge from them that can further be used to improve auto-tuners or manual code optimizations (Section 9.3). In Section 9.4 we take a closer look onto MATOG itself, how it performed and if there are aspects that should be improved/changed in future. Finally, we conclude our discussion by reflecting the goals we set for this thesis (Section 9.5).

9.1 Summary

In this thesis we have addressed the question if it is possible to automatically optimize the performance of memory access in GPU applications. We therefore developed an auto-tuner that specifically targets array access in NVIDIA CUDA applications. The main problem we faced has been how to efficiently determine optimal array layouts. As we decided to use empirical profiling instead of an analytical method, this analysis can take a huge amount of time for very complex applications if done with an exhaustive search, which is in general infeasible. We therefore developed multiple techniques to reduce this time to an absolute minimum, while achieving nearly optimal performance, comparable to the results of an exhaustive search. For this we employ an in-application profiling that uses a prediction algorithm to estimate the performance of huge parts of the solution space. This procedure only requires a very limited number of measurements. Given this data, we established a dependency graph to model the relation between multiple kernel executions and estimate the overall application execution time. This gives us optimal decisions, which we use to train decision models that adaptively select array layouts during runtime according to the data used in the application. The entire optimization process is designed to work fully automatic and therefore does not require any user interaction (except for integrating MATOG into the application).

9.2 Is auto-tuning useful?

In our introduction we stated that to achieve optimal performance it is necessary to adapt the code to the underlying hardware. For the evaluation we ran code on different kinds of GPUs (low-, mid-, high-end and HPC) of four different GPU generations. As we have seen, the performance of the reference code was usually optimal for the GPU that the code was developed for, but the more complex an application was the higher the chance that this reference code performed less optimal on newer hardware. This happened, e.g., for the COMIC benchmark, where starting with Kepler GPUs the reference code was even slower than the unoptimized baseline code of MATOG. On the Fermi it was up to 5% faster. This clearly shows the necessity of auto-tuning in terms of performance portability.

Further, we have seen that depending on the complexity of the application static optimizations (Bitonic Sort, SRAD, Hotspot, DPID and COMIC) may suffice, but adaptiveness did not reduce the performance. In some cases (REYES and KD-Tree) these dynamic optimizations achieved a significantly higher performance.

The results showed that optimal memory access is crucial in many applications and can significantly influence the performance of the code. However, solely optimizing the GPU code does not necessarily improve the overall performance of an application as can be seen in the REYES benchmark, where MATOG achieves a significant GPU speed up on all cards with CC 3.5 or higher (Figure 7.10), but the total application speed up is slightly negative in most cases (Figure 7.15). This is caused by memory layouts that perform less optimal on the CPU and by framework overhead. Section 9.4 discusses this in more detail.

Our results further show that auto-tuning an application can be done quite fast, as for all benchmarks we required only a few minutes to optimize these using MATOG. Even for applications with longer execution times, this is most likely still significantly faster than optimizing the code manually. It also does not require any knowledge of the hardware or software and can therefore be executed by any person. Manual code optimizations always require significant programming skills and a sophisticated knowledge of the application, applied algorithms and the hardware.

9.3 Which optimizations are optimal?

A key question is, whether we can find any regularity in which optimizations are optimal since this would allow to perform static analysis. In Table 9.1 to 9.4 we show how often which layout, cache size, transposition or memory have been used by MATOG. Figure 9.1 shows the different indexing schemes used by MATOG.

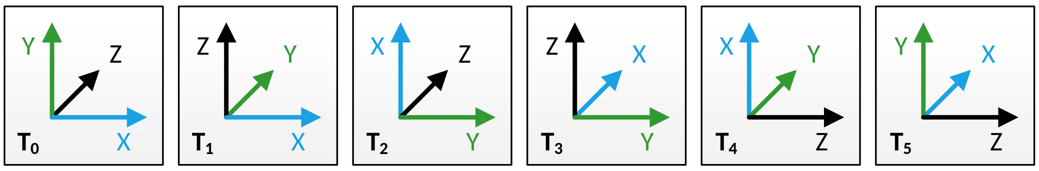


Figure 9.1: Different indexing schemes (transpositions) for a 3D matrix and the corresponding internal identification (T_0 to T_5). Lower dimensional matrices behave equally. C++ code for T_0 is: `x + y * size_x + z * size_x * size_y`

The results (Table 9.1 to 9.4) show that not only between the different benchmarks significantly different configurations are used, but also between the GPUs even within the same architecture. There is no clear tendency towards a specific struct layout, transposition or the L1 cache size visible. Solely the usage of local memory instead of shared memory proves only to be beneficial in some really rare cases (Table 9.4). REYES (Table 9.3) is the only benchmark where array sizes are known at compile time. This benchmark greatly benefits from using constant memory, although again the usage of texture, global and constant memory varies significantly between the different architectures. Further, it can be seen that the usage of texture memory is preferred on all newer platforms, whereas in most of the cases a certain balance between global and texture is chosen. What we can see is the obvious fact that using texture memory solely benefits arrays that are read more than once. Not using texture memory in this case allows to use the cache more efficiently as read-once data does not suppress other data. Be aware that the percentage does not represent the amount of memory that is used with the respective memory but the number of arrays!

Benchmark	GPU	Cache Size			Layout			Transposition						Local Arrays		Global Arrays		
		SM	L1	EQ	AoS	SoA	AoSoA	T_0	T_1	T_2	T_3	T_4	T_5	Shared	Local	Global	Texture	Constant
Bitonic	GT 440	76%	24%		0%	75%	25%									100%	0%	
	GTX 480	76%	24%		0%	75%	25%									100%	0%	
	Tesla C2070	71%	29%		0%	75%	25%									100%	0%	
	GTX 560 Ti	71%	29%		25%	75%	0%									100%	0%	
	GTX 570	79%	21%		0%	75%	25%									100%	0%	
	GTX 590	80%	20%		0%	75%	25%									100%	0%	
	GT 620	71%	29%		0%	100%	0%									100%	0%	
	GTX 680	50%	21%	29%	0%	100%	0%									100%	0%	
	GT 730	71%	0%	29%	25%	75%	0%									100%	0%	
	GTX 780	75%	25%	0%	25%	75%	0%									50%	50%	
	Tesla K20c	51%	29%	21%	0%	100%	0%									50%	50%	
	GTX 980				0%	75%	25%									50%	50%	
	GTX TITAN X				0%	100%	0%									100%	0%	
	GTX 1080				25%	75%	0%									100%	0%	
SRAD	GT 440	100%	0%					56%	44%							60%	40%	
	GTX 480	100%	0%					52%	48%							60%	40%	
	Tesla C2070	100%	0%					52%	48%							60%	40%	
	GTX 560 Ti	100%	0%					56%	44%							50%	50%	
	GTX 570	100%	0%					52%	48%							60%	40%	
	GTX 590	100%	0%					52%	48%							50%	50%	
	GT 620	100%	0%					65%	35%							60%	40%	
	GTX 680	100%	0%	0%				69%	31%							90%	10%	
	GT 730	100%	0%	0%				66%	34%							80%	20%	
	GTX 780	100%	0%	0%				66%	34%							0%	100%	
	Tesla K20c	100%	0%	0%				52%	48%							0%	100%	
	GTX 980							56%	44%							20%	80%	
	GTX TITAN X							61%	39%							10%	90%	
	GTX 1080							70%	30%							100%	0%	

Table 9.1: Usage of optimizations for the Bitonic and SRAD benchmark.

Benchmark	GPU	Cache Size			Layout			Transposition						Local Arrays		Global Arrays		
		SM	L1	EQ	AoS	SoA	AoSoA	T_0	T_1	T_2	T_3	T_4	T_5	Shared	Local	Global	Texture	Constant
Hotspot	GT 440	0%	100%					50%	50%							100%	0%	
	GTX 480	0%	100%					50%	50%							100%	0%	
	Tesla C2070	0%	100%					50%	50%							100%	0%	
	GTX 560 Ti	0%	100%					50%	50%							33%	67%	
	GTX 570	0%	100%					50%	50%							100%	0%	
	GTX 590	0%	100%					50%	50%							100%	0%	
	GT 620	0%	100%					50%	50%							100%	0%	
	GTX 680	0%	0%	100%				50%	50%							100%	0%	
	GT 730	0%	0%	100%				50%	50%							33%	67%	
	GTX 780	0%	0%	100%				50%	50%							33%	67%	
	Tesla K20c	100%	0%	0%				50%	50%							33%	67%	
	GTX 980							50%	50%							67%	33%	
	GTX TITAN X							50%	50%							67%	33%	
	GTX 1080							17%	83%							67%	33%	
DPID	GT 440	67%	33%		58%	42%	0%	100%	0%							100%	0%	
	GTX 480	100%	0%		42%	48%	10%	83%	17%							100%	0%	
	Tesla C2070	67%	33%		28%	72%	0%	94%	6%							100%	0%	
	GTX 560 Ti	17%	83%		58%	42%	0%	100%	0%							100%	0%	
	GTX 570	50%	50%		28%	72%	0%	94%	6%							100%	0%	
	GTX 590	0%	100%		42%	48%	10%	75%	25%							100%	0%	
	GT 620	33%	67%		58%	42%	0%	100%	0%							100%	0%	
	GTX 680	50%	0%	50%	27%	31%	42%	76%	24%							100%	0%	
	GT 730	17%	0%	83%	47%	53%	0%	100%	0%							42%	58%	
	GTX 780	33%	0%	67%	69%	31%	0%	75%	25%							42%	58%	
	Tesla K20c	83%	0%	17%	31%	69%	0%	89%	11%							42%	58%	
	GTX 980				10%	48%	42%	37%	63%							42%	58%	
	GTX TITAN X				22%	37%	42%	31%	69%							42%	58%	
	GTX 1080				0%	42%	58%	31%	69%							94%	6%	

Table 9.2: Usage of optimizations for the Hotspot and DPID benchmark.

Benchmark	GPU	Cache Size			Layout			Transposition						Local Arrays		Global Arrays		
		SM	L1	EQ	AoS	SoA	AoSoA	T_0	T_1	T_2	T_3	T_4	T_5	Shared	Local	Global	Texture	Constant
COMIC	GT 440	52%	48%					8%	75%	0%	8%	8%	0%	100%	0%	100%	0%	
	GTX 480	75%	25%					26%	66%	0%	8%	0%	0%	100%	0%	100%	0%	
	Tesla C2070	84%	16%					28%	63%	0%	0%	0%	8%	100%	0%	100%	0%	
	GTX 560 Ti	50%	50%					3%	80%	0%	0%	8%	8%	100%	0%	100%	0%	
	GTX 570	66%	34%					26%	66%	0%	0%	8%	0%	100%	0%	100%	0%	
	GTX 590	52%	48%					12%	80%	0%	0%	0%	8%	100%	0%	100%	0%	
	GT 620	50%	50%					8%	75%	0%	8%	8%	0%	100%	0%	100%	0%	
	GTX 680	50%	34%	16%				38%	54%	0%	8%	0%	0%	100%	0%	100%	0%	
	GT 730	52%	39%	9%				36%	47%	0%	0%	0%	17%	100%	0%	50%	50%	
	GTX 780	50%	11%	39%				38%	46%	8%	0%	0%	8%	100%	0%	50%	50%	
	Tesla K20c	66%	11%	23%				49%	34%	0%	0%	0%	17%	100%	0%	50%	50%	
	GTX 980							54%	38%	8%	0%	0%	0%	100%	0%	63%	38%	
	GTX TITAN X							83%	8%	0%	8%	0%	0%	100%	0%	75%	25%	
	GTX 1080							54%	38%	0%	8%	0%	0%	100%	0%	75%	25%	
REYES	GT 440	79%	21%		25%	41%	34%	74%	14%	0%	0%	11%	0%			37%	11%	52%
	GTX 480	69%	31%		32%	48%	20%	48%	41%	0%	0%	0%	11%			49%	6%	45%
	Tesla C2070	77%	23%		12%	79%	9%	76%	23%	0%	0%	0%	0%			61%	7%	33%
	GTX 560 Ti	90%	10%		11%	65%	23%	84%	5%	0%	0%	11%	0%			56%	4%	40%
	GTX 570	88%	12%		22%	41%	36%	44%	45%	11%	0%	0%	0%			37%	22%	41%
	GTX 590	66%	34%		12%	53%	36%	47%	42%	0%	0%	11%	0%			48%	19%	33%
	GT 620	79%	21%		30%	56%	13%	83%	5%	0%	0%	11%	0%			52%	4%	45%
	GTX 680	44%	29%	27%	19%	42%	39%	46%	40%	13%	0%	0%	0%			7%	51%	42%
	GT 730	31%	20%	49%	54%	11%	35%	72%	26%	1%	0%	0%	1%			3%	44%	53%
	GTX 780	44%	2%	54%	29%	58%	13%	68%	19%	2%	0%	11%	0%			3%	42%	56%
	Tesla K20c	50%	10%	40%	14%	72%	13%	68%	18%	0%	0%	13%	0%			0%	45%	55%
	GTX 980				63%	26%	11%	53%	33%	1%	0%	14%	0%			12%	38%	50%
	GTX TITAN X				53%	29%	18%	43%	21%	1%	0%	33%	2%			16%	45%	39%
	GTX 1080				72%	6%	22%	42%	22%	2%	1%	33%	0%			7%	58%	35%

Table 9.3: Usage of optimizations for the COMIC and REYES benchmark.

Benchmark	GPU	Cache Size			Layout			Transposition						Local Arrays		Global Arrays		
		SM	L1	EQ	AoS	SoA	AoSoA	T_0	T_1	T_2	T_3	T_4	T_5	Shared	Local	Global	Texture	Constant
KD-Tree	GT 440	39%	61%		29%	70%	1%	39%	40%	16%	5%	1%	0%	100%	0%	100%	0%	
	GTX 480	80%	20%		27%	71%	2%	56%	26%	11%	7%	0%	0%	100%	0%	100%	0%	
	Tesla C2070	40%	60%		30%	68%	2%	50%	32%	11%	7%	0%	0%	100%	0%	100%	0%	
	GTX 560 Ti	57%	43%		68%	30%	2%	52%	23%	10%	11%	5%	0%	43%	57%	100%	0%	
	GTX 570	66%	34%		81%	17%	2%	28%	50%	10%	11%	1%	0%	100%	0%	99%	1%	
	GTX 590	61%	39%		50%	48%	2%	51%	29%	13%	6%	0%	1%	97%	3%	100%	0%	
	GT 620	60%	40%		27%	60%	13%	40%	44%	10%	5%	0%	0%	91%	9%	77%	23%	
	GTX 680	64%	22%	14%	69%	21%	10%	74%	14%	10%	0%	2%	0%	100%	0%	100%	0%	
	GT 730	47%	29%	24%	20%	66%	14%	54%	32%	12%	0%	1%	0%	96%	4%	81%	19%	
	GTX 780	48%	26%	27%	22%	76%	2%	55%	32%	10%	3%	0%	0%	100%	0%	46%	54%	
	Tesla K20c	23%	41%	36%	44%	30%	26%	65%	23%	11%	1%	0%	0%	100%	0%	50%	50%	
	GTX 980				22%	65%	13%	43%	36%	7%	0%	14%	1%	100%	0%	55%	45%	
	GTX TITAN X				8%	71%	21%	47%	35%	6%	1%	11%	0%	100%	0%	57%	43%	
	GTX 1080				32%	64%	4%	56%	25%	1%	3%	10%	4%	100%	0%	52%	48%	

Table 9.4: Usage of optimizations for the KD-Tree benchmark.

9.4 MATOG Implementation Improvements

In our results we have been able to see that MATOG achieves considerable speed ups over an unoptimized version and also over the hand-written code, especially on the cards the reference code has not been optimized for. Nevertheless, there are several cases where the implementation of MATOG could be improved.

Starting with the implementation of MATOG data structures – as previously mentioned – they are prone to pointer aliasing, which can cause less performance as the compiler cannot know if different pointers do overlap. Therefore the amount of ILP can be decreased as it is not possible to distinguish if the results depend on each other. Currently we use C++ classes to hide the memory access from the user. As classes are normally used to be initialized in multiple different instances, it is not possible to tell the compiler that pointers returned by the class do not overlap. For MATOG each class is uniquely instantiated using a template and the framework also ensures that the pointer only exists once in the code. As far as we know, there is no way to tell the compiler about this. One solution would be to switch away from code generation and use a source-to-source compiler, which then directly puts the optimization into the code, without classes. This would require a lot of work to establish and maintain such a compiler. As C++ – and therefore CUDA – allows all kinds of fancy typedef structures and preprocessor hacks, it could be difficult to build a compiler that can transform all types of possible C++ code into a MATOG compatible solution. Using an existing optimizing compiler, e.g., OpenARC [Lee and Vetter 2014], which automatically generates parallel CUDA code from OpenACC, could be an option, but it would remove the ability to directly write CUDA code.

Another limitation of MATOG is that it does not take any CPU times into consideration. Layouts that decrease the CPU performance are currently not regarded during the optimization. Capturing the CPU time during the execution requires to put in check points. We have two ideas for solving this. First, as MATOG already intercepts CUDA Driver API calls, it could track the time elapsing between these. This can be implemented very easily but would also capture all kinds of I/O, which can contain a lot of noise. The other option is to disallow memory access to MATOG data structures in the host code. Instead CPU kernels could be supported that are equal to GPU kernels but are executed on the CPU. In this case it would be possible not only to auto-tune CPU code inside this kernel – as it is done with the GPU code – but also to track explicitly the time and optimize for it. CPU memory hierarchies are getting more and more complex. In the next generation HPC processors of AMD [Vijayaragavan et al. 2017], not only a CPU and GPU will be integrated onto the same chip, it will also have on-chip (HBM) and off-chip (DDR) memory. This make it essential that also CPU applications become more aware of the underlying memory hierarchy, to draw benefit from the different kinds of memory. Zivanovic et al. [2017] explore the possibilities of such systems for common HPC applications and argue that the benefit comes with an increased development and optimization effort. This is where the functionality of MATOG could be used to assist the development. For MATOG this would mean that a compiler and module load/unload infrastructure equal to the existing CUDA infrastructure has to be implemented.

One other aspect that decreases the CPU performance is the CUDA module loading/unloading from MATOG. As it is possible that during the execution several hundred different modules are used, MATOG loads and unloads these in the background, while it maintains a cache that employs a *least recently used* ***(LRU)*** strategy. This part of MATOG has several issues. First, MATOG stores the compiled CUDA images inside its database. The advantage is a very easy and fast access to the modules (as parts of the database are kept inside the memory) and we do not need to store every image as a single file, which would certainly litter the hard drive. However, to load these images we have to use the function `cuModuleLoadData`, which in contrast to `cuModuleLoad` seems to require significantly more time: ~838.66 µs vs ~163.09 µs per module. We have not been able to figure out a reason for this and there is also no information given, neither in the CUDA programming guide [NVIDIA 2016a] nor the Driver API reference [NVIDIA 2016b] for this effect. Even more problematic is the execution time of unloading modules, which is significantly higher (~3.533 ms per module). At the moment we do not know a good method to decrease this overhead, most likely reducing the total number of configurations would be the best solution. This would require to cluster the configurations according to their performance. We will discuss this in more detail in Section 10.1. Loading an unlimited number of modules also does not work, as all modules require memory on the GPU and loading/keeping too many modules could cause an out-of-memory exception to the application, which is not desirable. Table 9.5 shows an example for the time required for CUDA Driver API calls using MATOG and the reference code for the KD-Tree on a GTX 1080.

9.5 Conclusion

Overall we can summarize that the goals that we have defined in Chapter 1 have been fulfilled. First, we were able to build a tool that can optimize array access in CUDA applications independent of the used hardware and the application domain. We showed this by our evaluation on 14 GPUs from four different hardware generations and seven applications, from various application domains (image processing, bio informatics, real-time rendering and simulation). Second, our application analysis achieves results comparable to an exhaustive search, in significantly less

	CUDA Reference			MATOG		
Function Name	**Time (µs)**	**Time (%)**	**Calls**	**Time (µs)**	**Time (%)**	**Calls**
cuMemcpyHtoD	795,830.0	51.39%	134	413,150.0	29.96%	133
cuCtxCreate	436,130.0	28.16%	1	422,030.0	30.61%	1
cuModuleUnload		**0.00%**		**194,440.0**	**14.10%**	**55**
cuCtxSynchronize	238,630.0	15.41%	138	183,360.0	13.30%	138
cuMemAlloc	46,455.0	3.00%	139	41,441.0	3.01%	209
cuModuleLoad(Data)	**1,304.7**	**0.08%**	**8**	**88,059.0**	**6.39%**	**105**
cuMemFree	17,190.0	1.11%	139	18,269.0	1.32%	209
cuLaunchKernel	6,550.1	0.42%	542	8,647.7	0.63%	542
cuDeviceGetAttribute	3,311.9	0.21%	365	3,352.4	0.24%	409
cuDeviceTotalMem	928.7	0.06%	4	1,875.8	0.14%	8
cuMemcpyDtoH	1,581.2	0.10%	69	1,600.0	0.12%	69
cuDeviceGetName	284.6	0.02%	4	592.9	0.04%	8
cuCtxGetCurrent		0.00%		823.8	0.06%	2671
cudaDeviceReset	506.1	0.03%	1	512.9	0.04%	1
cuFuncSetCacheConfig		0.00%		492.3	0.04%	542
cuCtxGetDevice		0.00%		180.0	0.01%	542
cuModuleGetFunction	**3.4**	**0.00%**	**8**	**106.8**	**0.01%**	**105**
cuDeviceGet	5.0	0.00%	13	5.9	0.00%	13
cuDeviceGetCount	2.2	0.00%	3	3.4	0.00%	4
cuInit	0.9	0.00%	1	1.0	0.00%	1
cuDriverGetVersion	0.7	0.00%	1	0.5	0.00%	1
Total Driver API	**1,548,714.4**			**1,378,944.3**		

Table 9.5: Time required for CUDA Driver API calls for running the KD-Tree benchmark on a GTX 1080 with the reference and optimized MATOG variant. The highlighted calls (`cuModuleLoadData`, `cuModuleUnload` and `cuModuleGetFunction`) require significantly more time in the MATOG variant than in the reference code.

time. The optimization process is fully automated and does not require any user interaction. Finally, our tool is capable of reaching performance comparable to hand-optimized code, or even outperform it. Further, it can dynamically react onto changing application workloads to achieve even higher performance. These can be higher than for purely static applications. In Chapter 8 we have analyzed how automatic performance models could be generated using the data that we have available in MATOG. Our results show that in the most cases these already work as expected, but before they can be used in MATOG the method has to be improved. Currently, in some rare situations, the performance models do not work as expected.

Chapter 10
Future Work

It is easy to answer the question *"Is auto-tuning at its end?"*, as it is definitively not. More so, it is at its beginning! There are many open research topics, which are presented in the following sections.

10.1 Future of MATOG

As already mentioned in Section 9.4, the implementation itself could be improved to achieve higher performance. First, by changing the data-structures, so that no pointer aliasing occurs. Second, reducing the overhead of the module (un-)loading by reducing the number of configurations that are used and finally by extending to other compute device types (e.g., CPUs or FPGAs). Another big topic in compute intensive applications are sparse matrices. They are very data dependent and can greatly benefit from auto-tuning [Muralidharan et al. 2014]. Therefore, it could be beneficial not only to integrate sparse matrix data structures in MATOG, but also an automatic detection whether a dense or sparse matrix should be used. Besides optimized memory access, it would also be possible to choose between different compilers. For NVIDIA GPUs not only NVIDIA's own NVCC compiler exists, but also an alternative, called GPUCC. Wu et al. [2016] have shown that their compiler can outperform NVCC in some applications, therefore it would be convenient to have a mechanism that further detects, which of these compilers works better for the given kernel. Thinking further, also adding an MPI layer into MATOG would be interesting, as especially in HPC performance is of the essence. This would allow to use multiple cluster nodes for the calculation, while MATOG would take care of the data layouts. Further, it could automatically partition the data, similar to MAPS [Ben-Nun et al. 2015]. However, in its current implementation it is not possible to run MATOG in a distributed environment, as its database system is based on SQLite[1], which is not designed to be operated in an network environment. One option would be to use one instance as master for accessing the database, or to entirely switch to a network capable database (e.g., MySQL[2] or MariaDB[3]) or even an entirely distributed database (e.g., Apache Cassandra[4]). Another very

[1]sqlite.org
[2]mysql.org
[3]mariadb.org
[4]cassandra.apache.org

interesting optimization is kernel fusion/splitting. This deals with the question, when it makes sense to put multiple different processing steps into the same kernel or to split these in different kernels. More functionality in the same kernel allows to use shared memory to store intermediate results and access these faster. This usually consumes more resources, which can decrease the utilization of the kernel. Furthermore, necessary synchronization between the different processing parts can decrease the performance. Splitting a kernel can reduce the resource consumption, but does not allow to share data using shared memory between two kernels. To our knowledge, some work has been conducted into this direction [Wang et al. 2010; Fousek et al. 2011; Filipovic et al. 2012; Filipovic et al. 2015] but there is no fully automatic tool for arbitrary complex kernels so far, only for some specialized applications. As laid out in Chapter 8, the usage of performance models could improve the decision-making during runtime.

10.2 Evaluation and comparability

One big problem of auto-tuning is the evaluation and comparability. Nearly no auto-tuning project publishes its code. We would have liked to compare MATOG against other auto-tuners [Sung et al. 2012; Kofler et al. 2015; Peng et al. 2016] in our evaluation, but as their projects are not publicly available, this was not possible. We want to encourage researchers to publish their work as open source as we did for MATOG from the beginning. Another option is to provide a web platform, as e.g., the 3D Web Reconstruction project[5] or the DawnCC project[6] that allow to use their tools without giving away the code. If such a web-based approach is feasible for auto-tuning has to be evaluated.

Another issue is, that there is no default way or application to evaluate auto-tuners and given the variety of possible optimizations and methods, this is most likely difficult to establish in the community. This causes that some authors compare their auto-tuner to hand-optimized code and *"only"* achieve a speed up of a few percents, while others compare against *"naïve"* solutions (which are most likely the worst performing they were able to find) and achieve speed ups of several orders of magnitude. This is equal to the *"GPUs are 100x faster than CPUs"*-myth, where an optimized GPU implementation is compared against an unoptimized CPU version [Lee et al. 2010]. Neither variant can be blamed but this variety does not help to compare the different approaches. Further, often the used hardware and software stack (drivers, CUDA version, operating system, ...) are omitted, which make it even more uncomparable. There are ambitions in the community to change this using benchmark suites such as Rodinia [Che et al. 2009], Parboil

[5] gccvmwebreconstruction.igd.fraunhofer.de
[6] cuda.dcc.ufmg.br/dawn

[Stratton et al. 2012] or Polybench [Grauer-Gray et al. 2012] and building upon this the project of Fursin et al. [2016]. However, yet none of these approaches provides a satisfying solution, as we will lay out in the following sections.

10.2.1 Benchmark Suites

The mentioned benchmark suites usually consist of too simplistic applications, as e.g., SRAD or Hotspot from our evaluation, whose runtime is in a millisecond range. To show the ability of auto-tuners, in our opinion it is necessary to test them on realistic applications with runtime of seconds, minutes or even hours. Usually most of the time is used for I/O or setup of the application and only a very small fraction of the already short execution time is used for the actual computation. Further, these benchmarks do not come with predefined tests, so it is impossible to recreate the same results and no author can be blamed to choose datasets and parameters that work good with his tool. Another issue is the maintainability of these benchmarks. They often come with no common build chain, but every single application has its own way to be build and usually requires some adaption to work on other machines. Tools such as CMake[7] could be a solution for this, as this would even allow to use the benchmarks on multiple platforms such as Windows, Linux and MacOS, without changes (if the source code does not use platform dependent functionality). Further, some of the benchmarks contain even programming errors or non-functioning code (Section 7.1). We only took a look on some of the benchmarks in the Rodinia suite but are certain that we would be able to find similar errors in the other benchmarks as well, which raises the question for quality and usability of these.

Better organized examples for benchmarks can be found in other communities, e.g., the *Common Visual Data Foundation* [CVDF 2016] that poses explicit challenges with precisely determined task descriptions that have to be completed and how the results are scored. The problem for auto-tuning in such challenges is most likely the scoring, as different hardware (even equal GPUs from different manufacturers can vary in performance, caused by customizations such as varying clock frequencies or modified cooling systems) could yield in different scoring. Another example is the Middlebury benchmark [Baker et al. 2011] which explicitly contains a training and an evaluation dataset and even allows to submit results to the author's homepage.

[7]cmake.org

10.2.2 Collective Knowledge

The project of Fursin et al. [2016] (cTuning Foundation) goes into the right direction, as they provide a set of predefined benchmarks, datasets and a repository to store the benchmark results. However, one major problem is the comparability of different hardware and software setups. Everyone is using different hardware, drivers and operating systems and even small changes (e.g., a newer GPU driver version) can result in a change of performance, making it difficult to compare the results. They also store information about the test setup, but this does not solve the comparability problem. Metrics as FLOPS are also no good alternative, as higher FLOPS not necessarily guarantee shorter execution times. We do not have a solution for this. One idea could be a community driven centralized evaluation cluster, where researchers can upload their code and evaluate it on a standardized software/hardware stack with a predefined set of benchmarks, parameter and datasets. This would enable comparability across different approaches, but setting up and maintaining such an infrastructure is a very costly endeavor.

10.3 How to combine different optimizations?

Today there is a wide variety of auto-tuners available for all kinds of optimizations such as data layouts (MATOG), data partitioning and multi-device scheduling (MAPS [Ben-Nun et al. 2015]), sparse matrix formats (Nitro [Muralidharan et al. 2014]), and many more. But all of these only focus on a particular optimization field and provide analysis and optimization methods that are suitable for exactly their solution, but how to combine them? When an auto-tuner has the option to choose a different algorithm, it is most likely that it has to reevaluate, which data layouts could be optimal for the different algorithms, but what about data partitioning and data layouts? Do they exclude each other? Are they orthogonal to each other? Do layouts have an influence on partitioning at all? To our knowledge there has been no research conducted into this direction so far.

In the case that different optimizations do not allow to draw any conclusions on each other it would be very difficult for empirical profiling, as this would significantly increase the number of configurations that actually have to be evaluated. In this case a more analytical or hybrid method could be the right choice. However, as already mentioned, this could again have difficulties with unknown future hardware and unknown internal implementation of proprietary hardware. One idea for research could be: *"How to design hardware that has a predictable performance, which can either be done entirely analytically or in a hybrid way?"* Even if this hardware would be slower than an unpredictable one, the option to optimize software optimally to the hardware without *"guessing"* and *"magic witchcraft"* of compilers or auto-tuners could in end be as fast or even faster.

10.4 Performance models, continuous monitoring and adaptive reoptimization

MATOG and many other auto-tuners rely on profiling the application with a reference dataset, which is supposed to represent a realistic workload. However, many users will most likely use a rather small dataset as they do not want to wait hours for the auto-tuner to optimize the application. This can – depending on the application – lead to suboptimal optimization results. A better approach would be to use the current techniques to establish an initial performance model and then continuously monitor the performance of the application during runtime. With performance models (based on the initial profiling) it would be possible to check whether the prediction differs from the monitoring results. The models then could be continuously improved in parallel to the application runtime, on realistic workloads. Depending on the applied optimizations and if it is possible to draw conclusions from running one configuration onto others (as it is possible with the prediction that MATOG is build upon), the auto-tuner could employ better decisions. Further, it could be constantly improved during runtime, even with changing workloads. In Chapter 8 we showed initial experiments for such a system, based on the data available in MATOG.

10.5 Incompatibility of Auto-Tuners

Another unresolved problem is that most auto-tuners are incompatible to each other, so that e.g., using MAPS [Ben-Nun et al. 2015] and MATOG together would not work, as MATOG data structures cannot be used together with the data partitioning of MAPS. The same applies to libraries, as MATOG can be used together with THRUST data structures in the same kernel, but MATOG data structures cannot be used in THRUST functions. This problem already starts with the array layouts, as partitioning an AoS is quite simple and can be done by a simple memcopy, whereas for SoA or even AoSoA the data itself has to be explicitly unweaved into different data segments.

However, the problem of incompatibility is rooted even lower at the hardware level. With increasing number of different compute platforms (CPUs, GPUs, FPGAs, ...) the number of languages, dialects and extensions to program these increases significantly. Approaches such as OpenCL tried to put all of these under the same roof, but actually failed as the support of vendors is more and more reduced. NVIDIA only supported OpenCL until v1.2 [NVIDIA 2015] and Intel no longer supports it for the Xeon Phi Knights Landing. Even AMD seems no longer to believe in it, as they released their *Heterogeneous-Compute Interface for Portability **(HIP)*** [AMD 2016] which is very similar to CUDA and even provides a CUDA to HIP converter.

The reason for this is most likely that OpenCL is too low level and requires to be optimized specifically towards the underlying hardware (as it is necessary with every other language as well), so the dream to write code once and execute it on all platforms is not fulfilled. On the contrary, it does not only require to write different kernels for all kind of devices, but also does not allow to use specialized features the hardware does provide (e.g., the shuffle operation in CUDA).

The problem of many new programming languages today is that they do not live very long. Every year new languages appear, with new fancy features and disappear some years later because someone came up with a new, even more "*hipster*" language. However, the only constant languages for years have been C/C++ and Java [TIOBE 2016]. But these require the software to be specifically tuned towards the hardware, including a very high maintenance overhead to keep long living software still be functional and performing on today's and future hardware. This misses the actual goal of code portability and maintainability. In order to establish a successor for OpenCL, it would be necessary to establish an entire system stack rather than only a programming language. The focus of such a system should explicitly be performance portability and maintainability. An idea for such a system is discussed in the next section.

10.6 Performance Portability Aware Software Stack

In this section we discuss the idea of a performance portability aware software stack. The main concept should be that the algorithm and implementation are separated. This could be achieved when the application only serves as a control-server that schedules tasks and receives results. The underlying automatic task scheduler either controls a single workstation with CPUs and accelerator cards or it controls an entire cluster and offloads the tasks onto various heterogeneous compute nodes. This should be transparent to the program and could use a vendor specific implementation similar to today's MPI implementations. There are several existing examples for such scheduling systems also for heterogeneous hardware today, as shown in Section 4.5. However, they do not really provide performance portability today, as they still rely on a tight coupling of algorithm and implementation. Figure 10.1 shows an overview of the system that we are proposing.

The key idea of our programming model is to split the program code into a high-level algorithmic and a low-level implementation part, because algorithms in general are portable but implementations are device specific. The algorithmic implementation has to be high-level so that the programmer can concentrate on the algorithm rather than on hardware specific functionality or limitations.

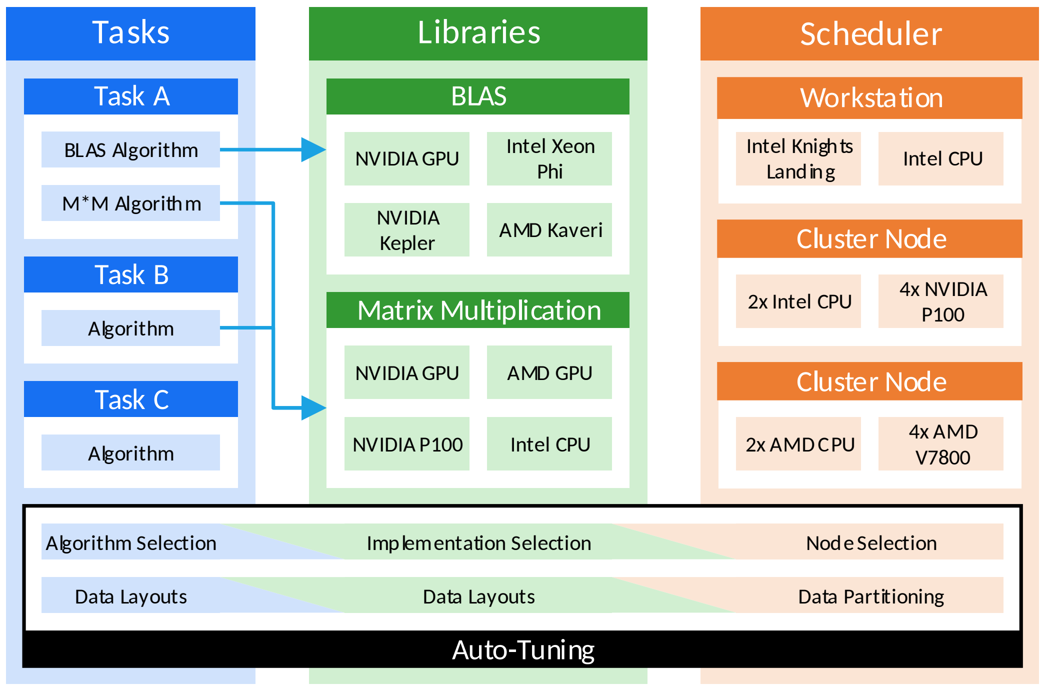


Figure 10.1: Schematic illustration of our proposed system. In the user-level (blue) a programmer can define his tasks and rely their functionality on libraries that provide highly optimized implementations for commonly used functions. An scheduler then takes control of task and data transfers in the system, which can be a workstation or even an entire cluster. As no application is similar to another as well as every workstation or cluster differs in its used hardware and interconnect, an auto-tuning layer should take care of selecting optimal parameters for the code, as well as the selection of hardware it is supposed to run on.

The low-level implementation part can use hardware specific functionality and should be provided by experienced programmers. This is complemented by an auto-tuning approach that chooses the actual implementation and processing unit on which the algorithm is executed in each specific instance. As auto-tuning is also always depending on the underlying hardware, it could be designed in a driver fashion, similar to operating systems, as these usually extend predefined APIs to plug into the system, but their actual operation is hidden inside the driver. The same way also different auto-tuning optimizations could be plugged into the system, depending on the demands of the application or hardware.

Algorithms should be programmed in capsuled functions (in the following called *tasks*), in any high-level language (e.g. in C#) but cannot access any low-level or vendor specific functionality. Instead parallelization features as known from

OpenMP and *OpenACC* or commonly used parallel functions as in *CUB* [NVIDIA 2013] or Intel's *Threading Building Blocks* ***(TBB)*** [INTEL 2016] should be available. As different processor types are better suited for certain algorithms, the user should able to provide multiple implementations of his algorithm, so that a suitable algorithm for a platform can be chosen automatically. For example, the user could provide an exact and a Monte Carlo based solving technique. However, code performance always benefits if it is specifically designed for a given hardware and when it uses vendor specific functionality. For these libraries with an API interface (e.g., BLAS) that can be used inside the tasks should be available. The actual implementation of the libraries can be specifically assigned to a hardware type (e.g., NVIDIA GPUs), a hardware generation (e.g., NVIDIA Pascal GPUs) or even a specific model (e.g., NVIDIA GTX 1080) and can be provided in a vendor specific language, e.g., CUDA, OpenCL or HIP. Again, multiple implementations are possible, which then could be chosen by an auto-tuner. The implementations of libraries are meant for experts and hardware enthusiasts only. However, novice users still would be able to use such a systems as basic functionality such as BLAS or matrix-multiplications can easily be provided by vendors, as they are already available today.

Additionally to the separation of algorithm and low-level implementation, the auto-tuning should dynamically adjust performance critical parameters at runtime. Optimizations should include algorithm selection, data layouts, data partitioning, device selection/scheduling, message passing, and a dynamic allocation of compute nodes to fit the needs of the current compute state, so that clusters can be used more efficiently.

Of course, establishing such a system requires a lot of work and definitively cannot be done by a single person or a small research group alone. It most likely requires support from the industry and community. Therefore, it would already be a great contribution if a concrete concept and suitable API would exist, equal to the MPI standard.

Appendix A
Benchmark Training-/Testing-Data

This appendix contains information about the training and testing datasets and parameters used in our evaluation.

A.1 Bitonic Sort

The datasets for the Bitonic Sort contain varying random, ascending or descending values, ranging from 0 to 1023 (255 for the 1 B field). **Format:** dataset name (number of items).

Training:

- dddd.dat (131.0 k)
- rddd.dat (65.5 k)

Testing:

- adad.dat (2.1 M)
- raaa.dat (262.1 k)
- dada.dat (524.3 k)
- aaaa.dat (1.0 M)
- rrrr.dat (4.2 M)

A.2 SRAD

For SRAD we used three different speckle parameter sets that we executed with the given sizes. For training purposes we only executed one iteration, while 100 for testing. We further varied the grid size, as shown below. The speckle values are:

- $X = (0, 127), Y = (0, 127), \lambda = 0.2$
- $X = (253, 213), Y = (32, 74), \lambda = 1.0$
- $X = (0, 53), Y = (74, 222), \lambda = 0.6$

Training:

- 512x512

Testing:

- 128x128
- 256x256
- 1024x1024
- 2056x2056

A.3 Hotspot

The Hotspot benchmark is only provided with three datasets.

Training:

- power_512

Testing:

- power_64
- power_1024

A.4 DPID

For this benchmark we used the video *"Fuerteventura 4K - A Timelapse Adventure*[1]". We want to thank S. Schall and J. Schmid for letting us use their video. The video has an input resolution of 3840x2178. For training we downscaled only one frame, while 100 frames for testing. Further, we changed the output resolution to:

Training:

- 1920x1123
- 768x436

Testing:

- 2048x1162
- 640x363
- 320x182

[1]youtu.be/40s_HSZkt3U

A.5 COMIC

For COMIC we used a series of different datasets / with varying counts of sequences and sequence lenghts. **Format:** dataset name (count of sequences / length of sequences).

Training:

- PF00520_mod (2,261 / 238)
- L4 (390 / 244)

Testing:

- alnComplete_A2Seqs1 (140 / 71)
- Calmodulin_MSA_clustalw (753 / 264)
- hcn (211 / 570)
- aln254718 (211 / 465)
- Calmodulin_MSA_muscle (753 / 275)
- PF07885_mod (4,204 / 120)
- ProtShort1_Selection (500 / 99)
- ProtShort1 (16,000 / 99)
- S2 (376 / 222)
- PF01007_mod (616 / 336)

A.6 REYES

All training runs have rendered two frames / while all testing runs rendered 100 frames. **Format:** dataset name (render resolution / patch counts).

Training:

- Utah Teapot (1920x1080 / 32)
- Utah Teapot (640x480 / 32)

Testing:

- Aphroidite (1920x1080 / 4,004)
- Bike (640x480 / 5,216)
- Cube (1024x786 / 6)
- Gumbo (720x480 / 128)
- Motor Bike (1440x960 / 826)
- Plato (1920x1200 / 224)
- Square (320x200 / 1)
- Utah Teapot (1280x854 / 32)

A.7 KD-Tree

All runs have been performed with 32 bins. **Format:** dataset name (number of triangles / type of scene).

Training:

- Happy Buddha (1,087,716 / 3D-Scan)

Testing:

- Bunny (69,451 / 3D-Scan)
- Conference Room (282,755 / artistic scene)
- Crytek Sponza (262,267 / artistic scene)
- Dabrovic Sponza (66,450 / artistic scene)
- Kitchen (425,504 / artistic scene)
- Mustang (787,668 / artistic scene)
- Sibenik (75,284 / artistic scene)
- San Miguel[2] (10,500,482 / artistic scene)
- Powerplant[3] (12,759,246 / artistic scene)

[2]only on GTX 1080 and Tesla K20c
[3]only on GTX 1080

Acronyms

ADG	Array Dependency Graph
AoS	Array of Structs
AoSoA	Array of Structure of Arrays
API	Application Programming Interface
APU	Accelerated Processing Unit
ASIC	Application Specific Integrated Circuits
ASTA	Array-of-Structure-of-Tiled-Arrays
AVX	Advanced Vector Extensions
Bit/s	Bits per second
BLAS	Basic Linear Algebra Subprograms
CC	Compute Capabilities
COMIC	Coevolution via MI on CUDA
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUPTI	CUDA Profiling Tools Interface
DDG	Decision Dependency Graph
DDR	Double Data Rate
DFT	Discrete Fourier Transformation
DM	Directional Model
DNA	Deoxyribonucleic acid
DPID	Detail-Preserving Image Downscaling
DRAM	Dynamic RAM
DSP	Digital Signal Processing
EE	exhaustive profiling/exhaustive analysis
EHP	Exascale Heterogeneous Processor
EP	exhaustive profiling/predictive analysis
ELF	Earliest Load First
FeRAM	Ferrorelectric RAM
FLOPS	Floating Point Opterations Per Second
FPGA	Field Programmable Gate Array
GDDR	Graphics DDR
GEMM	Dense Matrix-Matrix Multiplication
GEMV	Dense Matrix-Vector Multiplication
GP	Gaussian Processes
GPML	Gaussian Processes for Machine Learning

GP lin.	linear
GP lin. + SE	combined linear plus squared exponential
GP SE	squared exponential
GP SE + ARD	squared exponential with automatic relevance determination
GPU	Graphics Processing Unit
H2C	Heterogeneous Habanero-C
HBM	High Bandwidth Memory
HDD	Hard Drive Disk
HIP	Heterogeneous-Compute Interface for Portability
HMPP	Hybrid Multicore Parallel Programming
HPC	High Performance Computing
HPL	Heterogeneous Programming Library
I/O	input/output
ILP	Instruction Level Parallelism
IPS	Instructions per Second
ISPC	Intel Single-Program Multiple-Data Program Compiler
JSON	JavaScript Object Notation
lin. reg.	linear regression
LRU	least recently used
MATOG	“MATOG: Auto-Tuning on GPUs”
MIC	Many Integrated Core
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MMX	Multi Media Extension
MPI	Message Passing Interface
MRAM	Magnetoresitive RAM
NN	Neural Network
NVRAM	Non-volatile RAM
PCIe	Peripheral Component Interconnect Express
PCRAM	Phase-change RAM
PP	predictive profiling/predictive analysis
PTF	Periscope Tuning Framework
PTX	Parallel Thread eXecution architecture
PU	Processing Unit
QPI	QuickPath Interconnect
RAM	Random Access Memory
READEX	Runtime Exploitation of Application Dynamism for Energy-efficient eXascale computing
REYES	Renders Everything You Ever Saw
RRMSE	Relative Root Mean Squared Error
SATA	Serial AT Attachment

SD-RAM	Synchronous Dynamic RAM
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SISD	Single Instruction, Single Data
SM	Streaming Multi-Processor
SMT	Simultaneous Multithreading
SoA	Structure of Arrays
SoAoS	Structure of Array of Structures
SpMV	Sparse Matrix-Vector Multiplication
SRAD	Speckle Reducing Anisotropic Diffusion
SRAM	Static RAM
SSD	Solid State Disk
SVM	Support Vector Machine
TBB	Threading Building Blocks
TPU	Tensor Processing Unit
VLIW	Very Long Instruction Word

Bibliography

[7-CPU 2016] 7-CPU (2016). *Intel Skylake - Intel i7-6700*. www.7-cpu.com/cpu/Skylake.html [accessed 07.04.2017] (cit. on p. 15).

[AMD 2000] AMD (2000). *3DNow! Technology Manual*. support.amd.com/TechDocs/21928.pdf [accessed 07.04.2017] (cit. on p. 17).

[AMD 2015] AMD (2015). *High-Bandwidth Memory (HBM) - Reinventing Memory Technology*. www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf [accessed 07.04.2017] (cit. on p. 21).

[AMD 2016] AMD (2016). *It's HIP to be Open - Convert your CUDA Code to C++ Using AMD's New HIP Tool*. www.amd.com/Documents/HIP-Datasheet.pdf [accessed 07.04.2017] (cit. on p. 121).

[Afonso et al. 2016] Afonso, S., A. Acosta, and F. Almeida (2016). "Automatic Generation of OpenCL Code for ARM Architectures". In: *Proc. Euro-Par* (cit. on p. 44).

[Agakov et al. 2006] Agakov, F., E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams (2006). "Using Machine Learning to Focus Iterative Optimization". In: *Proc. CGO* (cit. on p. 44).

[Ahmed and Schuegraf 2011] Ahmed, K. and K. Schuegraf (2011). "Transistor Wars - Rival architectures face off in a bid to keep Moore's Law alive". In: *IEEE Spectrum*. spectrum.ieee.org/semiconductors/devices/transistor-wars [accessed 07.04.2017] (cit. on p. 17).

[Ainsworth and Jones 2017] Ainsworth, S. and T. M. Jones (2017). "Software Prefetching for Indirect Memory Accesses". In: *Proc. CGO* (cit. on p. 51).

[Amend 2017] Amend, S. C. (2017). "Predicting Execution Time of GPU Kernels using automatic Performance Models". In: *TU Darmstadt, Master Thesis* (cit. on pp. 5, 6, 95).

[Ansel 2014] Ansel, J. (2014). "Autotuning Programs with Algorithmic Choice". PhD thesis. MIT (cit. on pp. 39, 41, 75).

[Ansel et al. 2009] Ansel, J., C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe (2009). "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proc. PLDI* (cit. on p. 45).

[Ansel et al. 2011] Ansel, J., M. Pacula, S. Amarasinghe, and U.-M. O'Reilly (2011). "An Efficient Evolutionary Algorithm for Solving Incrementally Structured Problems". In: *Proc. GECCO* (cit. on p. 45).

[Ansel et al. 2014] Ansel, J., S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe (2014). "OpenTuner: An Extensible Framework for Program Autotuning". In: *Proc. PACT* (cit. on p. 48).

[Armstrong 2016] Armstrong, A. (2016). "Samsung 960 EVO M.2 NVMe SSD Review". In: *StorageReview.com*. www.storagereview.com/samsung_960_evo_m2_nvme_ssd_review [accessed 07.04.2017] (cit. on p. 15).

[Arslan et al. 2016] Arslan, E., K. Guner, and T. Kosar (2016). "HARP: Predictive Transfer Optimization Based on Historical Analysis and Real-time Probing". In: *Proc. SC* (cit. on p. 50).

[Babokin and Brodman 2016] Babokin, D. and J. Brodman (2016). *Intel SPMD Program Compiler - An open-souce compiler for high-performance SIMD programming on the CPU*. ispc.github.io/ [accessed 07.04.2017] (cit. on p. 18).

[Baghsorkhi et al. 2010] Baghsorkhi, S. S., M. Delahaye, S. J. Patel, W. D. Gropp, and W. mei W. Hwu (2010). "An Adaptive Performance Modeling Tool for GPU Architectures". In: *Proc. PPoPP* (cit. on p. 43).

[Baghsorkhi et al. 2012] Baghsorkhi, S. S., I. Gelado, M. Delahaye, and W. mei W. Hwu (2012). "Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors". In: *Proc. PPoPP* (cit. on p. 43).

[Bajrovic and Benkner 2014] Bajrovic, E. and S. Benkner (2014). "Automatic Performance Tuning of Pipeline Patterns for Heterogeneous Parallel Architectures". In: *Proc. PDPTA* (cit. on p. 50).

[Bajrovic et al. 2013] Bajrovic, E., S. Benkner, J. Dokulil, and M. Sandrieser (2013). "Autotuning of Pattern Runtimes for Accelerated Parallel Systems". In: *CSE* (cit. on p. 47).

[Baker et al. 2011] Baker, S., D. Scharstein, J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski (2011). “A Database and Evaluation Methodology for Optical Flow”. In: *IJCV* (cit. on p. 119).

[Bakhoda et al. 2009] Bakhoda, A., G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt (2009). “Analyzing CUDA Workloads Using a Detailed GPU Simulator”. In: *Proc. ISPASS* (cit. on p. 43).

[Bao et al. 2016] Bao, W., C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan (2016). “Static and Dynamic Frequency Scaling on Multicore CPUs”. In: *ACM TACO* (cit. on p. 47).

[Barman et al. 2011] Barman, S., R. Bodik, S. Jain, Y. Pu, S. Srivastava, and N. Tung (2011). “Parallel Programming with Inductive Synthesis”. In: *Proc. HotPar* (cit. on p. 45).

[Batcher 1968] Batcher, K. E. (1968). “Sorting Networks and Their Applications”. In: *Proc. SJCC* (cit. on p. 76).

[Bauer 2014] Bauer, M. (2014). “Legion: Programming Distributed Heterogeneous Architectures with Logical Regions”. PhD thesis. Stanford University (cit. on p. 50).

[Beaumont et al. 2016] Beaumont, O., T. Cojean, L. Eyraud-Dubois, A. Guermouche, and S. Kumar (2016). “Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources”. In: *Proc. HiPC* (cit. on p. 46).

[Bell and Garland 2008] Bell, N. and M. Garland (2008). *Efficient Sparse Matrix-Vector Multiplication on CUDA*. Tech. rep. NVIDIA (cit. on p. 46).

[Ben-Nun et al. 2015] Ben-Nun, T., E. Levy, A. Barak, and E. Rubin (2015). “Memory Access Patterns The Missing Piece of the Multi-GPU Puzzle”. In: *Proc. SC* (cit. on pp. 39, 50, 117, 120, 121).

[Bergstra et al. 2012] Bergstra, J., N. Pinto, and D. Cox (2012). “Machine Learning for Predictive Auto-Tuning with Boosted Regression Trees”. In: *Proc. InPar* (cit. on pp. 39, 42).

[Berkeley 2014] Berkeley (2014). *Moore’s Law and Computer Processing Power*. Tech. rep. datascience.berkeley.edu/moores-law-processing-power/ [accessed 07.04.2017]. Berkeley - University of California (cit. on pp. 1, 16).

[Bhat et al. 2006] Bhat, V., M. Parashar, H. Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed (2006). “Enabling Self-Managing Applications using Model-based Online Control Strategies”. In: *Proc. ICAC* (cit. on p. 48).

[Bianchin et al. 2008] Bianchin, S., P. Achenbach, S. Ajimura, O. Borodina, T. Fukuda, J. Hoffmann, M. Kavatsyuk, K. Koch, T. Koike, N.Kurz, F. Maas, S. Minami, Y. Mizoi, T. Nagae, D. Nakajima, A. Okamura, W. Ott, B. Özel, J. Pochodzalla, C. Rappold, T. R. Saito, A. Sakuguchi, M. Sako, M. Sekimoto, H. Suhimura, T. Takahashi, H. Tamura, K. Tanida, and W. Trautmann (2008). “The HyPHI Project: Hypernuclear Spectroscopy with Stable Heavy Ion Beams and Rare Isotope Beams at GSI and FAIR”. In: *ArXiv* (cit. on p. 47).

[Bischof et al. 2012] Bischof, C., D. an Mey, and C. Iwainsky (2012). “Brainware for green HPC”. In: *Computer Science - Research and Development* (cit. on pp. 1, 3).

[Bodin et al. 2016] Bodin, B., L. Nardi, P. H. J. Kelly, and M. F. P. O’Boyle (2016). “Diplomat: Mapping of multi-kernel applications using a static dataflow abstraction”. In: *Proc. MASCOTS* (cit. on p. 49).

[Bolchini et al. 2016] Bolchini, C., S. Cherubin, G. C. Durelli, S. Liutti, A. Miele, and M. D. Santambrogio (2016). “A Runtime Controller for OpenCL Applications on Heterogeneous System Architectures”. In: *Proc. ESWEEK* (cit. on p. 49).

[Bradski 2000] Bradski, G. (2000). “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (cit. on p. 78).

[Bruel et al. 2015] Bruel, P., M. A. Gonzalez, and A. Goldman (2015). “Autotuning GPU Compiler Parameters Using OpenTuner”. In: *Proc. SHPC* (cit. on p. 48).

[Buck et al. 2004] Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan (2004). “Brook for GPUs: Stream Computing on Graphics Hardware”. In: *Proc. SIGGRAPH* (cit. on p. 18).

[Buyya et al. 2009] Buyya, R., C. S. Yeo, S. Venugopal, J. Broberg, and I. Bradic (2009). “Cloud computing an emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”. In: *Future Generation Computer Systems* (cit. on p. 19).

[CRUCIAL 2015] CRUCIAL (2015). *Speed vs. Latency - Why CAS latency isn’t an accurate measure of memory performance*. www.crucial.com/usa/en/memory-performance-speed-latency [accessed 07.04.2017] (cit. on pp. 2, 15).

[CVDF 2016] CVDF (2016). *Common Visual Data Foundation*. www.cvdfoundation.org [accessed 07.04.2017] (cit. on p. 119).

[Calore et al. 2016] Calore, E., A. Gabbana, J. Kraus, S. F. Schifano, and R. Tripiccione (2016). "Performance and portability of accelerated lattice Boltzmann applications with OpenACC". In: *CCPE* (cit. on p. 44).

[Calotoiu et al. 2013] Calotoiu, A., T. Hoefler, M. Poke, and F. Wolf (2013). "Using automated performance modeling to find scalability bugs in complex codes". In: *Proc. SC* (cit. on p. 43).

[Calotoiu et al. 2016] Calotoiu, A., D. Beckingsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf (2016). "Fast Multi-Parameter Performance Modeling". In: *Proc. CLUSTER* (cit. on p. 43).

[Cantanzaro et al. 2014] Cantanzaro, B., A. Keller, and M. Garland (2014). "A Decomposition for In-place Matrix Transposition". In: *Proc. PPoPP* (cit. on p. 50).

[Catanzaro et al. 2010] Catanzaro, B., M. Garland, and K. Keutzer (2010). "Copperhead Compiling an Embedded Data Parallel Language". In: *Proc. PPoPP* (cit. on p. 45).

[Chan et al. 2009] Chan, C., J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman (2009). "Autotuning Multigrid with PetaBricks". In: *Proc. SC* (cit. on p. 45).

[Chang and Karamcheti 2001] Chang, F. and V. Karamcheti (2001). "A Framework for Automatic Adaptation of Tunable Distributed Applications". In: *Cluster Computing* (cit. on p. 48).

[Chang et al. 2016] Chang, L.-W., H.-S. Kim, and W. mei W. Hwu (2016). "DySel: Lightweight Dynamic Selection for Kernel-based Data-parallel Programming Model". In: *Proc. ASPLOS* (cit. on pp. 46, 50).

[Chase et al. 2008] Chase, J., B. Nelson, J. Bodily, and L. Dha-Jye (2008). "Real-Time Optical Flow Calculations on FPGA and GPU Architecture: A Comparison Study". In: *Proc. FPCCM* (cit. on p. 19).

[Che et al. 2009] Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron (2009). "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *Proc. IISWC* (cit. on pp. 75, 77, 78, 118).

[Cheng et al. 2017] Cheng, D., J. Rao, Y. Guo, C. Jiang, and X. Zhou (2017). “Improving Performance of Heterogeneous MapReduce Clusters with Adaptive Task Tuning”. In: *IEEE TPDS* (cit. on p. 49).

[Choi et al. 2010] Choi, J. W., A. Singh, and R. W. Vuduc (2010). “Model-driven autotuning of sparse matrix-vector multiply on GPUs”. In: *Proc. PPoPP* (cit. on pp. 39, 46).

[Christen et al. 2011] Christen, M., O. Schenk, and H. Burkhart (2011). “PATUS: A Code Generation and Auto-Tuning Framework For Parallel Stencil Computations”. In: *Proc. IPDPS* (cit. on p. 46).

[Chung and Hollingsworth 2004] Chung, I.-H. and J. K. Hollingsworth (2004). “Using Information from Prior Runs to Improve Automated Tuning Systems”. In: *Proc. SC* (cit. on pp. 41, 42, 48, 71).

[Collange et al. 2009] Collange, S., D. Defour, and D. Parello (2009). *Barra, a Modular Functional GPU Simulator for GPGPU*. Tech. rep. Univ. de Perpignan (cit. on p. 43).

[Cook 1997] Cook, D. (1997). “Performance Implications of Pointer Aliasing”. In: *SGI*. ftp.sgi.com/sgi/audio/audio.apps/dev/aliasing.html [accessed 07.04.2017] (cit. on p. 58).

[Cook et al. 1987] Cook, R. L., L. Carpenter, and E. Catmull (1987). “The Reyes Image Rendering Architecture”. In: *Proc. SIGGRAPH* (cit. on p. 79).

[Coplin and Burtscher 2015] Coplin, J. and M. Burtscher (2015). “Effects of Source-Code Optimizations on GPU Performance and Energy Consumption”. In: *Proc. GPGPU* (cit. on p. 47).

[Cruz et al. 2016] Cruz, E. H. M., M. Diener, L. L.Pilla, and P. O. A. Navaux (2016). “Hardware-Assisted Thread and Data Mapping in Hierarchical Multicore Architectures”. In: *ACM TACO* (cit. on p. 51).

[Datta et al. 2008] Datta, K., M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick (2008). “Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures”. In: *Proc. SC* (cit. on p. 46).

[Davidson et al. 2011] Davidson, A., Y. Zhang, and J. D. Owens (2011). “An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU”. In: *Proc. IPDPS* (cit. on p. 47).

[Devito et al. 2013] Devito, Z., J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek (2013). “Terra: A Multi-Stage Language for High-Performance Computing”. In: *Proc. PLDI* (cit. on p. 46).

[Dolbeau et al. 2007] Dolbeau, R., S. Bihan, and F. Bodin (2007). “HMPP: A Hybrid Multi-core Parallel Programming Environment”. In: *Proc. GPGPU* (cit. on p. 44).

[Dublish et al. 2016] Dublish, S., V. Nagarajan, and N. Topham (2016). “Cooperative Caching for GPUs”. In: *ACM TACO* (cit. on p. 51).

[Durillo and Fahringer 2015] Durillo, J. and T. Fahringer (2015). “From single- to multi-objective auto-tuning of programs: Advantages and implications”. In: *Scientific Programming* (cit. on p. 47).

[Edwards and Trott 2013] Edwards, H. C. and C. R. Trott (2013). “Kokkos: Enabling performance portablitity across manycore architectures”. In: *Proc. XSW* (cit. on pp. 39, 50).

[Elangovan et al. 2015] Elangovan, V. K., R. M. Badia, and E. Ayguadé (2015). “Auto-Tuning OmpSs-OpenCL Kernels Across GPU Machines”. In: *Proc. PARMA-DITAM* (cit. on p. 44).

[Enmyren and Kessler 2010] Enmyren, J. and C. W. Kessler (2010). “SkePU: a multi-backend skeleton programming library for multi-GPU systems”. In: *Proc. HLPP* (cit. on p. 50).

[Fabeiro et al. 2014] Fabeiro, J. F., D. Andrade, B. B. Fraguela, and R. Doallo (2014). “Writing self-adaptive codes for heterogeneous systems”. In: *Proc. Euro-Par* (cit. on p. 49).

[Fachada et al. 2016] Fachada, N., V. V. Lopes, R. Martins, and A. C. Rosa (2016). “cf4ocl: a C framework for OpenCL”. In: *ArXiv* (cit. on p. 50).

[Fatahalian et al. 2006] Fatahalian, K., T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan (2006). “Sequoia: Programming the Memory Hierarchy”. In: *Proc. SC* (cit. on p. 45).

[Filipovic et al. 2012] Filipovic, J., J. Fousek, and B. Lakomy (2012). “Automatically Optimized GPU Acceleration of Element Subroutines in Finite Element Method”. In: *Proc. SAAHPC* (cit. on p. 118).

[Filipovic et al. 2015] Filipovic, J., M. Madzin, J. Fousek, and L. Matyska (2015). “Optimizing CUDA code by kernel fusion: application on BLAS”. In: *SC* (cit. on p. 118).

[Flynn 1966] Flynn, M. J. (1966). “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* (cit. on p. 11).

[Fousek et al. 2011] Fousek, J., J. Filipovic, and M. Madzin (2011). “Automatic fusions of CUDA-GPU kernels for parallel map”. In: *ACM SIGARCH Computer Architecture News* (cit. on p. 118).

[Frigo 1999] Frigo, M. (1999). “A fast Fourier transform compiler”. In: *Proc. PLDI* (cit. on p. 47).

[Frigo and Johnson 1998] Frigo, M. and S. G. Johnson (1998). “FFTW: An adaptive software architecture for the FFT”. In: *Proc. CASS* (cit. on p. 47).

[Frigo and Johnson 2005] Frigo, M. and S. G. Johnson (2005). “The Design and Implementation of FFTW3”. In: *IEEE* (cit. on p. 47).

[Fursin et al. 2008] Fursin, G., C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O’Boyle (2008). “MILEPOST GCC: machine learning based research compiler”. In: *Proc. GCC* (cit. on p. 44).

[Fursin et al. 2016] Fursin, G., A. Lokhmotov, and E. Plowman (2016). “Collective Knowledge: Towards R&D sustainability”. In: *Proc. DATE* (cit. on pp. 119, 120).

[Gadioli et al. 2014] Gadioli, D., S. Libutti, G. Massari, E. Paone, M. Scandale, P. Bellasi, G. Palermo, V. Zaccaria, G. Agosta, W. Fornaciari, and C. Silvano (2014). “OpenCL Application Auto-Tuning and Run-Time Resource Management for Multi-Core Platforms”. In: *Proc. ISPA* (cit. on p. 49).

[Ganestam and Doggett 2012] Ganestam, P. and M. Doggett (2012). “Auto-tuning Interactive Ray Tracing using an Analytical GPU Architecture Model”. In: *Proc. GPGPU* (cit. on p. 47).

[Gao and Peterson 2015] Gao, S. and G. D. Peterson (2015). “Optimizing CUDA Shared Memory Usage”. In: *Proc. SC* (cit. on p. 49).

[Gasior 2014] Gasior, G. (2014). “The SSD Endurance Experiment: Two freaking petabytes”. In: *TechReport.com*. techreport.com/review/27436/the-ssd-endurance-experiment-two-freaking-petabytes [accessed 07.04.2017] (cit. on p. 14).

[Gaster and Howes 2011] Gaster, B. R. and L. Howes (2011). *The Future of the APU - Braided Parallelism*. AMD Fusion Developer Summit (cit. on p. 20).

[Gerndt 2016] Gerndt, M. (2016). “The READEX Project for Dynamic Energy Efficiency Tuning”. In: *Proc. SEM4HPC* (cit. on p. 47).

[Glatter 2015] Glatter, Z. (2015). “ATI 3D Rage”, “NVIDIA NV1” and “3dfx Voodoo”. In: *Vintage3D.org*. vintage3d.org/rage.php / vintage3d.org/nv1.php / vintage3d.org/3dfx1.php [accessed 07.04.2017] (cit. on p. 18).

[Goodacre 2011] Goodacre, J. (2011). “The evolution of the microprocessor - from single cputs to many core devices”. In: *NewElectronics*. www.newelectronics.co.uk/electronics-technology/the-evolution-of-the-microprocessor-from-single-cpus-to-many-core-devices/35556/ [accessed 07.04.2017] (cit. on p. 1).

[Götz et al. 2010] Götz, S., C. Wilke, M. Schmidt, S. Chech, and U. Assmann (2010). “Towards Energy Auto-Tuning”. In: *Proc. GREEN IT* (cit. on p. 47).

[Grasso et al. 2013] Grasso, I., K. Kofler, B. Cosenza, and T. Fahringer (2013). “Automatic problem size sensitive task partitioning on heterogeneous parallel systems”. In: *Proc. PPoPP* (cit. on p. 49).

[Grauer-Gray et al. 2012] Grauer-Gray, S., L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos (2012). “Auto-tuning a High-Level Language Targeted to GPU Codes”. In: *Proc. InPar* (cit. on pp. 44, 119).

[Gray and Stratford 2016] Gray, A. and K. Stratford (2016). “A Lightweight Approach to Performance Portability with targetDP”. In: *HPCA* (cit. on p. 50).

[Green 500 2016] Green 500 (2016). *November 2016*. www.top500.org/green500/lists/2016/11/ [accessed 07.04.2017] (cit. on p. 1).

[Guo and Wang 2010] Guo, P. and L. Wang (2010). “Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs”. In: *Proc. ICCIS* (cit. on p. 46).

[Guo et al. 2011] Guo, P., H. Huang, Q. Chen, L. Wang, E.-J. Lee, and P. Chen (2011). “A Model-Driven Partitioning and Auto-tuning Integrated Framework for Sparse Matrix-Vector Multiplication on GPUs”. In: *Proc. TeraGrid* (cit. on p. 46).

[Gupta 2016] Gupta, S. (2016). “IBM and NVIDIA present the NVLink server you’ve been waiting for”. In: *IBM Systems Blog*. www.ibm.com/blogs/systems/ibm-nvidia-present-nvlink-server-youve-waiting/ [accessed 07.04.2017] (cit. on p. 20).

[Gysi et al. 2016] Gysi, T., J. Baer, and T. Hoefler (2016). “dCUDA: hardware supported overlap of computation and communication”. In: *Proc. SC* (cit. on p. 50).

[Hall et al. 2009] Hall, M., J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan (2009). “Loop Transformation Receipes for Code Generation and Auto-Tuning”. In: *Proc. LCPC* (cit. on pp. 39, 45).

[Han and Abdelrahman 2009] Han, T. D. and T. S. Abdelrahman (2009). “HiCUDA A High-Level Directive-based Language for GPU Programming”. In: *Proc. GPGPU* (cit. on pp. 40, 45).

[Han and Abdelrahman 2011a] Han, T. D. and T. S. Abdelrahman (2011a). “Reducing Branch Divergence in GPU Programs”. In: *Proc. GPGPU* (cit. on p. 48).

[Han and Abdelrahman 2011b] Han, T. D. and T. S. Abdelrahman (2011b). “hiCUDA: High-Level GPGPU Programming”. In: *IEEE TPDS* (cit. on p. 45).

[Han and Abdelrahman 2013] Han, T. D. and T. S. Abdelrahman (2013). “Reducing Divergence in GPGPU Programs with Loop Merging”. In: *Proc. GPGPU* (cit. on p. 49).

[Han and Abdelrahman 2014] Han, T. D. and T. S. Abdelrahman (2014). “Automatic Tuning of Local Memory Use on GPGPUs”. In: *ArXiv* (cit. on p. 51).

[Hechtman et al. 2016] Hechtman, B. A., A. D. Hilton, and D. J. Sorin (2016). “TREES: A CPU-GPU Task-Parallel Runtime with Explicit Epoch Synchronization”. In: *ArXiv* (cit. on p. 50).

[Helal et al. 2016] Helal, A. E., P. Sathre, and W. chun Feng (2016). “MetaMorph: a library framework for interoperable kernels on multi- and many-core clusters”. In: *Proc. SC* (cit. on p. 50).

[Hoffmann et al. 2010] Hoffmann, H., J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal (2010). “Application Heartbeats A Generic Interface for Specifying Program Performance and Goals in”. In: *Proc. ICAC* (cit. on p. 47).

[Hoffmann et al. 2011] Hoffmann, H., S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard (2011). “Dynamic Knobs for Power-Aware Computing”. In: *Proc. ASPLOS* (cit. on p. 47).

[Hollingsworth and Keleher 1998] Hollingsworth, J. K. and P. J. Keleher (1998). “Prediction and Adaption in Active Harmony”. In: *Proc. HPDC* (cit. on pp. 38, 48).

[Hong et al. 2012] Hong, S., H. Chafi, E. Sedlar, and K. Olukotun (2012). “Green-Marl: A DSL for Easy and Efficient Graph Analysis”. In: *Proc. ASPLOS* (cit. on p. 46).

[Hong 2009] Hong, S. (2009). “An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness”. In: *Proc. ISCA* (cit. on pp. 40, 43).

[Hook and Graves 2016] Hook, C. and L. Graves (2016). “AMD introduces Radeon Instinct: Accelerating Machine Intelligence”. In: *AMD Press Releases*. www.amd.com/en-us/press-releases/Pages/radeon-instinct-2016dec12.aspx [accessed 07.04.2017] (cit. on p. 20).

[Hsu et al. 2014] Hsu, C.-C., C.-Y. Lin, S. K. Chen, C.-W. Liu, and J.-K. Lee (2014). “Optimized Memory Access Support for Data Layout Conversion on Heterogeneous Multi-core Systems”. In: *Proc. ESTIMedia* (cit. on pp. 39, 50).

[INTEL 1997] INTEL (1997). *Intel Architecture Software Developer’s Manual, Volume 1: Basic Architecture*. download.intel.com/design/intarch/manuals/24319001.PDF [accessed 07.04.2017] (cit. on p. 17).

[INTEL 2002] INTEL (2002). *Intel Hyper-Threading Technology - Get Faster Performance for Many Demanding Business Applications*. www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html [accessed 07.04.2017] (cit. on pp. 17, 25).

[INTEL 2005] INTEL (2005). *Intel Pentium D Processor 805, Specifications*. ark.intel.com/de/products/27511/ [accessed 07.04.2017] (cit. on p. 17).

[INTEL 2009] INTEL (2009). *An Introduction to the Intel QuickPath Interconnect*. www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf [accessed 07.04.2017] (cit. on p. 15).

[INTEL 2011] INTEL (2011). *Intel Core i7 2700K, Specifications*. ark.intel.com/de/products/61275/ [accessed 11.04.2017] (cit. on p. 2).

[INTEL 2013a] INTEL (2013a). *Intel Core i7 4765T, Specifications*. ark.intel.com/de/products/75121/ [accessed 11.04.2017] (cit. on p. 2).

[INTEL 2013b] INTEL (2013b). *Intel Xeon Phi Coprocessor Developer's Quick Start Guide*. software.intel.com/sites/default/files/managed/ee/4e/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf [accessed 07.04.2017] (cit. on p. 18).

[INTEL 2016] INTEL (2016). *Threading Building Blocks - Developer Guide*. software.intel.com/en-us/node/506045 [accessed 07.04.2017] (cit. on p. 124).

[Imani et al. 2017] Imani, M., D. Peroni, Y. Kim, A. Rhaimi, and T. Rosing (2017). "Efficient Neural Network Acceleration on GPGPU using Content Addressable Memory". In: *Proc. DATE* (cit. on p. 47).

[InfiniBand 2016] InfiniBand (2016). *InfiniBand Architecture Specification Volume 2, Release 1.3.1*. cw.infinibandta.org/document/dl/8125 [accessed 07.04.2017] (cit. on p. 19).

[Inggs et al. 2017] Inggs, G., D. B. Thomas, and W. Luk (2017). "A Domain Specific Approach to High Performance Heterogeneous Computing". In: *IEEE TPDS* (cit. on p. 50).

[Ipek et al. 2005] Ipek, E., B. R. de Supinski, M. Schulz, and S. A. McKee (2005). "An approach to performance prediction for parallel applications". In: *Proc. ECPP* (cit. on pp. 43, 96).

[Iwainsky et al. 2015] Iwainsky, C., S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf (2015). "How Many Threads will be too Many? On the Scalability of OpenMP Implementations". In: *Proc. EUROPAR* (cit. on p. 44).

[Jääskeläinen et al. 2014] Jääskeläinen, P., C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg (2014). "pocl: A Performance-Portable OpenCL Implementation". In: *Parallel Programming* (cit. on p. 49).

[Jia and Zhou 2016] Jia, Q. and H. Zhou (2016). “Tuning Stencil Codes in OpenCL for FPGAs”. In: *Proc. ICCD* (cit. on p. 46).

[Jordan et al. 2012] Jordan, H., P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch (2012). “A Multi-Objective Auto-Tuning Framework for Parallel Codes”. In: *Proc. SC* (cit. on p. 47).

[Jouppi 2016] Jouppi, N. (2016). “Google supercharges machine learning tasks with TPU custom chip”. In: *Google Cloud Platform Blog*. cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html [accessed 07.04.2017] (cit. on pp. 2, 19).

[Kamil et al. 2010] Kamil, S., C. Chan, L. Oliker, J. Shalf, and S. Williams (2010). “An Auto-Tuning Framework for Parallel Multicore Stencil Computations”. In: *Proc. IPDPS* (cit. on p. 46).

[Karsai et al. 2001] Karsai, G., A. Ledeczi, and J. Sztipanovits (2001). “An Approach to Self-Adaptive Software based on Supervisory Control”. In: *Self-Adaptive Software: Applications* (cit. on p. 38).

[Khan et al. 2013] Khan, M., P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame (2013). “A script-based autotuning compiler system to generate high-performance CUDA code”. In: *ACM TACO* (cit. on p. 46).

[Kim et al. 2016] Kim, J., Y.-J. Lee, J. Park, and J. Lee (2016). “Translating OpenMP Device Constructs to OpenCL using Unnecessary Data Transfer Elimination”. In: *Proc. SC* (cit. on p. 44).

[Klöckner et al. 2011] Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih (2011). “PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation”. In: *Parallel Computing* (cit. on p. 45).

[Kofler et al. 2015] Kofler, K., B. Cosenza, and T. Fahringer (2015). “Automatic Data Layout Optimizations for GPUs”. In: *Proc. Euro-Par* (cit. on pp. 23, 39, 40, 50, 75, 118).

[Kopf et al. 2013] Kopf, J., A. Shamir, and P. Peers (2013). “Content-Adaptive Image Downscaling”. In: *ACM TOG* (cit. on p. 78).

[Krajewski 1985] Krajewski, R. (1985). “Multiprocessing An Overview”. In: *BYTE Magazine* 5 (cit. on p. 17).

[Kurzak et al. 2012] Kurzak, J., S. Tomov, and J. Dongarra (2012). “Autotuning GEMM Kernels for the Fermi GPU”. In: *IEEE TPDS* (cit. on p. 46).

[Langdon et al. 2016] Langdon, W. B., B. Y. H. Lam, M. Modat, J. Petke, and M. Harman (2016). “Genetic Improvement of GPU Software”. In: *Genetic Programming and Evolvable Machines* (cit. on p. 45).

[Lashgar and Baniasadi 2016] Lashgar, A. and A. Baniasadi (2016). “OpenACC cache Directive: Opportunities and Optimizations”. In: *Proc. WACCPD* (cit. on p. 44).

[Lee et al. 2007] Lee, B. C., D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. a. McKee (2007). “Methods of inference and learning for performance modeling of parallel applications”. In: *Proc. PPPP* (cit. on pp. 43, 96).

[Lee and Vetter 2014] Lee, S. and J. S. Vetter (2014). “OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing”. In: *Proc. HPDC* (cit. on pp. 40, 44, 114).

[Lee et al. 2010] Lee, V. W., C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennpaty, P. Hammarlund, R. Singhal, and P. Dubey (2010). “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *Proc. ISCA* (cit. on p. 118).

[Levenson 2013] Levenson, M. D. (2013). “Lessons From Past Architecture Wars”. In: *Semiconductor Manufacturing and Design*. semiengineering.com/lessons-architecture-wars/ [accessed 07.04.2017] (cit. on p. 1).

[Li 2016] Li, A. (2016). “GPU Performance Modeling and Optimization”. PhD thesis. Technische Universiteit Eindhoven (cit. on p. 39).

[Li et al. 2015] Li, A., G.-J. van den Braak, A. Kumar, and H. Corporaal (2015). “Adaptive and Transparent Cache Bypassing for GPUs”. In: *Proc. SC* (cit. on pp. 39, 40, 49).

[Li et al. 2016a] Li, C., Y. Yang, Z. Lin, and H. Zhou (2016a). “Automatic Data Placement into GPU On-Chip memory resources”. In: *Proc. CGO* (cit. on p. 51).

[Li et al. 2016b] Li, C., Y. Yang, M. Feng, S. Chakradhar, and H. Zhou (2016b). “Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs”. In: *Proc. SC* (cit. on p. 47).

[Liu et al. 2014] Liu, W., I. A. C. Ureña, M. Gerndt, and B. Gong (2014). “Automatic MPI-I Tuning with the Periscope Tuning Framework”. In: *Proc. IPDPS* (cit. on p. 47).

[Liu et al. 2008] Liu, Y., E. Z. Zhang, and X. Shen (2008). *A Cross-Input Adaptive Framework for GPU Programs Optimization*. Tech. rep. College of William and Mary (cit. on pp. 39–41, 48).

[Long and Fursin 2005] Long, S. and G. Fursin (2005). “A heuristic search algorithm based on Unified Transformation Framework”. In: *Proc. ICPP* (cit. on p. 44).

[Luo et al. 2015] Luo, Y., G. Tan, Z. Mo, and M. Suo (2015). “FAST: A Fast Stencil Autotuning Framework Based on an Optimal-solution Space Model”. In: *Proc. ICS* (cit. on p. 46).

[Lutz 2015] Lutz, T. (2015). “Enhancing Productivity and Performance Portability of OpenCL Applications on Heterogeneous Systems using Runtime Optimizations”. PhD thesis. The University of Edinburgh (cit. on p. 49).

[Lutz et al. 2013] Lutz, T., C. Fensch, and M. Cole (2013). “PARTANS: An autotuning framework for stencil computation on multi-GPU systems”. In: *ACM TACO* (cit. on pp. 39, 46).

[Lutz et al. 2015] Lutz, T., C. Fensch, and M. Cole (2015). “Helium: A Transparent Inter-kernel Optimizer for OpenCL”. In: *Proc. GPGPU* (cit. on p. 39).

[Macri 2015] Macri, J. (2015). “AMD’s Next Generation GPU and High Bandwidth Memory Architecture: FURY”. In: *Proc. Hot Chips Symposium* (cit. on p. 21).

[Magni et al. 2014] Magni, A., C. Dubach, and M. O’Boyle (2014). “Automatic Optimization of Thread-Coarsening for Graphics Processors”. In: *Proc. PACT* (cit. on pp. 39, 41, 49).

[Majeti et al. 2016] Majeti, D., K. S. Meel, R. Barik, and V. Sarkar (2016). “Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures”. In: *Proc. CC* (cit. on p. 46).

[Marangoni and Wischgoll 2016] Marangoni, M. and thomas Wischgoll (2016). “Togpu: Automatic Source Transformation from C++ to CUDA using Clang-LLVM”. In: *Proc. Electronic Imaging* (cit. on pp. 44, 45).

[Matsumoto et al. 2012] Matsumoto, K., N. Nakasato, and S. G. Sedukhin (2012). “Performance tuning of matrix multiplication in OpenCL on different GPUS and CPUS”. In: *Proc. SCC* (cit. on p. 46).

[Mei and Chu 2017] Mei, X. and X. Chu (2017). “Dissecting GPU Memory Hierarchy Through Microbenchmarking”. In: *ArXiv* (cit. on p. 43).

[Meng et al. 2011] Meng, J., V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram (2011). “GROPHECY: GPU Performance Projection from CPU Code Skeletons”. In: *Proc. SC* (cit. on pp. 43, 54).

[Miceli and Bodin 2013] Miceli, R. and F. Bodin (2013). *The State-of-the-Art in Directive-Guided Auto-Tuning for Accelerator and Heterogeneous Many-Core Architectures*. Tech. rep. PRACE White Papers (cit. on p. 47).

[Miceli et al. 2013] Miceli, R., G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin (2013). “AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications”. In: *Proc. PARA* (cit. on p. 47).

[Michaud 2016] Michaud, P. (2016). “Some Mathematical Facts About Optimal Cache Replacement”. In: *ACM TACO* (cit. on p. 51).

[Mills and Mills 2015] Mills, N. and E. Mills (2015). “Taming the energy use of gaming computers”. In: *Energy Efficiency* (cit. on p. 1).

[Moammer 2016] Moammer, K. (2016). “Nvidia Plans GTX 2080 Ti, 2080 and 2070 Refresh With GDDR5X and Faster Clocks In 2017 - Volta GPUs With HBM2 and GDDR6 in 2018”. In: *WCCFTECH.com*. wccftech.com/nvidia-pascal-volta-gpu-leaked-2017-2018/ [accessed 07.04.2017] (cit. on p. 33).

[Monakov et al. 2010] Monakov, A., A. Lokhmotov, and A. Avetisyan (2010). “Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures”. In: *Proc. HiPEAC* (cit. on pp. 39, 46).

[Moore 1965] Moore, G. E. (1965). “Cramming more components onto integrated circuits”. In: *Electronics Magazine* (cit. on pp. 1, 16).

[Moreira et al. 2017] Moreira, R. E. A., S. Collange, and F. M. Q. Pereira (2017). “Function Call Re-Vectorization”. In: *Proc. PPoPP* (cit. on p. 49).

[Morton 1966] Morton, G. M. (1966). *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. Tech. rep. IBM Corporation (cit. on p. 22).

[Moskewicz et al. 2016] Moskewicz, M. W., A. Jannesari, and K. Keutzer (2016). "A Metaprogramming and Autotuning Framework for Deploying Deep Learning Applications". In: *ArXiv* (cit. on p. 47).

[Muralidharan et al. 2014] Muralidharan, S., M. Shantharam, M. Hall, M. Garland, and B. Catanzaro (2014). "Nitro: A Framework for Adaptive Code Variant Tuning". In: *Proc. IPDPS* (cit. on pp. 39, 41, 42, 46, 48, 71, 75, 117, 120).

[Muralidharan et al. 2016a] Muralidharan, S., A. Roy, M. Hall, M. Garland, and P. Rai (2016a). "Architecture-Adaptive Code Variant Tuning". In: *Proc. ASPLOS* (cit. on p. 48).

[Muralidharan et al. 2016b] Muralidharan, S., M. Garland, A. Sidelnik, and M. Hall (2016b). "Designing a Tunable Nested Data-Parallel Programming System". In: *ACM TACO* (cit. on p. 50).

[NVIDIA 2009] NVIDIA (2009). *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf [accessed 07.04.2017] (cit. on pp. 2, 33).

[NVIDIA 2013] NVIDIA (2013). *CUB*. nvlabs.github.io/cub/ [accessed 07.04.2017] (cit. on p. 124).

[NVIDIA 2014a] NVIDIA (2014a). *Kepler GK110/210 Whitepaper*. images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf [accessed 07.04.2017] (cit. on pp. 2, 33).

[NVIDIA 2014b] NVIDIA (2014b). *NVIDIA GeForce GTX 980*. international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF [accessed 07.04.2017] (cit. on pp. 2, 34).

[NVIDIA 2014c] NVIDIA (2014c). *NVIDIA NVLink High-Speed Interconnect: Application Performance*. info.nvidianews.com/rs/nvidia/images/NVIDIA NVLink High-Speed Interconnect Application Performance Brief.pdf [accessed 07.04.2017] (cit. on p. 20).

[NVIDIA 2015] NVIDIA (2015). *Release 349 Graphics Drivers for Windows, Version 350.12*. de.download.nvidia.com/Windows/350.12/350.12-win8-win7-winvista-desktop-release-notes.pdf [accessed 07.04.2017] (cit. on p. 121).

[NVIDIA 2016a] NVIDIA (2016a). *CUDA Programming Guide v8.0*. docs.nvidia.com/cuda/cuda-c-programming-guide/index.html [accessed 07.04.2017] (cit. on pp. 22, 25, 26, 28, 29, 33, 35, 54, 82, 115).

[NVIDIA 2016b] NVIDIA (2016b). *NVIDIA CUDA Driver API v8.0*. docs.nvidia.com/cuda/cuda-driver-api/index.html [accessed 07.04.2017] (cit. on p. 115).

[NVIDIA 2016c] NVIDIA (2016c). *NVIDIA CUDA Samples v8.0*. [included in CUDA Toolkit v8.0] (cit. on p. 76).

[NVIDIA 2016d] NVIDIA (2016d). *NVIDIA Tesla P100*. images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf [accessed 07.04.2017] (cit. on pp. 21, 29, 34).

[Nugteren and Codreanu 2015] Nugteren, C. and V. Codreanu (2015). "A Generic Auto-Tuner for OpenCL Kernels". In: *Proc. MCSoC* (cit. on pp. 39, 41, 48).

[Oliveira Castro et al. 2013] Oliveira Castro, P. de, E. Petit, A. Farjallah, and W. Jalby (2013). "Adaptive Sampling for Performance Characterization of Application Kernels". In: *Concurrency and Computation: Practice and Experience* (cit. on p. 43).

[Olofsson 2016] Olofsson, A. (2016). *Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip*. Tech. rep. www.parallella.org/2016/10/05/epiphany-v-a-1024-core-64-bit-risc-processor/ [accessed 07.04.2017]. Adapteva Inc. (cit. on p. 19).

[Öztireli and Gross 2015] Öztireli, A. C. and M. Gross (2015). "Perceptually Based Downscaling of Images". In: *ACM TOG* (cit. on p. 78).

[PCI-SIG 2010] PCI-SIG (2010). *PCI Express Base Specification Revision 3.0*. composter.com.ua/documents/PCI_Express_Base_Specification_Revision_3.0.pdf [accessed 07.04.2017] (cit. on p. 15).

[Pacula et al. 2012] Pacula, M., J. Ansel, S. Amarasinghe, and U.-M. O'Reilly (2012). "Hyperparameter Tuning in Bandit-Based Adaptive Operator Selection". In: *Proc. EvoApplications* (cit. on p. 45).

[Pai and Pingali 2016] Pai, S. and eshav Pingali (2016). “A Compiler for Throughput Optimization of Graph Algorithms on GPUs”. In: *Proc. OOPSLA* (cit. on p. 47).

[Panneerselvam and Swift 2016] Panneerselvam, S. and M. Swift (2016). “Rinnegan: Efficient Resource Use in Heterogeneous Architectures”. In: *Proc. PACT* (cit. on p. 50).

[Paone et al. 2014] Paone, E., D. Gadioli, G. Palermo, V. Zaccaria, and C. Silvano (2014). “Evaluating Orthogonality between Application Auto-Tuning and Run-Time Resource Management for Adaptive OpenCL Applications”. In: *Proc. ASAP* (cit. on p. 49).

[Papadonikolakis et al. 2009] Papadonikolakis, M., C.-S. Bouganis, and G. Constantinides (2009). “Performance comparison of GPU and FPGA architectures for the SVM training problem”. In: *Proc. FPT* (cit. on p. 19).

[Park et al. 2011] Park, E., L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan (2011). “Predictive Modeling in a Polyhedral Optimization Space”. In: *Parallel Programming* (cit. on p. 44).

[Park et al. 2015] Park, J. J. K., Y. Park, and S. Mahlke (2015). “ELF: Maximizing Memory-level Parallelism for GPUs with Coordinated Warp and Fetch Scheduling”. In: *Proc. SC* (cit. on p. 51).

[Patterson and Hennessy 2013] Patterson, D. A. and J. L. Hennessy (2013). *Computer Organization and Design*. Vol. 4. Morgan Kaufmann (cit. on pp. 7–10, 12, 17).

[Pauwels et al. 2011] Pauwels, K., M. Tomasi, J. D. Alonso, E. Ros, and M. M. V. Hulle (2011). “A Comparison of FPGA and GPU for Real-Time Phase-based Optical Flow, Stereo, and Local Image Features”. In: *IEEE TOC* (cit. on p. 19).

[Peng et al. 2016] Peng, Y., M. Grossman, and V. Sarkar (2016). “Static Cost Estimation for Data Layout Selection”. In: *Proc. PMBS* (cit. on pp. 23, 39, 40, 50, 75, 118).

[Pennycook et al. 2016] Pennycook, S. J., J. D. Sewall, and V. W. Lee (2016). “A Metric for Performance Portability”. In: *Proc. PMBS* (cit. on p. 91).

[Pimenta et al. 2013] Pimenta, A., E. Cesar, and A. Sikora (2013). “Methodology for MPI Applications Autotuning”. In: *Proc. EuroMPI* (cit. on p. 47).

[Popov et al. 2006] Popov, S., J. Günther, H.-P. Seidel, and P. Slusallek (2006). “Experiences with Streaming Construction of SAH KD-Trees”. In: *Proc. IEEE IRT* (cit. on p. 79).

[Püschel et al. 2004] Püschel, M., B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson (2004). “SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms”. In: *IJHPCA* (cit. on p. 47).

[Ragan-Kelley et al. 2012] Ragan-Kelley, J., A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand (2012). “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines”. In: *ACM TOG* (cit. on p. 46).

[Ragan-Kelley et al. 2013] Ragan-Kelley, J., C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe (2013). “Halide: A Language and Compiler for Optimizing Parallelism, Locality and Recomputation in Image Processing Pipelines”. In: *Proc. PLDI* (cit. on p. 46).

[Rasmussen and Williams 2006] Rasmussen, C. E. and C. K. I. Williams (2006). “Gaussian Processes for Machine Learning”. In: *The MIT Press* (cit. on pp. 96, 97).

[Reinders 2013] Reinders, J. (2013). “Intel AVX-512 instructions”. In: *Intel Developer Zone*. software.intel.com/en-us/blogs/2013/avx-512-instructions [accessed 07.04.2017] (cit. on pp. 2, 17).

[Rossbach et al. 2013] Rossbach, C., Y. Yu, J. Currey, and J.-P. Martin (2013). *Dandelion: a Compiler and Runtime for Heterogeneous Systems*. Tech. rep. Microsoft Research (cit. on p. 49).

[Rossi and Zhou 2016] Rossi, R. A. and R. Zhou (2016). “Hybrid CPU-GPU Framework for Network Motifs”. In: *ArXiv* (cit. on p. 50).

[Rubin et al. 2014] Rubin, E., E. Levy, A. Barak, and T. Ben-Nun (2014). “MAPS: Optimizing Massively Parallel Applications Using Device-Level Memory Abstraction”. In: *ACM TACO* (cit. on p. 50).

[Rudy et al. 2011] Rudy, G., M. M. Khan, M. Hall, C. Chen, and J. Chame (2011). “A Programming Language Interface to Describe Transformations and Code”. In: *Proc. LCPC* (cit. on pp. 45, 48).

[Ryoo et al. 2008] Ryoo, S., C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu (2008). “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA”. In: *Proc. PPoPP* (cit. on p. 50).

[Sakai et al. 2016] Sakai, R., F. Ino, and K. Hagihara (2016). “Towards Automating Multi-Dimensional Data Decomposition for Executing a Single-GPU Code on a Multi-GPU System”. In: *Proc. CSA* (cit. on p. 50).

[Sensi et al. 2016] Sensi, D. D., M. Torquati, and M. Danelutto (2016). “A Reconfiguration Algorithm for Power-Aware Parallel Applications”. In: *ACM TACO* (cit. on p. 47).

[Shilov 2016] Shilov, A. (2016). “Discrete Desktop GPU Market Trends Q2 2016: AMD Grabs Market Share, But NVIDIA Remains on Top”. In: *AnandTech.com*. www.anandtech.com/show/10613/discrete-desktop-gpu-market-trends-q2-2016-amd-grabs-market-share-but-nvidia-remains-on-top
[accessed 07.04.2017] (cit. on p. 2).

[Shudler et al. 2015] Shudler, S., A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf (2015). “Exascaling Your Library Will Your Implementation Meet Your Expectations”. In: *Proc. ICS* (cit. on p. 43).

[Siddiqui et al. 2014] Siddiqui, S., F. AlZayer, and S. Feki (2014). “Historic Learning Approach for Auto-tuning OpenACC Accelerated Scientific Applications”. In: *Proc. VECPAR* (cit. on p. 44).

[Sikora et al. 2016] Sikora, A., E. César, I. Comprés, and M. Gerndt (2016). “Auto-tuning of MPI Applications Using PTF”. In: *Proc. SEM4HPC* (cit. on p. 47).

[Sorensen 2012] Sorensen, H. H. B. (2012). “Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs”. In: *Proc. PPAM* (cit. on pp. 39, 46).

[Srivastava et al. 2016] Srivastava, P., M. Kotsifakou, and V. Adve (2016). “HPVM: A Portable Virtual Instruction Set for Heterogeneous Parallel Systems”. In: *ArXiv* (cit. on p. 50).

[Stephenson et al. 2003] Stephenson, M., S. Amarasinghe, M. Martin, and U.-M. O’Reilly (2003). “Improving compiler heuristics with machine learning”. In: *Proc. PLDI* (cit. on p. 44).

[Steuwer et al. 2016] Steuwer, M., T. Remmelg, and C. Dubach (2016). "Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation". In: *Proc. CASES* (cit. on p. 46).

[Stratton et al. 2012] Stratton, J. A., C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, D. Liu, and W. mei W. Hwu (2012). *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Tech. rep. UIUC (cit. on p. 119).

[Strzodka 2011] Strzodka, R. (2011). *Abstraction for AoS and SoA Layout in C++* (cit. on p. 50).

[Strzodka 2012] Strzodka, R. (2012). "Data Layout Optimization for Multi-Valued Containers in OpenCL". In: *Parallel and Distributed Computing* (cit. on p. 50).

[Sung et al. 2012] Sung, I.-J., G. D. Liu, and W.-M. W. Hwu (2012). "DL: A Data Layout Transformation System for Heterogeneous Computing". In: *Proc. InPar* (cit. on pp. 23, 39, 50, 75, 118).

[TIOBE 2016] TIOBE (2016). *TIOBE Index*. www.tiobe.com/tiobe-index [accessed 12.04.2017] (cit. on p. 122).

[Tang et al. 2015] Tang, W. T., R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh (2015). "Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi". In: *Proc. CGO* (cit. on p. 46).

[Tapus et al. 2002] Tapus, C., I.-H. Chung, and J. K. Hollingsworth (2002). "Active Harmony: Towards Automated Performance Tuning". In: *Proc. SC* (cit. on p. 48).

[Tausche et al. 2016] Tausche, K., M. Plauth, and A. Polze (2016). "dOpenCL: Evaluation of an API-Forwarding Implementation". In: *Proc. HPI Cloud Symposium* (cit. on p. 50).

[TechPowerUp.com 2017] TechPowerUp.com (2017). *GPU Database*. www.techpowerup.com/gpudb/ [accessed 07.04.2017] (cit. on p. 76).

[Tillmann et al. 2013] Tillmann, M., T. Karcher, C. Dachsbacher, and W. Tichy (2013). "Application-independent Autotuning for GPUs". In: *Proc. ParCo* (cit. on p. 48).

[Tillmann et al. 2016] Tillmann, M., P. Pfaffe, C. Kaag, and W. F. Tichy (2016). "Online-Autotuning of Parallel SAH kD-Trees". In: *Proc. IPDPS* (cit. on p. 48).

[Tiwari et al. 2009] Tiwari, A., C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth (2009). “A Scalable Auto-tuning Framework for Compiler Optimization”. In: *Proc. IPDPS* (cit. on p. 48).

[Tiwari et al. 2011] Tiwari, A., M. A. Laurenzano, L. Carrington, and A. Snavely (2011). “Auto-tuning for Energy Usage in Scientific Applications”. In: *Proc. EUROPAR* (cit. on p. 47).

[Tom’s Hardware 2017] Tom’s Hardware (2017). *Enterprise HDD Charts*. www.tomshardware.com/charts/enterprise-hdd-charts/benchmarks,156.html [accessed 07.04.2017] (cit. on p. 15).

[Tomusk et al. 2016] Tomusk, E., C. Dubach, and M. O’Boyle (2016). “Selecting Heterogeneous Cores for Diversity”. In: *ACM TACO* (cit. on p. 47).

[Top 500 2016] Top 500 (2016). *November 2016*. www.top500.org/lists/2016/11/ [accessed 07.04.2017] (cit. on p. 1).

[Veras et al. 2016] Veras, R. M., T. M. Low, T. M. Smith, and R. van de Geijn Franz Franchetti (2016). “Automating the Last-Mile for High Performance Dense Linear Algebra”. In: *ArXiv* (cit. on p. 46).

[Vijayaragavan et al. 2017] Vijayaragavan, T., Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karaunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan (2017). “Design and Analysis of an APU for Exascale Computing”. In: *Proc. HPCA* (cit. on pp. 20, 114).

[Viñas et al. 2013] Viñas, M., Z. Bozkus, and B. B. Fraguela (2013). “Exploiting heterogeneous parallelism with the Heterogeneous Programming Library”. In: *Parallel and Distributed Computing* (cit. on p. 49).

[Viñas et al. 2016] Viñas, M., B. B. Fraguela, D. Andrade, and R. Doallo (2016). “High Productivity Multi-device Exploitation with the Heterogeneous Programming Library”. In: *Parallel and Distributed Computing* (cit. on p. 49).

[Volkov 2010] Volkov, V. (2010). “Better Performance at Lower Occupancy”. In: *GPU Tech Conference* (cit. on p. 28).

[Vollmer et al. 2015] Vollmer, M., B. J. Svensson, E. Holk, and R. R. Newton (2015). “Meta-Programming and Auto-Tuning in the Search for High Performance GPU Code”. In: *Proc. FHPC* (cit. on p. 46).

[Vuduc et al. 2005] Vuduc, R., J. W. Demmel, and K. A. Yelick (2005). “OSKI: A library of automatically tuned sparse matrix kernels”. In: *Journal of Physics* (cit. on p. 46).

[Waechter et al. 2012] Waechter, M., K. Jaeger, S. Weissgraeber, S. Widmer, M. Goesele, and K. Hamacher (2012). “Information-theoretic Analysis of Molecular (Co)Evolution Using Graphics Processing Units”. In: *Proc. ECMLS* (cit. on p. 79).

[Wang et al. 2010] Wang, G., Y. Lin, and W. Yi (2010). “Kernel Fusion: an Effective Method for Better Power Efficiency on Multithreaded GPU”. In: *Proc. GreenCom* (cit. on p. 118).

[Wang and Chu 2017] Wang, Q. and X. Chu (2017). “GPGPU Performance Estimation with Core and Memory Frequency Scaling”. In: *ArXiv* (cit. on p. 43).

[Wang et al. 2015] Wang, Z., D. Grewe, and M. F. P. O’Boyle (2015). “Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-based Heterogeneous Systems”. In: *ACM TACO* (cit. on p. 44).

[Weber and Goesele 2014] Weber, N. and M. Goesele (2014). “Auto-Tuning Complex Array Layouts for GPUs”. In: *Proc. EGPGV* (cit. on pp. 4, 53).

[Weber and Goesele 2016] Weber, N. and M. Goesele (2016). “Adaptive GPU Array Layout Auto-Tuning”. In: *Proc. SEM4HPC* (cit. on pp. 4, 5, 69, 71, 72, 82).

[Weber and Goesele 2017] Weber, N. and M. Goesele (2017). “MATOG: Array Layout Auto-Tuning for CUDA”. In: *ACM TACO*. *in review* (cit. on pp. 4, 5).

[Weber et al. 2015] Weber, N., S. C. Amend, and M. Goesele (2015). “Guided Profiling for Auto-Tuning Array Layouts on GPUs”. In: *Proc. PMBS* (cit. on pp. 4, 5, 64, 65, 75).

[Weber et al. 2016] Weber, N., M. Waechter, S. C. Amend, S. Guthe, and M. Goesele (2016). “Rapid, Detail-Preserving Image Downscaling”. In: *Proc. SIGGRAPH Asia* (cit. on pp. VII, 78).

[Whaley and Dongarra 1998] Whaley, R. C. and J. J. Dongarra (1998). “Automatically Tuned Linear Algebra Software”. In: *Proc. SC* (cit. on pp. 37, 46).

[Wolf et al. 2014] Wolf, F., C. Bischof, T. Hoefler, B. Mohr, G. Wittum, A. Calotoiu, C. Iwainsky, A. Strube, and A. Vogel (2014). “Catwalk: A Quick Development Path for Performance Models”. In: *Proc. EUROPAR* (cit. on p. 95).

[Wong et al. 2010] Wong, H., M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos (2010). “Demystifying GPU Microarchtitecture through Microbenchmarking”. In: *Proc. ISPASS* (cit. on p. 43).

[Wu et al. 2015] Wu, G., J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou (2015). “GPGPU performance and power estimation using machine learning”. In: *Proc. HPCA* (cit. on pp. 43, 96).

[Wu et al. 2016] Wu, J., A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt (2016). “GPUCC - An Open-Source GPGPU Compiler”. In: *Proc. SCGO* (cit. on p. 117).

[Xu and Gregg 2015] Xu, S. and D. Gregg (2015). “Exploiting Hyper-Loop Parallelism in Vectorization to Improve Memory Performance on CUDA GPGPU”. In: *Proc. TRUSTCOM* (cit. on p. 49).

[Yamato 2017] Yamato, Y. (2017). “Optimum Application Deployment Technology for Heterogeneous IaaS Cloud”. In: *Information Processing* (cit. on p. 50).

[Yang et al. 2010] Yang, Y., P. Xiang, J. Kong, and H. Zhou (2010). “A GPGPU Compiler for Memory Optimization and Parallelism Management”. In: *Proc. PLDI* (cit. on p. 51).

[Yang et al. 2016] Yang, Y., S. Prestwood, and C. Barnes (2016). “VizGen: Accelerating Visual Computing Prototypes in Dynamic Languages”. In: *Proc. SIGGRAPH Asia* (cit. on p. 47).

[Yu and Cardona 2010] Yu, P. Y. and M. Cardona (2010). *Fundamentals of Semiconductors*. Vol. 4. Pearson (cit. on p. 16).

[Zenker et al. 2016] Zenker, E., B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann (2016). “Alpaka - An Abstraction Library for Parallel Kernel Acceleration”. In: *ArXiv* (cit. on p. 50).

[Zhang et al. 2005] Zhang, K., U. Bhattacharya, Z. Chen, F. Hamzaoglu, D. Murray, N. Vallepalli, Y. Wang, B. Zheng, and M. Bohr (2005). “SRAM Design on 65-nm CMOS Technology With Dynamic Sleep Transistor for Leakage Reduction”. In: *SSC* (cit. on p. 16).

[Zhang and Mueller 2013] Zhang, Y. and F. Mueller (2013). “Auto-Generation and Auto-Tuning of 3D Stencil Codes on homogeneous and Heterogeneous GPU Clusters”. In: *IEEE TPDS* (cit. on p. 46).

[Zhang et al. 2016] Zhang, Y., S. Li, S. Yan, and H. Zhou (2016). "A Cross-Platform SpMV Framework on Many-Core Architectures". In: *ACM TACO* (cit. on p. 46).

[Zheng et al. 2012] Zheng, M., V. T. Ravi, W. Ma, F. Qin, and G. Agrawal (2012). "GMProf: A Low-Overhead, Fine-Grained Profiling Approach for GPU Programs". In: *Proc. HiPC* (cit. on p. 43).

[Ziabari et al. 2016] Ziabari, A. K., Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli (2016). "UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs". In: *ACM TACO* (cit. on p. 51).

[Zivanovic et al. 2017] Zivanovic, D., M. Pavlovic, M. Radulovic, H. Shin, J. Son, S. A. McKee, P. M. Carpenter, P. Radojković, and E. Ayguadé (2017). "Main Memory in HPC: Do We Need More or Could We Live with Less?" In: *ACM TACO* (cit. on p. 115).

(Co-)Authored Publications

Patrick Weber, **Nicolas Weber**, Michael Goesele and Rüdiger Kabst. **Prospect for Knowledge in Survey Data – An Artificial Neural Network Sensitivity Analysis**. Social Science Computer Review (SSCR), 2017

Nicolas Weber and Michael Goesele. **MATOG: Array Layout Auto-Tuning for CUDA** ACM Transactions on Architecture and Code Optimization (TACO), 2017.

Nicolas Weber, Michael Waechter, Sandra C. Amend, Stefan Guthe and Michael Goesele. **Rapid, Detail-Preserving Image Downscaling**. ACM Transactions on Graphics (TOG), SIGGRAPH Asia, 2016.

Nicolas Weber and Michael Goesele. **Adaptive GPU Array Layout Auto-Tuning**. In proceedings of Software Engineering Methods for Parallel and High Performance Applications, SEM4HPC, 2016.

Nicolas Weber, Sandra C. Amend and Michael Goesele. **Guided Profiling for Auto-Tuning Array Layouts on GPUs**. In proceedings of 6th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS, 2015.

Nicolas Weber and Michael Goesele. **Auto-Tuning Complex Array Layouts on GPUs**. In proceedings of Eurographics Symposium on Parallel Graphics and Visualization, EGPGV, 2014.

Sven Widmer, Dominik Wodniok, **Nicolas Weber** and Michael Goesele. **Fast Dynamic Memory Allocator for Massively Parallel Architectures**. In proceedings of 6th Workshop on General Purpose Processing Using GPUs, GPGPU, 2013.