PERFORMANCE EXPLORATION OF SELECTED MANUALLY AND AUTOMATICALLY PARALLELIZED CODES ON GPUS

FRANZ XAVER BAYERL

Master's Thesis

Department of Informatics and Mathematics
Chair for Programming
University of Passau

Supervisor: Prof. Christian Lengauer, Ph.D.
Second reader: Prof. Dr. Martin Griebl
Tutor: Dr. Armin Größlinger

March 2012

ABSTRACT

General-Purpose computing on GPUs (GPGPU) provides the opportunity to utilize the tremendous computational power of graphics accelerators for a wider set of problems. These devices leverage massive parallelism to achieve high performance, however, creating highly parallelized code which is optimized for the characteristics of GPUs is no simple task. The polyhedron model is used successfully to parallelize code in many domains. We use polyhedral techniques to generate high performance CUDA code for specific problems on specific architectures and evaluate if automatically generated code is able to reach or exceed the performance of manually optimized code.

In this thesis we focus our research on tensor contractions and General Matrix-Matrix Multiplication (GEMM). We identify aspects that are crucial for high performance and deduce strategies for automatic code generation. These strategies are defined as transformations on polyhedral descriptions. Our experiments suggest that polyhedral code generators are able to generate Compute-Unified Device Architecture (CUDA) code that achieves the same level of performance as manually optimized codes.

ACKNOWLEDGMENTS

I thank Dr. Armin Größlinger for providing valuable advice and strong support for this thesis. His tutorial on parallel programming inspired me to seek a deeper insight into the topic. I also thank Prof. Christian Lengauer, Ph.D. and Prof. Dr. Martin Griebl for my academic background in this field.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ALU Arithmetic Logic Unit

BLAS Basic Linear Algebra Subprograms

CPU Central Processing Unit

CUDA Compute-Unified Device Architecture

GEMM General Matrix-Matrix Multiplication

GPGPU General-Purpose computing on GPUs

GPU Graphics Processing Unit

HPC High Performance Computing

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MPI Message Passing Interface

OpenMP Open Multi-Processing

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SISD Single Instruction Single Data

SM Streaming Multiprocessor

SPMD Single Program Multiple Data

1 INTRODUCTION

1.1 MOTIVATION

Driven by the continuous strong market demand for high quality computer graphics in end user systems, enormous efforts have been invested to enhance the graphical capabilities of computers. Since the appearance of 3D graphics and the first graphics cards in the mass market in the 1990s it finally became clear that a general purpose CPU is not the tool of choice for all types of computational problems when it comes to high performance.

Graphical processing requires a very high throughput of floating point operations - think of a state of the art video game that renders complex environments with over 60 frames per second. Each pixel of each frame needs to be calculated and painted within a very short time interval. The CPU was not able to handle this tremendous task on an adequate level and consequently new hardware, the Graphics Processing Unit (GPU), was developed. There is one characteristic that is inherent to many graphical calculations: the high potential for parallelism. GPUs are designed to exploit this characteristic and to deliver high performance in this scenario.

The result of the arrival of GPUs in the mass market is that today's computers are equipped with a massive parallel computing device which provides computing power that is orders of magnitude higher then that of CPUs (see Figure 1). Moreover, the calculations performed on GPUs are not limited to graphics anymore, GPGPU designates the process of moving all sorts of computationally expensive parts which can be parallelized to GPUs. Examples can be found in various fields, from scientific computing, e.g., electronic structure calculations in quantum chemistry, to multimedia applications, e.g., voice recognition or augmented reality.

The drawback of GPGPU is, apart from the limited problem space which can be handled efficiently, that developing parallel programs is very complex and more difficult than developing sequential programs. In order to achieve high performance, developers have to parallelize problems with respect to specific hardware characteristics of different GPU architectures, e.g., memory sizes, caches or access patterns. Efforts have been made to relieve the developers from this complex task. Although specialized libraries for specific problems exist and allow the developer to use optimized codes written by experts, automatic code generation for GPUs is needed to tackle a larger set of problems.

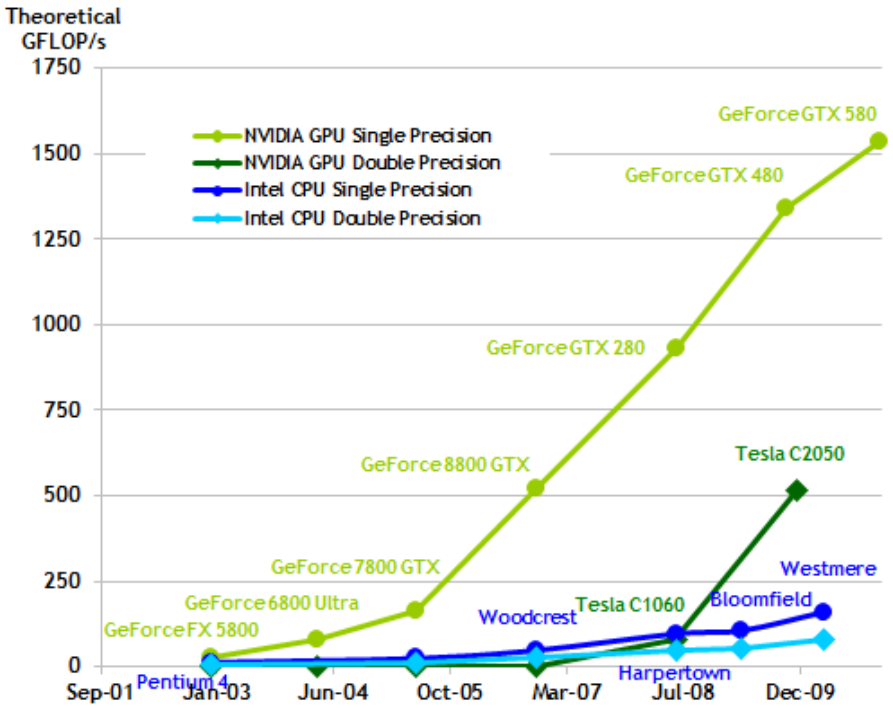

Figure 1: Theoretical Floating-Point Operations per Second for CPUs and GPUs (Image from[30])

The concept of parallel programming is not new and the technology exists to detect data dependencies in programs and to generate parallelized codes automatically. The difficulties arise from the aim to generate high performance code which is competitive to hand-optimized code. Autotuning and machine learning have been used to search the space of possible codes for optimal solutions. Another approach is to model a problem on an abstract level, e.g. with the polyhedron model [14], and to transform this model in a way that allows parallel execution. The performance of the generated code depends on the characteristics of the transformation.

1.2 OBJECTIVE

This thesis is concerned with performance optimization of generated polyhedral codes on GPUs. For this purpose, we analyse the best hand-optimized codes for specific problems and determine the aspects that are crucial for their high performance. In a next step, we use the results of this analysis and try to generate code that yields the same performance using polyhedral techniques automatically.

Because high performance codes have to be optimized with respect to architectural characteristics of the target platform, we limit our research to NVIDIA GPUs and CUDA. We differentiate between hardware of the compute capability 1.x and 2.x (see Section 2.2).

1.3 OUTLINE

The organization of the thesis is as follows. In Chapter 2 we give the required background information. Chapter 3 presents the hand-optimized codes on which our research is based and shows the measures necessary for optimizing the code. Subsequently, in Chapter 4, we use polyhedral techniques to generate code that aims for the same performance than the hand-optimized codes. We also present Treduda, a tool that generates CUDA code for tensor operations, which was used to implement and evaluate different transformations. Chapter 5 concludes the thesis.

2 PREREQUISITES

2.1 PARALLEL PROGRAMMING

In Section 2.1.1 we give a short overview of parallel computer architectures and show to which category GPUs belong and which principles are used for parallelization. In Section 2.1.2 we introduce the polyhedron model to the amount needed for this thesis.

2.1.1 *Parallel Computer Architectures*

Flynn's Taxonomy [15] is an established classification of computer architectures which differentiates them according to their ability to handle instruction or data streams sequential.

The classic von Neumann architecture, i.e., a non-parallel computer, implements the Single Instruction Single Data (SISD) principle. A control unit fetches instructions from a single instruction stream and uses only a single data stream as input, i.e., a control unit loads one piece of data in one Arithmetic Logic Unit (ALU) (provided that it is an arithmetical instruction and not an instruction that deals with control flow) and executes the instruction.

Data parallelism can be obtained by allowing several data streams as input, this principle is called Single Instruction Multiple Data (SIMD). In one clock cycle a control unit loads data from several data streams into several ALUs and executes the same instruction on each unit synchronously. Intel's Streaming SIMD Extensions (SSE) [33] is an example for this principle.

Another way to obtain parallelism is by using several instruction streams and one data stream, hence the name Multiple Instruction Single Data (MISD). This approach is suited for pipelining, since various instructions are executed on data from one input.

The last category Flynn describes is Multiple Instruction Multiple Data (MIMD). This architecture implements task parallelism by providing the possibility to control explicitly which instruction stream operates on which data stream. Multiple processors execute their own set of instructions on allocated data and there is no implicit synchronization. The input program is in charge of the parallelism, especially of synchronization and communication.

Mixtures and specializations of the four categories described by Flynn are possible and prevalent. Single Program Multiple Data (SPMD) [13] refers to an architectures that executes the same program on

many processors. Each processor is free to take a different control path (thread) in the program and operates on possibly different data. Like in MIMD, the parallelism has to be modeled explicitly in the program. SPMD has proven to be very successful in the field of High Performance Computing (HPC) and it is the principle predominant technologies like Message Passing Interface (MPI) [16] and Open Multi-Processing (OpenMP) [10] are based on.

The architecture GPUs are based on is Single Instruction Multiple Threads (SIMT) [30]. Processors of this architecture are arranged in groups called multiprocessors. Each multiprocessor consists of one control unit, several processors and several ALUs. Like SPMD, the same explicitly parallelized program is executed on all processors, each taking a possibly different control path. Like SIMD, one control unit is used per multiprocessor to fetch instructions from the instruction stream and to forward it to all processors in the group. Only processors of a multiprocessor that require the same instruction, i.e., that have taken the same control path in the program, are able to execute in the same clock cycle. If a processor in the group needs another instruction, the multiprocessor has to switch between control paths, serially executing each of them, and parallelism is lost.

We refer to Section 2.2 for further details on NVIDIA hardware.

2.1.2 *Polyhedron Model*

One objective of this thesis is to generate GPU code for specific problems automatically, i.e., we want to transform a problem in a way that it can be executed on a SIMT architecture. In order to accomplish this task, we require a representation of the problem to work with. In particular, we need a model of the loop structure of the problem. We are then able to apply transformations to the model that change the loop structure but that do not change the outcome. By choosing a transformation that changes the structure to fit the SIMT architecture, we are able to execute the problem on GPUs.

We use the polyhedron model [24, 14] to represent the loop structure of a computational problem. A program can be modeled as a set of statements. Each statement S consists of a ordered set T of instructions which are executed in the same iterations of a loop nest. We describe the iterations of a statement using an iteration domain D. D is a set of points in $\mathbb{Z}^n$ for some $n \in \mathbb{N}$. Each point $p \in D$ represents one specific iteration of the surrounding loop nest. We use a schedule Θ to define the order of execution for the iterations of a statement. Θ is a relation between D and a target space $O = \mathbb{Z}^m$ (for some $m \in \mathbb{N}$). The lexicographic order of O determines the order of execution. A statement $S = (T, D, \Theta)$ defines all instances of the instructions in T and their order of execution.

We describe the problems we want to execute on the SIMT architecture as a set of statements. Note that the polyhedron model has certain restrictions and that it cannot model all computational problems, e.g., the loop bounds have to be linear expressions. However, the model can be used to describe a wide range of problems and it is apt for our purpose.

The following code snippet is a very simple example of a program. Notice that all loop iterations can be executed independently from each other, i.e., no iteration depends on data from another iteration.

```
  for (int i=0; i<M; i++) {
S:  A[i] = i;
  }
```

We can express this program using the polyhedron model.

$$T = \{A[i] = i;\}$$
$$D = \{(i) \mid 0 \leqslant i < M\}$$
$$\Theta = \{(i) \to (i)\}$$

In order to execute this program efficiently on a SIMT architecture, we need to distribute all points in T among the multiprocessors. Let $x = 16$ be the number of processors in a multiprocessor. We want to transform the program in a way that allows us to calculate x points of T per multiprocessor. We can transform the model by modifying Θ, i.e., we change the order of execution.

$$\Theta' = \{(i) \to (j,k) \mid (1-x) + i \leqslant x \cdot j \leqslant i, \quad i = x \cdot j + k\}$$

We introduce another ordering dimension by increasing the dimension of the target space from 1 to 2. Then, we map all values of i to j and k. In this case we use a tiling that divides values of i into groups of size x. $S' = (T, D, \Theta')$ is a model of the following code snippet.

```
    for (int j=0; j<⌊M/16⌋; j++){
      for (int k=0; k<min(16, M-16·j); k++) {
        int i = 16·j+k;
S':     A[i] = i;
      }
  }
```

We see that we have divided the problem into equal parts of size x (if M is a multiple of x). We can use this order of execution to divide all points in T among different multiprocessors, each processing x different points.

In Section 4.2 we show another example usage of the polyhedron model and explain in detail how we use it to generate CUDA code.

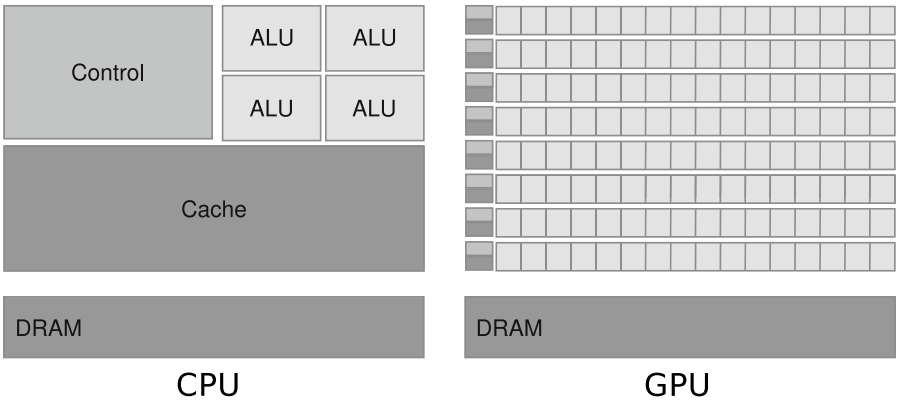

Figure 2: Different design philosphies of CPUs and GPUs (Image from [30])

2.2 CUDA

We introduce Compute-Unified Device Architecture (CUDA) in this section. At first, we explain NVIDIA's hardware implementation of the Single Instruction Multiple Threads (SIMT) architecture. Then we describe the programming model that is used to control the hardware.

2.2.1 *Hardware*

The reason why a GPU performs better than a traditional CPU when it comes to parallel computing is the fundamental difference in the processor design [30, 29, 30]. CPUs are optimized for sequential code performance and dedicate a large portion of the chip area to the implementation of control logic algorithms and data caches. In contrast, GPUs maximize the chip area that is used for floating point operations and, therefore, reduce the complexity of the control logic and the size of the data caches, as illustrated in Figure 2. While a CPU typically supports 1-2 hardware threads and uses sophisticated algorithms and large caches to load the ALUs with the work of these few hardware threads, the GPU takes advantage of massive multithreading to hide latency. If one hardware thread is stalled, the processor is able to find work within a large pool of hardware threads waiting for execution [22]. A GPU consists of hundreds of processors each supporting hundreds of hardware threads and, in order to utilize the full potential, a computational problem needs to be parallelized to divide the workload among this massive number of hardware threads.

2.2.1.1 *Streaming Multiprocessor*

The SIMT architecture, which is the basic concept behind GPUs, was introduced in Section 2.1.1. NVIDIA implements this concept using Streaming Multiprocessors (SM). A GPU has many SMs, each SM including several (8 or 32 in current implementations) CUDA cores, capable

of integer and single precision calculations, and a warp scheduler. All threads are organized in groups of 32 threads, called a warp. The warp scheduler is the implementation of the control unit in the SIMT architecture. It loads instructions required by the threads and forwards them to the corresponding CUDA cores. Parallelism is obtained if many threads require the same instruction, i.e., follow the same control path in the program.

A Streaming Multiprocessor (SM) creates, manages, schedules and executes warps [30], each thread of which has its own instruction address counter and register state. Even though all threads of a warp start with the same program adress, they are free to take different control paths and execute independently.

All 32 threads of a warp do not necessarily need to be executed in parallel, even if they require the same instruction. Since some NVIDIA GPUs have only 8 CUDA cores per SM, there is a further separation in half-warps and quarter-warps which are executed serially. In addition, CUDA cores are only capable of integer and single precision operations, if a double precision operation or a transcendental operation, like sin, needs to be calculated parallelism is lost, because a SM includes only small number of double precision and transcendental units. This is a result of the design which is optimized for a high throughput of single precision operations.

An SM has a set of 32-bit registers, as well as on-chip memory at his disposal. The registers are allocated to specific threads of a warp and the on-chip memory is shared between all threads of a thread block. A thread block is a group of warps, or from a different perspective, a thread block inherits a number of threads which are organized in warps. If the number of threads is not a multiple of the number of threads in a block, padding is used. All threads of a block have access to the same part of the on-chip memory. This is called the scratchpad memory. In Section 2.2.2.2 the memory hierarchy of CUDA is explained in detail.

2.2.1.2 *Hardware Multithreading*

Upon initialization, a set of warps and blocks is assigned to each SM. The context of each warp, particularly the instruction address and the register state of each thread, are maintained in the SM for the entire lifetime of the warp, making switches between different warps cost-free [30]. The warp scheduler of an SM is able to pick any warp from a set of warps waiting for execution and to process it. A warp is waiting for execution if it has threads waiting for the next instruction. The warp scheduler then fetches and forwards the required instruction to those threads.

The number of warps that can reside in one SM depends on the available registers and the available on-chip memory. Based upon the program, each warp requires a fixed amount of registers and each

block requires a fixed amount of scratchpad memory to execute. An SM can only support as many warps and blocks as memory is available. Additionally, there are conceptional restrictions to the maximum number of warps and blocks, e.g., current architectures allow a maximum number of 8 blocks per SM.

2.2.1.3 *Architectural Differences*

In this thesis we differentiate between NVIDIA devices of the compute capability 1.x, codename Tesla, and devices of the compute capability 2.x, codename Fermi. The Tesla architecture is based on the g80 architecture, NVIDIA's first generation GPU Computing architecture. The Fermi architecture is the newest generation of NVIDIA's devices and brings several improvements.

The most important improvement in terms of performance behavior and programmability is the introduced cache hierarchy. The Fermi architecture provides a configurable L1 cache and an L2 cache. The L1 cache resides in each SM and can be configured to use more or less of the available on-chip memory. If more memory is used for caching, less memory is available for the scratchpad memory of the blocks. Even though caching improves the general performance in most cases, the developer has to decide which configuration is better suited for his application in order to achieve the highest performance.

2.2.2 *Programming Model*

We will now introduce the parallel programming model NVIDIA uses to control the hardware. From the developers point of view CUDA is a set of language extensions for C that allows to control the hierarchy of thread groups, memory and barrier synchronization [30].

Code that is to run on a GPU is written as a C function and is called a kernel. Upon invocation, the exact number of threads and blocks has to be specified. The instructions of the kernel are then executed in parallel by all threads.

2.2.2.1 *Threads*

When calling a kernel, the number of blocks, called the grid, and the number of threads per block has to be specified. Figure 3 shows the organization of the threads.

Both the blocks in a grid and the threads in a block can be arranged in multiple dimensions. Each thread has knowledge about his exact coordinates within the grid and the block. This information is an essential part of CUDA programming, because the developer can exactly control each thread addressing it by its coordinates. Nevertheless, it is important that all threads in a block follow the same control path

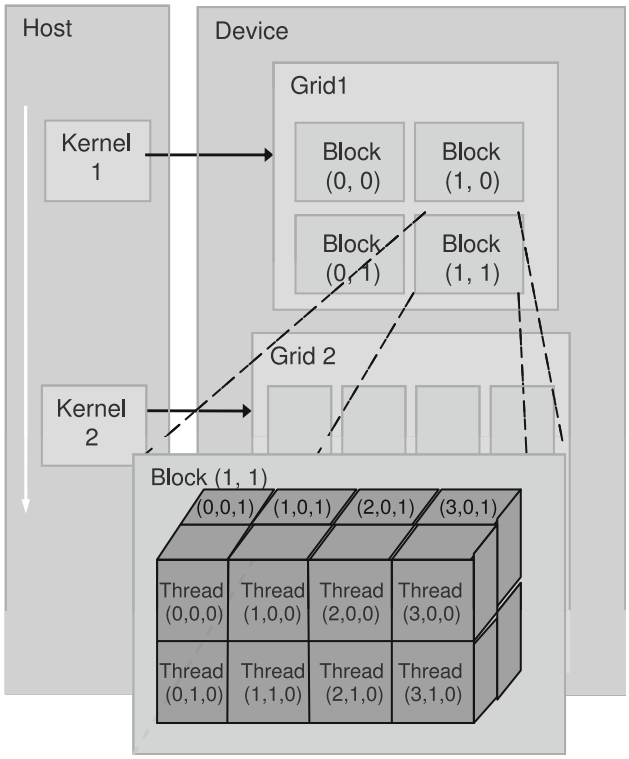

Figure 3: Cuda Thread organization (Image from [22])

in order do achieve parallelism. A typical CUDA program uses the coordinates of a thread as a parameter for memory accesses.

Because GPUs are based on data parallelism, the data needs to be distributed among all threads. Threads which require the same data or which require data from other threads are assigned to the same block. Parts of the data which can be processed independently from each other are divided into different blocks. The reason behind this division is that only threads in the same block are able to communicate with each other. This limitation exists, because of CUDAs memory hierarchy we describe in Section 2.2.2.2.

Each thread executes independently from all other threads. If synchronization is required, the developer has to implement explicit calls to a barrier synchronization in the program. These calls cause all threads of a block to be stalled until every thread in the block has reached the synchronization point. Synchronization is only possible within a single block and not between different blocks.

2.2.2.2 *Memory*

A CUDA device has four types of memory: main memory, constant memory, scratchpad memory and registers. Figure 4 shows how the memory is organized.

Typically, before a kernel is called the host allocates memory in the main or the constant memory and loads data to the device. After the

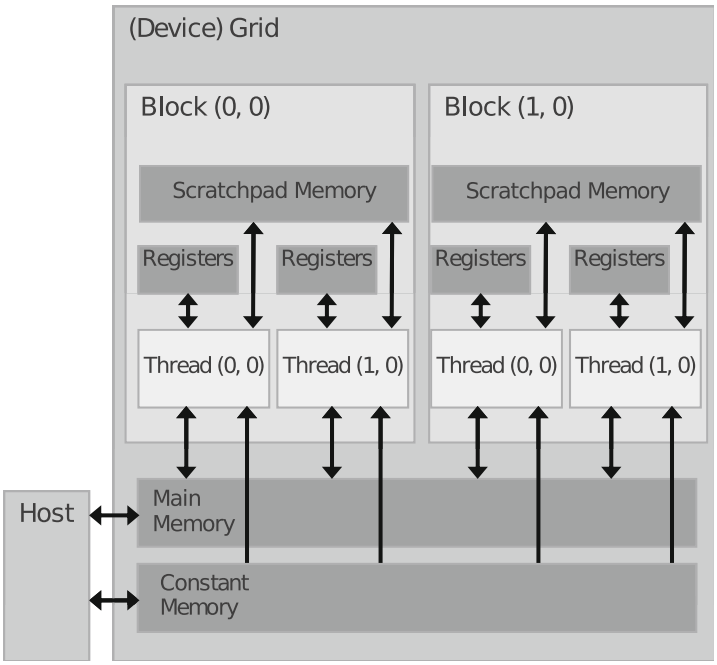

Figure 4: Cuda Device Memory Model (Image from [22])

kernel call is finished, the application can fetch the result from the main memory.

Every thread can read data from the main or the constant memory, but write access is limited to the main memory. Additionally, every thread has exclusive access to a set of registers and shared access to the scratchpad memory. Only threads in the same block can access the same scratchpad memory, a thread can not access the scratchpad memory of other blocks.

Memory conflicts are not handled by CUDA and result in undefined behavior. The developer is in charge of avoiding memory conflicts or resolving them using barrier synchronization or atomic operations.

Threads in a block can communicate with each other using the scratchpad memory. In one cycle a thread can write data into the scratchpad and in a later cycle, i.e., after a barrier synchronization, another thread can fetch this data. It is not possible to send data to a specific thread.

2.2.3 *Performance Considerations*

Many different aspects have to be considered when a CUDA program is developed for high performance. In fact, many different aspects are relevant to achieve even a moderate performance on the device. Solely parallelizing a problem, without specializing on the characteristics of the GPU, is likely to result in an unsatisfying performance.

Finding the optimal implementation on GPUs is very complex. Even small changes in the code can have a strong impact on the perfor-

mance and the difference in performance between implementations is significant. Using multi-variable optimization techniques to find the best solution is generally not feasible, because the optimization space on is large and discontinuous [34].

In this section we describe the aspects which are relevant for high performance.

2.2.3.1 *Device Memory*

The main and the constant memory are together called device memory, because they are not on-chip like the scratchpad memory or the registers. Accordingly, the device memory has a higher latency then the on-chip memory. Memory access to the device memory is to be minimized.

Since access to the device memory is inevitable, CUDA provides a way to achieve high throughput by coalescing memory requests. Data from the device memory is accessed in chunks, e.g., 128-byte words. If a thread requires only one byte from the device memory, a complete chunk has to be transferred from the device memory to the chip. To not leave the rest of the chunk unused, requests of threads which require data from the same chunk are coalesced. Because all threads of a warp are sure to execute the same instruction at a time, the hardware is able to scan for requests to the same chunk within a warp and to optimize the access.

Optimal coalescing would be achieved if all 32 threads of a warp would require consecutive 4 bytes of a 128-byte word. Only one chunk of data would be transferred from the device memory to the chip this way. If the threads of a warp access data in a very scattered pattern, i.e., data of different chunks, coalescing is not possible and a lot of unused data is transferred leading to a congestion and a performance loss.

2.2.3.2 *Scratchpad Memory*

The scratchpad memory is located on the chip and organized in memory banks. Memory banks are equally-sized memory modules that can be accessed in parallel. If all addresses contained in a memory access fall into distinct memory banks, the request can be handled simultaneously. If two addresses fall into the same memory bank a conflict appears.

Conflict-free access to the scratchpad memory has a very low latency and a very high throughput. If the required data is equally spread over N distinct memory banks, the overall bandwidth is N times higher than the bandwidth of a single memory module. Conflicting access to the memory banks enforces a serial execution of the request and, therefore, reduces the possible throughput.

Conflicts on memory banks need to be minimized in order to optimize the performance.

2.2.3.3 *Latency Hiding*

One of the principles of the SIMT architecture is latency hiding through massive multithreading [30]. While some threads are stalled, e.g., because they have to wait for memory transfers, other threads, which have already received data, are able to work. It is important to optimize an application in a way that all SMs have threads, i.e., warps, waiting for execution at all times in order to utilize the hardware [39, 34, 22] fully. Different approaches exist to achieve this optimization goal.

WORKLOAD PER THREAD A thread needs to be stalled if it has to wait for data dependencies or barrier synchronization calls. Instructions of a program that do not force the thread to be stalled are called independent instructions [34, 39]. A longer sequence of consecutive independent instructions allows threads and, therefore, warps to load the CUDA cores with work for a longer time. Hence, the number of consecutive independent instructions should be maximized in order to utilize latency hiding [39, 30].

A common practice to increase the number of consecutive independent instructions is preloading [22]. It means that a thread loads data which will be required for future calculations into a temporary memory, while currently processing data that has already been loaded. After the calculation is finished, the data from the temporary memory is prepared to be processed and the next set of data is preloaded. This way, the load statements can be added to the independent instructions, because only write statements enforce the use of synchronization.

SM RESOURCE ALLOCATION Warps and blocks are assigned to the SMs when a kernel is invoked. The number of blocks and warps per SM depends on the size of the scratchpad memory per block and the number of registers per thread (see Section 2.2.1.2). In general, the fewer registers and less scratchpad memory an application needs to execute, the more warps and blocks are assigned to each SM [30]. A higher number of warps and blocks per SM increases the possibility that warps are available for execution at any point in time, improving latency hiding [22].

When an application is designed, register and scratchpad memory usage have to be considered carefully. If certain thresholds to memory sizes are exceeded, the number of warps or blocks per SM decreases. NVIDIA provides a tool called Occupancy Calculator [28] that shows these thresholds.

The number of threads per block should always be a multiple of the architecture's warp size. If it is no multiple, some warps are not

completely filled with threads and not all CUDA cores can be fully utilized.

OPTIMIZING LATENCY HIDING In order to maximize latency hiding, the number of independent instruction and the number of warps and blocks per SM have to be maximized. Unfortunately, increasing the number of independent instructions typically requires more memory, e.g., for storing intermediate results or for data preloading. As a result, the number of warps or blocks per thread that can be allocated to a SM decreases. Finding the optimal solution is the task of the developer and requires experimentation [22, 34].

2.2.3.4 *Memory Usage*

The different types of memory on CUDA devices have different characteristics. While main and constant memory have a low latency, on-chip memory has a very high latency. It is important to note that registers are at least 3 times faster then the scratchpad memory and, because of that, we have to "use registers to run close to the peak" performance [39].

A CUDA application usually uses the scratchpad memory as a kind of custom implemented cache. Since all threads of a block have access to the scratchpad memory, each thread copies a small portion of data from the device to the chip in parallel, yielding a high throughput. After a synchronization, all threads have access to the full amount of the copied data. This technique can be used to minimize the number of data accesses to the device memory.

2.3 TENSORS

Tensor contractions are very important operations in the field of multilinear algebra [21]. Many problems in scientific computing can be expressed as operations on tensors, e.g., in fluid and solid mechanics, general relativity and quantum chemistry [20, 8]. They have a high potential for massive data-parallelism and are very compute intensive, making them apt for the execution on parallel architectures [8]. We will give a brief introduction to tensors based on a book by Nadir Jeevanjee [20].

A tensor of type (r,s) on a vector space V is defined as a multilinear function T on

$$\underbrace{V \times \cdots \times V}_{\text{r times}} \times \underbrace{V^* \times \cdots \times V^*}_{\text{s times}}$$

A multilinear function is linear in each argument, i.e.,

$$\begin{aligned} &T(v_1 + cw, v_2, \ldots, v_r, f_1, \ldots, f_s) \\ &\quad = T(v_1, \ldots, v_r, f_1, \ldots, f_s) + cT(w, v2, \ldots, f_1, \ldots, f_s) \end{aligned}$$

and similar for all the other arguments. Let $\{e_i\}_{i=1,\ldots,n}$ be a basis for V and $\{e^i\}_{i=1,\ldots,n}$ be the corresponding dual basis. We call

$$T^{j_1 \cdots j_s}_{i_1 \ldots i_r} \equiv T(e_{i_1}, \ldots, e_{i_r}, e^{j_1}, \ldots, e^{j_r})$$

the components of T in the basis $\{e^i\}_{i=1,\ldots,n}$. By choosing the basis accordingly, we can express tensors as multidimensional arrays. Scalars are defined as (0,0) tensors, vectors as (0,1) tensors and linear operators as (1,1) tensors.

3

MANUALLY PARALLELIZED EXPERIMENTS

In this chapter we try to find specific aspects and implementation strategies that are crucial for high performance. We create different implementations of a computational problem, each focusing on one or more different performance considerations we introduced in Section 2.2.3. By analyzing each implementation and comparing them to each other, we derive promising implementations strategies and use those results to create further optimized implementations. At the end of this process, we want to have two results, a set of good implementation strategies and a highly optimized implementation that yields a very high performance. Both results are needed for our approach to generate high performance CUDA code automatically for specific applications in Chapter 4.

We started our experiments with the well-known GEMM. Our research showed that finding an optimized implementation is no trivial task, even for seemingly simple problems. In order to find a high performance solution, more then 20 different implementations were necessary. Table 2 shows a list of these implementations. Additionally, our experiments showed that there is still potential for optimizations, even in well researched and very common applications. Our implementations of GEMM yield a higher performance than other state-of-the-art implementations. MAGMA [1] and NVIDIA's CUBLAS [31], two popular CUDA implementations of Basic Linear Algebra Subprograms (BLAS), were used for comparison.

Because of the intensive work on GEMM, it was not possible to analyze other computational problems at the same level of detail within the context of this thesis. The study of other problems is future work. However, the knowledge gained by optimizing GEMM can be used to optimize similar problems. In Chapter 4 we apply the same optimization strategies to tensor contractions, a general form of GEMM.

Our experiments were conducted with the hardware specified in Table 1. Because the same code may yield a vastly different performance on different hardware architectures, we used two different hardware architectures, namely Tesla and Fermi (see Section 2.2.1.3). We optimized the code for both architectures. CUDA Version 4.1 was used in the experiments.

	9800 GTX SSC	GTX 560 Ti
Codename	Tesla	Fermi
Clock (shader)	1944MHz	1645MHz
Memory	768MB	1024MB
Multiprocessors (SMs)	16	8
Scratchpad per SM	16kB	16kB/48kB
L1 data cache per SM	0kB	48kB/16kB
Peak performance	746 GFLOPS	1263 GFLOPS

Table 1: NVIDIA GPUs used in experiments

3.1 GEMM

The General Matrix-Matrix Multiplication (GEMM) is a fundamental routine in the field of dense linear algebra, which is used in many numerical algorithms [3]. Because of the importance of GEMM, a lot of highly optimized hardware specific implementations exist. Major hardware vendors offer BLAS libraries specialized for their products, for example Intel's MKL [19], AMD's ACML [4] or NVIDIA's CUBLAS [31]. There are also open source implementations like ATLAS [2], GotoBLAS [12] or MAGMA [1]. In this section, we explain how GEMM works and show a possibility for its parallelization.

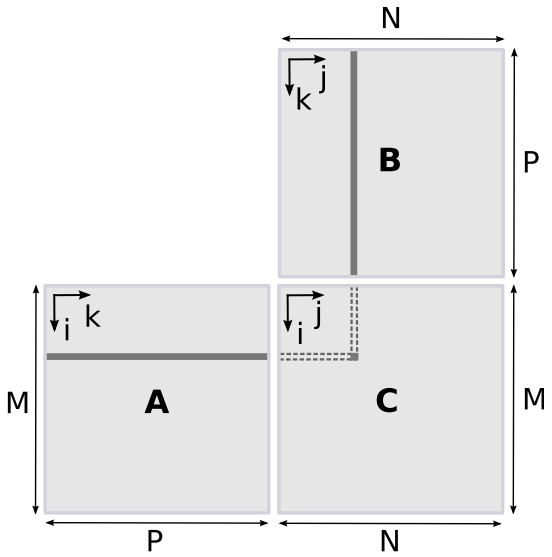

Figure 5: GEMM

GEMM is given by $C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C$, where A is a matrix of size MxP, B is a matrix of size PxN, C is a matrix of size MxN and α and β are scalar parameters. The calculation of the new value of each element of the matrix C is based on its old value and the original content is overwritten. For each element $C_{i,j}$ of C, we have to compute

$$C_{i,j} = \alpha \cdot \sum_{k=0}^{P-1} A_{i,k} \cdot B_{k,j} + \beta \cdot C_{i,j}$$

The calculation of each element of C requires $2 + P$ multiplications and $1 + P$ additions, giving a complexity of $\mathcal{O}(N^3)$ for quadratic matrices with $M = N = P$. Figure 5 illustrates GEMM. To calculate one value of C, the current value of C, one row of A and one column of B are required.

Elements of C have no dependencies on other values of C, i.e., elements of C can be calculated independently from each other. Therefore, we are able to parallelize GEMM by assigning all elements of C to different threads. Threads are able to calculate their values of C independently from other threads and can be executed in parallel. This is called data parallelism, because all threads perform the same instructions on different data, i.e., values of A and B. Although the values of C are independent from each other, there are synergies between threads we want to exploit to improve the performance. For example, all elements in a row of C require the same values from the same row of A for their calculations. Since load operations from the device memory to the on-chip memory are expensive, finding a way to reduce load instructions has a positive effect on the performance. One possibility is to load values of A only once within a block and store them in the scratchpad memory. This way, all threads within a block can reuse the values of A.

Figure 6 shows a schematic draft of a possible parallelization of GEMM that focuses on such synergies. Each value of C is assigned to a specific thread. In CUDA, threads are identified by their coordinates in the grid (see Section 2.2.2.1). In our example we use a two-dimensional grid which contains $gridSize = gridDim.x \cdot gridDim.y$ two-dimensional blocks. Each block can be identified using the coordinates blockIdx.x (bX) and blockIdx.y (bY) and contains $blockSize = blockDim.x \cdot blockDim.y$ threads. Threads within a block are identified using the threadIdx.x (tX) and threadIdx.y (tY) coordinates.

We apply a tiling to C and split it into $blockDim.x \cdot blockDim.y$ equally sized pairwise disjoint subsets of adjacent elements. We use these subsets TC as our blocks.

$$\begin{aligned} TC_{bX,bY} = \{C_{i,j} \mid & bX \cdot blockDim.x \leqslant i < (bX+1) \cdot tX, \\ & bY \cdot blockDim.y \leqslant j < (bY+1) \cdot blockDim.y\} \end{aligned}$$

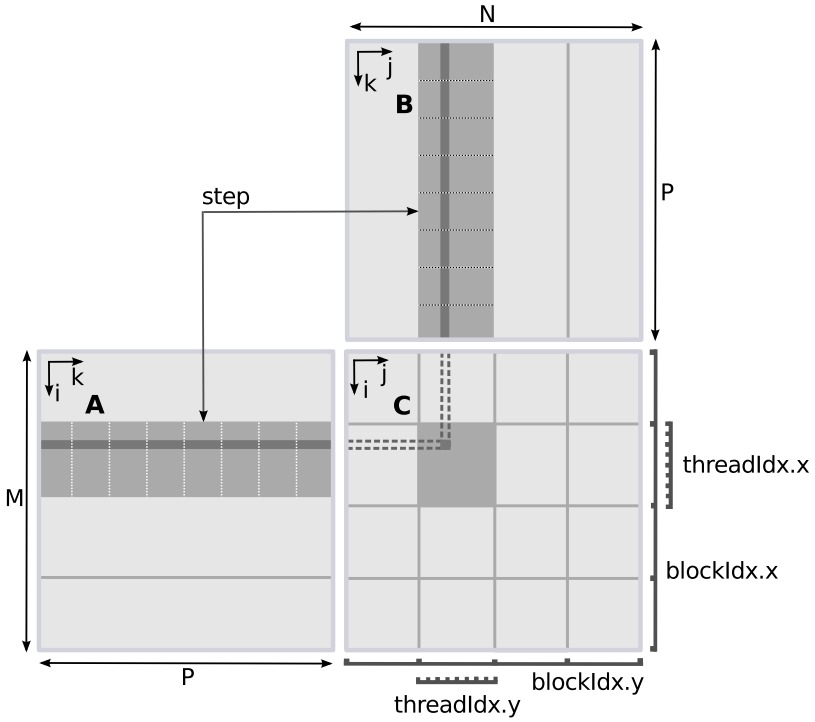

Figure 6: Parallelized GEMM

In the example in Figure 6 we define $\texttt{blockDim.x} = M/4$ and $\texttt{blockDim.y} = N/4$, giving us a tiling of C into 4x4 blocks. The elements of C in each block are then distributed between all threads in a block. Since all threads should follow the same control path, the number of elements of C per block should be a multiple of the number of threads per block. If this is the case, we can use the thread coordinates as parameters within the same instructions. If it is no multiple, we need to use branching, which results in a slower performance on SIMT architectures. Hence, we will only consider cases in which it is a multiple, especially because we are free to choose a fitting tiling to avoid branching. It is also arithmetical possible that the host which calls the kernel extends the dimensions of the matrices to the next multiple using zero values. The kernel would then carry out more work, but might still be faster because branching is avoided.

Several ways exist to assign values of C to threads. VT contains the elements of C that are assigned to a thread. If the number of elements of C per thread equals the number of threads per block, we can directly use the thread coordinates as mapping.

$$VT_{TB_{bX,bY},tX,tY} = \{(TC_{bX,bY})_{x,y} \mid x = tX, y = tY\}$$

If the number of threads per block does not equal the number of elements of C per block, we need to calculate multiple elements of C per thread. We designate lX and lY as the number of multiple

calculations a thread has to perform in the according dimensions. We can then assign adjacent elements to the thread.

$$VT'_{TB_{bX,bY},tX,tY} = \{(TC_{bX,bY})_{x,y} \mid tX \leqslant x < lX + tX, \\ ty \leqslant y < lY + tY\}$$

Another possibility is to use a cyclic assignment of elements to a threads within a block.

$$VT''_{TB_{bX,bY},tX,tY} = \{(TC_{bX,bY})_{(tX+lX\cdot x),(tY+lY\cdot y)} \mid \\ 0 \leqslant x < \frac{blockDim.x}{lX}, \\ 0 \leqslant y < \frac{blockDim.y}{lY}\}$$

It is also possible to assign statements cyclic in one dimension and adjacent in the other dimension.

After assigning all elements of C to specific threads, we can use the scratchpad memory to reduce the number of load instructions. Thereby, each thread in a block loads distinct values of A and B from the device memory and stores them in the scratchpad. After a synchronization, all threads of the block have access to all loaded values. This technique greatly reduces the number of overall load instructions, because values can be reused within a block.

Scratchpad memory is limited, so we cannot load all required values of A and B at once for big matrices. We need to split the calculations into steps. The number of calculations per step equals the stepsize. Again, we can use zero values if the stepsize is no multiple of P. We load data from A and B required for one step, calculate and store the intermediate value in a register. We also need to synchronize before and after the write accesses to the scratchpad memory.

Listing 1 shows a simple implementation of GEMM that follows the described concept. It uses a block containing 32x8 threads (blockDim.x = 32, blockDim.y = 8), each thread calculating 4 elements of C (lX = 4, lY = 1) and calculates 8 intermediate values per step (stepsize = 8). The parameters lda, ldb and ldc determine the leading dimension, i.e. the layout, of the arrays.

Listing 1: Simple CUDA implementation of GEMM. 32x8 blocksize, 32x32 tilesize, 8 stepsize, 4 calculations per thread, load A and B

```
/* threads(32,8), grid(M/32,N/32) */
extern "C" __global__ void
sgemm_custom_kernel(float *C, const float *A, const float *B,
                    int M, int N, int P, int lda, int ldb,
                    int ldc, float alpha, float beta)
{
    /* Allocate scratchpad memory */
    __shared__ float A_shared[32][8];
    __shared__ float B_shared[32][8];

    /* Allocate regsiters for calculations per thread */
    float C_thread[4] = {0,0,0,0};

    /* Adjust the starting position of the data arrays */
    A += blockIdx.x * 32;
    B += blockIdx.y * 32 * ldb;
    C += (blockIdx.x * 32 + threadIdx.x) +
         (blockIdx.y * 32 + threadIdx.y * 4) * ldc;

    /* Calculate steps */
    for(int step=0;step<P/8;step++) {
        /* Load data from the device memory to the scratchpad
            memory*/
        __syncthreads();
        A_shared[threadIdx.x][threadIdx.y] =
            A[(threadIdx.x) + (threadIdx.y + 8 * step) * lda];
        B_shared[threadIdx.x][threadIdx.y] =
            B[(threadIdx.y + step * 8) + (threadIdx.x)*ldb];
        __syncthreads();

        /* Calculate 8 intermediate values for each of the
           calculations per thread */
        for(int k=0;k<8;++k) {
            float a = A_shared[threadIdx.x][k];
            for(int j=0;j<4;++j) {
                float b = B_shared[threadIdx.y * 4 + j][k];
                C_thread[j] +=  a * b;
            }
        }
    }

    /* Write the results from the register to the device memory*/
    for(int j=0;j<4;++j){
        C[j * ldc] = beta * C[j * ldc] + alpha * C_thread[j];
    }
}
```

Table 2: List of SGEMM implemenations

Id	Name	blockDim.x	blockDim.y	gridDim.x	gridDim.y	stepsize	lX	lY	min regs	scraptchpad size	assignment	peak Tesla	peak Fermi
1	sgemm-8-8.cu	8	pY	M/BlockSize	N/8	K/(blockDim.y*pL)	1	8	14	BlockSize*pL	adjacent	10.87%	17.93%
2	sgemm-16.cu	pX	pY	M/BlockSize	N/16	K/((pX*pY)/pL)	1	16	22	16*((pX*pY)/pL)	cyclic in N	9.79%	13.80%
3	sgemm-16-alt.cu	16	pY	M/BlockSize	N/16	K/(blockDim.y*pL)	1	16	22	BlockSize*pL	adjacent	16.47%	21.03%
4	sgemm-24-alt.cu	24	pY	M/BlockSize	N/24	K/(blockDim.y*pL)	1	24	30	BlockSize*pL	adjacent	15.40%	8.76%
5	sgemm-32-alt.cu	32	pY	M/BlockSize	N/32	K/(blockDim.y*pL)	1	32	38	BlockSize*pL	adjacent	11.95%	23.43%
6	sgemm-64-alt.cu	64	pY	M/BlockSize	N/64	K/(blockDim.y*pL)	1	64	70	BlockSize*pL	adjacent	4.44%	9.58%
7	sgemm-8.cu	pX	pY	M/BlockSize	N/8	K/(blockSize/pL)	1	8	14	8*(BlockSize/pL)	cyclic in N	10.86%	20.10%
8	sgemm-8-alt.cu	8	pY	M/BlockSize	N/8	K/(blockDim.y*pL)	1	8	14	BlockSize*pL	adjacent	15.19%	18.05%
9	Sgemm-nooptimize.cu	pX	pY	M/BlockSize	N/16	K/(blockSize/pL)	1	16	22	16*(BlockSize/pL)	cyclic in N	18.84%	22.00%
10	sgemm-norest.cu	pX	pY	M/BlockSize	N/16	K/(blockSize/pL)	1	16	22	16*(BlockSize/pL)	cyclic in N	27.83%	13.93%
11	sgemm-16x16.cu	16	16	M/256	N/16	128	1	16	22	2048	adjacent	38.23%	21.67%
12	sgemm-AB_4local.cu	32	8	M/32	N/32	32	1	4	10	2048	adjacent	6.54%	17.37%
13	sgemm-AB_4local_1l.cu	32	8	M/32	N/32	8	1	4	10	512	adjacent	11.97%	10.86%
14	sgemm-AB_4local_2l.cu	32	8	M/32	N/32	16	1	4	10	1024	adjacent	6.97%	11.04%
15	sgemm-32x32.cu	32	2	M/32	N/32	32	4	4	22	2048	adjacent	-	29.96%
16	sgemm-48x48.cu	16	4	M/48	N/48	16	6	6	42	1536	cyclic in M,N	8.14%	29.23%
17	sgemm-48x48-32k.cu	16	4	M/48	N/48	32	6	6	42	3072	cyclic in M,N	-	17.99%
18	sgemm-64x64.cu	32	8	M/64	M/64	16	4	4	22	2048	adjacent	-	29.87%
19	sgemm-64x64-2.cu	64	4	M/64	N/64	16	4	4	22	2048	adjacent	-	37.68%
20	sgemm-72x72.cu	16	9	M/72	N/72	16	6	6	42	2304	adjacent	-	14.82%
21	sgemm-80x80.cu	16	10	M/80	N/80	16	4	10	46	2560	adjacent	-	5.92%
22	sgemm-96x48.cu	16	8	M/96	N/48	16	6	6	42	2304	adjacent	-	3.51%
23	sgemm-96x96.cu	16	16	M/96	N/96	16	6	6	42	3072	adjacent	-	8.07%
24	sgemm-96x96-6x6.cu	16	16	M/96	N/96	16	6	6	42	3072	adjacent	-	16.75%
25	sgemm-96x96-6x6-opt.cu	16	16	M/96	N/96	16	6	6	42	3072	cyclic in M,N	-	35.51%
26	CUBLAS											37.53%	35.64%
27	MAGMA											37.55%	35.61%

Table 2 shows a list of GEMM implementations that are based on the same principle illustrated in Figure 6. At a conceptional level, each implementation differs in the parameters used for the assignment of elements of C to specific threads and the stepsize. The table shows those parameters for each implementation. Additionally, "scratchpad size" denotes the number of bytes used as scratchpad memory for each block and "min regs" denotes the minimal number of registers which are required by each thread. The minimal number of registers depends on the number of elements of C that are assigned to each thread, i.e. $lX \cdot lY$, on required pointers to the memory and on the number of loops with parametric bounds that can not be fully unrolled at compile time. Since the compiler is in charge of the register management, "min regs" can only be used as an indicator for the actual number of registers the compiler uses for each thread.

In Listing 1 we assume that the minimal number of registers is 10. It requires 4 registers to store C_thread, 5 registers to store the the pointers to the memory (A, B, C, A_shared, B_shared) and one register for the iteration variable step. This is of course a very rough estimate.

Table 2 also shows the percentage of the peak performance an implementation achieves on our test hardware. On the Tesla device some implementations could not be executed, because the resource requirements of a kernel could not be met.

Some implementations, i.e., implementations 1-10, are parameterized and allow us to search a wider space of codes for possible high performance candidates. The peak performance in the table for those implementations is the best performance achieved. Implementations with fixed parameters are optimized implementations of a possible high performance candidate.

We see a large difference in the performance of the implementations. On Tesla the lowest achieved performance is 4.4% of the theoretical peak performance and the highest performance is 38.23%, making a difference of 33.83 percentage points. On Fermi the best performance is 35.94% with a difference of 32.43 percentage points to the lowest performance. We also see that the best implementation for Tesla achieves only average results on Fermi.

Although all implementations are based on the same concept, selecting the optimal parameters, i.e., the optimal assignment of elements of C to threads is crucial for high performance. In order to find the optimal assignment we have to consider all aspects described in Section 2.2.3. One aspect which makes the optimization difficult is latency hiding. We can either increase the workload per thread, which results in an increased register and scratchpad memory usage, or we can optimize the distribution of warps among SMs, which restricts the available registers and scratchpad memory per thread. Finding an optimal balance between those opposing factors is crucial. Additionally, other aspects like memory access patterns, e.g., for coalesced

memory access or for memory banks, lay further restrictions on the implementations.

In order to find an optimized implementation of GEMM, we need to search the space of all possible codes in a target-oriented manner and identify configuration with a potential for high performance. We are then able to implement an optimized version of those configurations.

3.2 EXPLORATION OF THE CODE SPACE

The first step in this process is to implement parameterized versions of GEMM that allow us to easily change crucial characteristics. We can then use this parameterized versions to test the performance of different configurations automatically. The most promising configurations found by the parameterized implementations are the candidates for optimization in Section 3.3.

3.2.1 *Tesla Architecture*

Implementation 3, called sgemm-16-alt.cu, from Table 2 is a parameterized version of GEMM. It has two parameters, pY and pL. Each thread calculates 16 vertical elements of C, i.e., $lY = 16$ and $lX = 1$. This non-quadratic pattern is the consequence of the matrices being in column-major order. For each element of C we need to load all values of a row of A and a column of B. Since B is in column-major order we can read consecutive values of B and leverage coalesced data accesses to the device memory. To maintain the generic approach of this implementation, we store only values of B in the scratchpad memory. Values of A are read by each thread directly from the device memory.

The number of threads in a block can be changed with the parameter pY. In this implementation threadDim.x is fixed to 16 and threadDim.y is set to pY upon execution. If pY is increased, more threads are allocated to a block and more elements of C are calculated within a block, resulting in a higher amount of required scratchpad memory. The second parameter pL can be used to change the stepsize, i.e. the number of values of B each thread loads from the device memory to the scratchpad memory per step. The more intermediate values are calculated in a step, the more independent instructions exist within a thread. Increasing the stepsize also requires more scratchpad memory. With the parameters pY and pL we can change the work per thread and the required amount of scratchpad memory. Hence, we can conduct experiments on latency hiding.

Figure 7 shows the performance of implementation 3 on quadratic matrices of the size 2048 using different configurations. On the x-axis we see the blocksize which was used in the configuration. Parameter

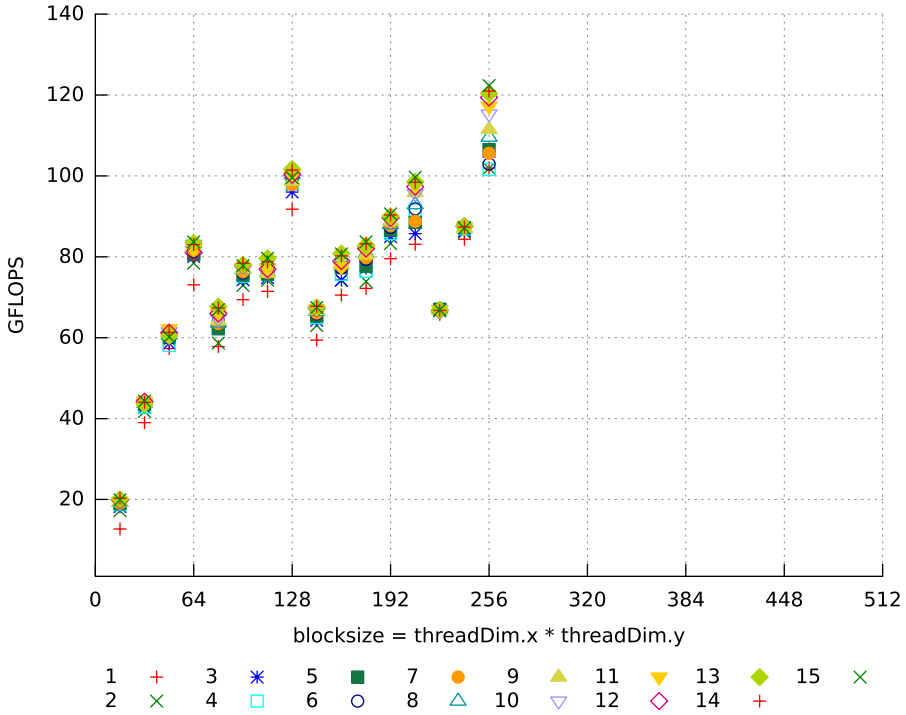

Figure 7: Experiment: Tesla, sgemm-16-alt.cu, threadDim.x=16, M=N=P=2048

pY has a direct influence on the blocksize, because in this implementation threadDim.y equals pY. Parameter pL is illustrated by points of different shapes. In this experiment, we tested $0 < pY \leqslant 32$, because on Tesla devices the maximum blocksize is 512, and $0 < pL < 16$ due to memory restrictions. Not every combination of pY and pL is valid or can be executed on the Tesla device, those combinations are not shown in the graph.

At first, we see that we achieve the best performance with a blocksize of 256 threads, i.e., $pY = 16$, and $pL = 15$. We mark this configuration for later optimization. We also see that parameter pY has a stronger effect on the performance then pL. This indicates that, for this implementation, finding a suitable distribution of warps on the SMs is more important then increasing the number of independent instructions. There are other local maxima at 64, 128 and 208 threads per block. These configurations might also be interesting candidates for further optimization, since some optimization techniques might not be applicable on the best configuration we found.

We conducted many similar experiments with parameterized versions of GEMM, see Figures 18, 19 and 20 in Appendix A. The result of these experiments is a list of characteristics we want to implement in an optimized version of GEMM for Tesla.

Blocksizes of 256, 128 and 64 threads showed the best performance in many experiments. For Tesla, we narrowed down the space of possible codes to those blocksizes, with the priority on 256 threads.

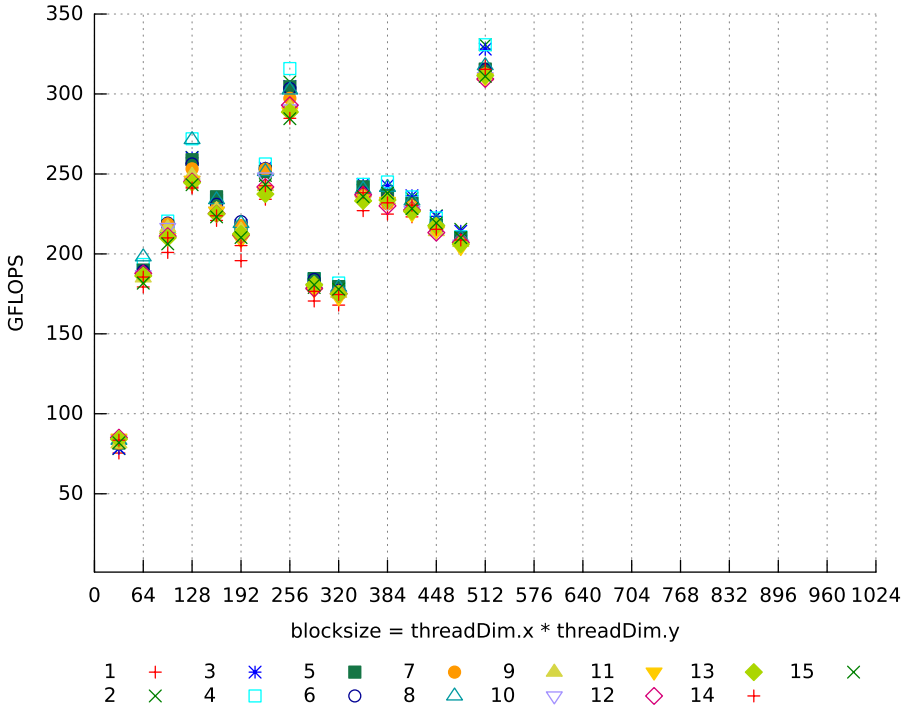

Figure 8: Experiment: Fermi, sgemm-32-alt.cu, threadDim.x=32, M=N=P=4096

Using all threads in a block to load a long sequence of consecutive values of B into the scratchpad memory turned out to be a successful strategy on Tesla. As a result, we want to implement a non-quadratic assignment of elements of C to threads and the usage of all available scratchpad memory for values of B. Hence, we maximise the stepsize. In Section 3.3.2 we evaluate an implementation of GEMM which is based on this result.

3.2.2 *Fermi Architecture*

On the Fermi architecture we follow the same principle to find characteristics of possible high performance implementations as on Tesla (see Section 3.2.1) and we conduct the same experiments.

Figure 8 shows the result of Implementation 3. We see that the best performance is achieved with 512 threads per block, each thread loading either 2 or 4 values from the device memory to the scratchpad memory. We also see similar peaks at a blocksize of 128 and 256 threads. Again, those blocksizes seem to achieve high performance. Figures 24, 25 and 26 in Appendix A show the results of additional experiments.

The most important difference between the Tesla and the Fermi architecture is the introduced cache hierarchy (see Section 2.2.1.3). Implementation 15 was created to leverage the cache by loading values of A and B into the scratchpad and by using a quadratic assignment

of elements of C to the threads. This technique proved to be successful for Fermi devices. Table 2 shows that the best performance of a non-quadratic implementation is 23.43 percent of the peak performance and Implementation 15 achieves 29.96 percent. It is clear that focusing on the cache is crucial for the performance.

Our experiments showed that implementations of GEMM that use 512, 256, 128 or 64 threads per block and fully utilize the cache hierarchy using a quadratic assignment of elements of C to threads seem to have the highest potential for high performance on Fermi.

3.3 OPTIMIZATION

We identified specific characteristics for GEMM which have a strong potential for high performance in Section 3.2. In this section, we present the optimized implementations of GEMM and evaluate if our assumptions were correct.

3.3.1 *Tesla Architecture*

Implementation 11 in Table 2 achieves the highest performance on the Tesla architecture, reaching a higher performance than the state of the art implementations of CUBLAS and MAGMA. Detailed graphs of its performance are shown in Figures 21, 22 and 23 in Appendix A. Listing 3 in Appendix B shows the kernel.

The number of threads per block in Implementation 11 is 256. Each thread calculates 16 consecutive horizontal values of C, therefore, each block calculates 256x16 values of C. The stepsize is 128 and each thread loads 8 values of B from the device memory to the scratchpad memory per step. The threads are divided into two groups, the first 128 threads and the last 128 threads. Each group loads 128 consecutive values of B, resulting in an optimized coalesced memory access. Conflicts on memory banks are minimized in this implementation. Each step can execute a sequence of at least 64 independent instructions per step. Each block requires 2048 bytes of scratchpad memory and each thread requires a minimum of 22 registers. Because this kernel fully utilizes the hardware of the Tesla device, we have to ensure that the compiler does not use more then 32 registers per thread using the compiler flag -maxrregcount=32. If it used more than 32 registers, the device can not allocate enough resources to execute the kernel. Additionally, unrolling the loops significantly improves the performance.

All identified requirements of Section 3.2.1 are met in this implementation with respect to all performance considerations of Section 2.2.3. Table 3 shows that our implementation of GEMM achieves a higher performance then than state-of-the-art implementations. This indicates that we have reached a maximum. However, we can not be sure that

we have reached a global maximum. Further optimizations might be possible.

Table 3: Comparison between CUBLAS and our optimized GEMM for Tesla (M = N = K)

N	256	512	1024	1536	2048	3072	3840
CUBLAS	161.17	254.94	275.52	279.35	281.03	275.17	269.17
optimized	207.94	266.3	281.25	284.56	285.76	287.02	287.43
speedup	22.49%	4.27%	2.04%	1.83%	1.66%	4.13%	6.35%

Benchmarks of Implementation 11 (Figures 21, 22 and 23) show that the dimensions of the matrices have a strong effect on the performance. It is natural that the peak performance can only be achieved if the dimensions of matrix C are a multiple of the thread block, since this is the only way we can avoid branching. But we see that the size of the M dimension has a far stronger effect on the performance than N and P. The reason for this behavior is the non-quadratic assignment of elements of C to threads. If M is no multiple of blockDim.x a many threads are idle during the whole execution of the kernel. If N is no multiple of blockDim.y or P is no multiple of the stepsize, all threads are still able to do some work.

3.3.2 *Fermi Architecture*

Finding an optimized kernel for the Fermi architecture required more implementations of GEMM than for the Tesla architecture. Loading values of matrices A and B into the scratchpad memory, while focusing on coalesced memory access and memory banks, enforces many restrictions to the possible implementations and limits the space of possible codes tremendously. For example, the number of elements in each matrix (A, B and C) needs to be a multiple of the warpsize in order to avoid branching. Listing 5 in Appendix B shows a script which uses these restrictions to determine possible configurations. In Table 4 we see the result of the script for a quadratic assignment of elements, i.e., lX equals lY.

We restrict our search space to this small number of configurations. Previous experiments (see Section 3.2) show that 512, 256, 128 or 64 threads per block have a strong potential for high performance on the Fermi architecture. Therefore, we selected the corresponding 5 configurations (1, 2, 4, 5, 6) for implementation.

The results from the experiments is that Configuration 6 achieves a similar performance as CUBLAS or MAGMA (35.51% of peak). Configurations 1 and 2 that use only 64 threads per block achieve a surprisingly good performance (29.23% and 29.96% of peak), which approves Volkov [39]. He stated that using less threads and putting more work into a single thread is a good strategy to achieve high

Table 4: Possible quadratic configurations for GEMM

id	threads	tileDim.x	tileDim.y	stepsize	lX	lY
1	64	32	32	8	4	4
2	64	48	48	8	6	6
3	144	48	48	12	4	4
4	256	64	64	16	4	4
5	256	80	80	16	5	5
6	256	96	96	16	6	6
7	400	80	80	20	4	4

performance. The best configuration we found is Configuration 4 (37.68% of peak) and its implementation is illustrated in Listing 4 in Appendix B.

Each of the 16x16 threads of Implementation 19 calculates 4x4 elements of C, therefore each block calculates 64x64 values. A shared memory size of 2048 and a minimum of 22 registers was used. Each thread loads 4 adjacent values of A and B from the device memory to the scratchpad memory. Within a block, loads from B are coalesced. Loads from A leverage the cache hierarchy. In each step a sequence of 64 independent instructions is used for the calculation. We use preloading to increase the number of independent instructions to 74 for the cost of 8 registers. With this implementation we outperform CUBLAS as shown in Table 5.

Table 5: Comparison between CUBLAS and our optimized GEMM for Fermi ($M = N = K$)

N	256	512	1024	1536	2048	3072	3840
CUBLAS	161.95	359.09	425.32	442.63	441.46	440.85	447.97
optimized	265.24	413.15	454.55	461	464.15	466.46	467.49
speedup	38.94%	13.08%	6.43%	3.98%	4.89%	5.49%	4.18%

Implementations 23 - 25 are based on the same configuration, however, they achieve a very different performance. This is caused by different memory access patterns to the device memory and the scratchpad memory. Only one of three implementations is able to leverage the cache hierarchy of the Fermi devices. We see that not only the optimal configuration is important, access patterns to the memory are also crucial for high performance.

4 AUTOMATICALLY PARALLELIZED EXPERIMENTS

In the previous Chapter 3 we optimized GEMM for high performance. Based on hardware specific performance considerations, we searched the space of possible implementations in a target oriented way and identified characteristics that are crucial for high performance. The assignment of the elements of the result tensor C to specific threads is a fundamental aspect for the performance of GEMM and we identified different patterns to distribute the workload among the threads. In this chapter we show a concept that allows us to apply these patterns to a high-level description of GEMM in order to generate CUDA code automatically. Moreover, we do not limit this concept to GEMM, but we extend it to tensor contractions, a generalization of GEMM. We are then able to apply the same patterns that yield high performance on GEMM to other tensor contractions.

We use the polyhedron model to describe tensor operations and express the patterns as transformations of the model. The tool Treduda implements this concept and it is able to generate CUDA code automatically for tensor contractions. The GEMM implementations generated with Treduda achieve the same performance as the manually optimized codes. On the Fermi architecture, the generated implementation even outperforms the manually optimized implementation.

In Section 4.1 we discuss related work. Section 4.2 presents Treduda and the concept used for code generation. The benchmarks of the generated codes are shown in Sections 4.3 and 4.4.

4.1 RELATED WORK

With the increasing distribution of multi-core CPUs arose the requirement to generate parallelized codes automatically. Software developers should be able to utilize the computational power of the hardware without having to deal with platform specific details. Polyhedral compilation is being used successfully in this domain [35, 17].

With the distribution of modern GPUs that support GPGPU comes the same requirement. Compilers should be able to utilize the additional computational power of these devices, however, generating a kernel that yields acceptable performance is not easy (see Section 2.2.3). Different approaches exist to search the space of possible codes for good solutions and to optimize the code.

Baskaran et al. [5] implemented CUDA support for PLuTo [11] focusing on coalesced memory access to the device memory and conflict-free access to the scratchpad memory. They use empirical search to

find good parameters for unrolling and tiling and report reaching up to 96% of the performance of CUBLAS for GEMM. Ryoo et al. [34] derive metrics from static code and are able to reduce the optimization space by up to 98%. The PoCC compiler [32] is able to explore the space of legal polyhedral transformations in order to find good transformations. Another approach is to analyze the program inputs. Liu et al. [25] use this technique to find good codes. The unrolling of inner loops [27], the management of data transfers between the device memory and scratchpad memory [18] and dealing with multiple levels of parallelism [6] are also crucial for high performance.

Lee et al.[23] presented a compiler framework that translates OpenMP to CUDA code. The tool C-to-CUDA [7] is able to generate CUDA code from sequential C code. It uses PLuTo as polyhedral framework and ClooG to generate code. The performance of the generated code is reported to be "quite close" to hand-optimized CUDA code.

Mint [36] is a programming model that is specialized on 3D stencil methods. It generates CUDA code from annotated C code and achieves 80% of the performance of hand-optimized code.

Efforts are made to optimize the performance of tensor contractions. The tensor contraction engine is able to generate parallel C or Fortran code (OpenMP, MPI) from a high-level description of a tensor contraction [8, 9]. Ma et al. [26] created a framework that maps tensor contrations to a cluster of GPUs. They focus on the management of data movement between levels of a memory hierarchy.

The focus of our work is polyhedral code generation for specific problems on specific architectures. We try to reach or exceed the performance of hand-optimized code using transformations which are optimized for these specific problems. Our work complements previous efforts by giving examples of good transformations.

4.2 TREDUDA

The tool Treduda automatically generates CUDA code for tensor operations, especially tensor contractions. It requires a high-level description of a tensor operation as input and the user can choose among a predefined set of generation strategies. Each strategy is parameterized and is able to generate a set of implementations. All informations required for the code generation can be deduced from the high level description.

Tensor operations are expressed as operations on multidimensional arrays and we use at least two arrays for every tensor operation. The user defined operations of the high level description are performed in array O. Array R is used as input and to store the final result. All values of R are calculated in the form of

$$R_{a_1,a_2,\ldots,a_n} = \alpha \cdot O_{b_1,b_2,\ldots,b_n} + \beta \cdot R_{a_1,a_2,\ldots,a_n}$$

overwriting the original contents. The scalar parameters α and β are scaling factors. In order to use Treduda all elements of R have to be independent from each other, i.e., the calculation of one value of R must not effect other values of R. In Section 3.1 we expressed GEMM as

$$C_{i,j} = \alpha \cdot \sum_{k=0}^{P-1} A_{i,k} \cdot B_{k,j} + \beta \cdot C_{i,j}$$

We are able to simplify this expression by applying the definition above. Additionally, we can eliminate the summation symbol using the Einstein notation, a convention which states that an index is implicitly summed if it appears in two tensors that multiply each other. GEMM can then be expressed as

$$C_{i,j} = A_{i,k} \cdot B_{k,j}$$

All tensor contractions can be expressed in this simple form. Another example is the Riemann-Christoffel curvature tensor which is useful in general relativity.

$$C_{i,j,k,l} = A_{i,j,k,l} - A_{i,l,k,j} + B_{m,j,k} \cdot B_{i,m,l} - B_{m,l,k} \cdot B_{i,m,j}$$

From this high-level description of a tensor operation, we are able to deduce a representation in the polyhedron model (see Section 2.1.2). A predefined set of strategies is used to transform this model in a structure apt for GPUs. We can then use the transformed model as input for a polyhedron code generator. Finally, we adapt the generated code to CUDA.

4.2.1 *External Tools*

Treduda uses ISL [38] and ClooG [37]. ISL is a library that is able to handle sets and relations of integer points. We use it as implementation of the polyhedron model. The tool ClooG is a polyhedral code generator. Based on a polyhedron model represented in ISL, it is able to generate a corresponding loop structure.

4.2.2 *Concept*

Treduda uses a high-level description of a tensor operation to generate CUDA code. We represent the high-level description as a tree structure. Figure 9 shows the representation of GEMM. Each node of the tree has the attributes type, name and vars. Some nodes have additional attributes that are specific to the type, e.g., a node of the type Write requires the name of a data array. The attribute vars specifies the iterators which are required in each node. Only the iterators of the leaf nodes are given, but we are able to deduce the required iterators for the inner nodes, e.g., the node S_2 requires all iterators of its child

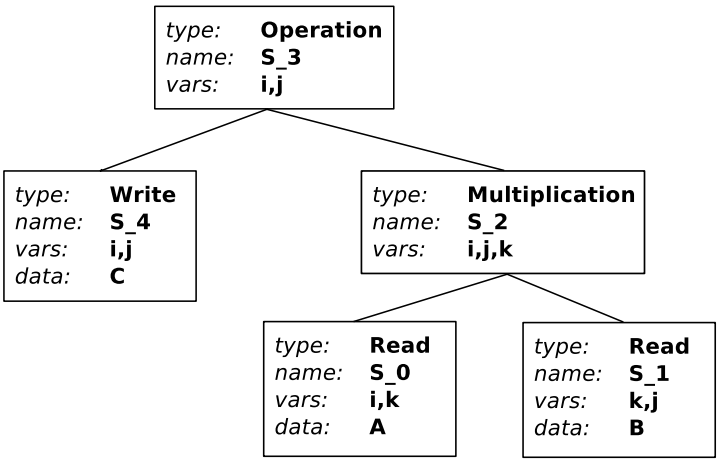

Figure 9: Internal tree representation of GEMM

nodes. We identify the semantic meaning of an element by its type and its context. For example, an element of type Multiplication could be a scalar multiplication or a matrix multiplication. In Figure 9 we can deduce that node S_2 is a matrix multiplication, because its parent does not require the iterator k. Basically, we apply the Einstein notation. Notice that the root node is always of the type Operation and that the only node of type Write is its first child node.

This representation of tensor operations can be transformed into a polyhedron model. We use all iterators that are used in the tree as our iteration domain D.

$$D = \{(i, j, k) \mid 0 \leqslant i < M, 0 \leqslant j < N, 0 \leqslant k < P\}$$

Each statement $S_i = (T_i, D, \Theta_i) \in P$ uses the same iteration domain but a different schedule Θ_i and executes a different set of instructions T_i. Multiple possibilities exist to translate the semantics of the tree to P, i.e., to the instructions T_i and their order of execution Θ_i. A generation strategy specifies a specific mapping between the high-level description in form of a tree and P.

Figure 10 shows one possible representation of GEMM in the polyhedral model. Five statements are required to express the semantic of the high-level description. S_1 is used to define and initialize a temporary variable that stores intermediate results. We need to make sure that the instruction of S_1 is executed before any other statement requires the variable and that it is initialized only once, i.e., we need to specify a order between the statements. A new dimension in the target space of the schedule is introduced for that cause. The other statements correspond directly to nodes in the tree. S_2 corresponds to node named S_0, S_3 to S_1, S_4 to S_2 and S_5 to S_3. We can use the polyhedral description in Figure 10 as input for a polyhedral code generator. In Listing 2 we see how the code for this description looks like.

$$D = \{(i, j, k) \mid 0 \leqslant i < M, 0 \leqslant j < N, 0 \leqslant k < P\}$$
$$S_i = (T_i, D, \Theta_i)$$
$$T_1 = \{\text{"float tmp = 0;"}\}$$
$$\Theta_1 = \{(i, j, k) \rightarrow (i, j, 0, 0)\}$$
$$T_2 = \{\text{"float a = A[i][k];"}\}$$
$$\Theta_2 = \{(i, j, k) \rightarrow (i, j, 1, k)\}$$
$$T_3 = \{\text{"float b = B[k][j];"}\}$$
$$\Theta_3 = \{(i, j, k) \rightarrow (i, j, 1, k)\}$$
$$T_4 = \{\text{"tmp += a·b;"}\}$$
$$\Theta_4 = \{(i, j, k) \rightarrow (i, j, 2, k)\}$$
$$T_5 = \{\text{"C[i][j] = } \alpha\text{·tmp + } \beta\text{·C[i][j];"}\}$$
$$\Theta_5 = \{(i, j, k) \rightarrow (i, j, 3, 0)\}$$

Figure 10: Polyhedral representation of GEMM code shown in Listing 2

Listing 2: Code generated from polyhedral representation in Figure 10

```
  for (int i=0; i<M; i++) {
    for (int j=0; j<N; j++) {
S1:   float tmp = 0;
      for (int k=0; k<P; j++)
S2:     float a = A[i][k];
S3:     float b = B[k][j];
S4:     tmp += a·b;
S5:   C[i][j] = α·tmp + β·C[i][j];
    }
  }
```

In order to execute a tensor operation on a GPU we need to transform the polyhedral model in a form that allows us to assign iterations to threads. Each thread is identified by the 4-tuple (blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y). We want to map elements of the result tensor (C) to specific threads. Because of the restriction that all elements of the result tensor have to be independent from each other, each thread can calculate values of the result tensor in parallel. In order to assign elements of the result tensor to threads, we modify the target space of the schedule. The first four dimensions of the target space are associated with the 4-tuple that identifies a thread in CUDA. The values of the result tensor that are mapped to these dimension are assigned to a specific thread. We then remove the loops that correspond to these 4 dimensions from the generated code and replace their iterators with the corresponding variable of the 4-tuple. These values are then executed in parallel.

We will now describe the different generation strategies implemented in Treduda.

4.2.3 *Generation Strategies*

Treduda currently implements 9 generation strategies which are all based on the experiments from Chapter 3. Therefore, they all take a similar approach to generating CUDA code, e.g., all of them use a tiling on two loops in order to distribute the elements of the result tensor among threads. The implementation of other approaches and generation strategies is future work. In Section 4.2.4 we see that Treduda is designed for easy extensibility and how new generation strategies can be added.

As mentioned, all implemented generation strategies use a tiling on two loops, i.e., they distribute two iterators of the result tensor among all threads. Hence, they all use a variation of schedule Θ for $lX = \frac{\texttt{tileDim.x}}{\texttt{blockDim.x}}$ and $lY = \frac{\texttt{tileDim.y}}{\texttt{blockDim.y}}$.

$$
\begin{aligned}
\Theta = \{[\ldots l1 \ldots l2 \ldots] \to [o0, o1, o2, o3, \texttt{order}, o5, \texttt{order}, o7, \ldots] \mid \\
& (1 - \texttt{tileDim.x}) + l1 \leqslant \texttt{tileDim.x} \cdot o0 \leqslant l1, \\
& 0 \leqslant o2 \leqslant \texttt{blockDim.x}, \\
& l1 = \texttt{tileDim.x} \cdot o0 + lX \cdot o5 + o3, \\
& (1 - \texttt{tileDim.y}) + l2 \leqslant \texttt{tileDim.y} \cdot o1 \leqslant l2, \\
& 0 \leqslant o4 \leqslant \texttt{blockDim.y}, \\
& l2 = \texttt{tileDim.y} \cdot o1 + lY \cdot o7 + o4\}
\end{aligned}
$$

The user can select the dimensions l1 and l2 which are to be parallelized. The other dimensions of the iteration domain are executed serially in each thread. Notice that generation strategies have certain constraints, e.g., the result tensor must have a rank $\geqslant 2$. The parameters tileDim.x and tileDim.y specify the size of a tile, i.e., the number elements of the result tensor that are assigned to a thread block. The tileSize has to be a multiple of the blocksize, hence, $lX \cdot lY$ elements of the tiled dimensions are assigned to each thread. In Θ we use the dimensions o5 and o7 to address multiple elements within a thread. In all strategies we use ordering dimensions to define the order of execution between statements. One ordering dimensions is inserted after every dimension of the target space beginning after o3.

4.2.3.1 *Strategy 1*

This strategy is very basic. Each thread calculates multiple elements of the tiled dimensions and all operations are performed in a single statement S. The generated code for GEMM that is executed by each thread has the following form.

```
loop o5 {
  loop o7 {
    loop k {
      S(i,j,k)
    }
  }
}
```

The indices i and j are defined by schedule of the strategy. In this example they are defined as

$$i = tileDim.x \cdot blockIdx.x + lX \cdot o5 + threadIdx.x$$
$$j = tileDim.y \cdot blockIdx.y + lY \cdot o7 + threadIdx.y$$

We see that i and j depend on the coordinates of the thread, hence, all threads access different points in the iteration domain, i.e., elements of the result tensor.

We use the compiler's preprocessor to insert the instructions in the code. In this example we generate the following statements from the tree description of GEMM.

```
#define S_1(i,j,k) B[(k)+(j)*P]
#define S_0(i,j,k) A[(i)+(k)*M]
#define S_2(i,j,k) S_0(i,j,k)*S_1(i, j, k)
#define S_4(i,j,k) C[(i)+(j)*M]
#define S(i,j,k) S_4(i,j,k)=alpha*S_2(i,j,k)+beta*S_4(i,j,k)
```

This strategy requires β to equal 1 to be correct, so it can not be used for tensor operations in general. Nevertheless, it can be used for testing.

4.2.3.2 *Strategy 2*

Strategy 2 is an extension of strategy 1. It uses the same tiling but it separates the final write statement from the other instructions. A two-dimensional temporary array is needed to store the intermediate values of the calculation. In order to forward the mapping between the elements of the result tensor and the temporary array to the statements, we need to extend the iteration domain by two dimensions.

For GEMM the generated code has the following form.

```
loop o5 {
  loop o7 {
    loop k {
      CALC(i,j,k,reg1,reg2)
    }
  }
}
loop o5 {
  loop o7 {
    WRITE(i,j,k,reg1,reg2)
```

```
  }
}
```

In addition to the definitions of Strategy 1 we add CALC and WRITE.

```
#define CALC(i,j,k,reg1,reg2) (tmp[reg1][reg2] += S_2(i,k,j))
#define WRITE(i,j,k,reg1,reg2) (S_4(i,j,k) = beta*S_4(i,j,k) +
    alpha*tmp[reg1][reg2])
```

4.2.3.3 *Strategy 3*

In strategies 1 and 2 we used a fixed amount of statements (1 and 2). This strategy generates a dynamic amount of statements based on the high-level description. Beginning from the innermost contraction in the tree (if existent), it analyzes for each operation what to do with its result. Based on the arithmetical meaning of an operation, the result is either stored in a new temporary variable or it is merged with an existing one. We try to reuse as many temporary variables, i.e., intermediate results, as possible.

The main purpose of this strategy is to implement the algorithm to reuse intermediate values. The tiling is not focused in this strategy, hence, we use a very general one. A single element is assigned to one thread. Strategies 4 to 9 are based on this strategy and apply other tilings.

The code generated for GEMM has the following form.

```
INIT_S_2(i,j,k,reg1,reg2)
loop k {
    INIT_S_0(i,j,k,reg1,reg2)
    S_0(i,j,k,reg1,reg2)
    S_1(i,j,k,reg1,reg2)
    S_2(i,j,k,reg1,reg2)
}
S_4(i,j,k,reg1,reg2)
```

The INIT statements are used to allocate and initialize temporary variables.

4.2.3.4 *Strategy 4, 5, 6*

Different possibilities exist to assign elements of a tile to threads in a block (see Section 3.1). Strategy 4 implements a cyclic distribution, Strategy 6 an adjacent distribution and Strategy 5 is adjacent in one of the two tiling dimensions and cyclic in the other.

4.2.3.5 *Strategy 7*

Strategy 7 is based on Strategy 4 and implements scratchpad memory usage. In order to use the scratchpad memory efficiently we have to

apply specific memory access patterns that come with many restrictions, e.g., the tile and the block need to be quadratic and a multiple of 16 in this strategy.

The contraction dimension is split into equally sized steps. Let s be the stepsize, i.e., the number of iterations of the contraction dimension per step. In each step we load data from the device memory to the scratchpad memory. After all required values for the current step are loaded, we calculate s intermediate values of the contraction.

The code generated for GEMM has the following form.

```
loop o5 {
  SYNC(i,j,k,reg1,reg2)
  loop o9 {
    LOAD_S_1(i,j,k,reg1,reg2)
  }
  loop o11 {
    LOAD_S_0(i,j,k,reg1,reg2)
  }
  SYNC(i,j,k,reg1,reg2)
  INIT_S_2(i,j,k,reg1,reg2)
  loop o7 {
    loop o9 {
      loop o11 {
        INIT_S_0(i,j,k,reg1,reg2)
        S_0(i,j,k,reg1,reg2)
        S_1(i,j,k,reg1,reg2)
        S_2(i,j,k,reg1,reg2)
      }
    }
  }
}
loop o9 {
  loop o11 {
    S_4(i,j,k,reg1,reg2)
  }
}
```

In this strategy we use o5 as iterator for the steps and o7 as iterator the calculations within a step, i.e., $k = o5 \cdot s + o7$. We need to insert barrier synchronizations (SYNC) for the correct access to the scratchpad memory.

4.2.3.6 *Strategy 8*

Strategy 8 extends Strategy 5 to use the scratchpad memory. It is optimized for coalesced memory access to the device memory. This strategy is able to generate code for GEMM which has the same structure as the best manually optimized code for the Tesla architecture.

4.2.3.7 *Strategy 9*

This strategy is based on Strategy 6 and implements scratchpad memory usage. It is able to generate the best code for the Fermi architecture. We also implement a preloading technique in this strategy. Preloading means that we load values from the device memory into the register before we write them into the scratchpad memory. The purpose of this technique is to increase the number of independent instructions (see Section 2.2.3).

```
<< load values from device memory to register >>

loop step {
  << load values from register to scratchpad >>
  << load values from device memory to register >>
  << calculate >>
}

<< load values from register to scratchpad >>
<< calculate >>
<< write result to device memory >>
```

Preloading can be achieved easily in the polyhedral model. We just need to shift the step dimension ($[\ldots, step, \ldots] \rightarrow [\ldots, step + 1, \ldots]$) for the statements that load values from the devide memory.

4.2.4 *Structure*

Treduda is designed as a C++ library. The user defines the high-level description of a tensor operation using a tree of TensorStatement objects. The root of the tree has to be a object of the class TensorRoot. Subclasses of TensorStatement are used to specify the type of each node. The iterators of a tensor operation are represented with the TensorVariable class and data arrays with the TensorData class. We use TensorParameter objects to define the upper bound of iterators and the size of the data arrays. A set of iterators is assigned to each node. Nodes of the type TensorDataAccess can either be read (TensorDataRead) or write (TensorDataWrite) statements and a single TensorData object is assigned to them. Each node in a tree knows the complete context using a reference to a TensorContext object. Nodes of the type TensorOperation are used to represent operations on tensors. Currently, addition, subtraction and multiplication are implemented.

Figure 11 illustrates a class diagram of Treduda. The high-level description of a tensor operation is passed to a CodeGenerator object which generates the CUDA code. CodeGenerator and CodeGeneratorCudaIsl are abstract classes and all generation strategies are implemented as a subclass of CodeGenerator. This way, we can reuse large portions of the code and extend existing strategies.

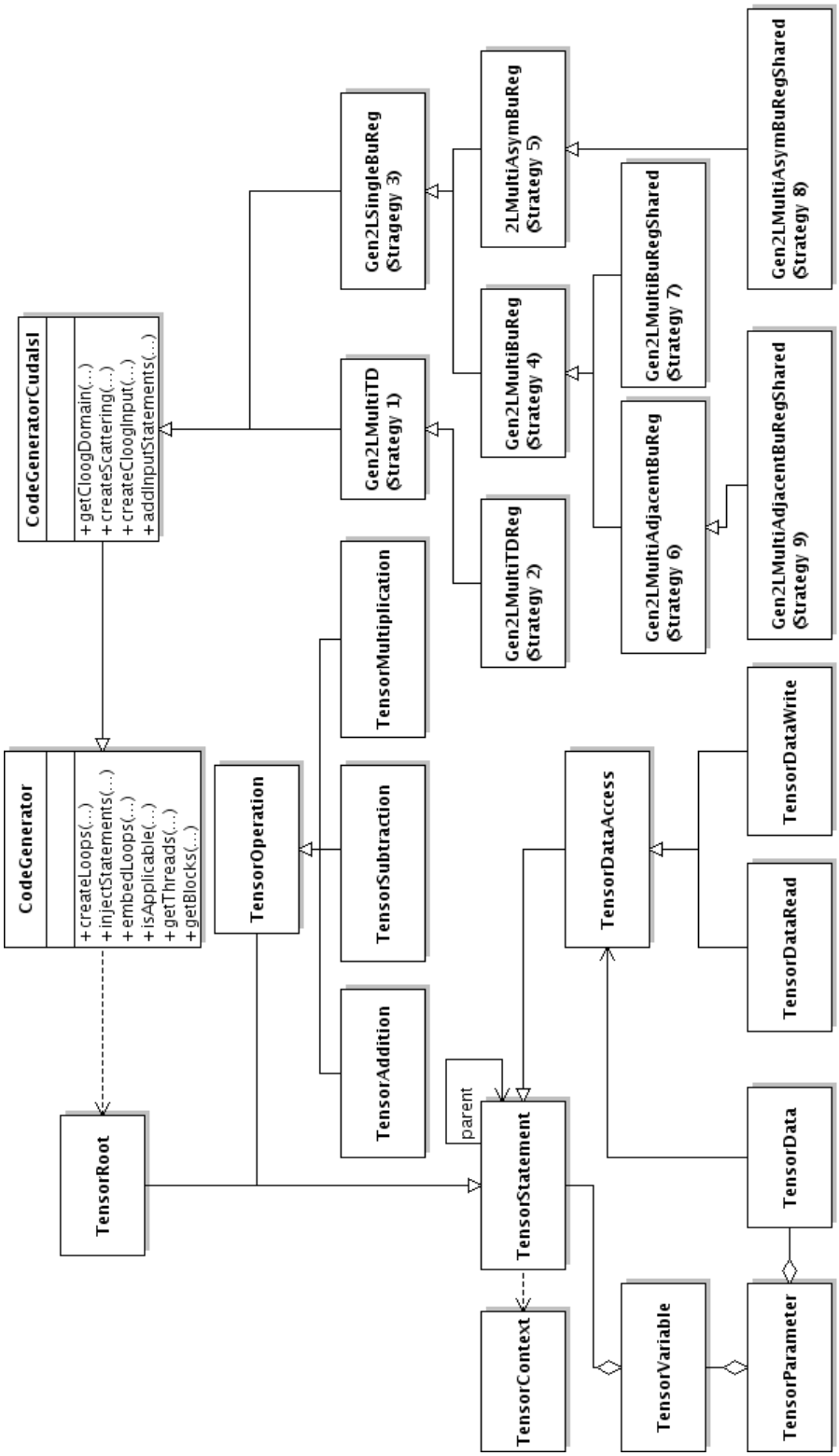

Figure 11: Class diagram Treduda

4.3 GEMM

In the previous Section 4.2 we were introduced to Treduda. The tool implements 9 (see Section 4.2.3) strategies for code generation. Strategies 1 to 6 do not use the scratchpad memory and have less restrictions. Therefore, these strategies can be used for a wider range of tensor operations. The specific strategies 7 to 9 use the scratchpad memory and can only be used to generate code for a limited number of tensor operations. In this section we evaluate the performance of all strategies on the Tesla and Fermi architecture.

The graph in Figure 12 shows the performance of implementations of GEMM that were automatically generated with Treduda using the general strategies 1 to 6. We see that Strategy 4 is able to achieve about 250 GFLOPS, i.e., about 20% of the peak performance of the Fermi device. This strategy uses a cyclic distribution of elements to threads. In contrast, strategies 5 and 6 use a adjacent distribution (in at least one dimension). We also see that strategies that are based on Strategy 3 show a better performance then strategies 1 and 2. Strategies that are based on it try to reuse intermediate values and use another algorithm to generate instructions for statements.

Figure 13 shows a graph of specific strategies on Fermi that use the scratchpad memory. We see that the best strategy in this graph achieves about 450 GFLOPS which is about 35% of the peak performance. This performance equals the performance of MAGMA and CUBLAS (see Table 2). Strategy 8 shows the lowest performance with about 340 GFLOPS. This is still about 100 GFLOPS higher then the best of the general strategies. We also see that the best specific strategy is an extension of the best general strategy. Strategies 7 and 9 both use a quadratic tile and load values from A and B into the scratchpad memory. Strategy 8 uses a non-quadratic tile and loads only values of B into the scratchpad.

The performance of the general strategies on the Tesla device was very low, i.e., about 1% of the peak performance. Figure 14 shows the performance of the strategies that use scratchpad memory. The best strategy in this graph achieves about 25%, the second best Strategy 10% and the worst Strategy 1% of the peak performance. We see that only one of the 9 strategies achieves a good performance. The Tesla device seems to require code with higher degree of specialization. The best strategy uses a non-quadratic tiling and loads only values of array B into the scratchpad memory.

In Figure 15 we compare the best manually optimized codes to the best automatically generated codes for GEMM on the Tesla device. We see that the best implementation is the manually optimized version. MAGMA and CUBLAS reach the second best performance. Both show a very similar curve and it seems that MAGMA uses the implemen-

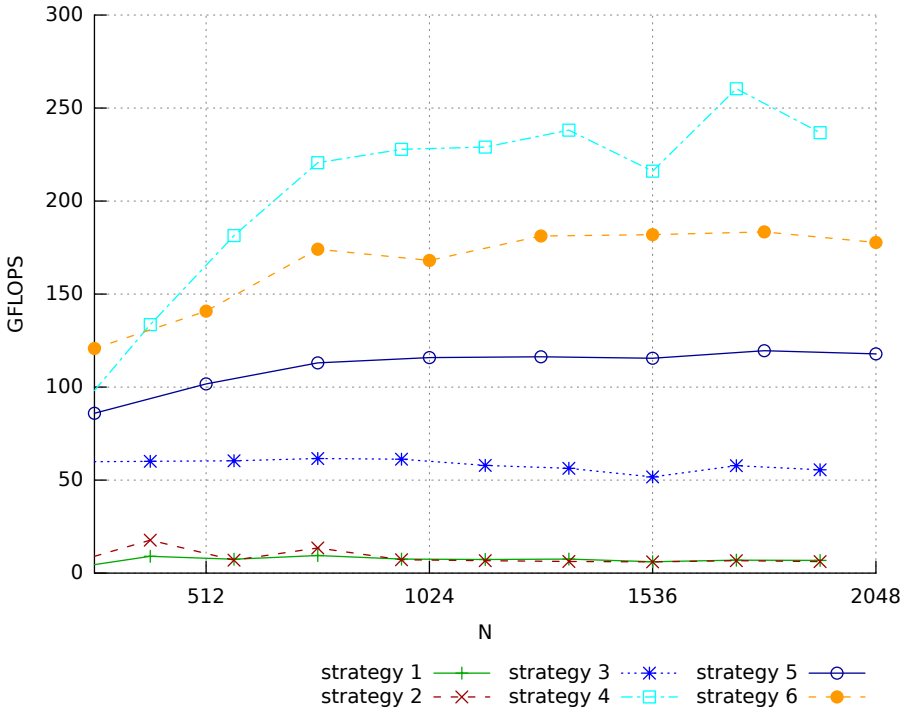

Figure 12: Application of general strategies on Fermi (M = N = K)

tation of CUBLAS. Also, the performance of MAGMA and CUBLAS decreases at a N = 2048. The performance of the generated code shows the lowest performance. It is not able to compete with the other implementations. However, we were able to identify the reason for the lower performance. The generated code has the same loop structure as the manually optimized code, the difference lies in a seemingly superfluous if-clause in the manually optimized code. Listing 3 in Appendix B shows the code of the manually optimized version. The if-clause that has influence on the performance is marked in the code and looks like "if(K % 128 > 0) {". We were not able to produce exactly the same code using polyhedral techniques. Seemingly, the compiler does not use the same optimization for both versions.

Figure 16 shows the best codes on the Fermi device. We have tested the generated code for Fermi in two versions, with and without the usage of the texture memory. We see that the highest performance is achieved by the generated code that uses the texture memory. The manually optimized version is slightly behind. Again, MAGMA and CUBLAS show a similar performance. The codes optimized for the Tesla architecture are not able to compete with the performance of the other codes. However, on the Fermi architecture the manually optimized and the generated code for Tesla show the exactly same performance. Seemingly, the compiler uses the same optimizations here. We see that the usage of the texture memory can have a influence on the performance. But our experiments showed that the texture memory can also decrease the performance.

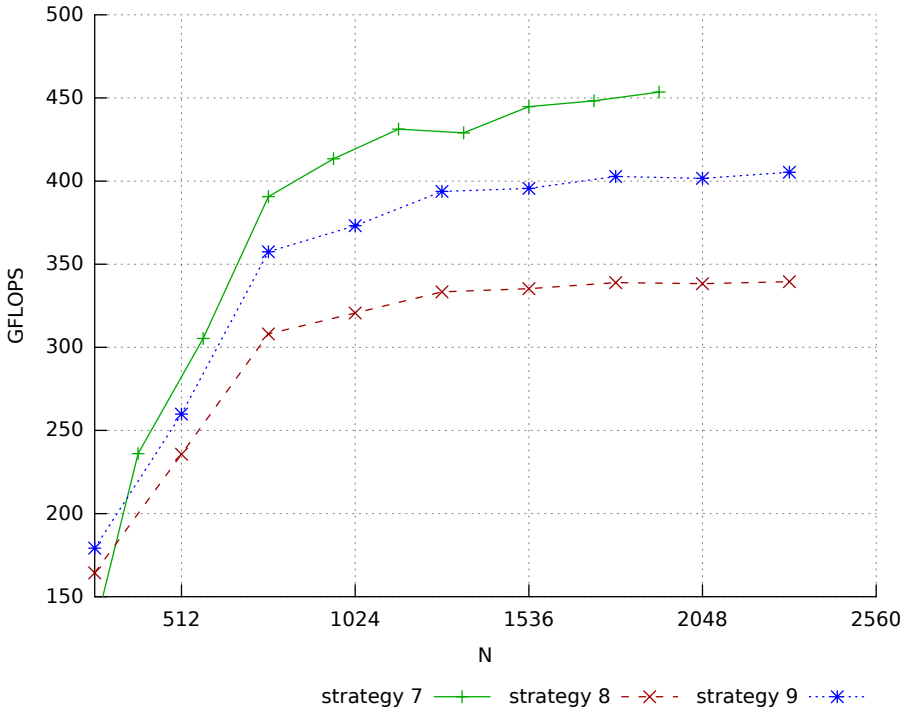

Figure 13: Application of specific strategies on Fermi (M = N = K)

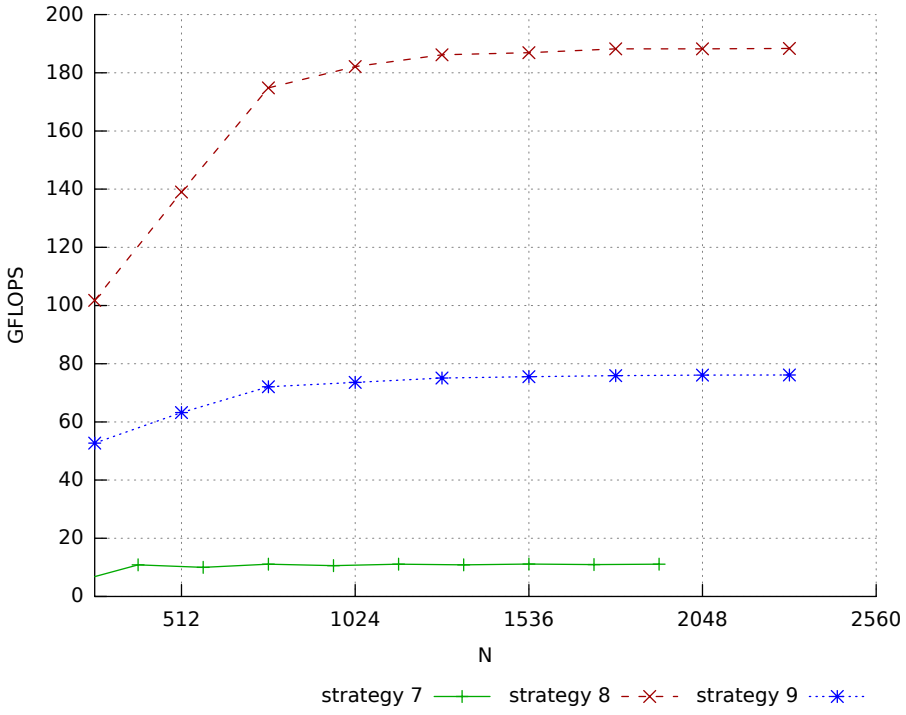

Figure 14: Application of specific strategies on Tesla (M = N = K)

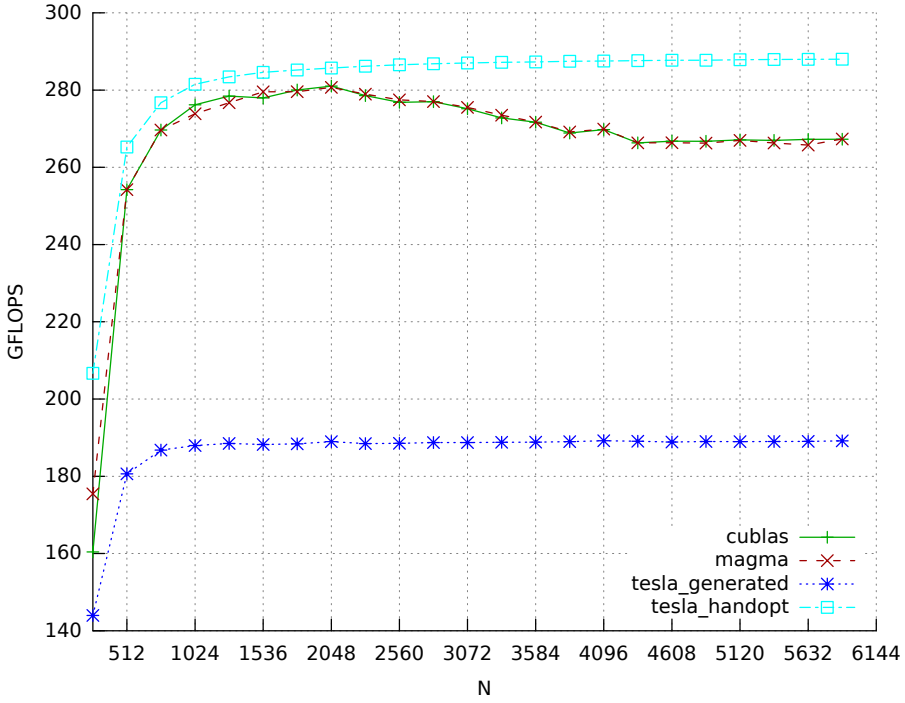

Figure 15: SGEMM benchmark on Tesla (M = N = K)

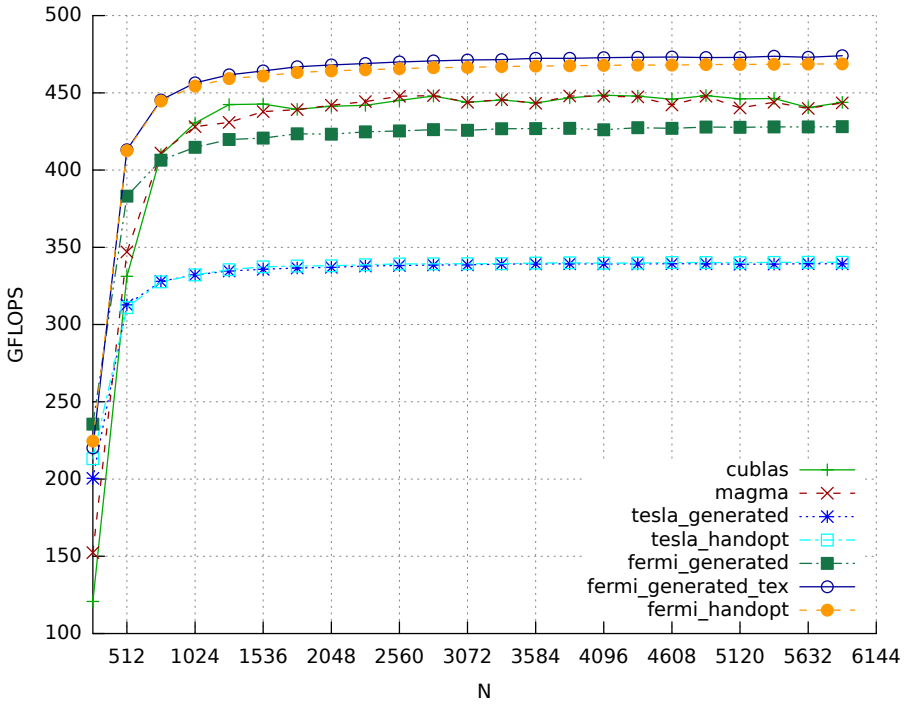

Figure 16: SGEMM benchmark on Fermi (M = N = K)

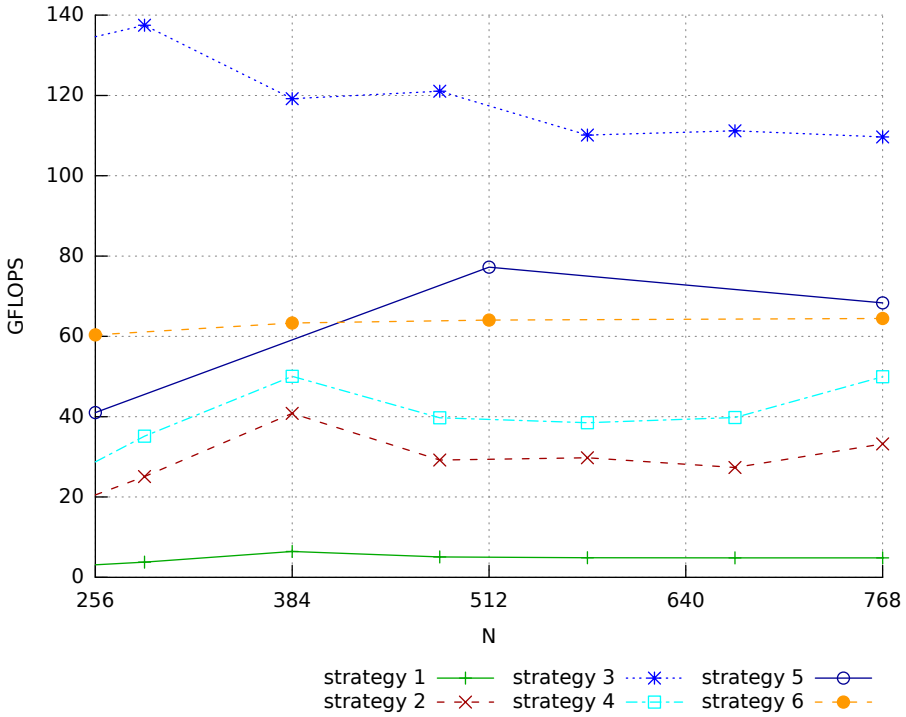

Figure 17: General strategies applied on $S(i,k) = A(i,j) + B(l,j) \cdot C(l,k)$ on Fermi

4.4 TENSOROPERATIONS

We have also tested the performance of other tensor operations generated by Treduda. One example is $S(i,j) = A(i,k) + B(l,k) \cdot C(l,j)$. Figure 17 shows the performance we achieved with the general strategies on Fermi. We see that the performance achieved is not trivial, but we are not able to achieve high performance. As with GEMM, we need to leverage the scratchpad memory to achieve a competitive performance. Unfortunately, our specific strategies can only be used for a limited set of tensor operations.

5 CONCLUSION

The objective of this thesis was to investigate whether it is possible to generate competitive high-performance GPU code using polyhedral techniques.

In a first step, we optimized GEMM manually in order to find the aspects that are crucial for high performance. We wanted to find an implementation that achieves the same performance as state-of-the-art implementations like NVIDIA's CUBLAS or MAGMA. Devices of the Tesla (compute capability 1.x) and the Fermi (compute capability 2.x) architecture were used as the target platforms for our optimizations. We were able to find competitive implementations of GEMM for both architectures. In fact, our implementations of GEMM achieve a 4% higher performance than state-of-the-art implementations.

The next step was to generate GEMM code automatically using polyhedral techniques. We developed the tool Treduda for this purpose. Treduda is able to transform a high-level description of tensor contractions (a generalization of GEMM) into a polyhedral representation. The tool is then able to generate CUDA code from this representation using generation strategies. A generation strategy defines a transformation on a polyhedral model. Based on the results of the manual optimization of GEMM, we developed different generation strategies. Finally, we were able to generate GEMM code that achieves the same performance as manually optimized code. On the Fermi architecture the generated code is even slightly faster then the manually optimized code and outperforms CUBLAS by about 5%.

Part I

APPENDIX

A

PLOTS

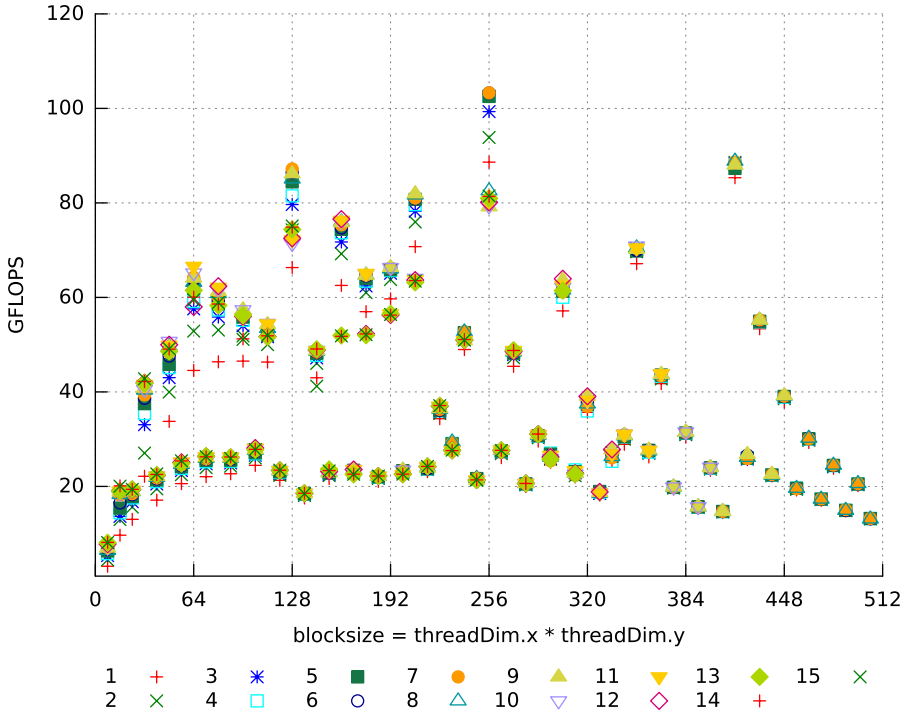

Figure 18: Experiment: Tesla, sgemm-8-alt.cu, threadDim.x=8, M=N=P=2048

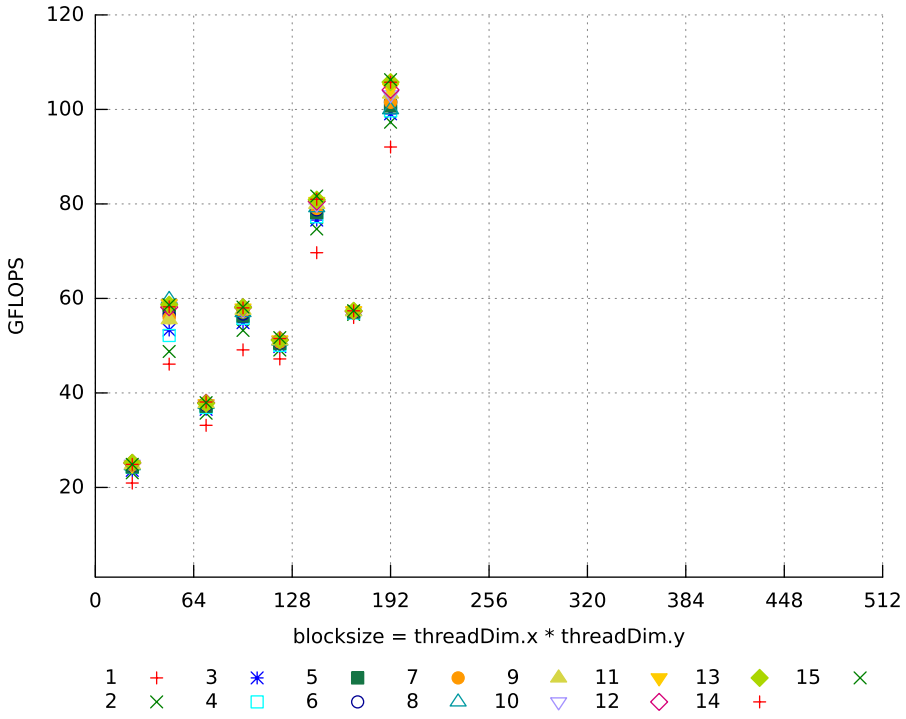

Figure 19: Experiment: Tesla, sgemm-24-alt.cu, threadDim.x=24, M=N=P=2048

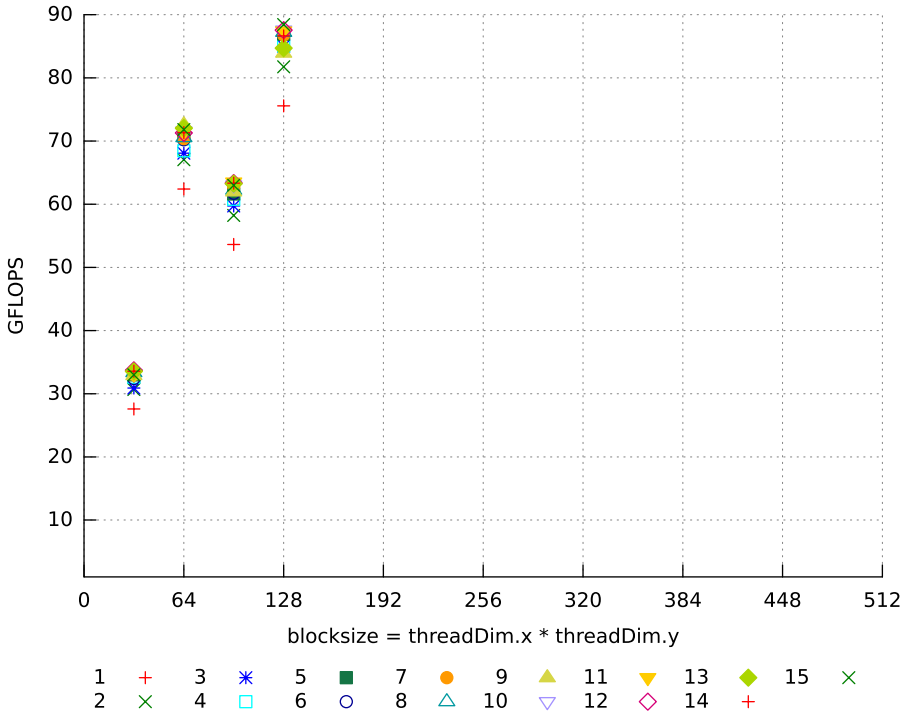

Figure 20: Experiment: Tesla, sgemm-32-alt.cu, threadDim.x=32, M=N=P=2048

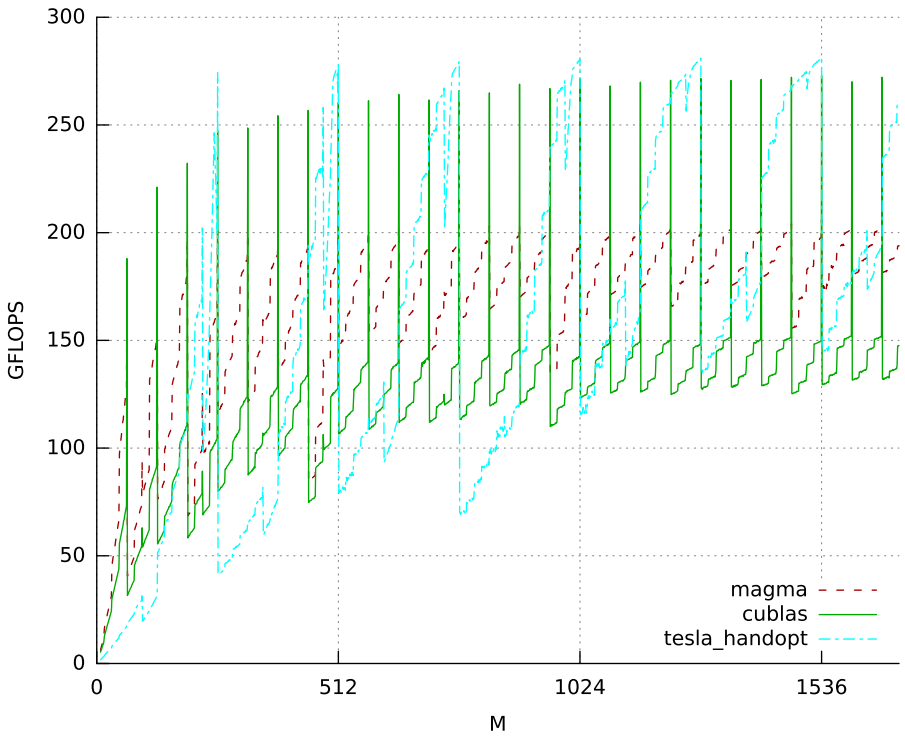

Figure 21: Experiment: Tesla, sgemm-16x16.cu, step=1, N=1024, K=1024

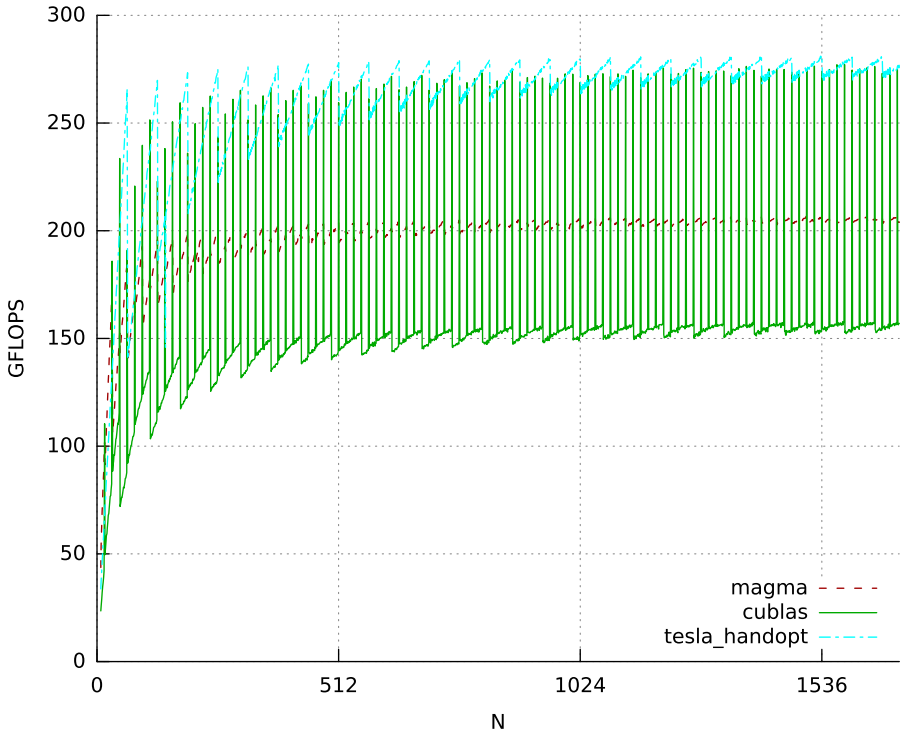

Figure 22: Experiment: Tesla, sgemm-16x16.cu, step=1, M=1024, K=1024

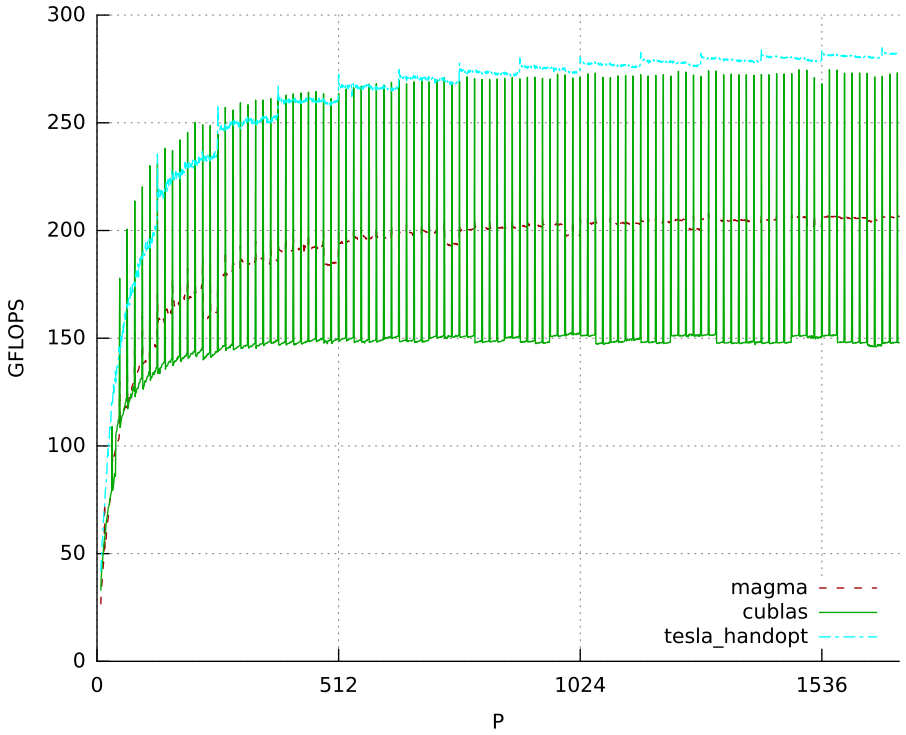

Figure 23: Experiment: Tesla, sgemm-16x16.cu, step=1, M=1024, N=1024

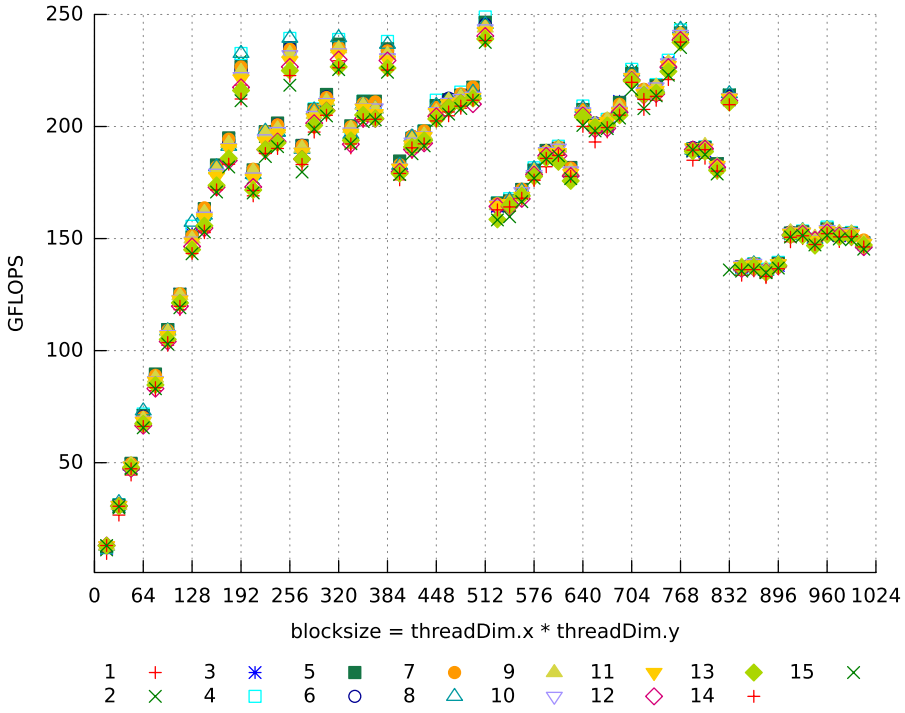

Figure 24: Experiment: Fermi, sgemm-8-alt.cu, threadDim.x=8, M=N=P=4096

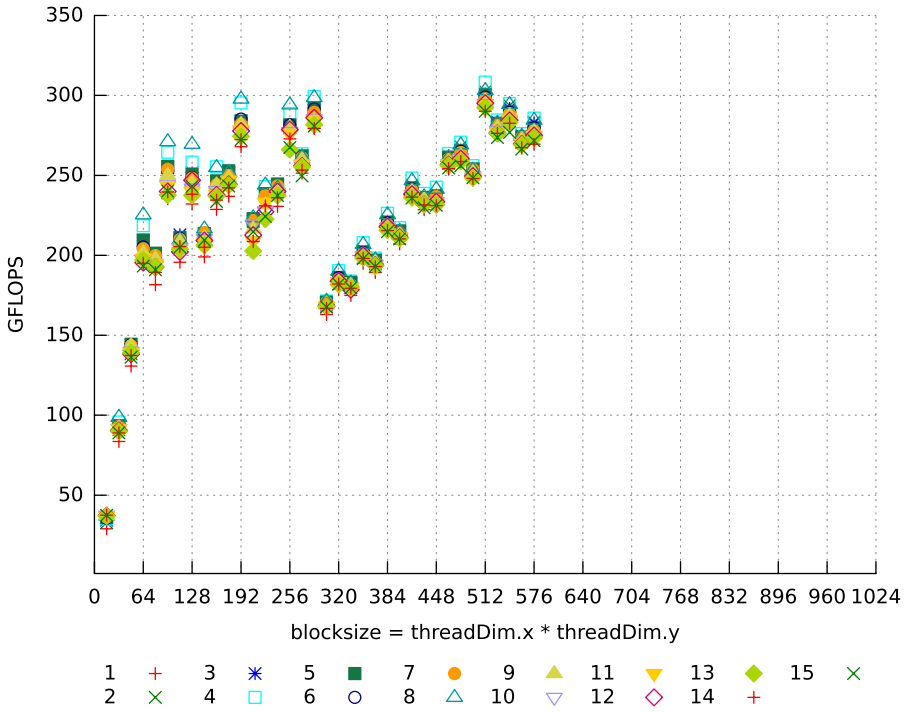

Figure 25: Experiment: Fermi, sgemm-16-alt.cu, threadDim.x=16, M=N=P=4096

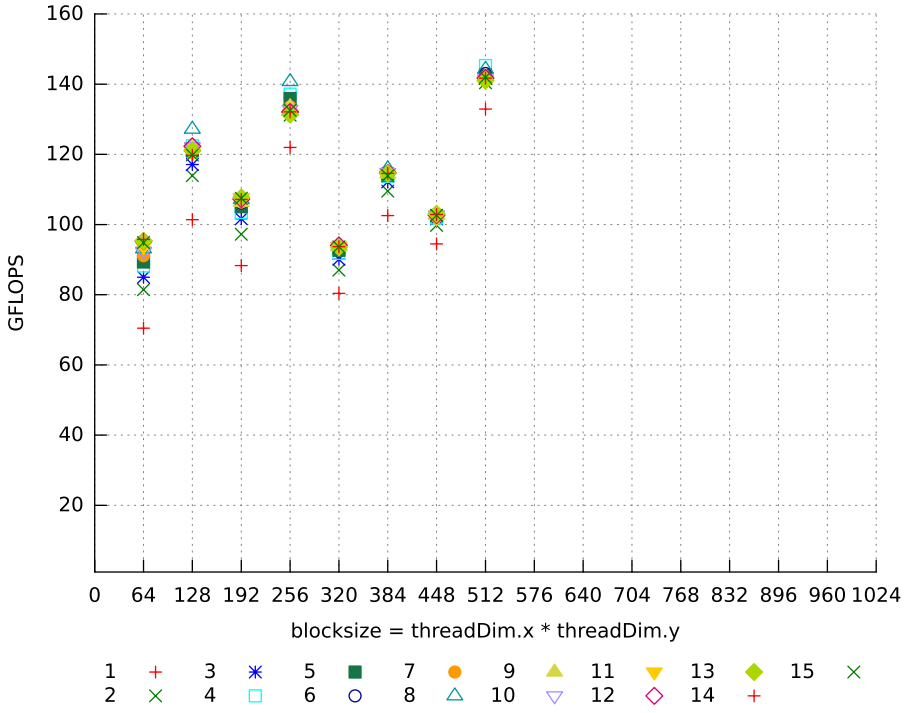

Figure 26: Experiment: Fermi, sgemm-64-alt.cu, threadDim.x=64, M=N=P=4096

B

CODES

Listing 3: Optimized CUDA implementation of GEMM for Tesla. 16x16 blocksize, 256x16 tilesize, 128 stepsize, 16 calculations per thread, load only B

```
static __device__ void calc16(float a,float *b, float *c) {
    c[0] += a * b[0];
    c[1] += a * b[1];
    c[2] += a * b[2];
    c[3] += a * b[3];
    c[4] += a * b[4];
    c[5] += a * b[5];
    c[6] += a * b[6];
    c[7] += a * b[7];
    c[8] += a * b[8];
    c[9] += a * b[9];
    c[10] += a * b[10];
    c[11] += a * b[11];
    c[12] += a * b[12];
    c[13] += a * b[13];
    c[14] += a * b[14];
    c[15] += a * b[15];
}

/* threads(16,16), grid(ceild(M,256),ceild(N,16)) */
extern "C" __global__ void
sgemm_custom_kernel(float *C, const float *A, const float *B,
                    int M, int N, int K,
                    int lda, int ldb, int ldc,
                    float alpha, float beta)
{
    //16x16 threads, each thread loading 8 floats into shared mem

    __shared__ float B_shared[2048];
    float C_thread[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    int thread_id = threadIdx.x + threadIdx.y * 16;
    int thread_id_adj = thread_id % 128;
    bool adj_flag = (thread_id >= 128);

    //fix B and C to starting n
    int n_start = blockIdx.y * 16;
    B += (n_start + adj_flag) * ldb;
    C += n_start * ldc;
```

```
    //fix C and A to m
    int m = 256 * blockIdx.x + thread_id;
    if (m >= M) {
        //recalculate value if m is out of bounds
        C += m - thread_id;
        A += m - thread_id;
    } else {
        C += m;
        A += m;
    }

    //Calculate divideable k's
    int steps = K / 128 + (K%128 != 0); //128 calculations per
        step
    while(steps > 0) {

        //handle rest in last step
        if(steps <= 1) {
            int rest = K % 128;
            if(rest > 0) {
                if(thread_id_adj < rest) {
                    #pragma unroll
                    for(int i=0;i<8;++i){
                        B_shared[(adj_flag+(2*i)) + thread_id_adj
                            *16] =
                            B[thread_id_adj + (2*i) * ldb];
                    }

                } else {
                    #pragma unroll
                    for(int i=0;i<8;++i){
                        B_shared[(adj_flag+(2*i)) + thread_id_adj
                            *16] = 0;
                    }
                }
            } else {
                //load B into shared memory
                B_shared[(adj_flag) + thread_id_adj*16] =
                    B[thread_id_adj];
                B_shared[(adj_flag+2) + thread_id_adj*16] =
                    B[thread_id_adj + 2 * ldb];
                B_shared[(adj_flag+4) + thread_id_adj*16] =
                    B[thread_id_adj + 4 * ldb];
                B_shared[(adj_flag+6) + thread_id_adj*16] =
                    B[thread_id_adj + 6 * ldb];
                B_shared[(adj_flag+8) + thread_id_adj*16] =
                    B[thread_id_adj + 8 * ldb];
                B_shared[(adj_flag+10) + thread_id_adj*16] =
                    B[thread_id_adj+10 * ldb];
                B_shared[(adj_flag+12) + thread_id_adj*16] =
                    B[thread_id_adj+12 * ldb];
                B_shared[(adj_flag+14) + thread_id_adj*16] =
```

```
                B[thread_id_adj+14 * ldb];
        }
    } else {
        //load B into shared memory
        B_shared[(adj_flag) + thread_id_adj*16] =
            B[thread_id_adj];
        B_shared[(adj_flag+2) + thread_id_adj*16] =
            B[thread_id_adj + 2 * ldb];
        B_shared[(adj_flag+4) + thread_id_adj*16] =
            B[thread_id_adj + 4 * ldb];
        B_shared[(adj_flag+6) + thread_id_adj*16] =
            B[thread_id_adj + 6 * ldb];
        B_shared[(adj_flag+8) + thread_id_adj*16] =
            B[thread_id_adj + 8 * ldb];
        B_shared[(adj_flag+10) + thread_id_adj*16] =
            B[thread_id_adj+10 * ldb];
        B_shared[(adj_flag+12) + thread_id_adj*16] =
            B[thread_id_adj+12 * ldb];
        B_shared[(adj_flag+14) + thread_id_adj*16] =
            B[thread_id_adj+14 * ldb];
    }

    B += 128;
    __syncthreads();



    //calculate C
    if(steps <= 1) {

/************* Important If-clause *********************/
        if(K % 128 > 0) {
/*******************************************************/

          int ub = ((K % 128)/4)+1;


            #pragma unroll
            for(int i=0;i<ub;++i) {
                float a[4] = {A[0], A[lda], A[2*lda], A[3*lda
                    ]};
                calc16(a[0], &B_shared[i*64], C_thread);
                calc16(a[1], &B_shared[16+i*64], C_thread);
                calc16(a[2], &B_shared[32+i*64], C_thread);
                calc16(a[3], &B_shared[48+i*64], C_thread);
                A += 4 * lda;
            }


        } else {
            #pragma unroll
            for(int i=0;i<32;++i) {
```

```
                float a[4] = {A[0], A[lda], A[2*lda], A[3*lda
                    ]};
                calc16(a[0], &B_shared[i*64], C_thread);
                calc16(a[1], &B_shared[16+i*64], C_thread);
                calc16(a[2], &B_shared[32+i*64], C_thread);
                calc16(a[3], &B_shared[48+i*64], C_thread);
                A += 4 * lda;
            }
        }
    } else {
        #pragma unroll
        for(int i=0;i<32;++i) {
            float a[4] = {A[0], A[lda], A[2*lda], A[3*lda]};
            calc16(a[0], &B_shared[i*64], C_thread);
            calc16(a[1], &B_shared[16+i*64], C_thread);
            calc16(a[2], &B_shared[32+i*64], C_thread);
            calc16(a[3], &B_shared[48+i*64], C_thread);
            A += 4 * lda;
        }
    }

    --steps;
    __syncthreads();
}

//write result to gobal memory
if((blockIdx.y+1) * 16 <= N & m < M) {
    #pragma unroll
    for(int i=0;i<16;++i){
        C[i*ldc] = beta * C[i*ldc] + alpha * C_thread[i];
    }
} else {
    if(m < M) {
        int n_rest = N - (blockIdx.y * 16);
        switch(n_rest){
            case 16:
                C[15*ldc] = beta * C[15*ldc] + alpha *
                    C_thread[15];
            case 15:
                C[14*ldc] = beta * C[14*ldc] + alpha *
                    C_thread[14];
            case 14:
                C[13*ldc] = beta * C[13*ldc] + alpha *
                    C_thread[13];
            case 13:
                C[12*ldc] = beta * C[12*ldc] + alpha *
                    C_thread[12];
            case 12:
                C[11*ldc] = beta * C[11*ldc] + alpha *
                    C_thread[11];
            case 11:
```

```
                    C[10*ldc] = beta * C[10*ldc] + alpha *
                        C_thread[10];
                case 10:
                    C[9*ldc] = beta * C[9*ldc] + alpha * C_thread
                        [9];
                case 9:
                    C[8*ldc] = beta * C[8*ldc] + alpha * C_thread
                        [8];
                case 8:
                    C[7*ldc] = beta * C[7*ldc] + alpha * C_thread
                        [7];
                case 7:
                    C[6*ldc] = beta * C[6*ldc] + alpha * C_thread
                        [6];
                case 6:
                    C[5*ldc] = beta * C[5*ldc] + alpha * C_thread
                        [5];
                case 5:
                    C[4*ldc] = beta * C[4*ldc] + alpha * C_thread
                        [4];
                case 4:
                    C[3*ldc] = beta * C[3*ldc] + alpha * C_thread
                        [3];
                case 3:
                    C[2*ldc] = beta * C[2*ldc] + alpha * C_thread
                        [2];
                case 2:
                    C[1*ldc] = beta * C[1*ldc] + alpha * C_thread
                        [1];
                case 1:
                    C[0] = beta * C[0] + alpha * C_thread[0];
                    break;
            }
        }
    }
}
```

Listing 4: Optimized CUDA implementation of GEMM for Fermi. 16x16 blocksize, 64x64 tilesize, 16 stepsize, 16 calculations per thread, load A and B, no rest calculation

```
texture<float,1>  texture_A;
texture<float,1>  texture_B;

static __device__ void calc(float* a_reg, float* b_reg,
                            float* c_reg, float* A_shared,
                            float* B_shared, int idx_a_shared,
                            int idx_b_shared, int l){
    #pragma unroll
    for(int k=0;k<4;++k){
        #pragma unroll
        for(int i=0;i<4;++i){
```

```
            a_reg[i+k*4] = A_shared[idx_a_shared + i + (k+l*4)
                *64];
        }
    }
    #pragma unroll
    for(int i=0;i<4;++i){
        #pragma unroll
        for(int k=0;k<4;++k){
            b_reg[k+i*4] = B_shared[idx_b_shared + k + l*4 + i
                *16];
        }
    }

    #pragma unroll
    for(int n=0;n<4;++n){
        #pragma unroll
        for(int m=0;m<4;++m) {
            #pragma unroll
            for(int k=0;k<4;++k) {
                c_reg[m+n*4] += a_reg[m+k*4] * b_reg[k+n*4];
            }
        }
    }
}

/* threads(16,16), grid(M/64,N/64) */
extern "C" __global__ void
sgemm_64_opt_kernel(float *C, const float *A, const float *B,
                    int M, int N, int K,
                    int lda, int ldb, int ldc,
                    float alpha, float beta,
                    int off_A, int off_B)
{
    __shared__ float A_shared[1024];   //64x16
    __shared__ float B_shared[1024];   //16x64

    float c_reg[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    int idx = threadIdx.x + threadIdx.y*16;
    int threadIdx_16_x = idx%16;
    int threadIdx_16_y = idx/16;
    int threadIdx_64_x = idx%64;
    int threadIdx_64_y = idx/64;

    int m_start = blockIdx.x * 64;
    int idx_A = off_A + m_start + threadIdx_64_x +
                threadIdx_64_y * lda;
    C += m_start + threadIdx_16_x*4;

    int n_start = blockIdx.y * 64;
    int idx_B = off_B + threadIdx_16_x +
                (threadIdx_16_y + n_start) * ldb;
```

```
C += (n_start + threadIdx_16_y*4) * ldc;

int idx_a_shared = threadIdx_16_x * 4;
int idx_b_shared = (threadIdx_16_y * 4) * 16;

float a_reg[16];
float b_reg[16];
float a_load[4];
float b_load[4];

#pragma unroll
for(int i=0;i<4;++i) {
    a_load[i] = tex1Dfetch(texture_A, idx_A + i*4 * lda);
}
#pragma unroll
for(int i=0;i<4;++i) {
    b_load[i] = tex1Dfetch(texture_B, idx_B + i*16 * ldb);
}
idx_B += 16;
idx_A += 16*lda;


int step = K / 16;
while(step > 0) {

    __syncthreads();
    #pragma unroll
    for(int i=0;i<4;++i) {
        A_shared[idx + i*256] = a_load[i];
    }
    #pragma unroll
    for(int i=0;i<4;++i) {
        B_shared[idx + i*256] = b_load[i];
    }
    __syncthreads();

    #pragma unroll
    for(int i=0;i<4;++i) {
        a_load[i] = tex1Dfetch(texture_A, idx_A + i*4*lda);
    }

    #pragma unroll
    for(int i=0;i<4;++i) {
        b_load[i] = tex1Dfetch(texture_B, idx_B + i*16*ldb);
    }

    calc(a_reg, b_reg, c_reg, A_shared, B_shared,
        idx_a_shared, idx_b_shared, 0);
    calc(a_reg, b_reg, c_reg, A_shared, B_shared,
        idx_a_shared, idx_b_shared, 1);
    calc(a_reg, b_reg, c_reg, A_shared, B_shared,
        idx_a_shared, idx_b_shared, 2);
```

```
        calc(a_reg, b_reg, c_reg, A_shared, B_shared,
            idx_a_shared, idx_b_shared, 3);

        idx_B += 16;
        idx_A += 16*lda;
        --step;
    }

    #pragma unroll
    for(int n=0;n<4;++n){
        #pragma unroll
        for(int m=0;m<4;++m) {
            C[m+n*ldc] = beta * C[m+n*ldc] + alpha * c_reg[m+n
                *4];
        }
    }
}
```

Listing 5: Script to find possible configurations

```
module Main where

import System.Environment

block :: Int -> Int -> Int -> Int -> [Int]
block maxX maxY maxThread warp = [warp*y | y <-[2..maxY*maxX],
                                          warp*y <= maxThread]

sharedMem :: Int -> Int -> Int -> Int -> Int -> Int -> [(Int,Int,
    Int)]
sharedMem maxM maxN maxK blockSize maxMem warp = [(m,n,k) |
      m <- [1..maxM],
      m `mod` warp == 0,
      n <- [1..maxK],
      n `mod` warp == 0,
      k <- [1..maxK],
      (m*k) `mod` blockSize == 0,
      (n*k) `mod` blockSize == 0,
      (m*n) `mod` blockSize == 0,
      (m*k+k*m) <= maxMem]

registers :: Int -> Int -> Int -> Int -> Int -> [(Int,Int,Int)]
registers registerCount m n k threads = [(f,t1,t2) |
      f <- [15..registerCount],
      f == (m*n) `div` threads,
      t1 <- [1..32],
      t2 <- [1..32],
      t1 == (m*k) `div` threads,
      t2 == (k*n) `div` threads,
      f == t1*t2,
      f <= registerCount]
```

```
gen :: Int -> Int -> Int -> Int -> Int -> Int
        -> Int -> Int -> Int -> [(Int,Int,Int,Int,Int,Int,Int)]
gen maxX maxY maxThread warp maxM maxN maxK memSize registerCount
     =
              [(s,m,n,k,r,t1,t2) |
                    s <- block maxX maxY maxThread warp,
                    (m,n,k)<- sharedMem maxM maxN maxK s memSize
                        warp,
                    (r,t1,t2) <- registers registerCount m n k s]

main :: IO ()
main = print l where
            l = gen 512 512 1024 32 512 512 512 5000 63
```

BIBLIOGRAPHY

[1] Matrix algebra on GPU and multicore architectures (MAGMA). http://icl.cs.utk.edu/magma/index.html.

[2] ATLAS - Automatically Tuned Linear Algebra Software, 2012. URL http://math-atlas.sourceforge.net.

[3] LAPACK - Linear Algebra PACKage, 2012. URL http://www.netlib.org/lapack.

[4] AMD. ACML - AMD Core Math Library, 2012. URL http://developer.amd.com/libraries/acml.

[5] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proc. 22nd International Conference on Supercomputing (ICS '08)*, pages 225–234. ACM, 2008. ISBN 978-1-60558-158-3. doi: http://doi.acm.org/10.1145/1375527.1375562. URL http://doi.acm.org/10.1145/1375527.1375562.

[6] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 1–10. ACM, 2008. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345210.

[7] Muthu M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proc. Compiler Construction (CC '10)*, pages 244–263, 2010.

[8] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276 –292, feb. 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840311.

[9] Gerald Baumgartner, David Bernholdt, Daniel Cociorva, Robert Harrison, Chi-Chung Lam, Marcel Nooijen, J. Ramanujam, and P. Sadayappan. A performance optimization framework for compilation of tensor contraction expressions into parallel programs.

In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 106–114, 2002.

[10] OpenMP Architecture Review Board. OpenMP Application Program Interface, 2012. URL `http://openmp.org/wp/openmp-specifications/`.

[11] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI)*, pages 101–113, 2008.

[12] Texas Advanced Computing Center. GotoBLAS, 2012. URL `http://www.tacc.utexas.edu/tacc-projects/gotoblas2`.

[13] P. Duclos, F. Boeri, M. Auguin, and G. Giraudon. Image processing on a simd/spmd architecture: OPSILA. In *Pattern Recognition, 1988., 9th International Conference on*, pages 430–433 vol.1, 1988. ISBN 0-8186-0878-1. doi: 10.1109/ICPR.1988.28259.

[14] Paul A. Feautrier and Christian Lengauer. Polyhedron model. In David Padua et al., editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer-Verlag, September 2011.

[15] Michael J. Flynn. Some computer organizations and their effectiveness. In *IEEE Transactions on Computers*, pages 948–960, 1972.

[16] Message Passing Interface Forum. MPI - a Message Passing Interface standard, 2012.

[17] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly – Polyhedral optimization in LLVM. In *Proc. First Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*, 2011.

[18] Armin Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In Ooge de Moor and M. Schwartzbach, editors, *Compiler Construction (CC)*, Lecture Notes in Computer Science 5501, pages 236–250. Springer-Verlag, 2009.

[19] Intel. MKL - Math Kernel Library, 2012. URL `http://software.intel.com/en-us/articles/intel-mkl`.

[20] Nadir Jeevanjee. *An Introduction to Tensors and Group Theory for Physicists*. Birkhäuser, 2011. ISBN 978-0-8176-4714-8.

[21] R.W. Johnson, C. Huang, and J.R. Johnson. Multilinear algebra and parallel programming. In *Supercomputing '90. Proceedings of*, pages 20–31, 1990.

[22] David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1st edition, 2010.

[23] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *PPoPP 2009*, pages 101–110, 2009.

[24] Christian Lengauer. Loop parallelization in the polytope model. In *Eike Best, editor, CONCUR'93, number 715 in Lecture Notes in Computer Science*, pages 398–416, 1993.

[25] Yixun Liu, E.Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for gpu program optimizations. In *Proc. Int. Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10, 2009. doi: 10.1109/IPDPS.2009.5160988.

[26] Wenjing Ma, Sriram Krishnamoorthy, and Gagan Agrawal. Practical loop transformations for tensor contraction expressions on multi-level memory hierarchies. In Jens Knoop, editor, *Compiler Construction (CC)*, Lecture Notes in Computer Science 6601, pages 266–285. Springer-Verlag, 2011. ISBN 978-3-642-19860-1. URL `http://dl.acm.org/citation.cfm?id=1987237.1987258`.

[27] G.S. Murthy, M. Ravishankar, M.M. Baskaran, and P. Sadayappan. Optimal loop unrolling for gpgpu programs. In *Parallel Distributed Processing (IPDPS)*, pages 1–11, 2010. doi: 10.1109/IPDPS.2010.5470423.

[28] NVIDIA Corporation. CUDA occupany calculator. `http://www.developer.nvidia.com/nvidia-gpu-computing-documentation`, 2012.

[29] NVIDIA Corporation. CUDA C best practices guide, version 4.1. `http://www.developer.nvidia.com/nvidia-gpu-computing-documentation`, 2012.

[30] NVIDIA Corporation. CUDA C programming guide, version 4.1. `http://www.developer.nvidia.com/nvidia-gpu-computing-documentation`, 2012.

[31] NVIDIA Corporation. cuBLAS version 4.1, 2012. `http://developer.nvidia.com/cublas`.

[32] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proc. Supercomputing (SC '10)*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: http://dx.doi.org/10.1109/SC.2010.14. URL `http://dx.doi.org/10.1109/SC.2010.14`.

[33] Srinivas K. Raman, Vladimir Penkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. In *Micro, IEEE, vol. 20*, pages 47–57, 2000. doi: 10.1109/40.865866.

[34] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, and J. Stratton. Program optimization space pruning for a multithreaded gpu. In *In Code generation and optimization: Proceedings of the Sixth Annual IEEE/ACM International Symposium on code generation and optimization*, pages 195–204, 2008.

[35] Konrad Trifunovic, Albert Cohen, David Edelsohn, Li Feng, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. GRAPHITE two years after. In *Proc. 2nd Int. Workshop on GCC Research Opportunities (GROW'10)*, 2010. `http://cTuning.org/workshop-grow10`.

[36] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proc. Int. Conf. on Supercomputing (ICS '11)*, pages 214–224. ACM, 2011. ISBN 978-1-4503-0102-2. doi: http://doi.acm.org/10.1145/1995896.1995932. URL `http://doi.acm.org/10.1145/1995896.1995932`.

[37] Sven Verdoolaege. ClooG, 2012. URL `http://www.cloog.org`.

[38] Sven Verdoolaege. ISL- Integer Set Library, 2012. URL `www.kotnet.org/~skimo/isl/user.html`.

[39] Vasily Volkov. Better performance at lower occupancy. Presentation at GPU Technology Conference (GTC 2010), 2010. `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf`.

DECLARATION

Hiermit versichere ich, dass ich diese Masterarbeit selbsständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 23.03.2012

Franz Xaver Bayerl