CHRISTOPH WOLLER

POLYHEDRAL OPTIMIZATION FOR GPU OFFLOADING IN THE EXASTENCILS CODE GENERATOR

POLYHEDRAL OPTIMIZATION FOR GPU OFFLOADING IN THE EXASTENCILS CODE GENERATOR

CHRISTOPH WOLLER

Master's Thesis

Programming Group
Department of Informatics and Mathematics
University of Passau

Supervisor: Prof. Christian Lengauer, Ph.D.
Tutor: Dr. Armin Größlinger

October 2016

Christoph Woller: *Polyhedral Optimization for GPU Offloading in the ExaStencils Code Generator,*, Master's Thesis, © October 2016

ABSTRACT

A popular subject in high-performance computing (HPC) is the engineering of massive parallel algorithms for stencil codes. Modern supercomputers approach exascale performance, yet HPC software engineering suffers from the effective exploitation of the massive parallelism. Project ExaStencils adopts a new course to take stencil code engineering to the next level. In the context of this project, a new domain-specific language (DSL) is designed, which facilitates the development of multigrid methods for solving systems arising from a discretization of partial differential equations (PDEs). Furthermore, project ExaStencils offers a DSL compiler that is able to generate high-performance hybrid target code.

This thesis presents a polyhedral CUDA code generation for the ExaStencils generator. Besides exploitation of the polyhedron model, the workflow features further extensions such as shared memory utilization, spatial blocking with shared memory, and spatial blocking with read-only cache. A polyhedral schedule exploration reveals the best performing schedule for graphics processing unit (GPU) targets. Several experiments demonstrate that the polyhedral CUDA code generation is capable of dealing with a real world problem from the ExaStencils' domain. The limitations of the workflow are discussed and a hybrid tiling approach for further performance improvements is examined.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [8]

ACKNOWLEDGMENTS

I would like to thank the following people for their help throughout this thesis:

Prof. Christian Lengauer, Dr. Armin Größlinger, and Stefan Kronawitter for supporting and guiding me towards this thesis. All members of the Programming Group and all participants in project ExaStencils for providing helpful hints.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LISTINGS

ACRONYMS

DSL domain-specific language v

HPC high-performance computing v

PDE partial differential equation v

AST abstract syntax tree 39

LSE linear system of equations 69

GPGPU general-purpose computation on graphics hardware 4

GPU graphics processing unit v

CPU central processing unit xi

ALU arithmetic logic unit 9

SM streaming multiprocessor 10

SIMT single-instruction-multiple-thread 10

SFU special function unit 12

FMA fused multiply-add 12

CWD CUDA work distributor 14

MPI Message Passing Interface 3

JIT just-in-time 21

SCOP static control program 25

TPDL target platform description language 32

ISL integer set library 38

Part I

BACKGROUND AND FUNDAMENTALS

1

INTRODUCTION

This chapter explains the topic of this thesis and the motivation behind it. The first part of the chapter gives an overview of the ExaStencils research project and introduces high-performance computing with GPUs. The second part points out the idea behind the new polyhedral CUDA code generation. The last part outlines the structure of this thesis.

1.1 PROJECT EXASTENCILS

Supercomputers begin to scratch the surface of exascale performance [12, p. 553]. In the domain of parallel programming the programming languages of choice are Fortran, C, or C++. At best these general-purpose languages are adorned with the Message Passing Interface (MPI) and OpenMP to achieve an high degree of parallelism. The overall goals in HPC engineering range from high performance to power consumption. The Holy Grail is software that is highly parallel and efficient with low energy consumption that can be ported to different execution platforms with minimal effort [12, p. 554]. However, the step towards exascale performance of the supercomputers creates one of the toughest problems in HPC software engineering: the necessity of the explicit treatment of the massive parallelism inside one node of a high-performance cluster [12, p. 553f.].

A highly recurring subject in HPC is the engineering of massive parallel algorithms for stencil codes that scale with the increasing performance of supercomputers [12, p. 554]. Stencil codes are compute-intensive algorithms, in which data points arranged in a large grid are being recomputed repeatedly from the values of data points in a predefined neighborhood [13, p. 56]. This fixed neighborhood pattern is called a stencil. Among other things, the application field of stencil codes covers multigrid methods for solving systems arising from a discretization of PDEs [12, p. 554]. Implementing stencil codes and numerical solvers like multigrid methods requires knowledge not only about the application domain and the mathematical model but also about the programming of modern, increasingly heterogeneous, HPC clusters [26, p. 42]. Programmers have to take huge efforts to get the mandatory expertise for writing optimized programs for an HPC cluster. Project ExaStencils[1] provides an escape from this scenario by pursuing a new way of stencil code engineering [12, p. 554].

Project ExaStencils follows two new ideas. Instead of using a general-purpose source language, a dedicated DSL called ExaSlang[2] is introduced. The main benefit of an ex-

1 http://www.exastencils.org/

2 ExaSlang stands for **ExaS**tencils **lang**uage.

ternal domain-specific language is the separation of algorithm and implementation [26, p. 42]. Domain experts are able to provide an algorithm written in ExaSlang without taking care of the implementation details. The DSL compiler, in our case the ExaStencils generator, is responsible for compiling the ExaSlang code into a target program with good performance. The second idea is to perform domain-specific optimizations at every refinement step to reach exascale performance for the domain of stencil codes [12, p. 555]. This explains the name of the project: ExaStencils. The vision of project ExaStencils is one automatic tool that is able to engineer a variety of stencil codes [12, p. 562]. There is already a code generator written in Scala that compiles ExaSlang to high-performance multigrid code written in C++ for upcoming exascale supercomputers [26, p. 42f.]. The output C++ code is parallelized with MPI, OpenMP, and also CUDA is possible.

The ExaStencils code generator and the ExaSlang domain-specific language are discussed in greater detail in Chapter 4. More information about project ExaStencils itself can be found elsewhere [12].

1.2 HIGH-PERFORMANCE COMPUTING WITH GPUS

In the previous section we briefly addressed high-performance computing and its quest towards exascale performance. Another topic in HPC that is getting more and more important is the use of GPUs. HPC can no longer be imagined without general-purpose computation on graphics hardware (GPGPU) [24, p. 21]. Commodity computer graphics chips, better known as GPUs, belong to the most powerful computational hardware in relation to its cost. The reasons for the prevalence of GPUs in HPC are varied.

On the one hand, GPUs are powerful and inexpensive [24, p. 21f.]. On the other hand, GPUs are getting more and more flexible and programmable [24, p. 22]. A modern GPU offers fully programmable processing units with support of vectorized floating-point operations. Furthermore, in comparison to CPUs, the evolution of the computation capabilities and the theoretical bandwidth of GPUs is far more impressive. Figure 1 illustrates the performance growth of CPUs and GPUs. The diagram reveals that the theoretical peak performance of modern GPUs, measured in GFLOP/s, is much higher than the one of modern CPUs. For example the GeForce GTX TITAN achieves a theoretical peak performance of about 4500 GFLOP/s with single precision, whereas an Ivy Bridge CPU scratches just the 750 mark. Additionally, the performance jumps made by the latest GPU generations are more promising than on the CPU side.

Beside the tremendous performance growth, the increasing bandwidth of GPUs as opposed to CPUs is noteworthy [22, p. 1ff.]. Figure 2 shows the bandwidth change of CPUs and GPUs. The bandwidth of GPUs is increasing just as the performance of GPUs and leaves CPUs behind. Overall, the power of modern GPUs eclipses that of CPUs and, in addition, such commodity computer graphics chips can be bought as off-the-shelf graphics cards built for the PC video game market that are available for $400-500 at release [24, p. 22].

One drawback of GPGPU computing is the unusual programming model [24, p. 22]. A developer has to learn a new language and express computations in graphics terms. This

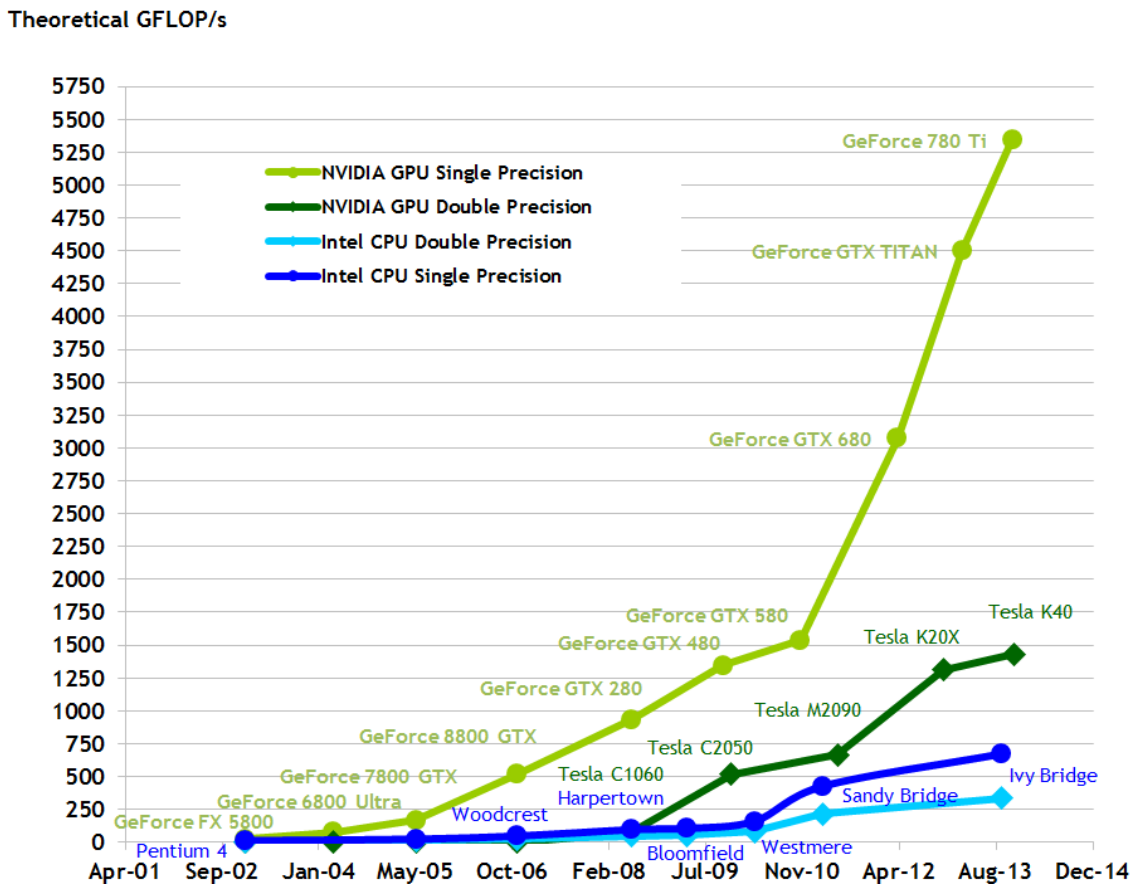


Figure 1: Performance evolution of selected GPUs and CPUs [22, p. 2]

requires a fundamental knowledge of the underlying hardware, its design, limitations, and evolution.

In summary, the theoretical performance, the increased precision, and the rapidly expanding programmability of the hardware make GPUs an attractive platform for general-purpose computation [24, p. 22]. The application of GPUs and data-parallel processing goes far beyond image rendering and processing [22, p. 4]. For example, general signal processing, physics simulation to computational finance or computational biology, or partial differential equations belong to their field of application.

1.3 MOTIVATION

In the previous section we got an impression of how powerful modern GPUs are and which opportunities they provide with regard to high-performance computing. Retrospectively, we can state that project ExaStencils aims to develop a parallel code generator that is able to construct code reaching exascale performance and, on the other hand, modern GPUs offer enormous compute power. As a consequence, the ExaStencils code generator ought to produce code for GPUs.

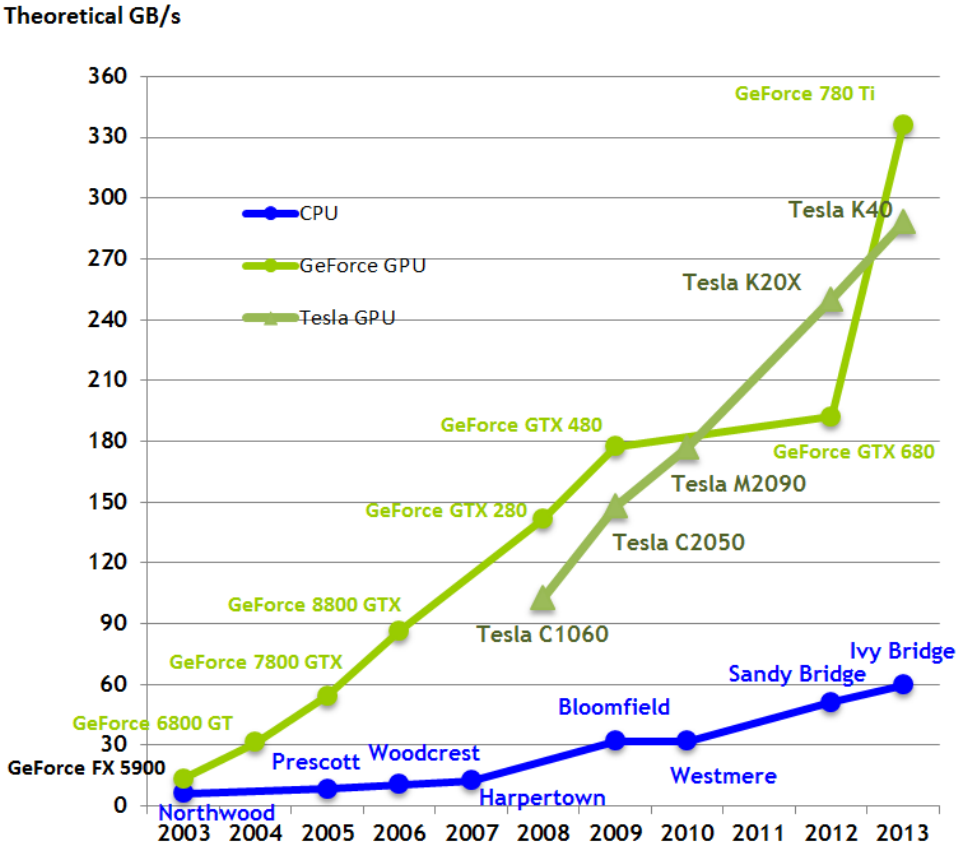


Figure 2: Bandwidth evolution of selected GPUs and CPUs [22, p. 3]

As mentioned in Section 1.1, the ExaStencils code generator applies code optimizations at different points and levels of the code generation process. In this context, the polyhedron model is used to apply transformations and optimizations for better parallel target code. The polyhedron model aims at improving the overall parallel speedup of a program but currently it is only used for CPU and not for GPU code generation in the ExaStencils generator.

The guiding questions of this thesis are:

- How can you utilize best a GPU's performance capabilities for project ExaStencils?
- How can you exploit the power of the polyhedron model for GPU code generation?
- What are the limits of GPUs in the context of project ExaStencils?

Hence, this thesis focuses on a new GPU code generation for the ExaStencils generator. We concentrate on NVIDIA GPUs and use NVIDIA's general-purpose parallel computing platform CUDA [22, p. 4]. CUDA is a programming model that simplifies programming of NVIDIA GPUs. Furthermore, the new workflow exploits the power of the polyhedron model and uses GPU-specific optimizations to get the most performance from GPUs.

1.4 OUTLINE OF THE THESIS

The next chapter lays the foundation for high-performance programming with NVIDIA GPUs. Chapter 3 examines the basic principles of the polyhedron model. In Chapter 4 the DSL ExaSlang and the ExaStencils code generator are discussed in greater detail. Additionally, the state of the art of GPU code generation and of polyhedral optimizations in the ExaStencils generator is explained. The second part of this thesis focuses on the implemented innovations in the ExaStencils generator. Chapter 5 presents the new GPU code generation workflow. Subsequently, the used polyhedral schedule exploration is explained. The last chapter presents the performed experiments and discusses the gathered results.

2

FUNDAMENTALS OF GPGPU COMPUTATION AND MODERN NVIDIA GPUS

This chapter describes the architecture of modern NVIDIA GPUs and focuses on the GK110 architecture in greater detail. Furthermore, programming with CUDA is explained.

2.1 ARCHITECTURE OF MODERN NVIDIA GPUS

After giving a brief introduction and motivation for GPGPU computing, this section covers the fundamental architecture of modern NVIDIA GPUs, which is the explanation for the difference in performance growth of CPUs and GPUs [22, p. 3f.]. The architectural contrasts of CPUs and GPUs are schematically shown in Figure 3.

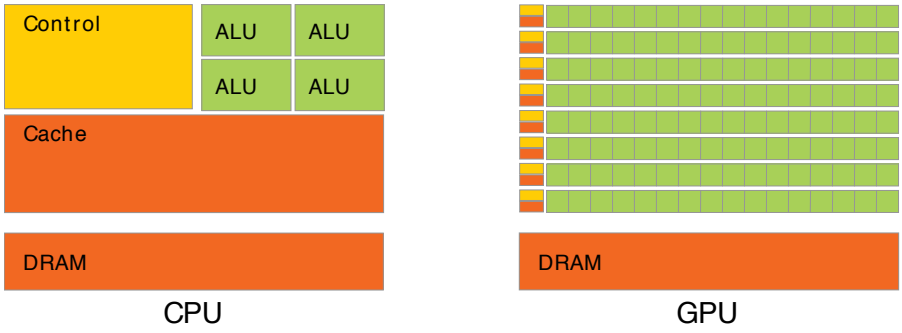


Figure 3: CPU design vs. GPU design [22, p. 3]

At the outset the main purpose of GPUs has been graphics rendering [22, p. 3]. Hence, they perform compute-intensive, highly parallel computations. However, CPUs are low-latency low-throughput processors. Thus, CPUs are designed to minimize latency, which requires caches. GPUs are high-latency high-throughput processors. As a consequence, there is no need of large caches and GPUs can dedicate more of the chip area to computational power. GPUs can have more arithmetic logic units (ALUs) for the same sized chip and therefore can support many more threads of computation. So, in contrast to CPUs, more transistors are devoted to data processing rather than data caching and control flow. In other words, the basic architecture of GPUs is designed for data-parallel computations. The idea of data parallelism is to perform the same task on many different data elements with high arithmetic intensity. This approach has two implications for

the GPU architecture. First of all if the same task is executed on a large number of data elements, there is a lower requirement for sophisticated control flow. Secondly, the memory access latency can be hidden with calculations instead of big data caches because the same task is executed on many data elements with high arithmetic intensity.

After this first explanation we take a closer look at the architectural design of NVIDIA GPUs [22, p. 69f.]. The core of the NVIDIA GPU architecture consists of a scalable array of multithreaded streaming multiprocessors (SMs). A multithreaded CUDA program is divided in thread blocks and the SMs execute them independently. This execution process is schematically shown in Figure 4.

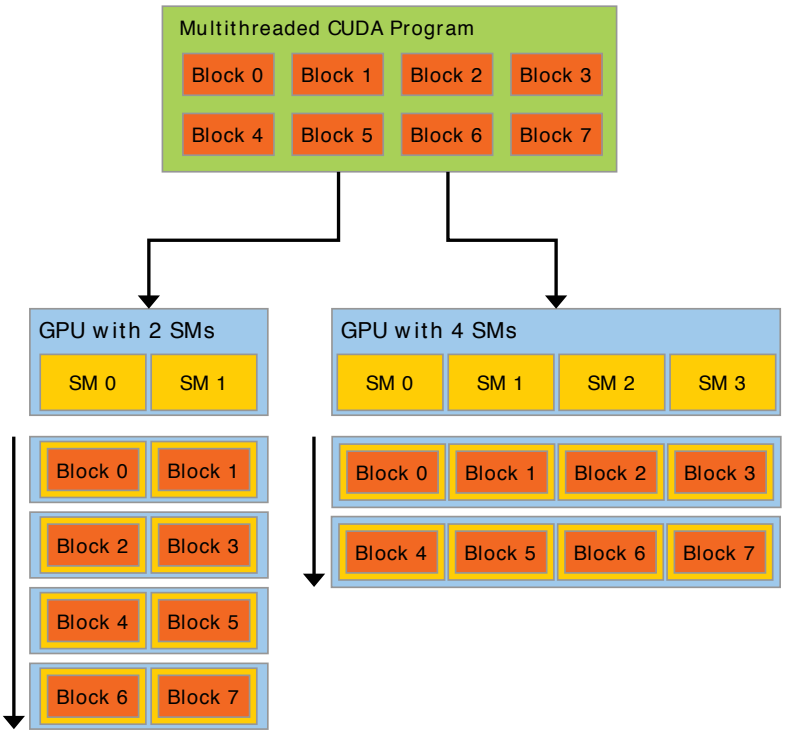


Figure 4: Design of a CUDA-capable GPU [22, p. 7]

Hence, an SM is designed to concurrently execute a large amount of threads [22, p. 69f.]. For this purpose, the multiprocessors follow the single-instruction-multiple-thread (SIMT) model. If a multiprocessor gets a thread block for execution, it partitions the block into warps. A warp is a group of 32 parallel threads. The threads within a warp have consecutive, increasing thread IDs. The part of an SM responsible for managing and scheduling warps is simply called the warp scheduler. All threads of a warp start at the same program address. Each thread has its own instruction address counter and register state, thus it is free to branch and execute independently. A warp executes one common instruction at a time. If there is a data-dependent conditional branch, it is possible that the threads of a single warp diverge. A warp serially executes every branch taken, while disabling threads that are not on that path. When all paths complete, the threads converge back to the same execution path. The full efficiency can be achieved if all threads composing a warp follow the same execution path. Each SM has a set of

32-bit registers that are partitioned among the warps, and a parallel data cache or shared memory that is partitioned among the thread blocks [22, p. 71].

2.2 THE NVIDIA KEPLER GK110 ARCHITECTURE

The last section provided an overview of the general design of a CUDA-capable NVIDIA GPU. In this section we will look in detail at the NVIDIA Kepler GK110 architecture because GPUs implementing this architecture are used for the experiments discussed in the later parts of the thesis.

The Kepler GK110/210 architecture was released in 2012 and comprises 7.1 billion transistors [20, p. 4]. NVIDIA puts the focus of this GPU architecture generation on compute performance. The Kepler GK110/210 architecture outperforms previous generation GPUs with regard to the raw compute power, the power consumption, and the heat output [20, p. 6]. GPUs implementing Kepler GK110 or GK 210 can perform double precision calculations at a rate of up to 1/3 of single precision compute performance. Figure 5 provides an overview of the compute capabilities of Fermi and Kepler GPU architectures.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110	KEPLER GK210
Compute Capability	2.0	2.1	3.0	3.5	3.7
Threads / Warp	32				
Max Threads / Thread Block	1024				
Max Warps / Multiprocessor	48		64		
Max Threads / Multiprocessor	1536		2048		
Max Thread Blocks / Multiprocessor	8		16		
32-bit Registers / Multiprocessor	32768		65536		131072
Max Registers / Thread Block	32768		65536		65536
Max Registers / Thread	63			255	
Max Shared Memory / Multiprocessor	48K				112K
Max Shared Memory / Thread Block	48K				
Max X Grid Dimension	2^16-1		2^32-1		
Hyper-Q	No			Yes	
Dynamic Parallelism	No			Yes	

Figure 5: Compute capability of Fermi and Kepler GPU architectures [20, p. 7]

2.2.1 *The streaming multiprocessor SMX*

One innovation in the GK 110 and GK 210 architecture is the new streaming multiprocessor called SMX that is schematically shown in Figure 6.

Figure 6: Structure of SMX unit in Kepler [20, p. 8]

Kepler GPUs feature 15 SMX units and six 64-bit memory controllers [20, p. 6]. Each SMX unit consists of 192 single-precision CUDA cores and every one of them ships with fully pipelined floating-point and integer arithmetic logic units [20, p. 9]. A Kepler SMX has eight times as many special function units (SFUs) for fast approximate transcendental operations as the Fermi GF110 SM and provides IEEE-754-2008 compliant single- and double-precision arithmetic including the fused multiply-add (FMA) operation. Each SMX has four warp schedulers, whose union is called a quad warp scheduler. Each warp scheduler has two instruction dispatch units, making in total eight instruc-

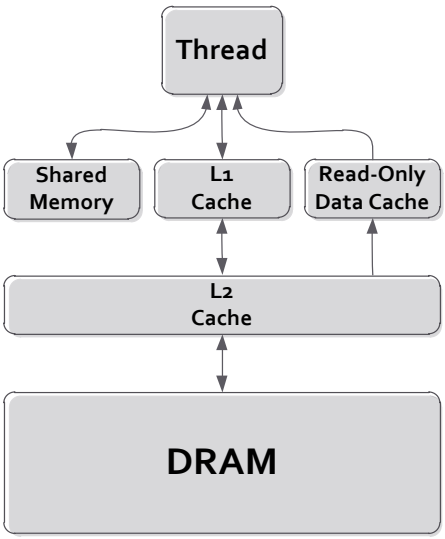


Figure 7: Kepler's memory subsystem [20, p. 13]

tion dispatch units, allowing four warps to be issued and executed concurrently. Hence, four warps can be selected by the scheduler and, in each cycle, two independent instructions per warp can be dispatched. It is noteworthy that double-precision instructions can be paired with other instructions. In a Kepler GK110 GPU a CUDA thread can access up to 255 registers [20, p. 11] and in comparison to GK 110, the Kepler GK210 architeture improves this further by doubling the overall register file capacity per SMX. Further performance improvements are achieved with a new shuffle instruction allowing threads within a warp to share data. CUDA threads within the same warp can read values from other threads in the warp in any permutation and there is no need for separate load and store operations to pass data through shared memory. The performance advantage of shuffle instructions over shared memory comes from the fact that the execution of a store-and-load operation requires only a single clock cycle. In addition to it, shuffle instructions go hand in hand with a lower amount of shared memory needed per thread block because data exchanged at warp level never needs to be stored in shared memory.

2.2.2 *Kepler's memory system*

The next point discusses Kepler's memory subsystem shown in Figure 7 [20, p. 13]. The memory request path for loads and stores is unified and each SMX unit has an L1 cache. In the Kepler GK110 architecture an SMX multiprocessor is equipped with a 64KB on-chip memory. This memory is split into shared memory and L1 cache providing the following configuration options. 48KB of shared memory with 16KB of L1 cache, or the other way round. The third option is to allocate 32KB of shared memory and 32KB of L1 cache. The shared memory bandwidth for 64-bit and larger load operations amounts to 256B per core clock. In the Kepler GK210 architecture an SMX multiprocessor has 128KB on-chip memory. The following splitting configurations into shared memory / L1 cache are possible: 112KB / 16KB, 96KB / 32KB, and 80 KB / 48 KB. Furthermore, Kepler

GK110 and GK210 have a 48KB read-only data cache that is directly accessible to the SMX units for general load operations [20, p. 14]. The benefit of using this cache arises from its higher tag bandwidth supporting, for example, full-speed unaligned memory access patterns. Besides the L1 and the read-only data cache, there is a 1536KB L2 cache, which is the primary point of data unification between the SMX units, servicing all load, store, and texture requests. All streaming multiprocessors have access to the same on-board DRAM memory called global memory. The global memory is banked, which means that simultaneous memory accesses to adjacent positions can be coalesced into a single memory transaction [30, p. 5]. In contrast to previous-generation GPUs, Kepler does not use the L1 cache for DRAM load caching, but only for register spilling [15, p. 1].

2.2.3 *Innovations of Kepler GPUs*

Apart from the new streaming multiprocessor SMX, GPUs of the Kepler generation offer two more noteworthy features [20, p. 5], [19]. First, the Dynamic Parallelism feature makes it easier for developers to exploit the massive parallel processing power of the GPU. This feature spawns new threads by adapting to the data without going back to the host CPU. Second, with the Hyper-Q feature it is possible that multiple CPU cores can start work on a single GPU simultaneously. The impact of this feature is an increased GPU utilization and a reduction of CPU idle times. A Kepler GPU allows up to 32 simultaneous hardware-managed connections. The previous Fermi architecture supports just a single connection.

2.2.3.1 *Dynamic Parallelism*

The new Dynamic Parallelism feature, visualized by Figure 8, was first presented by NVIDIA in its Kepler GK110/210 architecture [20, p. 15f.]. It enables kernels to launch new kernels, to create the necessary streams, events and to manage the dependences needed to process additional work. Thus, the GPU is able to create work without any CPU interaction. Hence, dynamic parallelism simplifies the development of recursive and data-dependent execution patterns. As an immediate consequence more code can be run on a Kepler GPU.

2.2.3.2 *Hyper-Q*

The Hyper-Q feature makes it possible to have 32 simultaneous, hardware-managed connections, called work queues, between the host and the CUDA work distributor (CWD) logic in the GPU [20, p. 17f.]. Figure 9 illustrates the Hyper-Q feature. It allows connections from multiple CUDA streams, from multiple MPI processes, or even from multiple threads within a process. There is a one-to-one mapping between CUDA streams and work queues. Hence, a CUDA stream is managed within its own hardware work queue. Furthermore, inter-stream dependences are optimized. Consequently, operations in one stream will no longer block other streams and concurrent execution of streams is possible.

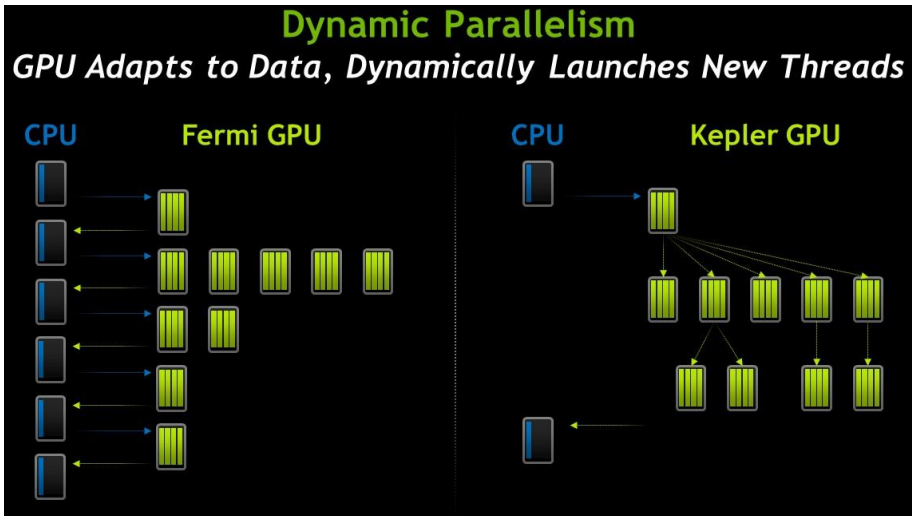


Figure 8: The left side shows the previous Fermi architecture without dynamic parallelism needing the CPU to launch new work and the right side shows the Kepler architecture with dynamic parallelism able to generate work by itself [20, p. 15].

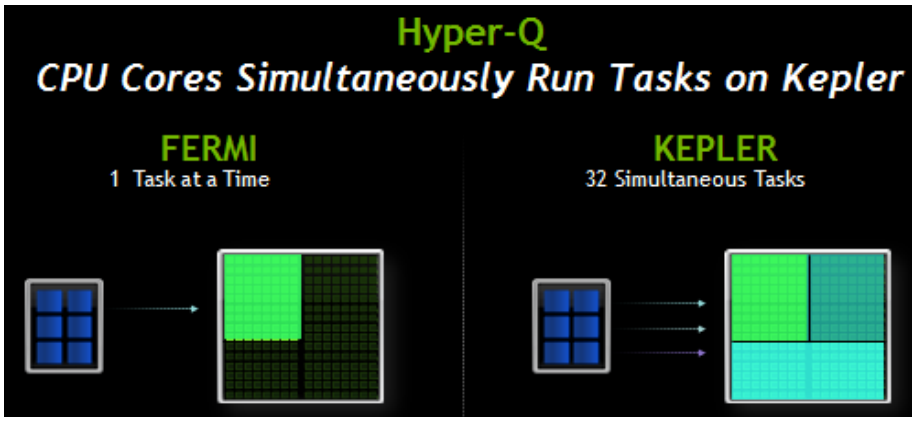


Figure 9: On the left side the previous Fermi architecture allowing just a single hardware-managed connection at a time and on the right side the Kepler architecture with Hyper-Q able to manage simultaneously 32 connections [20, p. 17].

2.3 NVIDIA GEFORCE GTX TITAN BLACK

The experiments discussed later in this thesis were performed on an NVIDIA GeForce GTX TITAN Black GPU. For the sake of completeness, this section provides an overview of the TITAN Black GPU and presents its technical details. The NVIDIA GeForce GTX TITAN Black GPU is part of the GeForce 700 Series and is based on the Kepler GK110 GPU microarchitecture discussed in Section 2.2 [18]. Table 1 lists its specification details.

GPU Engine Specifications	
CUDA Cores	2880
Base Clock	889 MHz
Boost Clock	980 MHz
Texture Fill Rate	213 GigaTexels/sec
Theoretical Peak Performance Single Precision (MAD)	5120,6 GFLOPS
Theoretical Peak Performance Double Precision (FMA)	1706,9 GFLOPS
Memory Specifications	
Memory Clock	7.0 Gbps
Standard Memory Configuration	6144 MB
Memory Interface	GDDR5
Memory Interface Width	384 Bit
Memory Bandwidth	336 GB/sec

Table 1: NVIDIA GeForce GTX TITAN Black specifications

2.4 CUDA GPU PROGRAMMING

NVIDIA introduced CUDA in November 2006 as general-purpose parallel computing platform [22, p. 4]. Furthermore, CUDA is a programming model that simplifies the application development for parallel compute engines in NVIDIA GPUs. The CUDA platform allows the developer to implement an application in one of the high-level programming languages C, C++, or Fortran and upgrade it with CUDA extensions or OpenACC directives for expressing parallelism. CUDA is available as NVIDIA CUDA Toolkit[1]. This toolkit covers a comprehensive development environment for C and C++ developers. It ships with a compiler for NVIDIA GPUs, math libraries, and tools for debugging and optimizing applications.

2.4.1 *Basic idea behind CUDA*

The development of a massive parallel application with CUDA for NVIDIA GPUs follows a basic principle [3]. First, the problem is solved in an higher-level programming language like C or C++. After this step, the application is enriched with CUDA keywords and extensions to shift computations from CPU to GPU. Figure 10 shows a simple C program on the left side and a version of this program with CUDA extensions on its right. For example, the compiler recognizes based on the `__global__` annotation that the function `saxpy` is destined for the GPU and the remaining program should run on the CPU. In addition, some memory transfer statements are added to manage the data transfer between CPU and GPU. In summation, there are only a few changes necessary to get a CUDA program that is parallel executable on GPUs.

1 https://developer.nvidia.com/cuda-toolkit

```
void saxpy(int n, float a,
    float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;
// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

Listing 1: Standard C code

```
__global__ void saxpy(int n, float a, float *x,
    float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);
// Perform SAXPY on 1M elements
saxpy<<<4096,256)>>>(N, 2.0, x, y);
cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

Listing 2: C with CUDA extensions

Figure 10: Simple CUDA example [3]

2.4.2 *The CUDA programming model*

This subsection addresses the main concepts of the CUDA programming model to get a better understanding of the semantic of the CUDA extensions in Figure 10 [7]. To put it in a nutshell, CUDA is a heterogeneous programming model, in which code is executed on the CPU and the GPU. This model refers to the CPU as *host*. The GPU is called the *device*. The programming model proceeds on the assumption that the host and the device maintain their own separate memory spaces in DRAM, which are respectively denoted by *host memory* and *device memory*. The device executes the CUDA parts of a program and operates as coprocessor to the host running the main program. Figure 11 illustrates the relation between host and device. A developer implements a program in an high-level programming language and adds special CUDA extensions to mark the parts of the code that should be executed on the device in parallel. The remaining code runs on the host. The main concepts of the CUDA programming model are explained with help of the following example in Listing 3 [22, p. 12]. The example shows a simple matrix addition of two square matrices.

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Listing 3: CUDA C matrix addition example

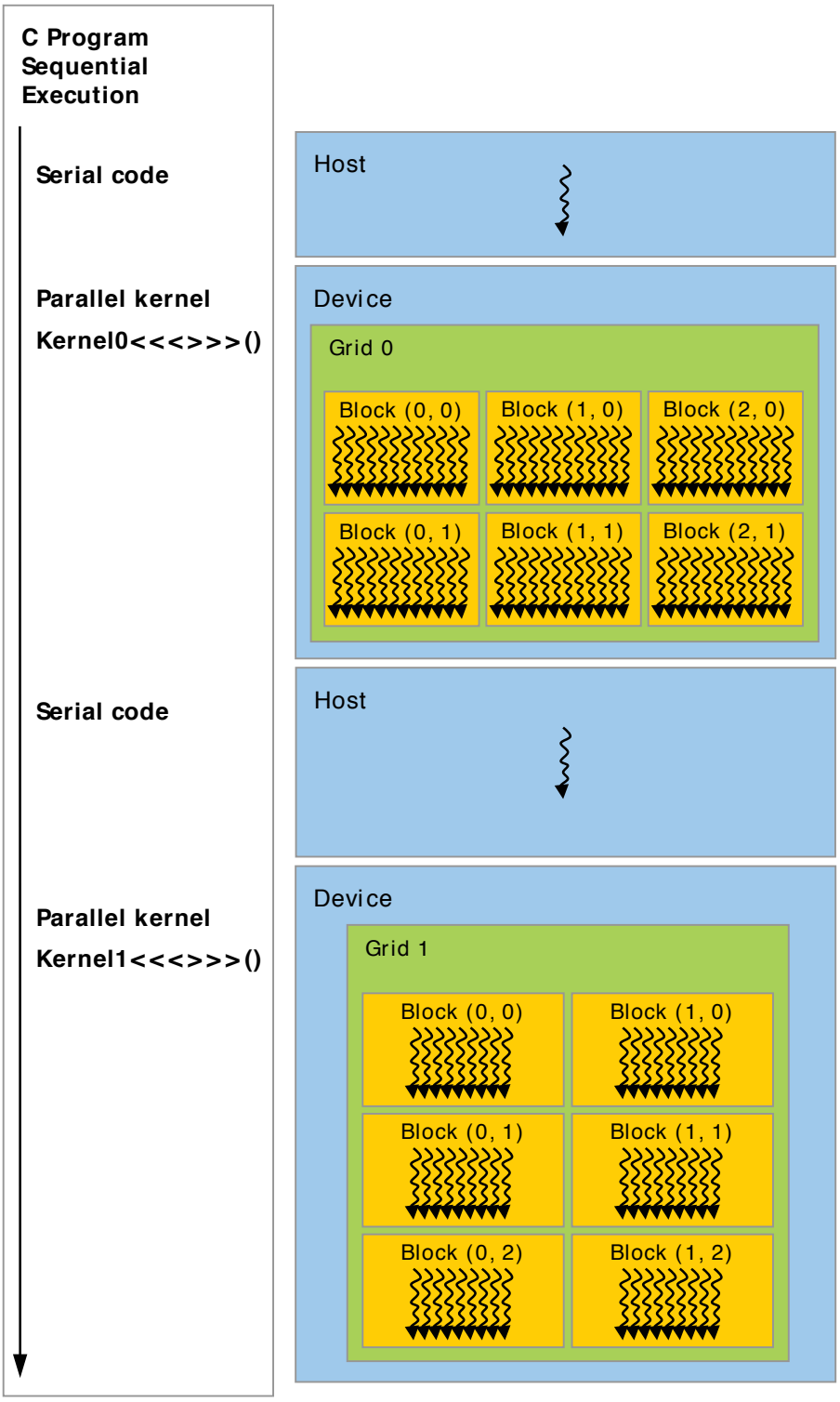


Figure 11: Heterogeneous programming with CUDA [22, p. 15]

2.4.2.1 *Kernels and Threads*

The central unit in the CUDA programming model is a kernel [22, p. 9]. The term kernel refers to a function that is meant to be executed on the device. The developer can extend a regular function definition with the `__global__` declaration specifier to mark it as kernel. For example the function `MatAdd` in line 1 in Listing 3 forms a CUDA kernel. Hence, the `main` function in line 7 is executed by the host and calls the kernel function `MatAdd` in line 12, which is then executed by the device. Beside the name of the kernel function and the function arguments, a kernel function call requires the specification of an execution configuration. The developer defines the execution configuration within `<<< ... >>>` like shown in line 12 in Listing 3. A kernel is executed N times in parallel by N different CUDA threads [22, p. 9]. In the CUDA programming model, these N threads are organized in one-dimensional, two-dimensional, or three-dimensional *thread blocks* and these thread blocks together form a one-dimensional, two-dimensional, or three-dimensional *grid* [22, p. 10f]. The freedom of defining blocks and grids of dimensionality 1, 2, or 3 should enable a simpler approach to vector, matrix, or volume calculations. Nonetheless, a kernel is executed by a number of equally shaped thread blocks and the total number of CUDA threads being launched equals the number of threads per block times the number of blocks in the grid. The number of threads per block and the number of blocks in the grid is determined by the kernel call's execution configuration, which has the following structure: `<<<numBlocks, threadsPerBlock, sharedMemory, cudaStream>>>` [22, p. 135].

- `numBlocks`: This variable of type `int` or `dim3` specifies the dimension and size of the grid [22, p. 135]. A one-dimensional grid is specified by providing an `int` variable. A two-dimensional grid is specified by defining `numBlocks` like in line 11 in Listing 3. To get a three-dimensional grid, one has to define a `dim3` variable like for a two-dimensional grid but adding a third component. Hence, the number of blocks being launched is equal to:
  - `numBlocks` for a one-dimensional grid
  - `numBlocks.x * numBlocks.y` for a two-dimensional grid
  - `numBlocks.x * numBlocks.y * numBlocks.z` for a three-dimensional grid
- `threadsPerBlock`: This variable of type `int` or `dim3` specifies the dimension and size of each block [22, p. 135]. One-dimensional, two-dimensional, and three-dimensional thread blocks are specified like for the grid mentioned in the previous point. Hence, the number of threads per block is equal to:
  - `threadsPerBlock` for a one-dimensional grid
  - `threadsPerBlock.x * threadsPerBlock.y` for a two-dimensional grid
  - `threadsPerBlock.x * threadsPerBlock.y * threadsPerBlock.z` for a three-dimensional grid.
- `sharedMemory`: This variable of type `size_t` is an optional argument and defaults to zero [22, p. 135]. It specifies the number of bytes in shared memory that are

dynamically allocated per block for this call in addition to the statically allocated memory.

- `cudaStream`: This variable of type **`cudaStream_t`** is an optional argument and defaults to zero [22, p. 135]. It specifies the associated stream.

Figure 12 illustrates the thread hierarchy in the CUDA programming model for a better understanding of the thread organization in thread blocks and the grid composition of thread blocks [22, p. 9ff.]. Each CUDA thread has a unique *thread ID* and belongs to exactly one thread block in a grid. As a consequence, the unique thread ID for some thread can be calculated with help of the built-in variable **`threadIdx`** of type `uint3` that contains the thread index within the respective thread block, the built-in variable **`blockIdx`** of type `uint3` that contains the block index within the grid, and the built-in variable **`blockDim`** of type **`dim3`** that contains the number of threads per dimension in a block. Hence, the unique thread ID is one-dimensional, two-dimensional, or three-dimensional according to the dimensionality of the grid and the thread block. Simply put, for a unique three-dimensional thread ID $\mathtt{tid} = (x, y, z)$ of a CUDA thread holds: $\mathtt{i} = \mathtt{blockIdx.i} * \mathtt{blockDim.i} + \mathtt{threadIdx.i}, \mathtt{i} \in \{x, y, z\}$. For a two-dimensional or a one-dimensional thread ID the calculation is straightforward. An example of the calculation of a two-dimensional thread ID is shown in line 2 and 3 in Listing 3.

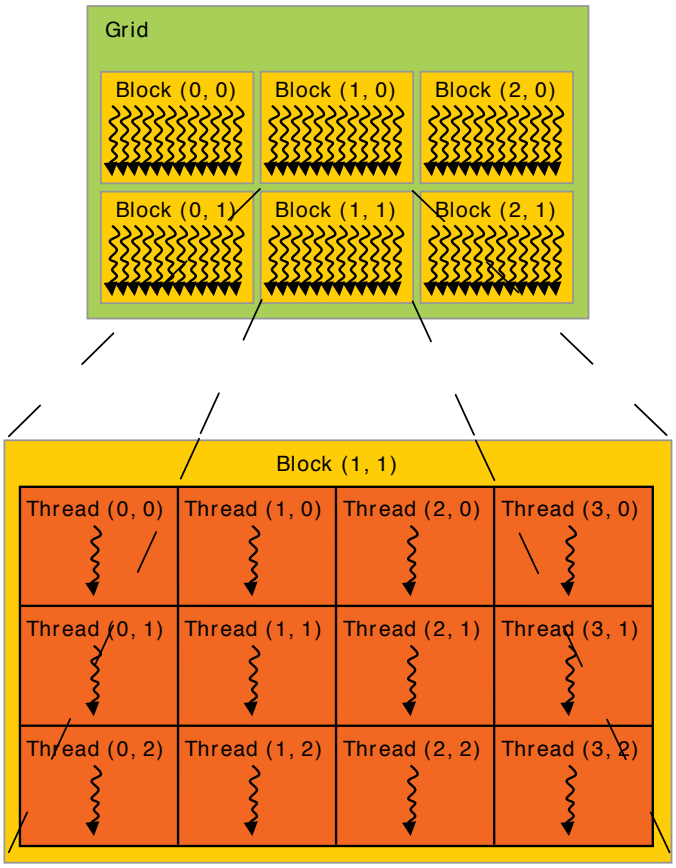


Figure 12: The CUDA thread hierarchy [22, p. 11]

As discussed previously in Section 2.1, the thread blocks are distributed to the different streaming multiprocessors, which execute the thread blocks in any order, in parallel or in series [22, p. 12]. Consequently, the thread blocks have to be independent of each other. Furthermore, the number of threads per block is limited to 1024 threads in modern GPUs because all threads of a block reside on the same streaming multiprocessor and must share its limited memory resources. Threads within the same block can share data through the shared memory of the SM and synchronize their execution with the intrinsic function `__syncthreads()`. This function acts as a barrier, which causes a thread to wait for all other threads of the same block before proceeding. There is no global synchronization between threads of different blocks.

2.4.2.2 *Memory Hierarchy*

Figure 13 provides an overview of the different memory spaces the CUDA threads have access to [22, p. 12f.]. Each thread has its own private local memory. For data exchange between threads of the same thread block there is a shared memory. Each thread block has its own shared memory. Moreover, there are three memory spaces that can be accessed by all threads no matter which grid or block. These include the global, the constant, and the texture memory space. The last two memory spaces, constant and texture, are read-only. All three are persistent across kernel launches by the same application.

2.4.3 *CUDA's programming interface in a nutshell*

After discussing the general CUDA programming model, this section gives an overview of CUDA's programming interface [22, p. 17]. As already mentioned in Section 2.4.1, the CUDA C toolkit provides a set of extensions to the C/C++ language and a runtime library. In order to run a CUDA application it must be compiled with NVIDIA's compiler driver `nvcc` that is introduced in the following subsection. Furthermore, we have a look at the CUDA runtime and the different compute modes of an NVIDIA GPU.

2.4.3.1 *CUDA compilation workflow*

Programs written in C or C++ with CUDA extensions must be compiled into binary code by the NVIDIA compiler driver `nvcc`[2] that is contained in the CUDA toolkit to execute on the GPU [22, p. 17f.]. The `nvcc` compiler driver supports offline and just-in-time (JIT) compilation. It expects source files containing a mix of host and device code as input and accomplishes the following tasks:

(1) Separation of device and host code.

(2) Compilation of the device code into an assembly form and/or binary form.

(3) Modification of the host code by replacing kernel invocations and the respective execution configurations with the necessary CUDA C runtime function calls to load and launch each compiled kernel from the assembly and/or binary form.

2 `http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/`

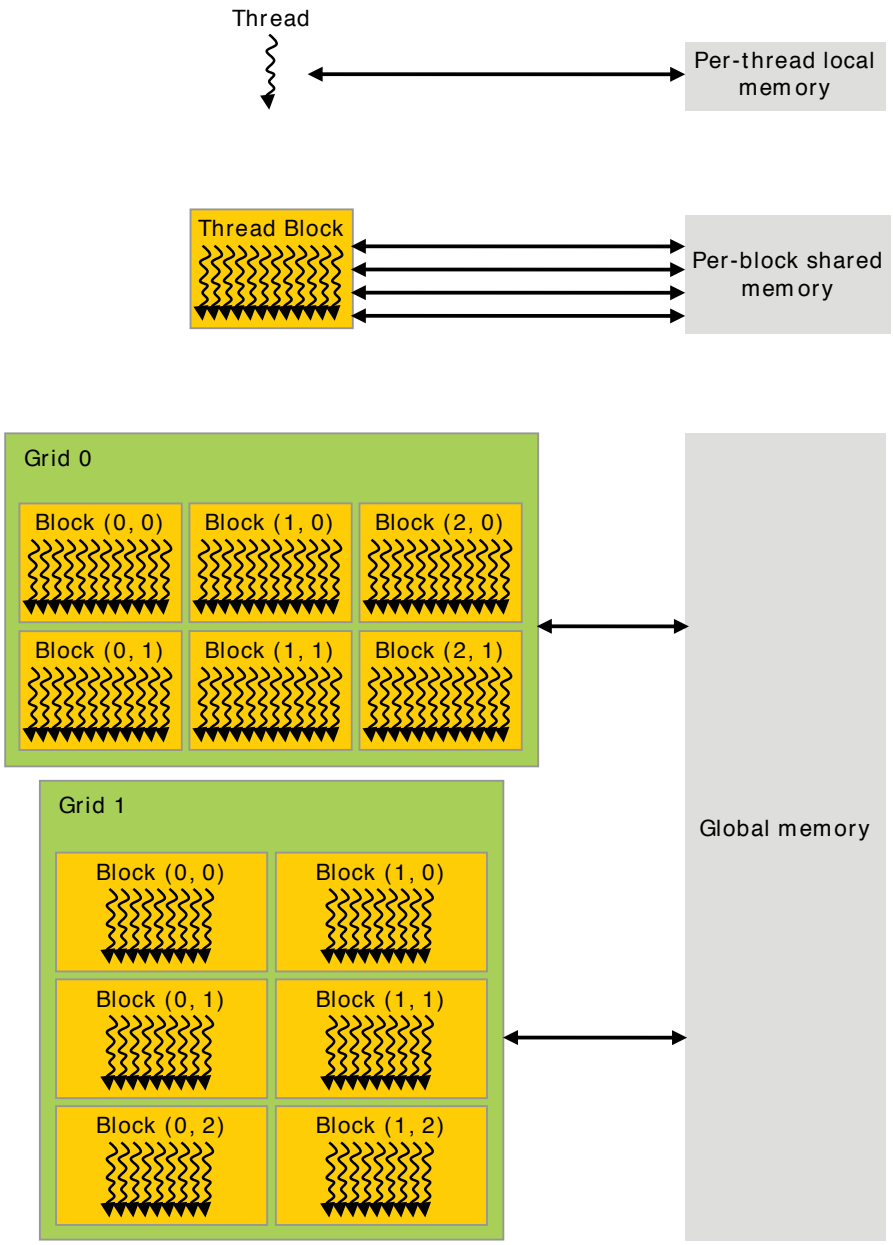


Figure 13: The CUDA memory hierarchy [22, p. 13]

(4) Output of the modified host code either as C code or as object code. In the first case the resulting C code must be compiled using another tool. In the latter case `nvcc` calls the host compiler itself during the last compilation stage.

2.4.3.2 *CUDA C runtime*

The CUDA C runtime is based on the CUDA driver API, which is a lower-level C API [22, p. 2of.]. It is implemented in the `cudart` library. If one executes a CUDA application, the runtime is initialized upon the first call of a runtime function. One part of this initializa-

tion is the context creation. For each device in the system an individual context, serving as the primary context of this device, is created. A CUDA context can be regarded as GPU analog of a CPU process [22, p. 224]. It encapsulates resources and actions performed within the CUDA driver API. If a context is destroyed, the system automatically cleans up all these resources. The primary context of a device is shared among all host threads of the program [22, p. 21]. Hence, there is an usage count for each context and if the usage count is equal to zero, the respective context is destroyed [22, p. 224]. If JIT compilation is used, the device code is compiled during the context creation and loaded into the device memory [22, p. 21].

A more detailed explanation of the CUDA C runtime and more information about the CUDA context can be found elsewhere [22, p. 20off.], [22, p. 224].

2.4.3.3 *Compute modes*

Modern NVIDIA GPUs offer different modes for compute applications [22, p. 67f.]. The compute mode of a device can be changed with help of the tool NVIDIA's System Management Interface (nvidia-smi)[3]. According to the GPU, there a four different compute modes:

(1) *Default*: Multiple host threads are able to use the device at the same time.

(2) *Exclusive-process*: Only one CUDA context may be created on the device across all processes in the system and that context may be accessible to as many threads as desired within the process that created the context.

(3) *Exclusive-process-and-thread*: Same as exclusive-process with the additional constraint that the context may only be accessible to one thread at a time.

(4) *Prohibited*: No CUDA context can be created on the device.

3 `https://developer.nvidia.com/nvidia-system-management-interface`

3

FUNDAMENTALS OF THE POLYHEDRON MODEL

Since the 1990s, the polyhedron model has been used as an abstract representation of loop programs to ease the process of code transformation and parallelization [4, 5, 11]. The abstract consideration of a loop program in the polyhedron model helps to answer the questions of the possibility of finding parallelism. In this chapter we will discuss the polyhedron model on the basis of an example program and how it can be used to parallelize a program. Listing 4 shows our example program.

```
1       for i = 1 ← 1 → n
2           for j = 1 ← 1 → i + m
3   S1 :        A[i][j] = A[i − 1][j] + A[i][j − 1]
4   S2 :    A[i][i + m + 1] = A[i − 1][i + m] + A[i][i + m]
```

Listing 4: Example source program

As its name implies, the main idea of the polyhedron model is to transform a loop program in an appropriate polyhedron [5, p. 1581], [11, p. 1f.]. The benefit of this transformation is to take advantage of the established theory of linear programming for working with polyhedra. The basic model expects a polyhedron to be convex, to have a flat surface, and to fulfill some more requirements [11, p. 2]. These required properties restrict the sets of programs that can be modeled to mainly for-loop programs acting on arrays. Furthermore, in a valid for-loop program, composed of n perfectly nested loops acting on arrays, the loop bounds and the array subscripts have to be affine expressions in the indices of the enclosing loops and in the problem size. Such a program is called a static control program (SCoP) and can then be represented by a polyhedron in $\mathbb{Z}^n$, where each of the n loops defines the extent of the polyhedron in one dimension. Hence, our example program in Listing 4 can be interpreted as a polyhedron in $\mathbb{Z}^2$. There are extensions to get a wider application of the model, but for the sake of simplicity we only consider basic for-loop programs.

3.1 FROM A FOR-LOOP PROGRAM TO A POLYHEDRON

It is still to be clarified how to transform a for-loop program into a polyhedron. In Listing 4 we can recognize two statements S_1 and S_2. S_1 is enclosed by two loops whereas only the outer loop surrounds S_2. Since S_1 and S_2 are part of a loop body, they are executed several times. These statement executions are called instances and the instances of a statement can be identified with the name of the statement and the surrounding

loop counters [5, p. 1581ff.]. Hence, the instances of S_1 can be identified in the form of $< S_1, i, j >$ and the one of S_2 with $< S_2, i >$. In the first step, a loop program is represented by the set of all instances occurring in it. This set is named the iteration domain or index space $\mathcal{IS}$. In this thesis we are only interested in terminating programs, thus the index space $\mathcal{IS}$ is finite. The index space $\mathcal{IS}$ has to be ordered by some relation $\prec$ to represent the source program. If $a, b \in \mathcal{IS}$, $a \prec b$ means that the instance a is executed before the instance b. In the case of a sequential program, $\prec$ is a total order and in the case of a fully parallel program, the relation $\prec$ is empty since all operations can be performed in any order. If $\prec_1$ and $\prec_2$ are two different execution orders for the same index space $\mathcal{IS}$, the concept of dependences is used to decide whether the tuples $(\mathcal{IS}, \prec_1)$ and $(\mathcal{IS}, \prec_2)$ represent the same program or not. The dependency relation is defined by $\delta = \{(a, b) \in \mathcal{IS} \times \mathcal{IS} : a \prec b \cap \mathcal{R}(a) \cap \mathcal{W}(b) \neq \emptyset \vee \mathcal{W}(a) \cap \mathcal{R}(b) \neq \emptyset \vee \mathcal{W}(a) \cap \mathcal{W}(b) \neq \emptyset\}$, where $\mathcal{W}(a)$ is the set of all memory cells written by instance $a \in \mathcal{IS}$ and $\mathcal{R}(a)$ is the set of all memory cells read by instance $a \in \mathcal{IS}$. To put it in a nutshell, two instances $a, b \in \mathcal{IS}$ are dependent, written $a \delta b$, if both access the same memory cell and at least one of them modifies it. The tuples $(\mathcal{IS}, \prec_1)$ and $(\mathcal{IS}, \prec_2)$ are equivalent (represent the same program), if dependent instances are executed in the same order in both. The index space $\mathcal{IS}$ of the example program in Listing 4 is given as $\mathcal{IS} = \{< S_1, i, j >: 1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant i + m\} \cup \{< S_2, i >: 1 \leqslant i \leqslant n\}$. A convex polyhedron can be defined as the intersection of a finite number of half-spaces. A half-space in $\mathbb{Z}^k$ can be specified with an affine inequality of the form $a_1 \cdot x_1 + a_2 \cdot x_2 + \cdots + a_k \cdot x_k \leqslant \beta$, where $a_1, \ldots, a_k \in \mathbb{Z} \wedge \beta \in \mathbb{Z}$ are constants and $x_1, \ldots, x_k \in \mathbb{Z}$ are variables. A convex polyhedron in $\mathbb{Z}^k$ can then be specified as a system of l affine inequalities of the form $A \cdot x \leqslant b$, where A is an $l \times k$ matrix, x is an $k \times 1$ vector of variables, and b is an $l \times 1$ vector of constants. In our example, we can separate the instances in $\mathcal{IS}$ into instances of S_1 and instances of S_2. For the sake of simplicity we just consider the instances of S_1 now, which fulfill in total four inequalities originating from the loop bounds. These four inequalities form a system of affine inequalities that describes a polyhedron. The following equivalent transformations describe how to get from the inequalities, derived from the loop bounds, to the system of affine inequalities that describe a polyhedron, which represents the instances of S_1:

$$\begin{array}{rcl} i & \geqslant & 1 \\ j & \geqslant & 1 \\ i & \leqslant & n \\ j & \leqslant & m + i \end{array} \Leftrightarrow \begin{bmatrix} -i \\ -j \\ i \\ -i + j \end{bmatrix} \leqslant \begin{bmatrix} 1 \\ 1 \\ n \\ m \end{bmatrix} \Leftrightarrow \underbrace{\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ -1 & 1 \end{bmatrix}}_{A:=} \cdot \underbrace{\begin{bmatrix} i \\ j \end{bmatrix}}_{x:=} \leqslant \underbrace{\begin{bmatrix} 1 \\ 1 \\ n \\ m \end{bmatrix}}_{b:=}$$

The rightmost representation corresponds to a system of affine inequalities that describes a polyhedron in $\mathbb{Z}^2$. We can also devise a system of affine inequalities for the instances of S_2, which can again be represented as a polyhedron in $\mathbb{Z}^2$. These two polyhedra together are illustrated in Figure 14 and represent the index space $\mathcal{IS}$ of the example program in Listing 4.

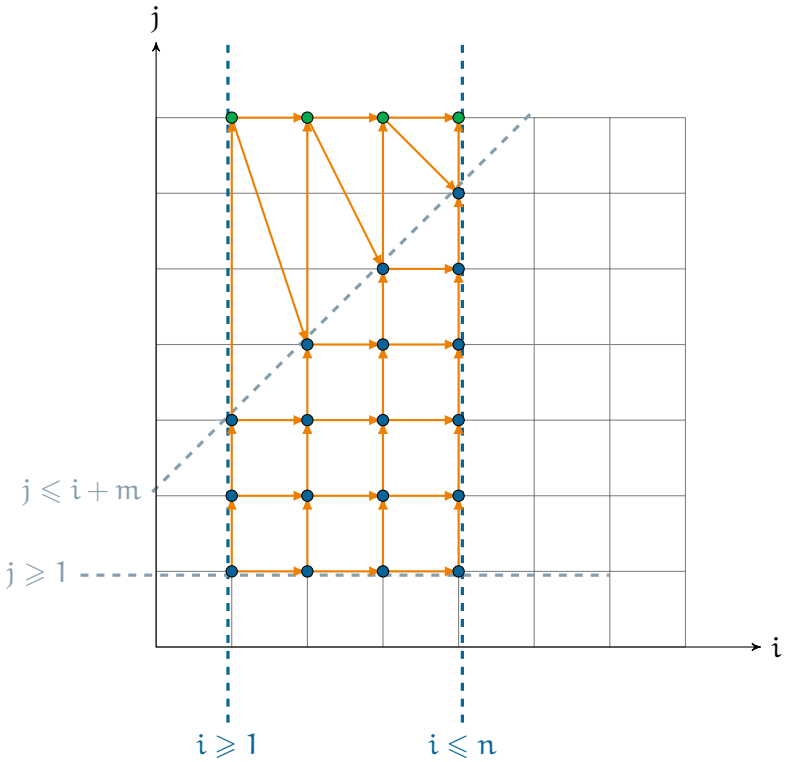


Figure 14: Source index space with $n = 4$ and $m = 2$

The dashed lines in Figure 14 highlight the half-spaces that bound the polyhedra and the arrows symbolize the dependences between the instances in $\mathcal{IS}$. Figure 14 visualizes the instances of S_1 with blue points and the ones of S_2 with green points.

3.2 POLYHEDRAL TRANSFORMATIONS

The previous subsection discussed the representation of a for-loop as polyhedron in $\mathbb{Z}^k$. This section addresses program optimizations in the polyhedron model. In general, optimizations in the polyhedron model are performed by coordinate transformations of the index space $\mathcal{IS}$ [5, p. 1584f.]. In the following we discuss one possible transformation of the index space shown in Figure 14. The loop nest in Listing 4 is imperfect because not all statements belong to the innermost loop body. Additionally, one notices that both the loop on i and on j are sequential since one iteration always accesses the value written by the previous iteration. However, with the aid of the polyhedron model we can expose parallelism by applying a skewing transformation. Figure 15 presents the result of this transformation. It reveals that there are no dependences between iterations of the p loop for a given value of t. Hence, the iterations of the p loop can be executed in parallel.

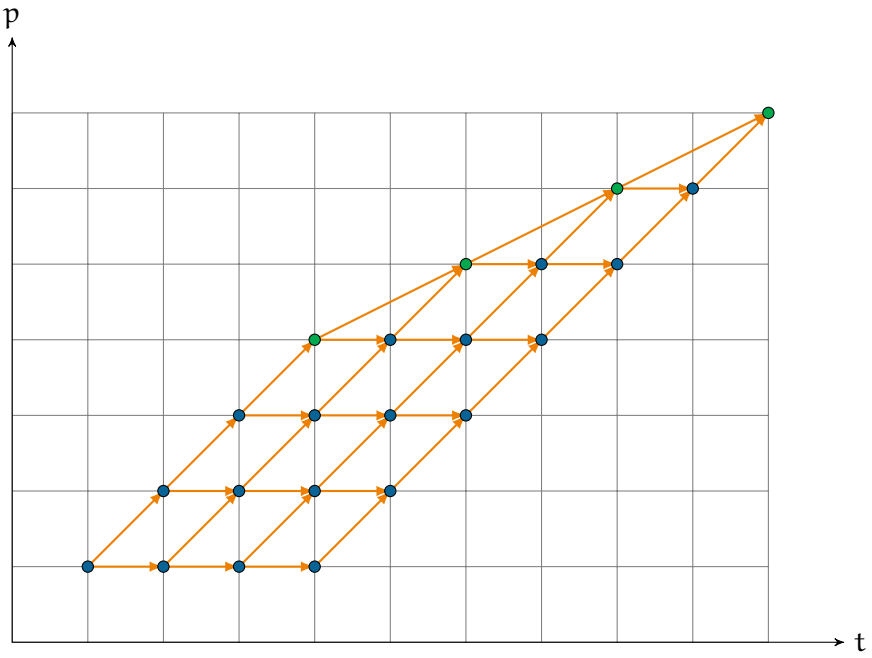


Figure 15: Target index space with $n = 4$ and $m = 2$

Essentially, the applied skewing transformation is a change of coordinates of the points in the polyhedra combined with a renaming [5, p. 1584f.]. The following equalities define the applied skewing transformation for S_1 and S_2:

$$S_1 : \begin{bmatrix} t \\ p \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}}_{T_1 :=} \cdot \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -2 \\ -1 \end{bmatrix}$$

$$S_2 : \begin{bmatrix} t \\ p \end{bmatrix} = \underbrace{\begin{bmatrix} 2 \\ 1 \end{bmatrix}}_{T_2 :=} \cdot \begin{bmatrix} i \end{bmatrix} + \begin{bmatrix} m-1 \\ m \end{bmatrix}$$

The matrices T_1, T_2 are called *space-time mapping* [11, p. 6] and are derived from a placement function $\hat{p}$ and a schedule function $\hat{t}$. The schedule function $\hat{t}$ assigns a point in time to every instance in the index space $\mathcal{IS}$ whereas the placement function $\hat{p}$ assigns a processor to each instance in $\mathcal{IS}$. This is why the dimensions in the target index space are named t and p. Linear programming offers methods and techniques to find good schedule and placement functions. A more detailed explanation of schedule and placement functions is outside the scope of this thesis and not essential in this context.

An important fact is that the transformation matrix T_1 is unimodular because its determinant is equal to 1. As a result of this unimodularity T_1 is bijective in the integers. As a consequence the target polyhedron resulting from T_1 is precisely the target space [11, p. 6].

3.3 FROM A POLYHEDRON BACK TO A FOR-LOOP PROGRAM

In the previous subsection we demonstrated how to expose parallelism in a supposed sequential program. The next step is to obtain a loop program from the representation in the polyhedron model. Figure 15 serves again as an example. As already mentioned, we have a sequential loop on t for the time dimension and a parallel loop on p for the spatial dimension. In our case a very simple approach is to define the loop on t as the outer loop and the loop on p as the inner one. Since the calculation of the loop bounds is slightly difficult, we do not discuss it at this point. Listing 5 depicts the target program retrieved from the target index space in Figure 15. The **parfor** keyword in line 2 in Listing 5 indicates that the iterations of this loop can be executed in parallel.

```
1       for t = 0 ← 1 → m + 2 * n − 1
2           parfor p = max(0, t − n + 1) ← 1 → min(t, ⌈(t + m)/2⌉)
3   S1 :        A[i][j] = A[i − 1][j] + A[i][j − 1]
4
5   S2 :    A[i][i + m + 1] = A[i − 1][i + m] + A[i][i + m]
```

Listing 5: Example target program after polyhedral transformation

4

EXASLANG AND THE EXASTENCILS GENERATOR

Project ExaStencils was already introduced in Section 1.1. This chapter has its focus now on the ExaStencils code generator and the DSL ExaSlang. First, the DSL is presented. After that, we discuss the ExaStencils generator and the workflow of the CUDA code generation prior to this thesis.

4.1 THE DOMAIN-SPECIFIC LANGUAGE EXASLANG

As already mentioned, the domain of ExaSlang is multigrid stencil codes on (semi-) structured grids [12, p. 556]. Such stencil codes appear in multigrid methods, which are asymptotically optimal solvers for elliptic PDEs. In other words one can say that ExaSlang is an external DSL for highly scalable multigrid solvers. One characteristic of ExaSlang is its composition of four hierarchically ordered layers [26, p. 44f.]. The division into four layers is a design decision regarding the major user groups of ExaSlang: engineers, natural scientists, mathematicians, computer scientists. The hierarchy of ExaSlang is shown in Figure 16.

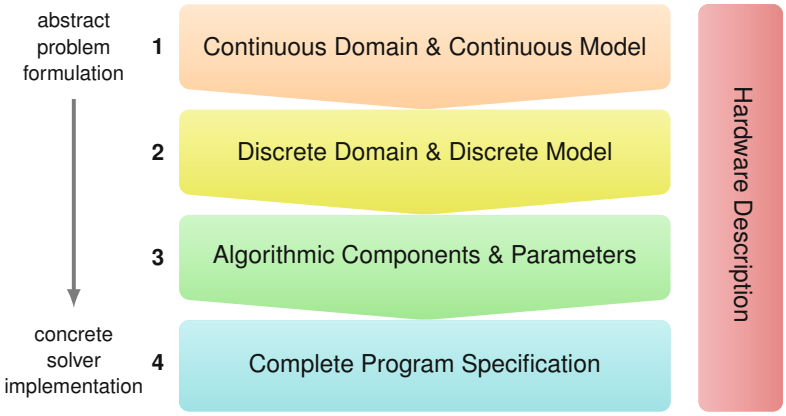


Figure 16: DSL Hierarchy in ExaStencils [12, p. 559]

The first layer, or just ExaSlang 1, is the most abstract layer and is designed for engineers and natural scientists [26, p. 44f.]. With ExaSlang 1 they are able to describe problems in mathematical formulation. Example problems would cover energy functionals that should be minimized or partial differential equations that should be solved on a given domain with corresponding boundary conditions. ExaSlang 2 is the layer for

mathematicians. It is a bit less abstract than ExaSlang 1 and allows to define a problem in a discretized formulation. In the third layer one can identify the multigrid method for the first time. ExaSlang 3 allows not only the specification of the problem in a discretized way, but also the modeling of algorithmic components, settings, and parameter values. This layer is intended for mathematicians. With the last layer, ExaSlang 4, the description of user-relevant parts of the parallelization is possible. For example data structures for data exchange can be chosen and the communication patterns can be selected. Most computer scientists will use ExaSlang 4. Additionally, there is an intermediate representation (ExaSlang IR), which appears only temporarily in the translation process and is not available to the user. This is the reason why ExaSlang IR is not depicted in Figure 16. Orthogonal to the functional description of programs using ExaSlang 1-4 is the target platform description. The target platform description covers the nature of the target system, for example the hardware components like CPUs, memory hierarchies, accelerators, cluster topology and available software like compilers, MPI implementations, and more. These properties can be specified with the target platform description language (TPDL).

The CUDA code generation operates on ExaSlang IR, which results from ExaSlang 4. Hence, we discuss only ExaSlang 4 in greater detail and refer to ExaSlang 4 as ExaSlang for the rest of this thesis.

4.1.1 *Multigrid methods in a nutshell*

Before we address the features of ExaSlang, a brief introduction to multigrid methods is in order. Multigrid methods are iterative solvers for systems arising from a discretization of PDEs [26, p. 44]. The iterative solver traverses between fine and coarse grids in a grid hierarchy. The basic idea is to approximate the unknown errors, to a given approximation on a fine grid, on a coarser grid. This is possible since a coarser grid has fewer discretization points. The combination of this coarse-grid principle and the smoothing property yields a fast rate of convergence. The smoothing property declares in example that classical iterative methods like Jacobi or Gauss-Seidel are able to smooth the error after few steps. The *V-cycle*, described in Algorithm 1, is an example for a multigrid iteration. Taken as a whole Algorithm 1 performs the following actions [26, p. 44]:

(1) *Pre-smoothing:* Smooth high-frequency error components

(2) *Compute residual:* Calculate a new error approximation called residual

(3) *Restrict residual:* Approximate the lower-frequency error components of the residual on coarser grids

(4) *Recursion:* Perform algorithm recursively on coarser grid

(5) *Prolongate error:* Prolongate residual back to the finer grids

(6) *Coarse grid correction:* Eliminate the residual on the finer grids

(7) *Post-smoothing:* Smooth remaining high-frequency error components

Algorithm 1 Recursive V-cycle to solve $u_h^{(k+1)} = V_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$ [26, p. 44]

if *coarsest level* **then**
  solve $A^h \cdot u^h = f^h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} = S_h^{\nu_1}(u_h^{(k)}, A^h, f^h)$ ▷ {pre-smoothing}
  $r^h = f^h - A^h \cdot \bar{u}_h^{(k)}$ ▷ {compute residual}
  $r^H = R \cdot r^h$ ▷ {restrict residual}
  $e^H = V_H(0, A^H, r^H, \nu_1, \nu_2)$ ▷ {recursion}
  $e^h = P \cdot e^H$ ▷ {prolongate error}
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ ▷ {coarse grid correction}
  $u_h^{(k+1)} = S_h^{\nu_2}(\tilde{u}_h^{(k)}, A^h, f^h)$ ▷ {post-smoothing}
end if

4.1.2 *ExaSlang by example*

The aim of this thesis is to compile an ExaSlang program into a well performing CUDA target program. Hence, we discuss in the following the most important points and features of the ExaSlang DSL on the basis of some examples [26, p. 45ff.]. ExaSlang is implemented as an external DSL and obeys the procedural programming paradigm [26, p. 45]. Its syntax partially follows Scala. Furthermore, ExaSlang comes with well-tried language elements, such as data types, functions, loops, variables, and constants. Additionally, the following paragraphs examine some main features of ExaSlang.

LEVEL SPECIFICATIONS The level specifications allow the user to map between functionality and multigrid level, which corresponds to the granularity of the grids [26, p. 45]. ExaSlang affords the opportunity to specify the level that corresponds to the coarsest grid and the higher the level number the finer the grid it refers to. For the sake of simplicity, ExaSlang provides a few keywords to work with grids of different granularity [26, p. 45]:

(1) `coarsest`: the lowermost multigrid level

(2) `finest`: the uppermost multigrid level

(3) `current`: can be used in functions for accessing objects on the same grid granularity

(4) `coarser`, `finer`: multigrid level adjacent to the current one

We clarify the use of level specifications by the example presented in Listing 6. Listing 6 provides a specification in ExaSlang of the V-cycle algorithm, which is defined in Algorithm 1. The example shows how the explained keywords can be used to perform calculations on grids with different granularity. Moreover, it demonstrates how arithmetic can be used to address different multigrid levels. Adding a positive number to a multigrid level results in a finer grid, for example `coarsest + 1`. On the other hand, subtracting a positive number leads to a coarser grid.

```
Function VCycle @((coarsest + 1) to finest) () : Unit {
    repeat 3 times {
        Smoother @current ()
    }
    UpResidual @current ()
    Restriction @current ()
    SetSolution @coarser (0)
    VCycle @coarser ()
    Correction @current ()
    repeat 2 times {
        Smoother @current ()
    }
}

Function VCycle @coarsest () : Unit {
    /* ... Solve directly ... */
}
```

Listing 6: V-cycle specification in ExaSlang equivalent to Algorithm 1

DATA TYPES AND VARIABLES ExaSlang supports a variety of data types, which can be separated into three different categories: *simple data types*, *aggregate data types*, and *algorithmic data types* [26, p. 45]. Table 2 gives an overview of the data types currently available in ExaSlang, ordered by its category.

Data type	Explanation	Example(s)
Simple data types		
`Real`	floating-point numbers	`1.2,0.8435`
`Integer`	whole numbers	`1,2,3,25`
`String`	character sequences	`"ExaSlang"`
`Boolean`	result of comparisons	`true, false`
`Unit`	return type of functions without return value	`Function T() : Unit`
Aggregate data types		
`Complex`	complex numbers; underlying data type must be `Real` or `Integer`	
Algorithmic data types		
`Stencil`	corresponds to matrices	
`Field`	corresponds to vectors	

Table 2: Overview of the data types available in ExaSlang

Variables of simple and aggregate data type can be specified with the keyword `Variable` or short `Var` [26, p. 46]. We can also define constants of simple and aggregate data type with the keyword `Value` or short `Val`. Variable and constant declarations are exemplified by Listing 7.

```
// variable definitions
Variable a : Real = 3.14
Var b : Integer = 3

// constant definitions
Value c : Integer = 0
Val d : Real = 0.8
```

Listing 7: Example of variable and constant definitions in ExaSlang

The category of algorithmic data types derives from the multigrid domain and is used for numerical calculations [26, p. 46]. In a mathematical sense a `Field` corresponds to a vector, for example discretized variables. Technically, a `Field` is an array of certain size, which is linked to the size of the computational domain. A `Field` is specified with the help of a predefined `Layout`. Listing 8 contains the specification of a computational domain, of a `Layout`, and a `Field` using the predefined `ExampleLayout`.

```
Domain global< [ 0, 0, 0 ] to [ 1, 1, 1 ] >
Layout ExampleLayout< Real, Node >@finest {
    innerPoints = [ 512, 512, 512 ]
    ghostLayers = [ 5, 5, 5 ]
    duplicateLayers = [ 1, 1, 1 ]
}
Field ExampleField< global, ExampleLayout, 0.0 >[2]@finest
```

Listing 8: Fields and Layouts

The `Stencil` data type represents matrices in a mathematical sense [26, p. 46]. Its application ranges from smoother implementation to correction or prolongation functions. A `Stencil` is defined by specifying offsets from the grid node and the weight of the grid node's neighbors. Listing 9 provides an example for the definition of a 5-point stencil.

```
Stencil Jacobi@finest {
    [ 0,  0] => 4.8
    [ 1,  0] => -0.8
    [-1,  0] => -0.8
    [ 0,  1] => -0.8
    [ 0, -1] => -0.8
}
```

Listing 9: 5-point Jacobi stencil example

CONTROL FLOW ExaSlang also features control flow elements like loops, branching, and function calls [26, p. 46f.]. These elements are mainly self-explaining and hence they are not discussed in detail at this point. Listing 10 sets some introductory examples. For instance, the `loop over` construct in line 4 specifies a loop over a defined `Field`. Line 11 shows a classical loop defined with the keywords `repeat` and `until`. The function with signature `Function Application() : Unit` in line 16 is something special because it is

the main entry point of a program written in ExaSlang. The `Application` function is translated into a C++ `main()` function by the ExaStencils code generator.

```
Field ExampleField...

Function InitExampleField() : Unit {
    loop over ExampleField@current sequentially {
        ExampleField[active]@finest = 0
    }
}

Function ExampleFunction() : Unit {
    Val repetitions = 10
    repeat until repetitions {
        InitExampleField()
    }
}

Function Application() : Unit {
    ExampleFunction()
}
```

Listing 10: Control flow examples in ExaSlang

A more detailed discussion of the ExaSlang language can be found elsewhere [26].

4.2 WORKFLOW OF THE EXASTENCILS GENERATOR

The previous subsection introduced the ExaSlang DSL and the purpose of the different DSL layers. As mentioned in Section 1.1 project ExaStencils provides a generator compiling ExaSlang to high-performance code. Figure 17 illustrates the principal workflow of the ExaStencils generator [12, p. 555ff.]. The idea behind the ExaStencils generator is to perform a stepwise translation from ExaSlang 1 into high-performance exascale target code. For the sake of simplicity, assume we have a program written in ExaSlang 1. The first step of the ExaStencils generator is to translate the program into ExaSlang 2. The program representation in ExaSlang 2 is then lowered to ExaSlang 3, which is lowered to ExaSlang 4. Domain and platform knowledge are used at every step of this lowering process to guarantee an appropriate translation and optimization.

Finally, the ExaStencils generator parses ExaSlang 4 into ExaSlang IR. The generator applies different transformations, in this context called strategies, on ExaSlang IR. These strategies refine and optimize the ExaSlang IR program using again the provided information about domain and platform. The applied strategies involve both polyhedral and traditional optimizations [10, p. 5]. Example traditional strategies that are called into action are loop unrolling and vectorization [10, p. 9]. For a more detailed description of the optimizations applied by the ExaStencils code generator please refer to the corresponding ExaStencils publication [10]. In the end, ExaSlang IR is compiled to hybrid target code for the specified target system.

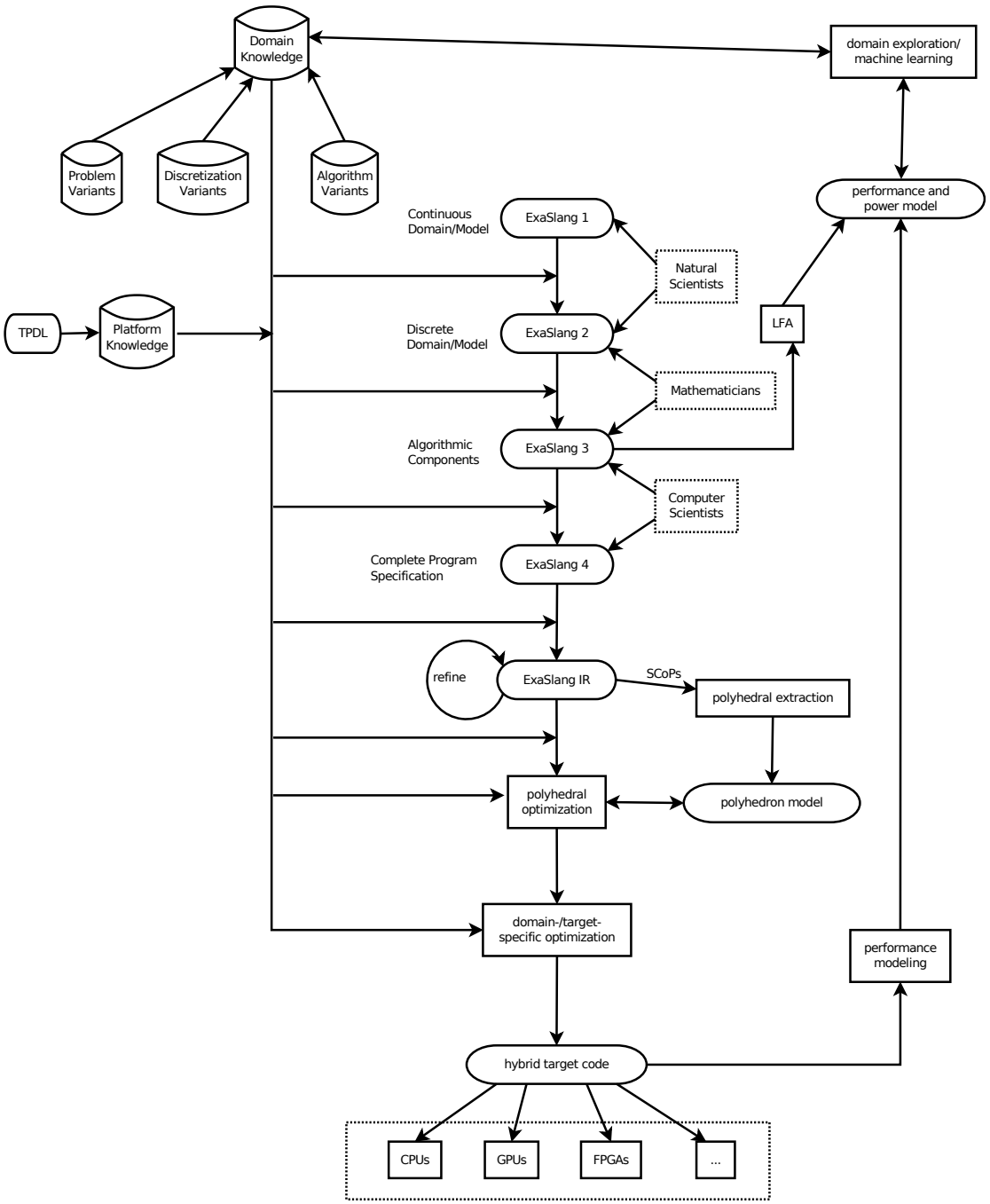


Figure 17: Workflow of the ExaStencils generator

4.3 POLYHEDRAL OPTIMIZATIONS APPLIED BY EXASTENCILS GENERATOR

This section explains the polyhedral optimization strategy of the ExaStencils code generator in greater detail [10, p. 5ff.]. The strategy contains all polyhedral optimizations available in the generator and it is applied on the ExaSlang IR representation of the input program as shown in Figure 17.

Algorithm 2 Polyhedral Optimization Strategy

```
1:  procedure POLY-OPT-STRATEGY(p)
2:      ▷ ExaSlang IR program p
3:      scops ← extract models from p
4:
5:      for all scop ∈ scops do
6:          mergeModels(scop)
7:          simplifyModel(scop)
8:          computeDependences(scop)
9:          if DCE is requested then
10:             deadCodeElimination(scop)
11:         end if
12:         ignoreReductionDependences(scop)
13:         simplifyModel(scop)
14:         if scop offers enough potential for optimization then
15:             searchParallelSchedule(scop)
16:         end if
17:         tileDimensions(scop)
18:         recreateAST(scop)
19:     end for
20: end procedure
```

Algorithm 2 describes the process of the polyhedral optimization strategy. For its polyhedral optimizations the ExaStencils generator takes advantage of the integer set library (isl), which is the most recent C library that supports the polyhedron model [10, p. 4], [29]. The first step of the polyhedral optimization strategy is to extract program parts that can be translated into the polyhedron model [10, p. 5]. These parts are called SCoPs. A program part has to fulfill certain constraints to be recognized as a valid SCoP. These constraints were already discussed in Chapter 3. In this step the polyhedral optimization strategy searches for loop statements in the ExaSlang IR code. If a loop statement forms a valid SCoP it is translated into the polyhedron model. The result of this first action is a list of SCoPs in the polyhedron model. Hence, every SCoP is equivalent to a polyhedron. Afterwards, the strategy traverses this list and performs the following actions on every SCoP in the list [10, p. 5ff.]:

(1) **mergeModels:** Every SCoP is equivalent to one polyhedron. If it is profitable, the strategy merges polyhedra of adjacent SCoPs.

(2) **simplifyModel:** The isl tries to simplify the representation of the SCoP.

(3) **computeDependences:** The generator computes the data dependences inside the polyhedron with the help of the isl.

(4) **deadCodeElimination:** Remove statements / instructions from the iteration domain, whose effect is not visible or not required.

(5) **ignoreReductionDependences:** If there is a reduction in the SCoP, its dependences get ignored to parallelize the reduction. To guarantee correctness, the generated code performs independent, sequential reductions in each thread and then a parallel tree reduction over the accumulators of all threads.

(6) **searchParallelSchedule:** This step forms the central polyhedral optimization. If the SCoP offers enough potential for optimization, the isl scheduler optimizes the schedule of the SCoP by attempting to increase data locality by minimizing the distance of input-dependences, with respect to all other dependences.

(7) **tileDimensions:** Apply classical tiling on the optimized SCoP inside the polyhedron model. The tile size is prescribed by the user.

(8) **recreateAST:** The abstract syntax tree (AST) builder of the isl generates an AST from the SCoP. This AST representation is then transformed to ExaSlang IR code.

4.4 CUDA CODE GENERATION IN PROJECT EXASTENCILS

In the previous subsections we discussed the general workflow of the ExaStencils generator and its polyhedral optimization strategy. This subsection concentrates now on the relevant parts for GPU code generation. At the time of writing the generator is capable of creating CUDA target code for NVIDIA GPUs. Figure 18 visualizes a detail of the ExaStencils generator's workflow. The important parts for CUDA code generation are highlighted. The ExaStencils generator compiles an ExaSlang input program in a CUDA program, only if the user enables the CUDA support at the invocation of the ExaStencils generator.

If CUDA support is enabled, the ExaStencils generator translates the ExaSlang input program into ExaSlang IR. In the next step the generator divides the ExaSlang IR code into a host part and a device part. As discussed in Section 2.4 the CPU is responsible for the host part and the device part is executed by the CUDA-capable GPU. For this division, the ExaStencils generator searches the ExaSlang IR code for loops. If a loop offers potential for parallelization, it is transformed into a kernel and its parallel dimensions are mapped to CUDA's thread identifiers. The CUDA code generation process is divided into the four strategies colored in Figure 18. For the sake of simplicity, the strategies are explained by an example. Listing 11 contains a loop nest L updating an array A. L serves as our starting point for the CUDA code generation. L consists of four nested loops L_1, L_2, L_3, and L_4. So as not to introduce the components of ExaSlang IR, Listing 11 is written in pseudo code.

```
1 for w = 0 ← 1 → 10
2     for z = 0 ← 1 → 10
3         for y = 0 ← 1 → 10
4             for x = 0 ← 1 → 10
5                 A[((1000 * w) + (100 * z) + (10 * y) + x)] = 1
```

Listing 11: Simple example of a loop nest L

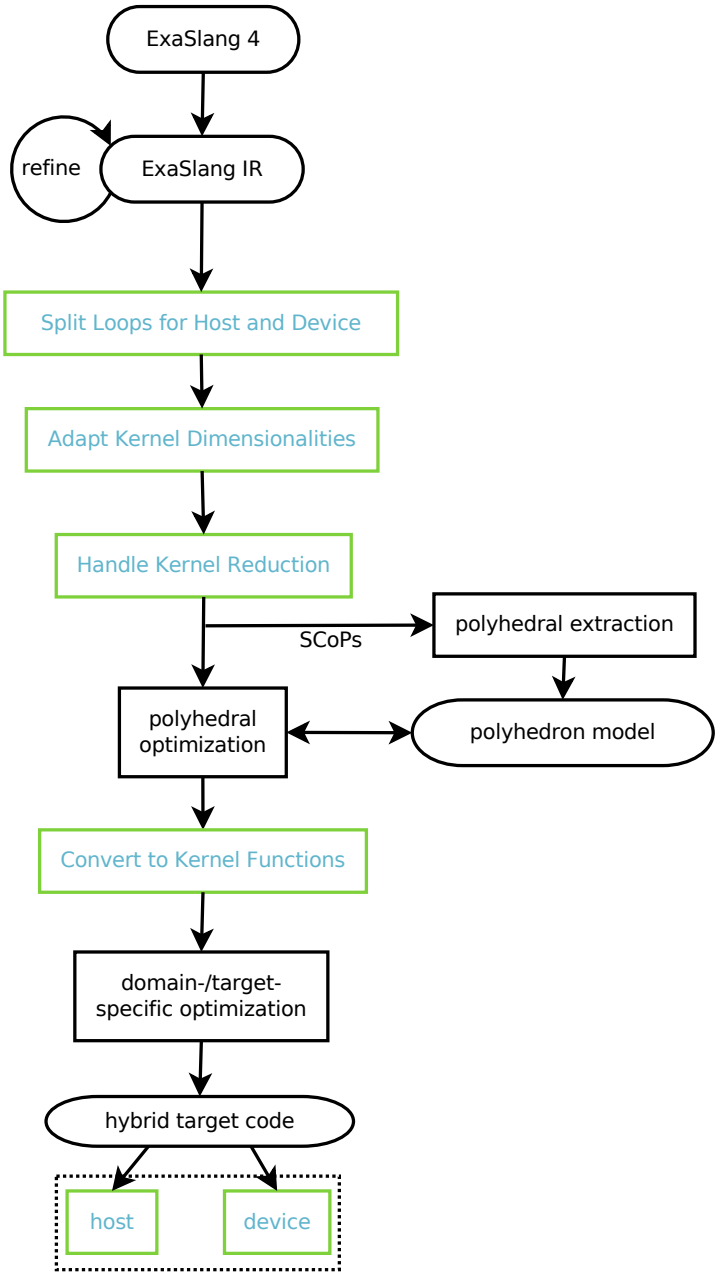


Figure 18: Workflow of the old CUDA code generation

At the end of the CUDA generation process, L will result in the CUDA code in Listing 12. Listing 12 shows a kernel K with the same semantic as L and a wrapper function for calling K.

```
__global__ void K(int _cu_begin_0, int _cu_end_0, int _cu_begin_1, int _cu_end_1
    , int _cu_begin_2, int _cu_end_2, float* A) {
  int _cu_global_x = ((blockIdx.x*blockDim.x)+threadIdx.x);
  int _cu_global_y = ((blockIdx.y*blockDim.y)+threadIdx.y);
  int _cu_global_z = ((blockIdx.z*blockDim.z)+threadIdx.z);
  bool _cu_condition = ((((_cu_global_x>=_cu_begin_0)&&(_cu_global_x<_cu_end_0))
    &&((_cu_global_y>=_cu_begin_1)&&(_cu_global_y<_cu_end_1)))&&((_cu_global_z>=
    _cu_begin_2)&&(_cu_global_z<_cu_end_2)));
  if (_cu_condition) {
    for (int _cu_global_w = 0; _cu_global_w<10; _cu_global_w += 1) {
      A[((1000*_cu_global_w)+(100*_cu_global_z)+(10*_cu_global_y)+_cu_global_x)]
     = 1;
    }
  }
}

extern "C" void K_wrapper() {
  K<<<dim3(2, 2, 2), dim3(8, 8, 8)>>>(0, 10, 0, 10, 0, 10, A);
}
```

Listing 12: CUDA code resulting from Listing 11

The following paragraphs explain the four strategies, which transform the loop L step by step into the kernel K.

SPLIT LOOPS FOR HOST AND DEVICE The first and central part of the CUDA code generation is the division of loops into host and device code. As its name implies, this strategy searches the ExaSlang IR code for loop statements. If a loop statement offers potential for parallelization, the strategy substitutes it by a kernel on the level of ExaSlang IR. Let L be the loop nest in Listing 11. The iterations of the nested loops L_1, L_2, L_3, and L_4 can be executed in parallel and thus L is substituted by a kernel K. The innermost body of L, the assignment of 1 to an array cell, forms the body of K. Additionally, the ExaSlang IR kernel K saves the loop bounds of L as context information required for the compilation. Furthermore, this strategy inserts statements for data synchronization between host and device in the ExaSlang IR code. Data synchronizations take place only if data on host or device is outdated. This guarantees that both, host and device, work with correct data.

ADAPT KERNEL DIMENSIONALITIES The previous strategy substituted the loop nest L from Listing 11 by a kernel K on the level of ExaSlang IR. As already mentioned the parallel dimensions of L are mapped to CUDA's thread identifiers. The current version of CUDA supports only up to three-dimensional thread identifiers. Hence, at most three parallel dimensions can be mapped to CUDA's thread identifiers. This strategy takes care of adapting a kernel's dimensionality by recreating a loop nest in the kernel's body if necessary. The example loop nest L contains four parallel loops, thus one loop is recreated in the body of kernel K by this strategy. This is the reason why the kernel K in Listing 12 contains a loop in its body.

HANDLE KERNEL REDUCTION The next step is to handle kernel reduction. If a kernel requires reduction handling for some assignment, this strategy inserts a temporary buffer and redirects the original assignment to it. Furthermore, a reduction kernel is created for the temporary buffer and an appropriate call of this kernel is inserted in order to calculate the correct result.

CONVERT TO KERNEL FUNCTIONS In the previous strategies the loop nest L was transformed into the kernel K, the kernel's dimensionality was adapted, and reduction was handled. In the last step the body of K is enriched with statements for calculating a thread's global id. The global id is then used for accessing fields appearing in the kernel's body. The lines 2 to 4 in Listing 12 contain the calculation of a thread's global id. This id is then used for accessing the array A in line 8. In addition to it this strategy calculates the kernel's function arguments and creates a wrapper function calling the kernel. The lines 13-15 contain the wrapper function for kernel K. The kernel call in line 14 requires an execution configuration. In the *Split Loops for Host and Device* strategy, the kernel K saved L's loop bounds as context information. This context information is now used to calculate the total number of iterations, which is equivalent to the total number of required threads. Since loop bounds are maybe unknown until runtime, the number of iterations is over approximated. In order to guarantee correct behavior, the conditional in line 6 checks if a thread's global id is out of bounds at runtime. The thread block size for the execution configuration is user-defined. The grid size is calculated from the total number of threads and the user-defined thread block size.

In summary, the ExaStencils generator compiles the loop nest L in Listing 11 to the CUDA code in Listing 12. Since the division into host and device code takes place before the polyhedral optimization strategy is applied, the CUDA code does not benefit from the polyhedron model and its maybe useful loop transformations.

Part II

POLYHEDRAL CODE GENERATION FOR GPUS

5

POLYHEDRAL CODE GENERATION FOR CUDA IN PROJECT EXASTENCILS

In the previous chapter we discussed the general workflow of the ExaStencils code generator and in particular the current CUDA code generation. In a nutshell the underlying idea is to look for multi-dimensional loops in the ExaSlang IR code and map these to kernels. Since the CUDA code generation substitutes loops with kernels before the polyhedral optimization takes hold, the substituted loops do not benefit from the polyhedral optimizations. In the recent past, a lot of different compilers targeting GPUs came up, which exploit the polyhedron model in their code generation process. The Polyhedral Parallel Code Generator or short PPCG [30], C-to-CUDA [2], and Par4All [1] are just a few noteworthy source-to-source compilers in this context. The first part of this chapter gives a summary of related work in the field of polyhedral compilation. The second part introduces the new CUDA code generation of the ExaStencils code generator, which exploits the polyhedron model. The last part introduces further transformations and optimizations for the CUDA code generation.

5.1 RELATED WORK

This section focuses on the state of the art in code generation and optimization tools for GPU targets. In this connection the domain of source-to-source polyhedral compilers is most attractive to project ExaStencils in reference to this thesis. The first noteworthy representative is C-to-CUDA, which was presented by Muthu Manikandan Baskaran et al. in 2010 [2]. C-to-CUDA is one of the first automatic code transformation systems that translates C code into parallel CUDA code. Pluto's algorithm serves as the basis for Baskaran's C-to-CUDA compiler and experiments show that the performance of automatically generated CUDA code can keep up with manually optimized CUDA code. At the time of writing C-to-CUDA remains still a prototype with a field of application limited to a small set of benchmarks. Since 2011 Reservoir Labs[1] offers another representative called the R-Stream compiler [16]. We do not discuss the R-Stream compiler any further because it is a commercial product and no source code is available. Additionally, the validation of this tool seems to be very restricted [30, p. 3]. The most promising candidate is Sven Verdoolaege's et al. PPCG source-to-source compiler developed in 2013 [30]. It expects C code as input and accelerates computations from any static control loop nest by generating CUDA kernels. PPCG uses polyhedral compilation techniques and

1 https://www.reservoir.com

applies new optimization techniques customized for GPU targets like hybrid tiling [6]. PPCG was evaluated on the entire PolyBench[2] suite and convinces with good performance results. PPCG is under further development and the latest release is available at `http://repo.or.cz/w/ppcg.git`. The CUDA code generation in PPCG executes the following steps [30, p. 8ff.]. First, the input C code is translated to the polyhedron model where a dependence analysis takes place. The PPCG compiler exploits parallelism found by an optimization based on the Pluto algorithm. The next step is the mapping to host and device with subsequent tiling. Furthermore, the memory management is addressed and analyses for shared memory and register usage are performed. The last step is the CUDA code generation for the schedule resulting from the polyhedron model. Aside from these polyhedral tools, there are also source-to-source compilers that are not directly based on the polyhedron model. For example Par4All is an automatic parallelizing and optimizing compiler for C and Fortran programs developed by the HPC project [1]. It is based on the source-to-source compiler infrastructure PIPS[3] and benefits from its powerful interprocedural analyses like reduction detection and parallelism detection. It does not use the polyhedron model, but works with an abstract interpretation for array regions, which also involves polyhedra [30, p. 3]. Par4All is capable of generating OpenMP, CUDA, or OpenCL code. In comparison with PPCG it achieves competitive speedup [30, p. 19ff.].

5.2 POLYHEDRAL CUDA COMPILATION WORKFLOW

As stated in [30, p. 6], polyhedral CUDA code generation faces some challenges. First, the source code has to be partitioned into host and device code. Device code has to be mapped to the CUDA environment (threads, blocks, grids). Data locality should be exploited by using shared memory and registers to reduce bandwidth requirements and finally, one has to consider, to schedule, and to control memory transfer. On the following pages we discuss the new CUDA code generation and how it masters the addressed challenges. Figure 19 illustrates its workflow and highlights the important parts of the CUDA code generation. The individual strategies are explained successively.

PREPARE CUDA RELEVANT CODE The first strategy searches the ExaSlang IR code for loop statements. If a loop statement offers potential for parallelization, it is annotated with advice for later CUDA transformation. Additionally, the strategy already inserts statements for data synchronization between host and device in the ExaSlang IR code. Data synchronization takes place only if data on host or device is outdated. This guarantees that both, host and device, work with correct data. In comparison to the old CUDA code generation there are two innovations. First, the strategy considers a wider variety of loops in the ExaSlang IR than the old workflow. The second difference is the noteworthy fact that at this stage no kernel is created and loop statements are not substituted with anything else.

2 `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`
3 `http://pips4u.org`

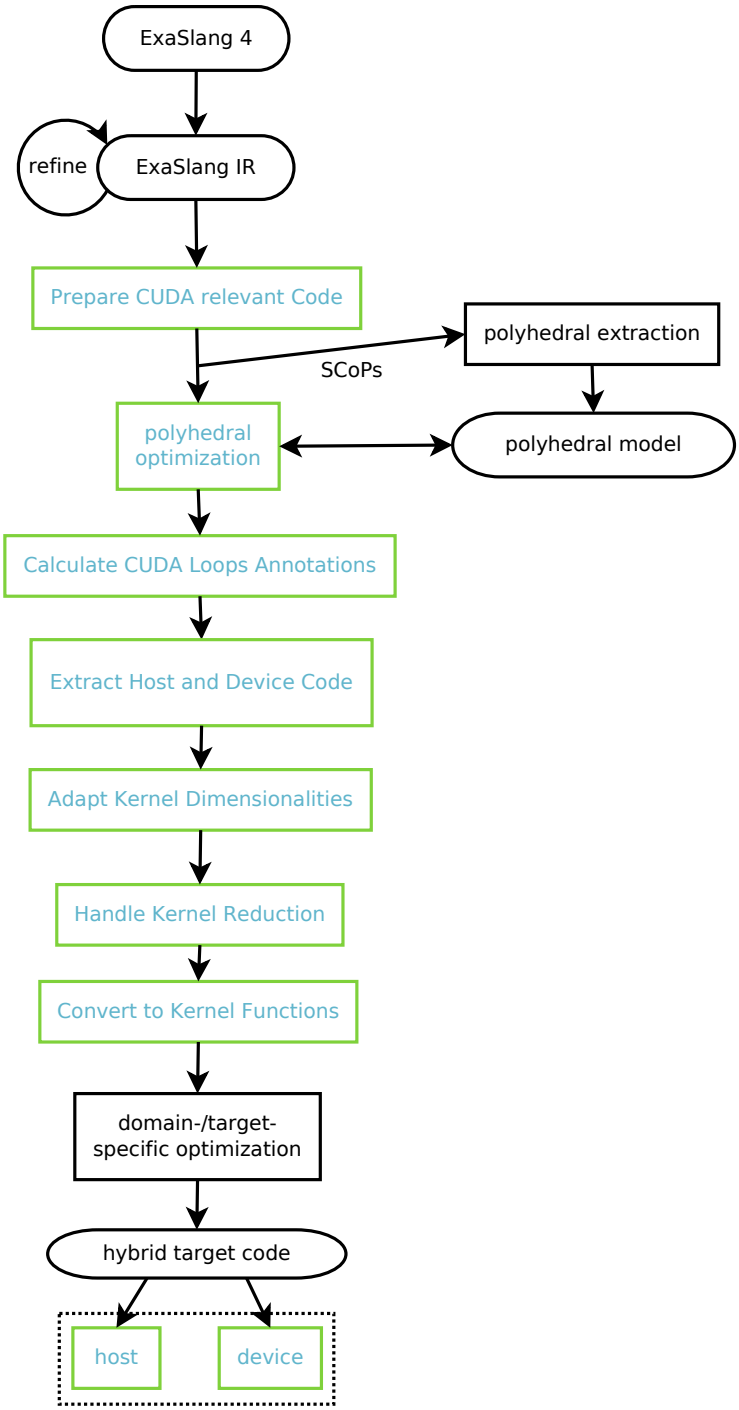


Figure 19: Workflow of the polyhedral CUDA compilation

POLYHEDRAL OPTIMIZATION The loop statements annotated in the previous step offer potential for parallelization. In other words, these statements form valid SCoPs and can be translated to the polyhedron model. Hence, loops intended for transformation into kernels can now benefit from the polyhedral optimizations discussed in Section 4.3.

CALCULATE CUDA LOOPS ANNOTATIONS This strategy is to look for loop statements that feature the annotation from the first step. Let L be such an annotated n dimensional loop nest looking as shown in Listing 13. Let $L_1, \ldots, L_n$ denote the individual loops.

```
L_1: for i_1 = l_1 ← 1 → u_1
      ⋮
L_n:    for i_n = l_n ← 1 → u_n
          // body...
```

Listing 13: n perfectly nested loops

The purpose of this strategy is to annotate the loops $L_1, \ldots, L_n$ with specific information to guarantee an easier and correct transformation into a kernel. This loop annotation process is shown in Algorithm 3. Moreover, the strategy determines, for each loop $L_i, 1 \leqslant i \leqslant n$, the extrema of the loop bounds l_i and $u_i, 1 \leqslant i \leqslant n$, and saves them as context information for later calculations.

Algorithm 3 Calculate CUDA relevant loop annotations

procedure CALCULATE-LOOP-ANNOTATIONS(L_1)
  ▷ The outermost loop L_1

  ▷ Find the first parallel loop in the loop nest
  $S \leftarrow L_1$
  while Is S not parallel? **do**
    $S \leftarrow$ get for loop in body of S
  end while
  if Is S parallel? **then**
    Annotate S with the information $Band - Start$ (indicating that this is the outermost parallel loop)

    ▷ Annotate all direct successive parallel loops with appropriate information
    $P \leftarrow$ get loop in body of S
    while Is P parallel? **do**
      Annotate P with the information $Band - Part$ (indicating that this loop is part of a parallel loop nest)
      $P \leftarrow$ get loop in body of P
    end while

    ▷ Annotate all remaining loops
    if Is P not parallel? **then**
      Annotate P with the information $Inner$
    end if
    Annotate all loops in the body of P with the information $Inner$
  end if
end procedure

EXTRACT HOST AND DEVICE CODE This strategy addresses loop statements with the annotated information $Band-Start$. Let L_1 denote such a loop statement. From the previous strategy we know that the body of L_1 contains $n-1$ nested loops $L_2, \ldots, L_n$ with annotation $Band-Part$. The strategy replaces $L_1, \ldots, L_n$ by a kernel K in the ExaSlang IR code. The body of L_n defines the body of K. The loops $L_1, \ldots, L_n$ are mapped to CUDA's thread identifiers because they can be executed in parallel according to their annotations.

ADAPT KERNEL DIMENSIONALITIES This strategy searches the ExaSlang IR code for kernels and verifies their dimensionalities. Let K be such a kernel and let $L_1, \ldots, L_n$ denote the loops, which were replaced by K in the previous step. As already mentioned $L_1, \ldots, L_n$ are mapped to CUDA's thread identifiers. The current version of CUDA supports only up to three-dimensional thread identifiers. Hence, at most three parallel dimensions can be mapped to CUDA's thread identifiers. This strategy takes care of adapting a kernel's dimensionality by recreating a $(n-3)$-dimensional loop nest in the kernel's body if $n > 3$.

HANDLE KERNEL REDUCTION This strategy has the same behavior as in the old CUDA generation workflow explained in Section 4.4.

CONVERT TO KERNEL FUNCTIONS The last step of the new CUDA code generation is quite the same as in the old workflow explained in Section 4.4. The strategy searches the ExaSlang IR code for kernels and completes the body of a kernel K. The body of K is enriched with statements for calculating a thread's global id. The global id is then used for accessing fields appearing in the kernel's body. In addition, this strategy calculates the kernel's function arguments and creates a wrapper function calling the kernel. The kernel call requires an execution configuration. Let L denote the loop nest the kernel K was created from. As mentioned above, the *Calculate CUDA Loops Annotations* strategy saved the extrema of the loop bounds of L as context information. This context information is now used to calculate the total number of iterations, which is equivalent to the total number of required threads. Since we work with extrema of loop bounds, the number of iterations is over-approximated. In order to guarantee correct behavior, a conditional is inserted that checks whether a thread's global id is out of bounds at runtime. The thread block size for K's execution configuration is user-defined. The grid size is calculated from the total number of threads and the user-defined thread block size. In contrast to the old workflow, further extensions like shared memory utilization are applied by this strategy. The new available extensions are discussed in the following subsection.

5.3 CUDA CODE GENERATION EXTENSIONS

The previous subsection discussed the new workflow of the CUDA code generation in the ExaStencils generator. In addition to modifications on the workflow itself, further extensions were installed as part of this thesis. This section concentrates on these ex-

tensions and explains each one in detail. We use the terms extension and optimization interchangeably. The principal aim of all extensions is to improve the speedup in comparison to the sequential program executed on a CPU.

Micikevicius [17] describes a parallelization of the 3D finite difference computation for GPU targets using the CUDA framework. In this context he discusses an efficient CUDA implementation for stencils. Maruyama and Aoki [15] compare different stencil optimizations for NVIDIA Kepler GPUs. Among other things, they examine spatial blocking with read-only cache and spatial blocking with shared memory. The latter is also used by Micikevicius. We adopt the ideas of Maruyama and Micikevicius and introduce shared memory, spatial blocking with shared memory, and spatial blocking with read-only cache to the ExaStencils code generator as part of this thesis.

5.3.1 *Shared memory utilization*

NVIDIA's SIMT execution model is based on the idea to apply a single instruction on multiple data by running many threads in parallel. As mentioned in Section 2.2.2, all threads simultaneously access the global memory, which causes a high pressure on the memory bandwidth [30, p. 6]. As reported by NVIDIA and several researchers the memory bandwidth utilization has a significant impact on the performance of a CUDA program [17, p. 2], [15, p. 2]. Hence, an important optimization is the reduction of global memory accesses because they are not implicitly cached by the hardware. This can be achieved by optimizing memory accesses and introducing shared memory. Furthermore, one exploits inter-thread locality and data reuse with an intelligent shared memory utilization. If the ExaStencils code generator takes shared memory into account, it performs two additional steps in the workflow presented in Figure 19. On the one hand, the *Extract Host and Device Code* strategy examines the newly created kernels and analyzes their field accesses. If a field access is suitable for shared memory, it is annotated with this information. Algorithm 4 describes this analysis process and explains the condition for shared memory suitability. On the other hand, the *Convert to Kernel Functions* strategy translates previously annotated field accesses to appropriate shared memory accesses in ExaSlang IR. Additionally, each kernel body that has shared memory accesses is enriched with statements for allocating shared memory and loading the necessary data from global into shared memory.

The allocation and loading process can be best explained with an example. Let K be a kernel, which corresponds to a simple two-dimensional 5-point stencil code. In this case each CUDA thread is responsible for updating exactly one point or more specifically the thread with id (x, y) updates the point $(x + 1, y + 1)$. (CUDA thread ids are zero-based and in our example we have one halo data point in every dimension, thus there is an offset of 1 when addressing the actual data point.) Assume that the CUDA threads are organized in 5×5 thread blocks. Since each thread block has its own shared memory, we concentrate on a single thread block in the following explanation. First a thread block allocates a two-dimensional chunk of shared memory for its corresponding subdomain. Figure 20 shows such a 5×5 subdomain of our example stencil, which is updated by one thread block. The white blocks are points of the stencil that need to be updated.

Algorithm 4 Analyze field accesses for shared memory

```
1: fieldToFieldAccesses ← extract all field accesses appearing in the kernel body and
   store them as map from field name to a list of all respective field accesses
2: availableSmem ← read in the shared memory amount of the target GPU
3:
4: ▷ Fields with the most accesses are considered first
5: Sort fieldToFieldAccesses according to the number of field accesses from high to
   low
6:
7: for all (field, fieldAccesses) ← fieldToFieldAccesses do
8:     ▷ Evaluate all conditions to be complied with
9:     requiredMem ← calculate memory required to store field in shared memory
10:    basic ← are there at least two field accesses?
11:    readOnly ← is field only read?
12:    enoughMem ← is there enough shared memory available?
13:    neighborsRequired ← are there at least two different accesses per iteration?
14:
15:    ▷ Is field suitable for shared memory?
16:    if basic ∧ readOnly ∧ enoughMem ∧ neighborsRequired then
17:        annotate field for shared memory
18:        availableSmem ← availableSmem − requiredMem
19:    end if
20: end for
```

The gray blocks illustrate halo data, which is introduced for the sake of simplicity to circumvent costly border updates. Assume that the current thread has the id $(3, 3)$ and consequently is responsible for the blue point in Figure 20. As we deal with a 5-point stencil we require the four adjacent green points and the old value of the blue point for an update of the blue point.

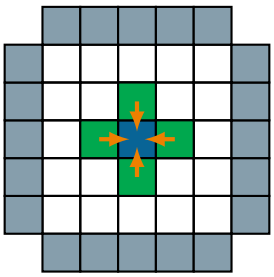
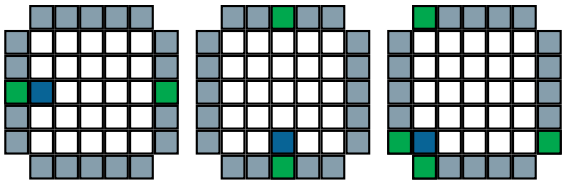


Figure 20: Two-dimensional 5-point stencil with halo data

The goal is that a thread reads the required values only from shared memory and not from global memory. Hence, each thread stores the old value of its point in shared memory. In this way all white points in Figure 20 are present in shared memory, but required halo data is still missing. The simplest approach would be to check if a thread is responsible for a margin point and to prompt this thread to load halo data. This

approach would require a check if a thread's id is at a lower margin and an additional check if it is at an upper margin. In order to save one check, we only verify whether a thread's id is at a lower margin. If this is true, the corresponding thread is responsible for loading halo data to shared memory as illustrated in Figure 21. The blue points in the subfigures are the points that should be updated by the current thread and the green points represent the halo data loaded by this thread. A thread with id $(0, y)$ additionally loads the halo data at $(0, y)$ and $(6, y)$. A thread with id $(x, 0)$ additionally loads the halo data at $(x, 0)$ and $(x, 6)$. Added together, a thread with id $(x, y), x = 0 \vee y = 0$, regards the required halo data.



(a) Left margin (b) Bottom margin (c) Corner point

Figure 21: Loading required halo data into shared memory

The current shared memory utilization of the ExaStencils generator lacks two important points. Firstly, the ExaStencils generator does not support a write back from shared memory to global memory. Hence, just read-only fields can be considered for shared memory. Secondly, the ExaStencils generator cannot handle diagonal neighbor relations. The latter point has no bearing on this thesis because most test cases involve Jacobi stencils, which do not have diagonal neighbors.

5.3.2 *Spatial blocking with shared memory*

The shared memory utilization can be improved further with explicit blocking [17, 15]. Currently, spatial blocking with shared memory can only be applied to kernels that arise from a parallel three-dimensional loop nest working on a three-dimensional field. As a rule, such a kernel would be called with a three-dimensional thread block configuration in order to process one point of the field per thread as discussed in the previous section.

In the case of spatial blocking with shared memory, this kernel is instead called with a two-dimensional thread block configuration [17, p.2f], [15, p. 3f.]. Let z be the outermost dimension of the source loop nest and let Figure 22 show the three-dimensional subdomain of the field that is processed by one thread block. The field points in the x-y-planes are computed in parallel by the threads in a thread block, whereas the computation over the z-direction is swept sequentially per thread. In other words, the three-dimensional subdomain showed in Figure 22 is sliced in two-dimensional pieces indicated by the blue plane. Threads in a thread block traverse the volume along the z-direction coherently and compute output for each slice. As a consequence, data reuse between threads in a thread block only exists for a single x-y-plane. Hence, we only need one two-dimensional chunk in shared memory. Additional values required in z dimension are

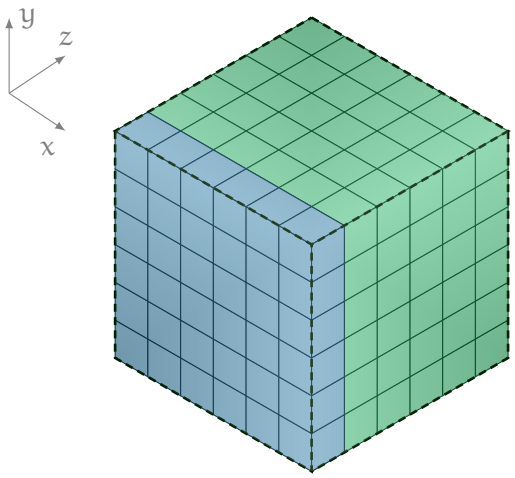


Figure 22: Three-dimensional subdomain processed by one thread block

stored in thread's registers. Once all threads in a thread block finished processing the current slice the next slice is loaded to shared memory and values in local variables are shifted in z-direction.

5.3.3 *Spatial blocking with read-only cache*

NVIDIA GPUs of the Kepler generation feature a 48KB read-only data cache that is directly accessible by the SMX units and can be used for general load operations as stated in Section 2.2.2. It is a separate cache equipped with a separate memory pipe [23, p. 6]. Furthermore, the read-only data cache has relaxed memory coalescing rules. For this reason NVIDIA recommends its usage in case of bandwidth-limited kernels for performance improvements. Recent experiments evidence this proposition [15]. The read-only data cache is automatically utilized by the NVIDIA compiler where data is guaranteed read-only for the duration of the kernel. The NVIDIA compiler detects that this condition is satisfied, if the data is marked with both the `const` and `__restrict__` qualifiers. This is implemented in the ExaStencils code generator as follows. If spatial blocking with read-only cache is enabled, the *Convert to Kernel Functions* strategy analyzes the kernel body and searches for read-only fields. If a located read-only field is provided as kernel argument, the related argument is marked with the `const` and `__restrict__` qualifiers. It should be noted that the current version of the ExaStencils code generator does not support shared memory usage in combination with spatial blocking with read-only cache because both optimizations target read-only fields and currently no heuristic is implemented that is able to decide whether a field should reside in shared memory or read-only cache.

5.4 POLYHEDRAL SCHEDULE EXPLORATION

The overall aim of this thesis is to produce performance optimal result code for GPU targets. The pre- and post-smoothing steps in the V-cycle algorithm are among the most

runtime intensive parts of multigrid methods. Hence, these steps offer a lot of potential for optimization. A smoother is implemented as stencil code like Jacobi, Red-Black Gauß-Seidel, or others. The stencil codes form valid SCoPs and can be translated into the polyhedron model as discussed in Chapter 3. In this context, a scheduler assigns a new execution time to each statement instance. As previously mentioned, the ExaStencils generator takes advantage of the isl, which offers a Pluto-like scheduler. The Pluto algorithm optimizes for parallelism and locality [9]. Its basic approach is to create the space of all legal transformations in the polyhedron model and to choose the best among them according to an affine objective function.

As part of project ExaStencils, Stefan Kronawitter examined the Jacobi smoother and its performance after optimizing with the isl scheduler [9]. The scheduler was not able to find the best transformation for several smoothing steps. As a consequence, Stefan Kronawitter applied a polyhedral search space exploration for CPU targets in order to identify the performance-optimal transformation among all legal ones. This chapter explains the performed polyhedral search space exploration and discusses the algorithm used for an exploration for GPU targets.

POLYHEDRAL SEARCH SPACE EXPLORATION The set of all legal one-dimensional transformations is a (unbounded) polyhedron [9]. A n-dimensional stencil requires n linearly independent schedules and a polyhedron for a higher dimension depends on all previous ones. The problem is to find the performance-optimal transformation among the legal ones. For this purpose there are several different approaches. On the one hand, the search space can be heuristically restricted and a full exploration can be performed. However, in many cases the search space is still too large or too complex to enumerate all elements efficiently. The second approach is to use a generator-based (dual) representation for the exploration. This is the chosen approach for ExaStencils.

The generator-based approach uses Chernikova's algorithm for finding an irredundant set of vertices V and rays R for a given polyhedron [9], [31]. With respect to schedules it holds that $V = \{\overrightarrow{0}\}$. The set of valid schedules can then be defined as $S = \{s : s = \sum_{r \in R} c_r \cdot r \wedge c_r \in \mathbb{Q}_0^+\}$. Since the exploration examines a subset $C \subset \mathbb{Q}_0^+$, the examined schedules are defined as $\hat{S} = \{s : s = \sum_{r \in R} c_r \cdot r \wedge c_r \in C\} \subset S$. Additionally, spatial blocking is required for larger grids, but this is only possible if no schedule dimension violates dependences carried by a previous one.

Stefan Kronawitter showed in his experiments that his polyhedral search space exploration is capable of finding schedules for the Jacobi smoother outperforming the isl scheduler [9]. Furthermore, he presented common properties of the schedules with the highest performance. These properties involve consecutive memory accesses, possibility of parallel execution, vectorizability, parallelism of the second dimension. In summary it can be said, therefore, that there are huge performance differences between valid schedules with regard to the Jacobi smoother targeting CPUs. Hence, this begs the question if the polyhedral search space exploration is also able to provide performance-optimal results for GPU targets. For this reason the polyhedral search space exploration is adapted to the new CUDA code generation workflow presented in Section 5.2. Section 6.2.2 discusses the performed experiments for the Jacobi smoother targeting GPUs.

6

EXPERIMENTS

This chapter assesses the new CUDA code generation on the basis of several experiments. Furthermore, the effect of the implemented extensions is evaluated. The last part analyzes the performance limitations of ExaStencils' generated CUDA code. In the end, we discuss the hybrid tiling feature of PPCG and evaluate its benefit for the ExaStencils domain.

6.1 EXPERIMENTAL FRAMEWORK

All experiments were conducted on an NVIDIA GeForce GTX TITAN Black GPU as described in Section 2.3. All timings include the data transfer overhead to and from the GPU. Furthermore, all calculations were performed as single precision floating point computations. Experiments concerning the CPU performance were run on an Intel Xeon E5-2690v2 CPU with 3,0 GHz operating frequency, 10 cores, and two hyper-threads per core, plus 64 GB RAM. We worked on Ubuntu 16.04 with CUDA Toolkit 7.5 and g++ 5.4.0. The compilation was performed with the flags described in Table 3. A detailed explanation of the flags can be found elsewhere [21, 27].

g++ flags	
-O3	Apply certain optimizations
-std=c++11	Select C++11 standard
-DNDEBUG	No debugging
nvcc flags	
-O3	Apply certain optimizations
-std=c++11	Select C++11 standard
-DNDEBUG	No debugging
-lineinfo	Generate line-number information for device code
-arch=sm_35	Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled

Table 3: Applied g++ and nvcc compilation flags

If we generated CUDA code with the ExaStencils generator, we used a thread block size of $(8, 8, 8)$ for three-dimensional problem sizes and a thread block size of $(16, 16)$

for two-dimensional problem sizes. According to NVIDIA, it is beneficial to choose a thread block size that is a multiple of the warp size (32).

6.2 EXPERIMENTAL SETUP

This section describes the performed experiments and discusses their results. For better understanding, some results are displayed graphically.

All test programs used in the following experiments were compiled by the ExaStencils generator. We compiled some test programs with 14 different configurations in order to assess available optimizations and the extensions introduced in Section 5.3. Table 4 lists the 14 configurations.

Variant	poly opt[a]	smem[b]	L1 cache[c]	sb smem[d]	sb ROC[e]	old[f]
0	false	true	false	true	false	false
1	false	true	false	false	false	false
2	false	false	true	false	true	false
3	false	false	true	false	false	false
4	false	false	false	false	true	false
5	false	false	false	false	false	true
6	false	false	false	false	false	false
7	true	true	false	true	false	false
8	true	true	false	false	false	false
9	true	false	true	false	true	false
10	true	false	true	false	false	false
11	true	false	false	false	true	false
12	true	false	false	false	false	true
13	true	false	false	false	false	false

a apply polyhedral optimizations
b shared memory is used
c use 48KB of L1 cache and 16KB of shared memory
d apply spatial blocking with shared memory
e apply spatial blocking with read-only cache
f use the old CUDA code generation workflow without any optimizations

Table 4: Overview on the different generated program variants with information about the used extensions and optimizations.

Unless otherwise specified in the following experiments, the compiled test programs / variants were executed five times and the median performance of the five runs is used for comparison.

6.2.1 *Experiment 1 - sequential performance and worthwhile parts for optimizations*

In the first experiment we address the sequential performance on the CPU of an example problem from the ExaStencils domain, in order to identify the performance-critical parts. As example problem we consider the steady-state heat equation with Dirichlet boundary conditions on the unit square, which describes the asymptotic heat distribution in a square region. The steady-state heat equation is better known as Laplace's equation [14, p. 66]. We investigated two different configurations of this problem, which represent a constant or smoothly changing thermal conductivity. We refer to the constant changing thermal conductivity as the *constant test case* and the smoothly changing thermal conductivity as the *variable test case*. Furthermore, we examined a two-dimensional and a three-dimensional problem size for both the constant and the variable test case.

We executed all test cases five times on the CPU and examined the best run in greater detail. In the following we discuss the performance results of the three-dimensional variable test case. Table 5 depicts an excerpt of its runtime results. The excerpt contains the results on the finest multigrid level, level eight, since this is the costliest level. The results reveal that the pre-smoothing and post-smoothing steps are clearly the most runtime-intensive parts. All test cases use the Laplace operator for smoothing.

Solver step	**Time in ms**
Total time to setup	191.545
Total time to solve in 4 steps	7609.83
Mean time per V-cycle	460.414
Total time spent on level 8 in pre-smoothing	557.707
Total time spent on level 8 in updating residual	186.886
Total time spent on level 8 in restricting	66.5284
Total time spent on level 8 in setting solution	2.9271
Total time spent on level 8 in prolongating and correcting	249.355
Total time spent on level 8 in post-smoothing	558.966

Table 5: Sequential runtime of the three-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity

Appendix A provides the complete runtime results and further runtime information about all considered test cases. All show the same effect: the smoother is the performance-critical part of the example problem.

6.2.2 *Experiment 2 - best performing schedule for smoother*

The previous experiment showed that the pre-smoothing and post-smoothing steps are performance-critical parts of a V-cycle. Hence, we used the polyhedral search space

exploration explained in Section 5.4 to find the optimal performing schedule for a smoother. Appendix B shows in Listing 20 a test program examining the performance of a defined smoother in MLUPs. The smoother uses the three-dimensional Laplace operator based on the Jacobi stencil. Furthermore, the smoother applies a technique called temporal blocking, which performs multiple updates on a small block of the computational domain before proceeding to the next block [32, p. 579]. Hence, temporal blocking introduces dependences, which allow to explore several different schedules for the smoother. The polyhedral search space exploration is able to obtain 12049 schedules. The test program was compiled 12049 times with the ExaStencils generator, once for each schedule. The ExaStencils generator applied the new CUDA code generation workflow to obtain CUDA code and used no further extensions. Figure 23 visualizes the performance results of the different schedules.

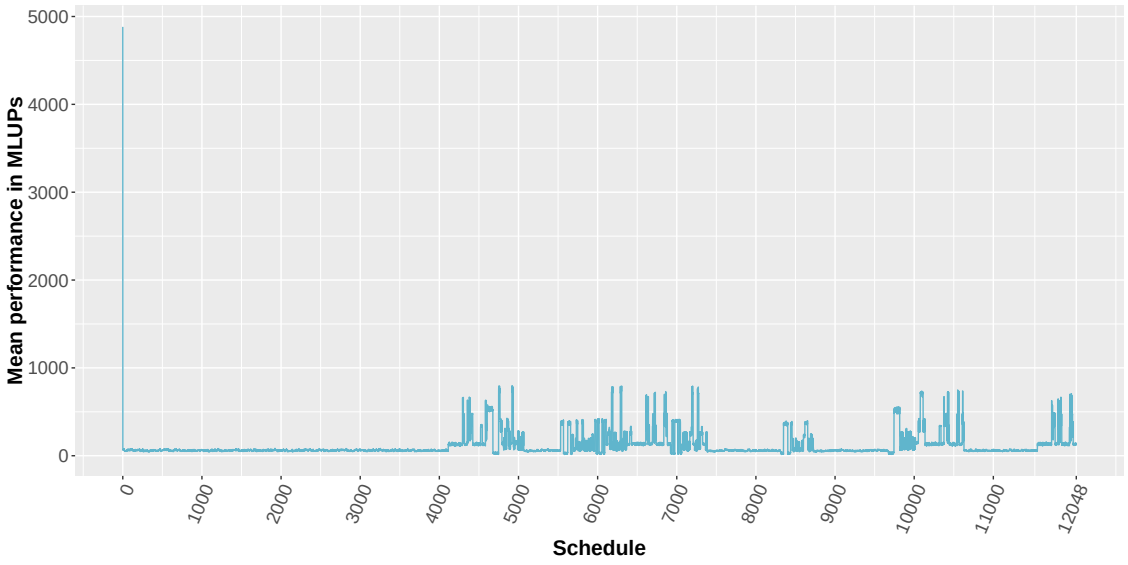


Figure 23: Polyhedral schedule exploration results

Remarkably enough, the initial schedule 0 outperforms all other schedules. The reason seems to be that the initial schedule already offers full parallelism. The iterations can be performed in parallel with low compute intensity per iteration and thus per CUDA thread. All transformations lead to schedules with higher compute intensity per CUDA thread and with constraints on parallel execution. For a better understanding we compare a smoother kernel of schedule 0 with the equivalent smoother kernel of schedule 6304, which is the second best schedule. We start our comparison with the internal smoother loop nest, which results from the polyhedral optimizations. Figure 24 shows the loop nest of schedule 0 on its left and the loop nest of schedule 6304 on its right. For a better readability unimportant details are omitted. There are two differences that cause the worse performance of schedule 6304. First, the ExaStencils generator is able to identify the whole loop nest of schedule 0 as parallel, but the generator recognizes that the loop on `_i3` of schedule 6304 is not parallel. Hence, the loop on `_i3` will be executed sequentially by the host. Second, the loop on `_i5` of schedule 6304 depends on the loop

on _i3. This dependency is the reason why the generator is not able to map the loop on _i5 to CUDA's thread identifiers.

```
for (int _i1 = ((3 * iterationOffsetBegin
    [2]) - 2); _i1 <= ((3 *
    iterationOffsetEnd[2]) + 515); _i1
    += 1) {
  for (int _i2 = ((3 *
    iterationOffsetBegin[1]) - 2); _i2 <
    = ((3 * iterationOffsetEnd[1]) + 515)
    ; _i2 += 1) {
    for (int _i3 = ((3 *
    iterationOffsetBegin[0]) - 2); _i3 <
     ((3 * iterationOffsetEnd[0]) + 516);
     _i3 += 1) {
      solutionData[...] = ...;
    }
  }
}
```

```
for (int _i3 = (-517 - (3 *
    iterationOffsetEnd[1])); _i3 < (-515
     - (2 * iterationOffsetEnd[1])); _i3
     += 1) {
  for (int _i4 = (_i3 - ((3 *
    iterationOffsetEnd[0]) + 515)); _i4
    <= ((_i3 + 2) - (3 *
    iterationOffsetBegin[0])); _i4 += 1)
     {
    for (int _i5 = (((3 *
    iterationOffsetBegin[2]) + _i3) - 2);
     _i5 < ((3 * iterationOffsetEnd[2]) +
     _i3 + 516); _i5 += 1) {
      solutionData[...] = ...;
    }
  }
}
```

Figure 24: Smoother loop comparison between schedule 0 and schedule 6304

If we consider the resulting CUDA code, these two differences become more obvious. Figure 25 depicts the kernel of schedule 0 on the left and the kernel of schedule 6304 on its right.

```
__global__ void Smoother(...) {
  int _cu_global_x = ...;
  int _cu_global_y = ...;
  int _cu_global_z = ...;
  bool _cu_condition = ...;
  if (_cu_condition) {
    solutiontData_0_o1[...] = ...;
  }
}

extern "C" void Smoother_wrapper(...) {
  Smoother<<<dim3(65, 65, 65), dim3(8, 8,
      8)>>>(...);
}
```

```
__global__ void Smoother(...) {
  int _cu_global_x = ...;
  bool _cu_condition = ...;
  if (_cu_condition) {
    int _start = ...;
    int _end = ...;
    int _intermediate = ...;
    for (int _i5 = _start; _i5<
    _intermediate; _i5 += 2) {
      solutiontData_0_o1[...] = ...;
      solutiontData_0_o1[...] = ...;
    }
    for (int _i5 = _intermediate; _i5<
    _end; _i5 += 1) {
      solutiontData_0_o1[...] = ...;
    }
  }
}

extern "C" void Smoother_wrapper(...) {
  Smoother<<<66, 8>>>(...);
}
```

Figure 25: Smoother kernel comparison between schedule 0 and schedule 6304

The wrapper function of schedule 0 launches the kernel with a three-dimensional execution configuration. Since the loop on _i3 and _i5 cannot be mapped to CUDA's thread identifiers, the kernel of schedule 6304 is called with an one-dimensional execution configuration. Furthermore, the kernel of schedule 6304 sequentially executes the loop on _i5, which increases the compute intensity per CUDA thread.

Figure 26 reveals how the wrapper functions in Figure 25 are called. Schedule 6304 calls its wrapper function inside of the loop on _i3 whereas schedule 0 calls the wrapper just once.

```
Smoother(...);
```

```
for (int _i3 = (-517-(3*
    iterationOffsetEnd[1])); _i3<
    (-515-(2*iterationOffsetEnd[1]));
    _i3 += 1) {
    Smoother(...);
}
```

Figure 26: Smoother kernel call comparison between schedule 0 and schedule 6304

In summary, schedule 6304 performs worse than schedule 0 because of the sequential loops executed both on host and device. Overall, the polyhedral schedule exploration is not able to find a schedule better than the initial schedule in case of the examined smoother.

6.2.3 *Experiment 3 - performance impact of the CUDA code generation extensions on smoothing*

The next step is to examine the impact of the extensions presented in Section 5.3 on the performance of a smoother. Appendix C describes in Listing 21 a test program written in ExaSlang. It measures the performance of a smoother in MLUPs. The smoother represents the Laplace operator implemented with the Jacobi stencil. The test program executes the smoother once for cache warmup and nine times for measurement. We compiled the test program with the ExaStencils generator and provided 14 different configurations. As a result we obtained 14 different variants of the test program. Table 4 lists the 14 configurations.

Figure 27 visualizes the performance in MLUPs. A larger value indicates a better performing program. The red horizontal line in Figure 27 indicates the performance of the sequential program execution on the CPU without optimizations. As expected, the performance of all variants is considerably improved in comparison to the sequential program. The variants 0 to 6 use no polyhedral optimizations, whereas variants 7 to 13 do. Figure 27 reveals that the polyhedral optimizations do not improve the performance. There is one simple reason for that. The loop nest in ExaSlang IR that servers as input for the polyhedral optimizations already offers full parallelism. Hence, the scheduler does not perform any transformations.

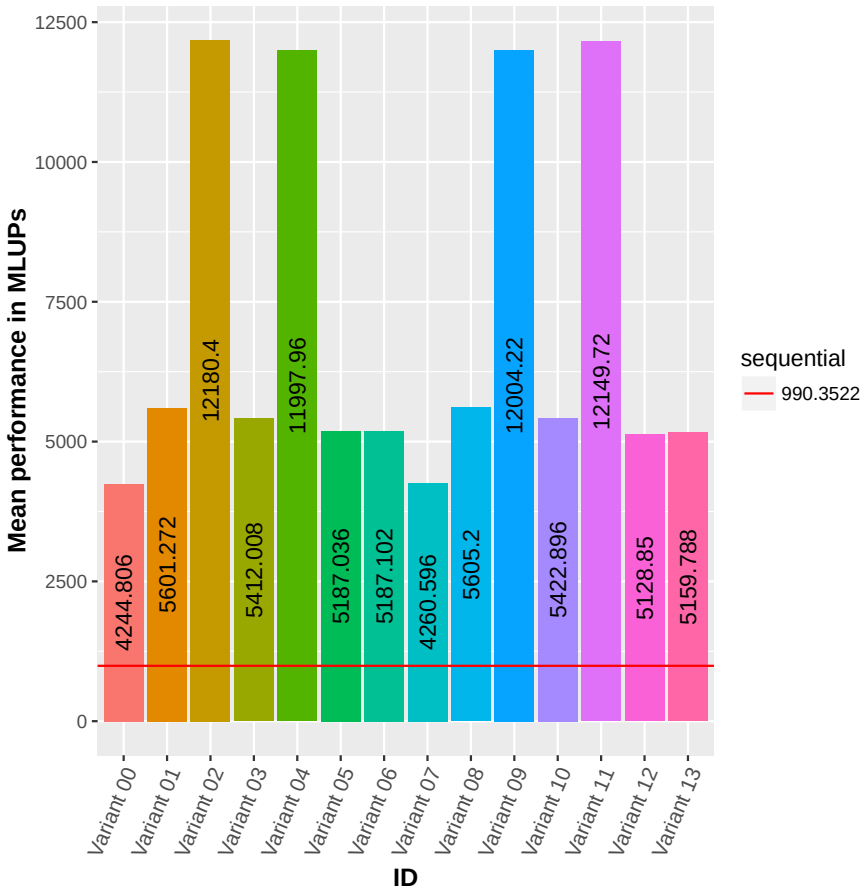


Figure 27: Performance comparison of smoother variants in MLUPs

Furthermore, the variants 2, 4, 9, and 11 exceed all other variants. All four variants rely on spatial blocking with the read-only cache and use no shared memory. The two worst performing variants are the ones using spatial blocking with shared memory. Interestingly enough, they are even worse than the variants 5, 6, 12, and 13, which utilize no extensions. In comparison to the other variants, spatial blocking with shared memory performs poorly because only a two-dimensional execution configuration is used for the kernels and each of the CUDA threads traverses the third dimension. Hence, in the case of spatial blocking with shared memory a smaller number of CUDA threads is launched and the compute intensity per CUDA thread is higher than in the other variants. The variants 1 and 8 utilize shared memory and do best next to the variants 2, 4, 9, and 11. However, shared memory does not gain very much performance in comparison to the variants without extensions. Variants 5 and 12 were generated with the old CUDA code generation discussed in Section 4.4. If we compare their performance with the performance of the variants generated with the new CUDA code generation, we observe that the new CUDA code generation provides far more performant target code than the old CUDA code generation. The variants exploiting spatial blocking with read-only cache are actually more than two times faster than the variants 5 and 12.

In summary, it can be stated that the smoother's performance benefits most from spatial blocking with the read-only cache and polyhedral optimizations do not have any performance impact at all. Furthermore, the new CUDA code generation is able to generate more performant CUDA code than the old CUDA code generation.

6.2.4 *Experiment 4 - performance impact of tiling on smoothing*

This experiment examines whether classical tiling increases the performance of the smoother presented in Listing 21 in Appendix C. As a starting point for tiling, we use the smoother variant 11 described in Table 4, which exploits spatial blocking with read-only cache, because the last experiment showed that this variant performs best. We consider all combinations of the following settings:

- tile size of dimension x: $t_x \in \{4, 8, 16, 32, 64, 128\}$
- tile size of dimension y: $t_y \in \{4, 8, 16, 32, 64, 128\}$
- tile size of dimension z: $t_z \in \{4, 8, 16, 32, 64, 128\}$
- allow tiling outer loop dimension: $\mathtt{outer} \in \{\mathtt{true}, \mathtt{false}\}$

In total these are $6^3 \cdot 2 = 432$ variants. Figure 28 illustrates the performance of the different variants in MLUPs. A larger value indicates again a better performing program. The red horizontal line shows the performance of the sequential program executed on the CPU. The blue horizontal line depicts the performance of the starting point.

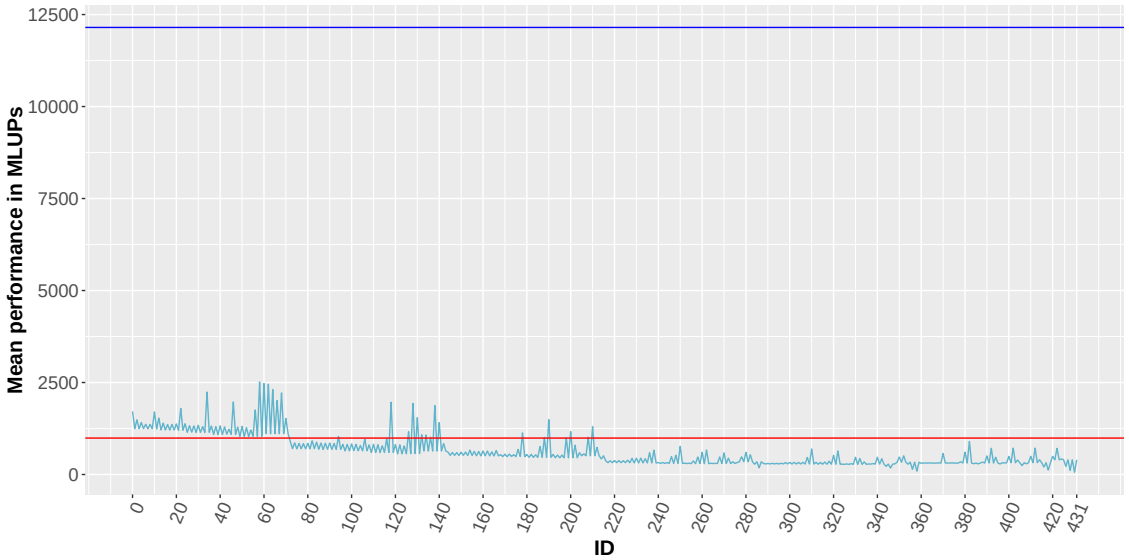


Figure 28: Performance of smoother variants with different tiling configurations in MLUPs

No matter what tiling configuration is applied, its performance is far behind the performance of the starting point. Actually, the performance obtained with tiling approximates the sequential performance. The main reason for the performance collapse is

the relationship between tiling and the mapping of loop dimensions to CUDA's thread identifiers. As mentioned in Section 4.4, at most three parallel loop dimensions can be mapped to CUDA's thread identifiers. The problem that arises from this mapping and tiling is best explained with a simple example. Listing 14 shows a loop nest L.

```
1 for z = 0 ← 1 → 10
2     for y = 0 ← 1 → 10
3         for x = 0 ← 1 → 10
4             ...
```

Listing 14: Source loop nest L

If we apply classical tiling on the innermost dimension of L with tile size 5, we get the loop nest in Listing 15 as a result.

```
1 for x_1 = 0 ← 5 → 10
2     for z = 0 ← 1 → 10
3         for y = 0 ← 1 → 10
4             for x_2 = x_1 ← 1 → min(10, x_1 + 5)
5                 ...
```

Listing 15: Target loop nest L after tiling innermost dimension with tile size 5

Regarding the tiled loop in Listing 15, the loop dimensions have increased because of tiling. We have four loop dimensions, but only three of them can be mapped to CUDA's thread identifiers. As a consequence either only the three innermost dimensions are mapped to CUDA's thread identifiers and the remaining dimension is executed on the CPU, or the outermost dimensions are mapped to CUDA's thread identifiers and the remaining dimension is executed on the GPU. In both cases, a loop is executed sequentially, which reduces the performance of the code. Furthermore, Listing 15 shows that the iterator of the point loop depends on the iterator of the tile loop. Due to this dependency the ExaStencils generator does not allow a parallelization of this loop with CUDA.

6.2.5 *Experiment 5 - performance evaluation of advanced smoother examples*

The last experiment investigated the performance of a Jacobi smoother dependent on the applied optimizations. We go now one step further and discuss the runtime of a smoother not isolated but in a complete multigrid solver. We considered three different smoothers: Block-Smoother (BS), Jacobi (Jac), and Red-Black Gauss-Seidel (RBGS). For each smoother we generated again the 14 different variants listed in Table 4. This experiment considers the mean time per V-cycle as reference value, since it deals with a complete multigrid method. Additionally, we can deduce the impact of a single optimization on the V-cycle. The following figures show the achieved speedup over the sequential execution on the CPU. Hence, a larger value indicates a better performing program.

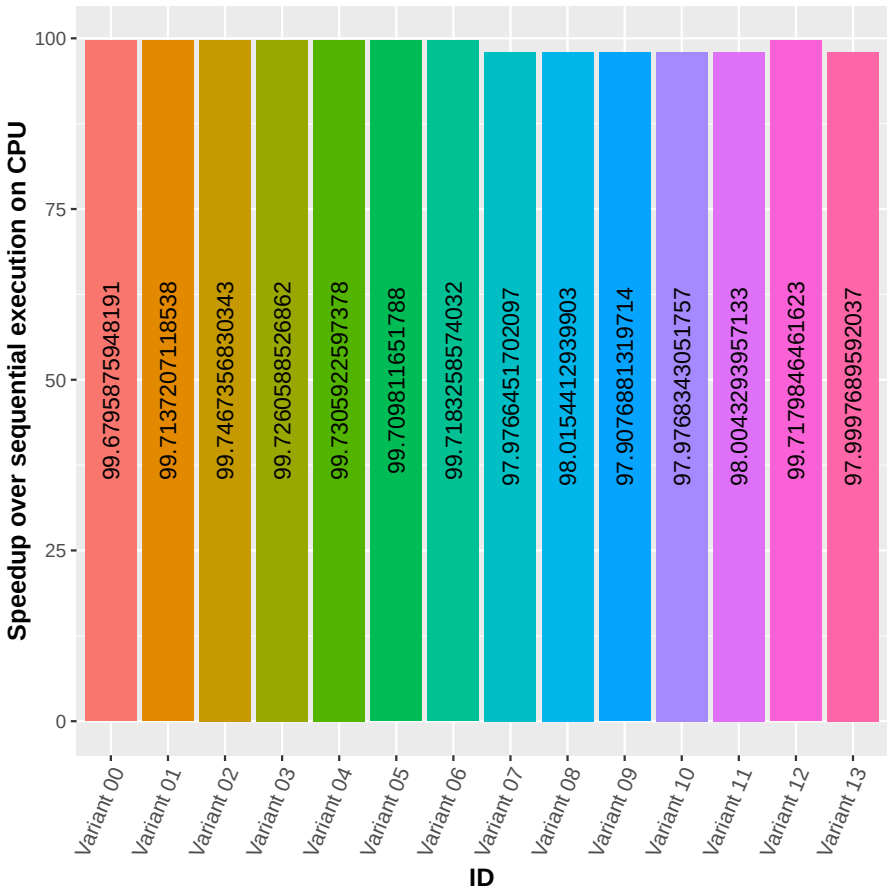


Figure 29: Performance comparison of variants of BS smoother in mean time per V-cycle

BLOCK-SMOOTHER Figure 29 visualizes the results for the Block-Smoother. The results are very close to each other. This has one main reason. Almost all fields are updated in-place. Thus, these fields cannot be cached by the read-only cache and shared memory cannot be used, since a write back from shared memory to global memory is not supported yet. For example, the stencil updates in the smoother are in-place and smoothing cannot benefit from shared memory or the read-only cache. However, the variants exploiting spatial blocking with the read-only cache perform best. Interestingly enough, polyhedral optimizations have a negative impact on the performance (variants 7-11,13) because some loop nests are not parallel anymore after polyhedral optimizations. Hence, the performance is worse because fewer computations are executed in parallel on the GPU.

JACOBI SMOOTHER Figure 30 shows the results for the Jacobi smoother. Experiment 3 in Section 6.2.3 already discussed the impact of optimizations on the Jacobi smoother. We can observe similar results in this experiment. The variants exploiting spatial blocking with read-only cache perform best again. The variants 5 and 12 were created with the old GPU code generation workflow. The code destined for GPU is the same in both cases, but there are differences in CPU code due to the polyhedral optimizations. These differences

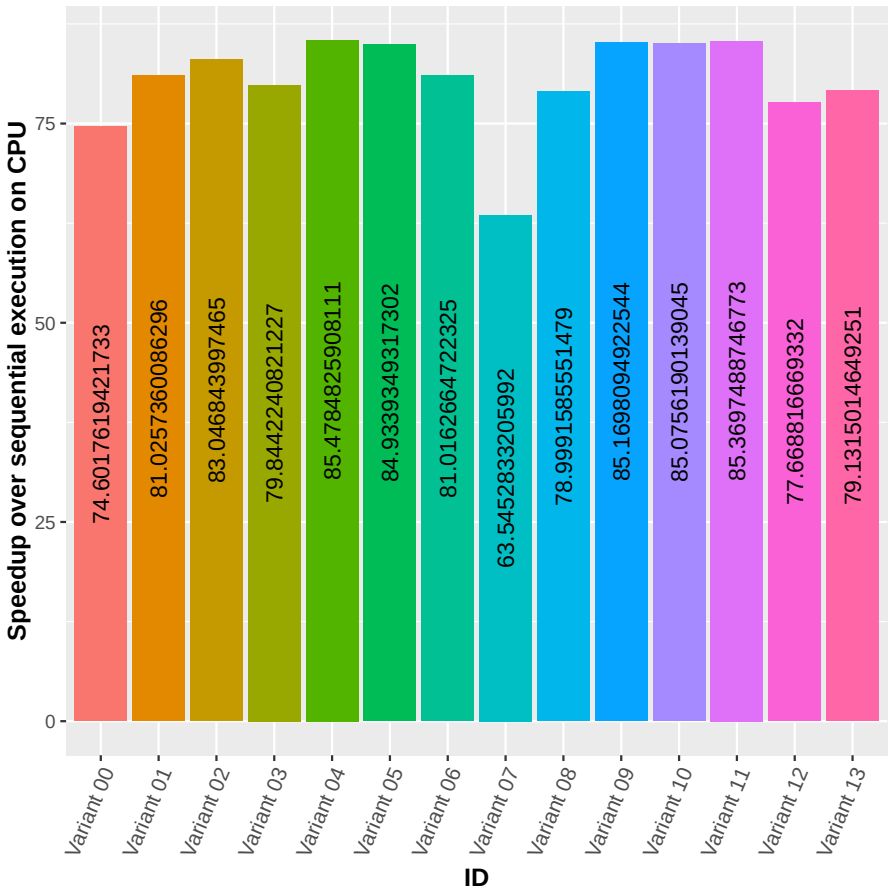


Figure 30: Performance comparison of variants of Jac smoother in mean time per V-cycle

cause a worse efficient program in case of the polyhedral optimizations. Spatial blocking with shared memory is applied in the variants 0 and 7, whereby variant 7 is worse than variant 0 because of the polyhedral optimizations.

RED-BLACK GAUSS-SEIDEL SMOOTHER Figure 31 illustrates the results for the RBGS smoother. The variants, which have been optimized in the polyhedron model, show a lower performance than the others. After polyhedral optimizations some loop nests have inner loop counters that depend on outer loop iterators. As exposed by the previous experiments, such loop nests can only be partially executed in parallel on the GPU. Hence, these variants offer less GPU parallelism than the variants without polyhedral optimizations. Moreover, spatial blocking with read-only cache does not promote the performance to the extent observed in other test cases. In particular, spatial blocking with read-only cache cannot be applied to a lot of fields because the fields are updated in-place as with the Block-Smoother. The same applies for the utilization of shared memory.

In summary, this experiment shows that the polyhedral optimizations sometimes have a negative impact on the performance because loop nests are not parallel anymore or there are additional dependences that render a mapping to CUDA impossible.

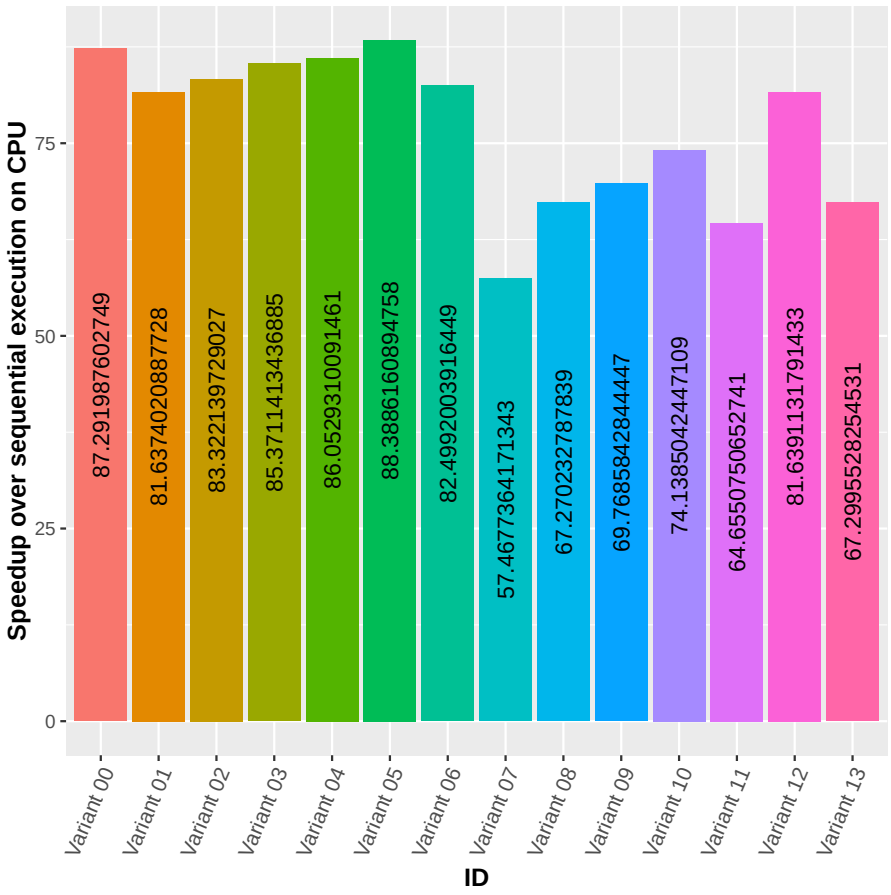


Figure 31: Performance comparison of variants of RBGS smoother in mean time per V-cycle

6.2.6 *Experiment 6 - multigrid solver evaluation*

The idea behind this experiment is to demonstrate that the new CUDA code generation workflow and the introduced extensions are also able to handle a complete multigrid solver as presented in Algorithm 1.

We consider again the steady-state heat equation with Dirichlet boundary conditions on the unit square, which is already known from experiment 1. One more time we refer to the constant changing thermal conductivity as the *constant test case* and the smoothly changing thermal conductivity as the *variable test case*. Furthermore, we examined a two-dimensional and a three-dimensional problem size for both, the constant and the variable test case. All test cases use the Laplace operator for smoothing. For each test case we examined the 14 variants listed in Table 4. We measure again the mean time per V-cycle. In the following, the plots detail the performance speedup over the sequential execution of the test cases on the CPU. A larger value indicates a better performance.

THREE-DIMENSIONAL CONSTANT TEST CASE Figure 32 shows the results for the three-dimensional constant test case. Unsurprisingly, spatial blocking with read-only cache boosts the performance and outperforms the variants generated with the old

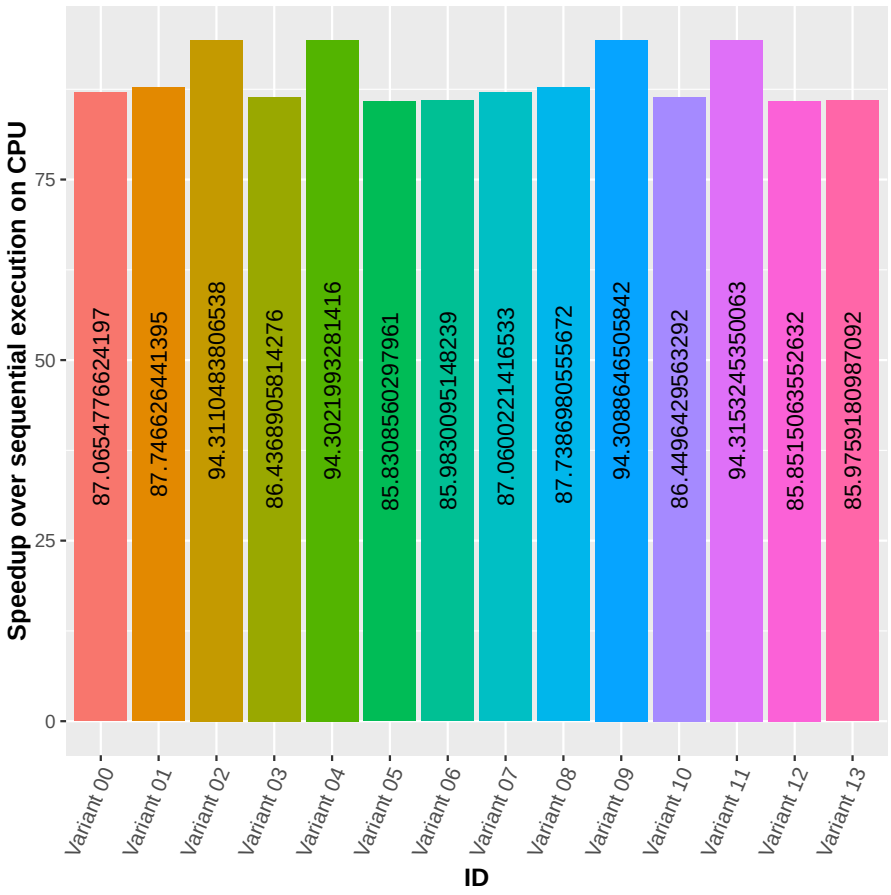


Figure 32: Runtime comparison of variants of the three-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and constant changing thermal conductivity

CUDA code generation. The runtimes of the remaining variants are very close to each other. The utilization of shared memory improves the performance only slightly as against the variants without optimizations.

THREE-DIMENSIONAL VARIABLE TEST CASE The performance results of the three-dimensional variable test case are very similar to the ones of the constant test case. Figure 33 reveals that variants exploiting spatial blocking with read-only cache do best and the remaining variants are hardly differentiable. However, this test case exposes the negative impact of spatial blocking with shared memory on the runtime of a V-cycle. As mentioned in the third experiment, spatial blocking with shared memory launches a fewer number of CUDA threads and the compute intensity per CUDA thread is higher than in the other variants. Nevertheless, the variants generated with the new CUDA code generation, which use spatial blocking with read-only cache, do again better than the variants generated with the old CUDA code generation.

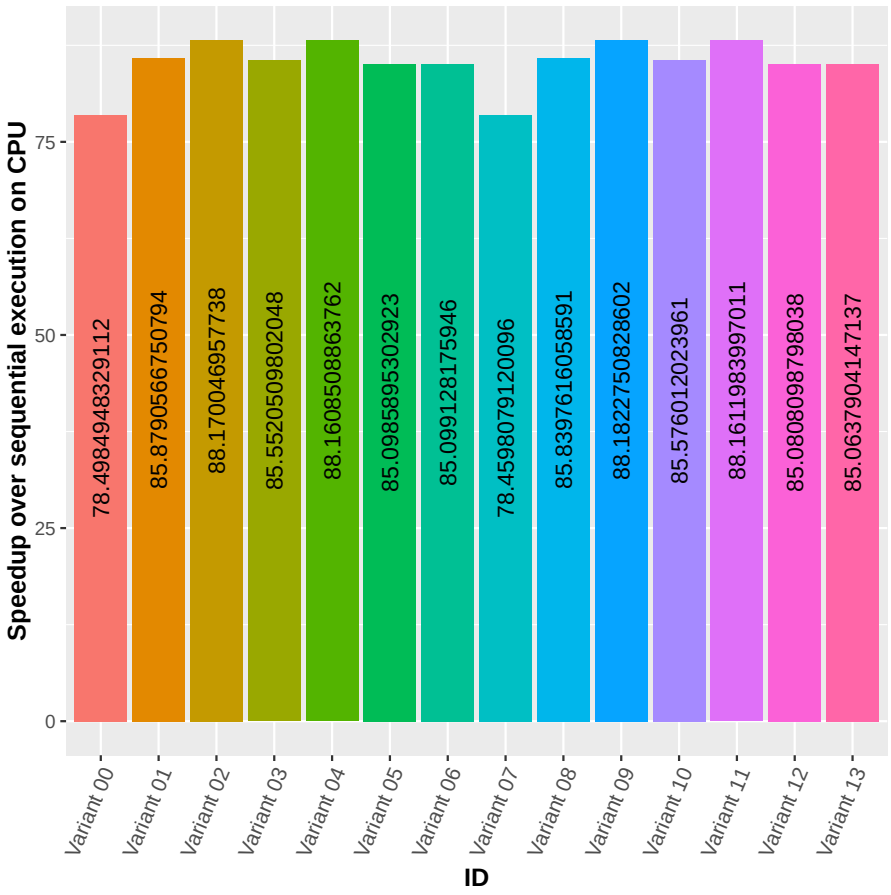


Figure 33: Runtime comparison of variants of the three-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity

Appendix D provides the runtime results of the two-dimensional test cases that were part of this experiment. In summary, the new CUDA code generation is able to generate better performing CUDA code than the old CUDA code generation for all considered test cases.

6.2.7 *Experiment 7 - fluid flow simulation*

The last experiment focuses on a complete application program. The idea behind this experiment is to demonstrate the applicability of the new CUDA code generation workflow to a relevant program in the domain of project ExaStencils. Hence, we examine the simulation of non-isothermal and non-Newtonian fluid flows. The targeted fluids have a high relevance in academia and industry alike. They are usually given by suspensions of particles or macromolecules and can be encountered as gels, pastes or foams. Relevant examples include organic fluids such as blood, food products such as fruit juice, and industrial fluids such as drilling fluids and mining pulps.

The fluid flow simulation is implemented based on the SIMPLE algorithm (Semi-Implicit Method for Pressure Linked Equations). A detailed derivation is beyond the scope of this thesis and can be found elsewhere [28, 25]. To put it in a nutshell, the SIMPLE algorithm works as follows: Instead of solving the entire non-linear system at once, linear systems of equations (LSEs) are set up for each of the velocity components. This step corresponds to freezing all other unknowns. Next, the single LSEs are solved. In our case, we use dedicated geometric multigrid solvers for this step. Since freezing components introduces some errors, a subsequent pressure correction has to be calculated and applied. In the classical SIMPLE algorithm, these steps are repeated until convergence is reached.

This fluid flow simulation is implemented in ExaSlang. We compiled four different variants of the ExaSlang source program with the ExaStencils generator:

(1) *CPU optimized*: a CPU variant using OpenMP for parallelization. We applied polyhedral optimizations and common subexpression elimination to this variant in order to boost its performance. The CPU variant is executed with 10 OpenMP threads.

(2) *GPU*: a GPU variant compiled with the new CUDA code generation workflow exploiting spatial blocking with read-only cache.

(3) *GPU + OMP*: same as variant two but additionally using OpenMP.

(4) *GPU old*: a GPU variant compiled with the old CUDA code generation workflow.

We do not consider other optimizations in this experiment because the previous experiments evinced that spatial blocking with read-only cache is most efficient. All four variants were executed five times, all calculations were performed as double precision, and four different execution times were extracted:

1. *update quantities*: the time to update physical properties such as viscosity
2. *compile LSEs*: the time to set up the LSEs for all variables
3. *solve LSEs*: the time to solve the LSEs
4. *total*: the total time, which consists of the previous mentioned and other factors such as convergence checks

Figure 34 shows the median of the execution times. The new GPU compilation workflow is able to handle the complete fluid flow simulation and to map it to the CUDA programming model. The new CUDA code generation performs significantly better than the old CUDA workflow. Furthermore, the new CUDA workflow is about two times faster in setting up the LSEs for all variables than the optimized CPU version. The combination of CUDA and OpenMP offers no additional performance improvement because all program parts relevant for OpenMP parallelization are already parallelized with CUDA and there is no heuristic, which decides if OpenMP or CUDA should be used for a program part. Overall the CPU optimized variant outperforms the GPU variants generated by the GPU compilation workflow in the ExaStencils generator.

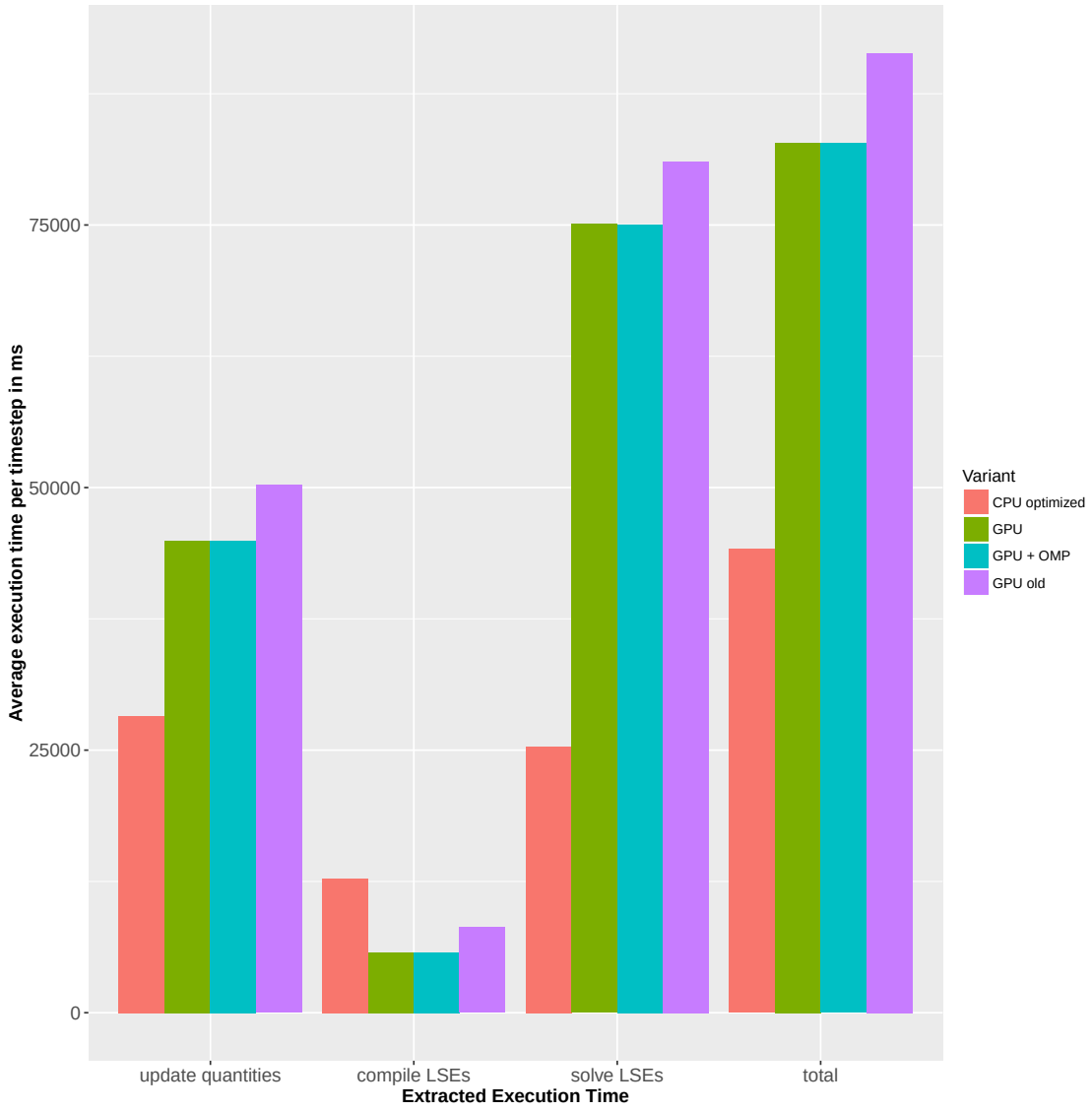


Figure 34: Runtime comparison of variants of Fluid Flow

6.3 SMOOTHER ANALYSIS - PERFORMANCE LIMITERS AND OPTIMIZATION OPPORTUNITIES

The previous experiments assessed the CUDA code generation in the ExaStencils generator. We discussed the performance of different programs and program variants depending on the enabled optimizations. In comparison to sequential programs or programs generated with the old CUDA generation workflow, the new one produces better performing code. However, one remaining question is how much potential for optimization is left or whether we already reached the performance limit. For this purpose, we focus again on the ExaSlang program provided in Appendix C. We already know that the program variant 2 described in Table 4 performs best among the examined variants. This section analyzes variant 2 in greater detail to reveal possible optimization opportunities and performance delimiters. We concentrate on the smoothing function of our example program shown in Listing 16.

```
1 Function SmootherT : Unit {
2     loop over fragments {
3         loop over SolutionT@finest {
4             SolutionT[nextSlot]@finest = SolutionT[active]@finest + (0.8 / diag(
      Laplace@finest) * (RHST@finest - Laplace@finest * SolutionT[active]@finest))
5         }
6         advance SolutionT@finest
7     }
8 }
```

Listing 16: Smoothing function extracted from the ExaSlang program listed in Appendix C

If the ExaStencils generator is encouraged to produce sequential CPU code, the function `SmootherT` results in the C++ loop nest exposed in Listing 17. We use this loop nest for further analysis purposes.

```
1 for (int z=iterationOffsetBegin[2]; z<(iterationOffsetEnd[2]+514); z+=1) {
2   for (int y=iterationOffsetBegin[1]; y<(iterationOffsetEnd[1]+514); y+=1) {
3     for (int x=iterationOffsetBegin[0]; x<(iterationOffsetEnd[0]+514); x+=1) {
4       field[((slot + 1) % 2)][((276672 * z) + (528 * y) + x + 1386008)] =
      ((0.19999999999999996f * field[(slot % 2)][((276672 * z) + (528 * y) + x +
      1386008)]) + (0.16666666666666669f * fieldData_RHST[((275616 * z) + (528 * y
      ) + x + 1104584)]) + (0.13333333333333336f * (field[(slot % 2)][((276672 * z
      ) + (528 * y) + x + 1109336)] + field[(slot % 2)][((276672 * z) + (528 * y)
      + x + 1385480)] + field[(slot % 2)][((276672 * z) + (528 * y) + x + 1386007)
      ] + field[(slot % 2)][((276672 * z) + (528 * y) + x + 1386009)] + field[(
      slot % 2)][((276672 * z) + (528 * y) + x + 1386536)] + field[(slot % 2)
      ][((276672 * z) + (528 * y) + x + 1662680)])));
5     }
6   }
7 }
```

Listing 17: Loop nest extracted from `SmootherT` in Listing 16

In the context of variant 2 in Table 4, the ExaStencils generator compiles the `SmootherT` function to a C++ function calling a CUDA kernel, which is presented in Listing 18.

```
__global__ void SmootherT(int _cu_begin_0, int _cu_end_0, int _cu_begin_1, int
    _cu_end_1, int _cu_begin_2, int _cu_end_2, float *solution_0_o1, const float
     *__restrict__ rhstData_0, const float *__restrict__ solution_0_o0, int
    slot_0, int fragmentIdx) {
  int _cu_x = ((blockIdx.x * blockDim.x) + threadIdx.x);
  int _cu_y = ((blockIdx.y * blockDim.y) + threadIdx.y);
  int _cu_z = ((blockIdx.z * blockDim.z) + threadIdx.z);
  bool _cu_condition = ((((_cu_x >= _cu_begin_0) && (_cu_x < _cu_end_0)) && ((
    _cu_y >= _cu_begin_1) && (_cu_y < _cu_end_1))) && ((_cu_z >= _cu_begin_2) &&
     (_cu_z < _cu_end_2)));
  if (_cu_condition) {
    solution_0_o1[((276672 * _cu_z) + (528 * _cu_y) + _cu_x + 1386008)] =
    ((0.19999999999999996f * solution_0_o0[((276672 * _cu_z) + (528 * _cu_y) +
    _cu_x + 1386008)]) + (0.16666666666666669f * rhstData_0[((275616 * _cu_z) +
    (528 * _cu_y) + _cu_x + 1104584)]) + (0.13333333333333336f * (solution_0_o0
    [((276672 * _cu_z) + (528 * _cu_y) + _cu_x + 1109336)] + solution_0_o0
    [((276672 * _cu_z) + (528 * _cu_y) + _cu_x + 1385480)] + solution_0_o0
    [((276672 * _cu_z) + (528 * _cu_y) + _cu_x + 1386007)] + solution_0_o0
    [((276672 * _cu_z) + (528 * _cu_y) + _cu_x + 1386009)] + solution_0_o0
    [((276672 * _cu_z) + (528 * _cu_y) + _cu_x + 1386536)] + solution_0_o0
    [((276672 * _cu_z) + (528 * _cu_y) + _cu_x + 1662680)])));
  }
}
```

Listing 18: CUDA kernel function resulting from `SmootherT` in Listing 16

6.3.1 *Performance estimates and actual performance*

The ExaStencils generator affords the opportunity to calculate performance estimates for a single call of a function F. It estimates the performance of F cumulatively with the performance of all calls to subfunctions appearing in F. In simplified terms, the ExaStencils generator traverses the AST of F and estimates the performance of its nodes on the basis of the provided platform configuration. With this in mind, the estimated performance corresponds to the technically possible performance. Table 6 details the calculated performance estimates for the loop nest in Listing 17.

Estimated time for loop on GPU	**8.3499 ms**
Max iterations	135796744
Optimistic memory transfer per iteration	12 byte
Optimistic device time for memory ops	4.8499 ms
Optimistic device time for computational ops	$7.4255e^{-4}$ ms
Assumed kernel call overhead	3.5 ms

Table 6: Performance estimates for loop nest shown in Listing 17

The ExaStencils generator calculated the performance estimates based on the technical details of the NVIDIA GeForce GTX TITAN Black presented in Section 2.3. For comparison, we executed variant 2 in Table 4 of the program in Appendix C again 10

times. Table 7 lists the execution time of the best run. The average runtime of smoother includes memory synchronization statements between host and device and a call to the kernel in Listing 18. Comparing this average smoother runtime with the estimated time detailed in Table 6, our generated smoother is only 2,7049 ms behind the technically possible performance.

Smoother runtime for 9 runs	99.4933 ms
Smoother MLUPs for 9 runs	12141.1 MLUPs
Average runtime of smoother	**11.0548 ms**

Table 7: Measured performance of variant 2 in Table 4 of the program presented in Appendix C

6.3.2 *Kernel analysis with NVIDIA profiling tools*

In the next step we perform a detailed analysis of the kernel presented in Listing 18 with the help of NVIDIAs profiling tools. The main work is done by the NVIDIA Visual Profiler (`nvvp`[1]), which is a cross-platform performance profiling tool that delivers feedback for optimizing CUDA C/C++ applications. Furthermore, we used `nvprof`[2] to collect the metrics needed by the guided analysis system of `nvvp` for an individual kernel. `nvprof` was called with the following arguments:

```
nvprof -kernels SmootherT -analysis-metrics -o sm.nvprof exa
```

`exa` is the compilation of our example program in Appendix C. The resulting output file `sm.nvprof` was then imported by `nvvp` in order to perform the guided analysis for the `SmootherT` kernel shown in Listing 18. The following paragraphs explain the analysis results.

APPLICATION'S OVERALL GPU USAGE The first analysis with `nvvp` addresses the overall GPU usage of our example program. Table 8 details the analysis results, a short explanation provided by `nvvp`, and an evaluation of the result's meaning. Overall, the results of the GPU usage analysis are expected and do not indicate further potential for optimization.

The following paragraphs discuss the results of the analysis, which concentrate on the smoother kernel. Hence, results, explanations, and evaluations refer to the smoother kernel presented in Listing 18.

1 https://developer.nvidia.com/nvidia-visual-profiler
2 http://docs.nvidia.com/cuda/profiler-users-guide/

Result	Explanation	Evaluation
Low Compute / Memcpy Efficiency	The amount of time performing compute is low relative to the amount of time required for memcpy.	This is expected. Our example program uses a Jacobi stencil as smoother. The update of a field is completely performed on the GPU. Hence, all data required for the field updates has to be copied from host to device. In comparison to that, the computation of field updates is quite fast because the TITAN Black offers enough CUDA cores.
Low Compute / Memcpy Overlap	The percentage of time when memcpy is being performed in parallel with compute is low.	This is expected. There is just a single kernel and memcpy is only performed before the smoother kernel is called.
Low Kernel Concurrency	The percentage of time when two kernels are being executed in parallel is low.	This is expected. Our example program contains only one kernel.
Low Compute Utilization	The multiprocessors of one or more GPUs are mostly idle.	This is expected. Regarding the timeline presented by `nvvp`, the most execution time is spent on CPU for CUDA context creation, memory transfer, and CUDA context destruction. The smoother kernel itself needs comparatively few time and the GPU is not used elsewhere.

Table 8: Results of `nvvp`'s overall GPU usage analysis

PERFORMANCE LIMITERS The first step in analyzing the smoother kernel is to determine if its performance is bounded by computation, memory bandwidth, or instruction/memory latency.

Figure 35 visualizes that the smoother kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of the NVIDIA GeForce GTX TITAN Black. According to `nvvp` these utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations because achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

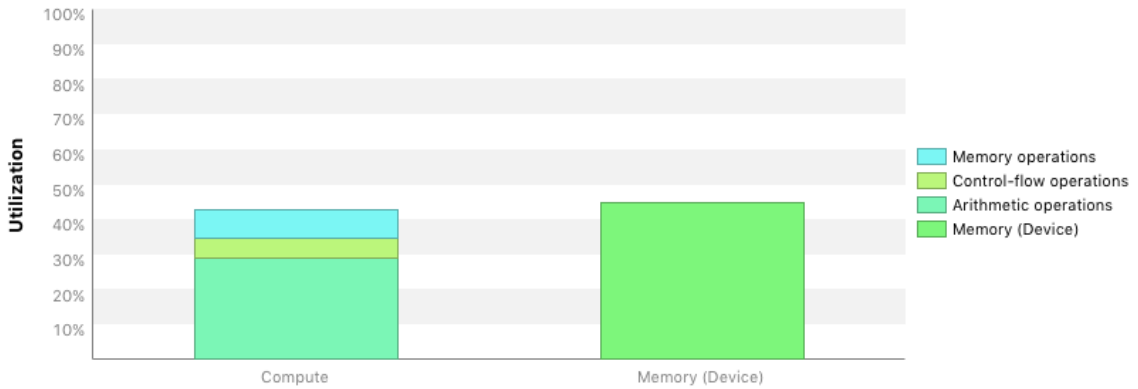


Figure 35: Compute throughput and memory bandwidth utilization of the kernel in Listing 18 relative to the peak performance of the NVIDIA GeForce GTX TITAN Black

INSTRUCTION AND MEMORY LATENCY ANALYSIS The previous analysis revealed that the smoother kernel is most likely latency bound. The next step is to analyze how the instruction and memory latency limits the kernel's performance.

The NVIDIA Visual Profiler examined different instruction stall reasons, which indicate the condition that prevents warps from executing on any given cycle. Figure 36 shows `nvvp`'s break-down of stall reasons averaged over the entire execution of the smoother kernel. As a result, `nvvp` states that the kernel has good theoretical and achieved occupancy indicating that there are likely sufficient warps executing on each SM. Furthermore, `nvvp` concludes that the smoother's performance is probably limited by the instruction stall reasons listed below because occupancy is not an issue.

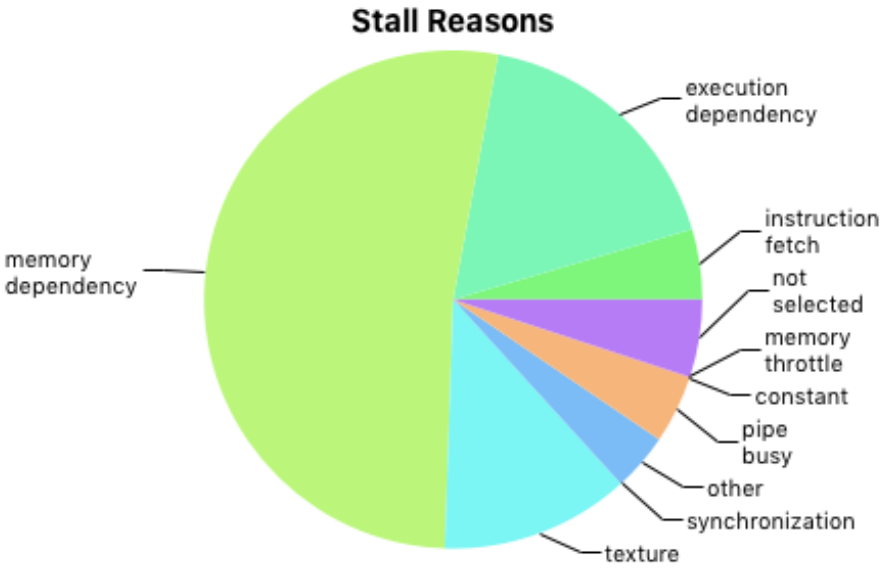


Figure 36: `nvvp`'s break-down of instruction stall reasons averaged over the entire execution of the smoother kernel

The following instruction stall reasons are sorted by the frequency of their occurrence from high to low. The remarks on the instruction stall reasons are extracted from `nvvp`'s analysis result.

(1) *Memory Dependency*: A loadstore cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.

(2) *Execution Dependency*: An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

(3) *Texture*: The texture sub-system is fully utilized or has too many outstanding requests.

(4) *Instruction Fetch*: The next assembly instruction has not yet been fetched.

(5) *Not Selected*: Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.

(6) *Pipeline Busy*: The compute resource(s) required by the instruction is not yet available.

(7) *Constant*: A constant load is blocked due to a miss in the constants cache.

(8) *Memory Throttle*: Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.

(9) *Synchronization*: The warp is blocked at a syncthreads() call.

The two most important instruction stall reasons are *memory dependency* and *execution dependency*. We first discuss the *execution dependency*. Regarding the smoother kernel in Listing 18, a field update is already performed completely in parallel. Hence, increasing instruction-level parallelism may not be useful or even possible. Since this point offers no obvious potential for optimization, we face now the *memory dependency*. The optimization of memory alignment and access patterns is difficult because every performed load operation is required for the stencil update. However, aligning data in shared memory and consequently optimizing the access patterns results in a poor smoother performance as already discussed in the previous experiments.

OCCUPANCY ANALYSIS The last analysis of the smoother examines its occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. The NVIDIA Visual Profiler argues that occupancy is not limiting the kernel's performance because the kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. Hence, the `nvvp` comes to the conclusion that a further investigation of the occupancy may not be useful.

In summary, it can be stated that the smoother kernel in Listing 18 is limited by instruction and memory latency. As already revealed in the previous subsection, the smoother's performance is close to the technically possible performance calculated by the ExaStencils generator. However, NVIDIA's profiling tools state that the CUDA code generation in the ExaStencils generator can possibly be improved by optimizing instruction-level parallelism, memory alignment, and access patterns. Furthermore, the fluid flow experiment in Section 6.2.7 illustrates that the current CUDA workflow cannot keep up with the optimized CPU code generated by the ExaStencils generator.

6.4 EVALUATION OF PPCG'S HYBRID TILING FEATURE FOR GPUS

Tobias Grosser et al. proposed an hybrid tiling algorithm to improve the performance of code generated for GPUs [6]. This hybrid tiling algorithm is a stencil-specific tiling scheme incorporated in the Polyhedral Parallel Code Generator (PPCG). In short, the algorithm addresses stencil codes and applies hexagonal tiling to the outer dimension along with classical tiling on all remaining dimensions. A detailed derivation of the complete hybrid tiling algorithm is beyond the scope of this thesis and can be found elsewhere [6]. They tested two-dimensional and three-dimensional Jacobi stencils in their experiments and showed that hybrid tiling provides a noteworthy performance boost. Listing 19 shows such a Jacobi stencil. There is an outer time loop performing T stencil iterations. The performance boost with hybrid tiling was examined with $T = 128$ and $T = 512$.

```
1 for (t = 0; t < T; t++) {
2   for (i = 1; i < N; i++)
3     for (j = 1; j < N; j++)
4       for (k = 1; k < N; k++)
5         A[(t+1) % 2][i][j][k] = (2.0f / 7.0f) * A[t%2][i][j][k] +
6             (1.0f / 7.0f) * (A[t%2][i+1][j][k] + A[t%2][i-1][j][k] + A[t%2][i][j
    +1][k] + A[t%2][i][j-1][k] + A[t%2][i][j][k+1] + A[t%2][i][j][k-1]);
7 }
```

Listing 19: Three-dimensional Jacobi stencil test case

The stencils appearing in the ExaStencils domain are slightly different from the presented test case. First, it is improbable that a stencil is performed more than four times iteratively and second there is no time loop because the handling of boundary points differ from time step to time step. The test case in Listing 19 reveals the same handling of boundary points for each time step. Hence, porting the hybrid tiling algorithm from PPCG to ExaStencils bears some challenges. Furthermore, it is arguable whether hybrid tiling performs well on only a few time steps. However, the detailed analysis of the smoother in the last section exposed a potential for optimization. Furthermore, the experiment in Section 6.2.4 showed that the tiling approach implemented in the ExaStencils generator does not improve the performance of generated CUDA code. In this section we examine the performance impact of hybrid tiling if only a few time steps are performed, in order to assess the benefits of hybrid tiling for the ExaStencils generator.

The following experiments were performed with the development version of PPCG (git version hash 4c0ecc8d6a316b1a802aad0b85c64073ceb52ca8). For benchmarks we used a Laplace kernel and a heat stencil. Both were tested with two space dimensions and three space dimensions. We compiled different versions of the test cases with PPCG, varying the problem size and the number of time steps. All programs were executed five times on the NVIDIA GeForce GTX TITAN Black. Figure 37 visualizes the performance results of the best run. Each plot names the stencil, the problem size, and the performed time steps. Interestingly enough, hybrid tiling improves the performance of every stencil independently from the executed time steps. One reason why PPCG's hybrid tiling is beneficial for a stencil in comparison to the tiling applied by the ExaStencils generator could be that PPCG does not only map loop dimensions to CUDA's thread identifiers, but also to CUDA's block identifiers. Hence, PPCG uses three more dimensions for the mapping from loop dimensions to the CUDA model.

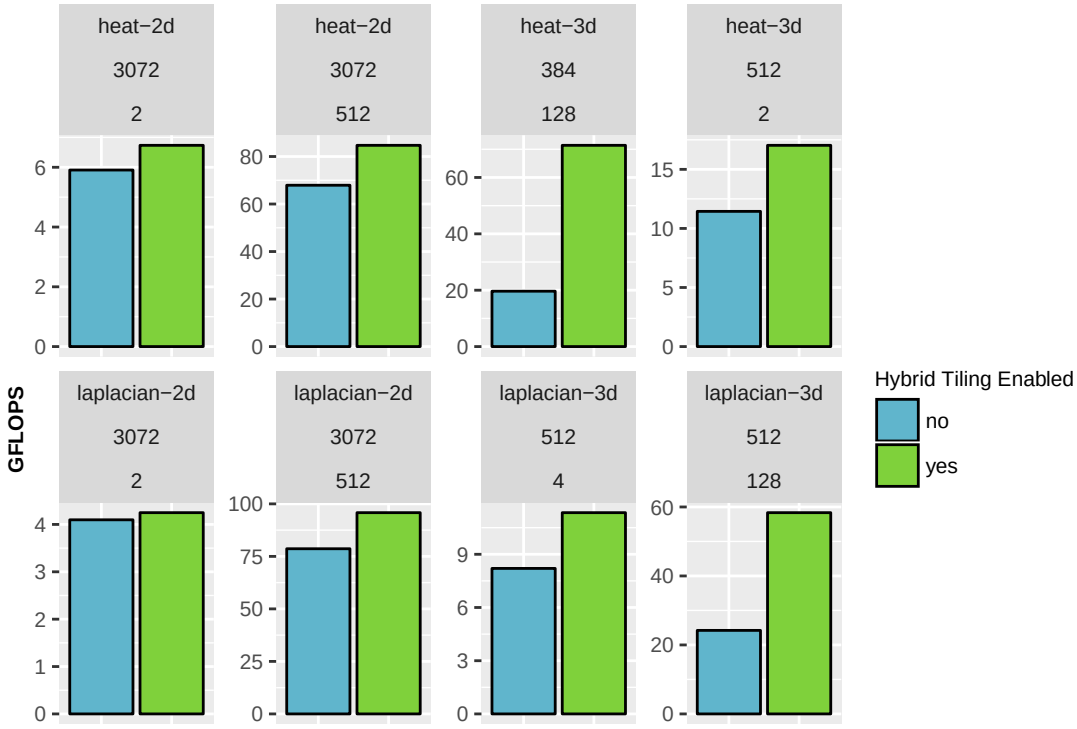


Figure 37: Performance comparison of different stencil test cases in GFLOPS

Appendix E presents the complete results of the hybrid tiling experiments. The experiments evince that PPCG's hybrid tiling approach may be useful to further improve the performance of the CUDA code generated by the ExaStencils generator.

7

CONCLUSION

This thesis presented a new CUDA code generation for the ExaStencils generator, which exploits the polyhedron model for its GPU code optimizations and features several extensions like spatial blocking with read-only cache and the utilization of shared memory. The new CUDA code generation is able to compile ExaSlang source code to high-performance GPU target code. Furthermore, we assessed this new workflow in extensive experiments.

The experiments revealed that the smoothing steps of a multigrid solver are the most performance-critical parts. Hence, we examined especially the smoother step. One important result of the experiments is the fact that polyhedral optimizations and polyhedral schedule exploration do not improve the performance of the generated CUDA code. In addition to it, the experiments exposed the negative impact of the current tiling approach on the performance of generated CUDA code. However, we were able to observe enormous performance boosts with the new CUDA workflow compared to sequential CPU variants. Furthermore, the experiments showed that the introduced extensions enable the new workflow to outperform the old CUDA code generation workflow. Additionally, the experiments successfully demonstrated that our new workflow cannot only deal with smoothers, but is also able to handle real world problems as discussed in Section 6.2.7.

The last part of the thesis concentrated on the performance limitations of the generated CUDA code. On the one hand, we compared a smoother's theoretical performance, predicted by the ExaStencils generator, with its actual performance. On the other hand, we examined a smoother's performance with NVIDIA's profiling tools. In summary, the results exhibited that the smoother's performance is close to the generator's predicted performance but also that it is bounded by instruction and memory latency. In reference to further performance improvements, we assessed PPCG's hybrid tiling feature with problem sizes of the ExaStencils' domain.

In conclusion, the thesis points out that the introduced polyhedral GPU code generation workflow compiles ExaSlang to well performing CUDA code. Furthermore, it shows that the polyhedron model has only marginal impact on the generated CUDA code. With regard to classical tiling and polyhedral schedule exploration, the thesis demonstrated that some techniques used for optimizing CPU performance fail at optimizing GPU performance.

OUTLINE OF FURTHER WORK The experiments just exposed that PPCG's hybrid tiling approach could further improve a smoother's performance. Hence, an implementation of hybrid tiling in the ExaStencils generator may be beneficial. As part of this implementation, the mapping of loop dimensions to the CUDA model has to be modified to deal with the increasing number of dimensions. A remaining task is also to extend the shared memory utilization to stencils with diagonal neighbors. Currently, shared memory is only used for read-only fields. This constraint can be removed by implementing a write back from shared memory to global memory. Furthermore, the ExaStencils generator does not use the read-only cache and shared memory in combination. A heuristic would be desirable, which decides if a field should reside in read-only cache or if it should be loaded in shared memory. Since we examined the performance with fixed CUDA thread block sizes, it is still to be clarified how varying thread block sizes influence the performance. Finally, the next level of polyhedral GPU code generation would be to combine CPU and GPU parallelism. On the one hand, a support for CUDA-aware MPI is imaginable and on the other hand, the utilization of multiple GPUs in the ExaStencils generator is possible. The new features of modern CUDA capable GPUs like *Hyper-Q* and *Dynamic Parallelism* are also interesting for further studies.

Part III

APPENDIX

A

SEQUENTIAL RUNTIME OF EXEMPLARY MULTIGRID SOLVERS

This appendix provides the complete runtime results of the experiment discussed in Section 6.2.1. The Tables 9, 10, 11, and 12 give an overview on the sequential runtime of four different test cases. All four test cases have in common that the pre-smoothing and post-smoothing steps are the most runtime intensive steps of the solver.

Table 9 details the results of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and constant changing thermal conductivity.

Solver step	**Time in ms**
Total time to setup	7.35611
Total time to solve in 100 steps	1813.78
Mean time per V-cycle	12.5436
Total time spent on level 5 in pre-smoothing	0
Total time spent on level 6 in pre-smoothing	1.1114
Total time spent on level 7 in pre-smoothing	3.80301
Total time spent on level 8 in pre-smoothing	16.4782
Total time spent on level 9 in pre-smoothing	61.3861
Total time spent on level 10 in pre-smoothing	257.317
Total time spent on level 5 in updating residual	0
Total time spent on level 6 in updating residual	0.403369
Total time spent on level 7 in updating residual	1.28279
Total time spent on level 8 in updating residual	5.21165
Total time spent on level 9 in updating residual	20.2505
Total time spent on level 10 in updating residual	85.2311
Total time spent on level 5 in restricting	0
Total time spent on level 6 in restricting	0.186424
Total time spent on level 7 in restricting	0.626223
Total time spent on level 8 in restricting	2.43679
Total time spent on level 9 in restricting	8.98563
Total time spent on level 10 in restricting	34.7119

Total time spent on level 5 in setting solution	0
Total time spent on level 6 in setting solution	0.045492
Total time spent on level 7 in setting solution	0.175215
Total time spent on level 8 in setting solution	0.514977
Total time spent on level 9 in setting solution	1.96414
Total time spent on level 10 in setting solution	6.92722
Total time spent on level 5 in prolongating and correcting	0
Total time spent on level 6 in prolongating and correcting	1.14755
Total time spent on level 7 in prolongating and correcting	4.53962
Total time spent on level 8 in prolongating and correcting	18.0047
Total time spent on level 9 in prolongating and correcting	71.891
Total time spent on level 10 in prolongating and correcting	287.028
Total time spent on level 5 in post-smoothing	0
Total time spent on level 6 in post-smoothing	1.09767
Total time spent on level 7 in post-smoothing	3.77848
Total time spent on level 8 in post-smoothing	16.3233
Total time spent on level 9 in post-smoothing	61.1968
Total time spent on level 10 in post-smoothing	254.29

Table 9: Sequential runtime of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and constant changing thermal conductivity

Table 10 details the results of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity.

Solver step	**Time in ms**
Total time to setup	15.151
Total time to solve in 100 steps	11099.9
Mean time per V-cycle	22.4803
Total time spent on level 5 in pre-smoothing	0
Total time spent on level 6 in pre-smoothing	2.04676
Total time spent on level 7 in pre-smoothing	7.60026
Total time spent on level 8 in pre-smoothing	31.1715
Total time spent on level 9 in pre-smoothing	122.575
Total time spent on level 10 in pre-smoothing	611.548
Total time spent on level 5 in updating residual	0

Total time spent on level 6 in updating residual	0.634189
Total time spent on level 7 in updating residual	2.26438
Total time spent on level 8 in updating residual	9.13774
Total time spent on level 9 in updating residual	34.0553
Total time spent on level 10 in updating residual	205.108
Total time spent on level 5 in restricting	0
Total time spent on level 6 in restricting	0.178953
Total time spent on level 7 in restricting	0.618911
Total time spent on level 8 in restricting	2.17718
Total time spent on level 9 in restricting	8.1937
Total time spent on level 10 in restricting	33.8058
Total time spent on level 5 in setting solution	0
Total time spent on level 6 in setting solution	0.060375
Total time spent on level 7 in setting solution	0.207412
Total time spent on level 8 in setting solution	0.65891
Total time spent on level 9 in setting solution	2.41712
Total time spent on level 10 in setting solution	10.0165
Total time spent on level 5 in prolongating and correcting	0
Total time spent on level 6 in prolongating and correcting	1.02688
Total time spent on level 7 in prolongating and correcting	4.00626
Total time spent on level 8 in prolongating and correcting	15.9245
Total time spent on level 9 in prolongating and correcting	63.5843
Total time spent on level 10 in prolongating and correcting	254.038
Total time spent on level 5 in post-smoothing	0
Total time spent on level 6 in post-smoothing	1.82079
Total time spent on level 7 in post-smoothing	6.97171
Total time spent on level 8 in post-smoothing	28.4941
Total time spent on level 9 in post-smoothing	110.24
Total time spent on level 10 in post-smoothing	645.728

Table 10: Sequential runtime of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity

Table 11 details the results of the three-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and constant changing thermal conductivity.

Solver step	Time in ms
Total time to setup	79.0469
Total time to solve in 3 steps	1133.98
Mean time per V-cycle	262.853
Total time spent on level 0 in pre-smoothing	0
Total time spent on level 1 in pre-smoothing	0.001457
Total time spent on level 2 in pre-smoothing	0.002382
Total time spent on level 3 in pre-smoothing	0.013281
Total time spent on level 4 in pre-smoothing	0.069628
Total time spent on level 5 in pre-smoothing	0.42937
Total time spent on level 6 in pre-smoothing	2.96025
Total time spent on level 7 in pre-smoothing	21.7716
Total time spent on level 8 in pre-smoothing	183.185
Total time spent on level 0 in updating residual	0
Total time spent on level 1 in updating residual	0.001122
Total time spent on level 2 in updating residual	0.001751
Total time spent on level 3 in updating residual	0.005489
Total time spent on level 4 in updating residual	0.022724
Total time spent on level 5 in updating residual	0.146541
Total time spent on level 6 in updating residual	0.96561
Total time spent on level 7 in updating residual	7.33021
Total time spent on level 8 in updating residual	58.9012
Total time spent on level 0 in restricting	0
Total time spent on level 1 in restricting	0.000316
Total time spent on level 2 in restricting	0.000973
Total time spent on level 3 in restricting	0.002099
Total time spent on level 4 in restricting	0.009769
Total time spent on level 5 in restricting	0.086894
Total time spent on level 6 in restricting	0.758946
Total time spent on level 7 in restricting	6.39969
Total time spent on level 8 in restricting	56.7561
Total time spent on level 0 in setting solution	0
Total time spent on level 1 in setting solution	0.000145
Total time spent on level 2 in setting solution	0.001404
Total time spent on level 3 in setting solution	0.001059
Total time spent on level 4 in setting solution	0.00312

Total time spent on level 5 in setting solution	0.008553
Total time spent on level 6 in setting solution	0.048855
Total time spent on level 7 in setting solution	0.340559
Total time spent on level 8 in setting solution	2.51204
Total time spent on level 0 in prolongating and correcting	0
Total time spent on level 1 in prolongating and correcting	0.000986
Total time spent on level 2 in prolongating and correcting	0.0013
Total time spent on level 3 in prolongating and correcting	0.005838
Total time spent on level 4 in prolongating and correcting	0.047941
Total time spent on level 5 in prolongating and correcting	0.414661
Total time spent on level 6 in prolongating and correcting	3.40126
Total time spent on level 7 in prolongating and correcting	26.6808
Total time spent on level 8 in prolongating and correcting	213.486
Total time spent on level 0 in post-smoothing	0
Total time spent on level 1 in post-smoothing	0.000266
Total time spent on level 2 in post-smoothing	0.001294
Total time spent on level 3 in post-smoothing	0.010383
Total time spent on level 4 in post-smoothing	0.062223
Total time spent on level 5 in post-smoothing	0.424309
Total time spent on level 6 in post-smoothing	2.94058
Total time spent on level 7 in post-smoothing	21.8311
Total time spent on level 8 in post-smoothing	176.492

Table 11: Sequential runtime of the three-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and constant changing thermal conductivity

Table 12 details the results of the three-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity.

Solver step	**Time in ms**
Total time to setup	191.545
Total time to solve in 4 steps	7609.83
Mean time per V-cycle	460.414
Total time spent on level 0 in pre-smoothing	0
Total time spent on level 1 in pre-smoothing	0.002376
Total time spent on level 2 in pre-smoothing	0.004782

Total time spent on level 3 in pre-smoothing	0.028628
Total time spent on level 4 in pre-smoothing	0.139943
Total time spent on level 5 in pre-smoothing	0.956643
Total time spent on level 6 in pre-smoothing	7.48184
Total time spent on level 7 in pre-smoothing	65.8215
Total time spent on level 8 in pre-smoothing	557.707
Total time spent on level 0 in updating residual	0
Total time spent on level 1 in updating residual	0.001866
Total time spent on level 2 in updating residual	0.002391
Total time spent on level 3 in updating residual	0.00875
Total time spent on level 4 in updating residual	0.045019
Total time spent on level 5 in updating residual	0.290869
Total time spent on level 6 in updating residual	2.07604
Total time spent on level 7 in updating residual	22.7912
Total time spent on level 8 in updating residual	186.886
Total time spent on level 0 in restricting	0
Total time spent on level 1 in restricting	0.000534
Total time spent on level 2 in restricting	0.001292
Total time spent on level 3 in restricting	0.002207
Total time spent on level 4 in restricting	0.011277
Total time spent on level 5 in restricting	0.10144
Total time spent on level 6 in restricting	0.885046
Total time spent on level 7 in restricting	7.57429
Total time spent on level 8 in restricting	66.5284
Total time spent on level 0 in setting solution	0
Total time spent on level 1 in setting solution	0.00025
Total time spent on level 2 in setting solution	0.001309
Total time spent on level 3 in setting solution	0.001392
Total time spent on level 4 in setting solution	0.003375
Total time spent on level 5 in setting solution	0.011117
Total time spent on level 6 in setting solution	0.053897
Total time spent on level 7 in setting solution	0.393639
Total time spent on level 8 in setting solution	2.9271
Total time spent on level 0 in prolongating and correcting	0
Total time spent on level 1 in prolongating and correcting	0.001337
Total time spent on level 2 in prolongating and correcting	0.001212
Total time spent on level 3 in prolongating and correcting	0.006538

Total time spent on level 4 in prolongating and correcting	0.055504
Total time spent on level 5 in prolongating and correcting	0.483209
Total time spent on level 6 in prolongating and correcting	3.97127
Total time spent on level 7 in prolongating and correcting	31.142
Total time spent on level 8 in prolongating and correcting	249.355
Total time spent on level 0 in post-smoothing	0
Total time spent on level 1 in post-smoothing	0.000377
Total time spent on level 2 in post-smoothing	0.002156
Total time spent on level 3 in post-smoothing	0.02076
Total time spent on level 4 in post-smoothing	0.131732
Total time spent on level 5 in post-smoothing	0.914085
Total time spent on level 6 in post-smoothing	7.24866
Total time spent on level 7 in post-smoothing	66.5939
Total time spent on level 8 in post-smoothing	558.966

Table 12: Sequential runtime of the three-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity

B

POLYHEDRAL SEARCH SPACE EXPLORATION TEST CASE

This chapter provides further information on the polyhedral search space exploration examined in Section 6.2.2. Listing 20 shows a program written in ExaSlang. This test program measures the performance of the defined smoother in MLUPs. Therefor the smoother is executed once for cache warmup and afterwards three times for the measurement. Furthermore, the smoother applies temporal blocking. The smoother is implemented with the help of a Jacobi stencil.

```
1  Domain global< [ 0, 0, 0 ] to [ 1, 1, 1 ] >
2
3  Layout FullTempBlockable< Real, Node >@finest {
4      innerPoints = [ 512, 512, 512 ]
5      ghostLayers = [ 5, 5, 5 ]
6      duplicateLayers = [ 1, 1, 1 ]
7  }
8  Layout PartTempBlockable< Real, Node >@finest {
9      innerPoints = [ 512, 512, 512 ]
10     ghostLayers = [ 4, 4, 4 ]
11     duplicateLayers = [ 1, 1, 1 ]
12 }
13
14 Field SolutionT< global, FullTempBlockable, 0.0 >[2]@finest
15 Field RHST< global, PartTempBlockable, None >@finest
16
17 Stencil Laplace@finest {
18     [ 0,  0,  0] => 4.8
19     [ 1,  0,  0] => -0.8
20     [-1,  0,  0] => -0.8
21     [ 0,  1,  0] => -0.8
22     [ 0, -1,  0] => -0.8
23     [ 0,  0,  1] => -0.8
24     [ 0,  0, -1] => -0.8
25 }
26
27 Globals {
28 }
29
30 Function LUPs() : Real {
31     Variable dimSize : Integer = 512
32     return(dimSize * dimSize * dimSize)
33 }
34
```

```
35 Function SmootherTTempBlock() : Unit {
36     loop over fragments {
37         repeat 3 times with contraction [1,1,1] { // marker;
38             loop over SolutionT@finest {
39                 SolutionT[nextSlot]@finest = SolutionT[active]@finest + (0.8 /
       diag(Laplace@finest) * (RHST@finest - Laplace@finest * SolutionT[active]@
       finest))
40             }
41             advance SolutionT@finest
42         }
43     }
44 }
45
46 Function InitFields ( ) : Unit {
47     loop over SolutionT@finest sequentially {
48         SolutionT[active]@finest = native('((double)std::rand()/RAND_MAX)')
49     }
50     loop over RHST@finest sequentially {
51         RHST@finest = 0
52     }
53 }
54
55 Function BenchmarkT() : Unit {
56     print('--------------------------------')
57     print('Smoother 3D')
58     print('Cache warmup')
59     repeat 1 times {
60         SmootherTTempBlock()
61     }
62
63     print('Starting benchmark')
64     startTimer(benchTTimer)
65     repeat 3 times { // marker
66         SmootherTTempBlock()
67     }
68     stopTimer(benchTTimer)
69
70     Variable time : Real = getTotalFromTimer(benchTTimer)
71     print('Runtime: ', time)
72     print('MLUPs: ', (LUPs() * 9) / time / 1e3)
73 }
74
75 Function Application() : Unit {
76     startTimer(setupWatch)
77     initGlobals()
78     initDomain()
79     InitFields()
80     stopTimer(setupWatch)
81     print('Total time to setup: ', getTotalFromTimer(setupWatch))
82     BenchmarkT()
83     destroyGlobals()
84 }
```

Listing 20: Example program in ExaSlang measuring the performance of a smoother in MLUPs. The smoother definition uses temporal blocking.

C

EXAMPLE SMOOTHER DEFINITION IN EXASLANG

This chapter contains an example program written in ExaSlang, which is used by several performed experiments. The ExaSlang program is shown in Listing 21. It measures the performance of the defined smoother in MLUPs. Therefor the smoother is executed once for cache warmup and afterwards nine times for the measurement. The smoother is implemented with the help of a Jacobi stencil.

```
1  Domain global< [ 0, 0, 0 ] to [ 1, 1, 1 ] >
2
3  Layout FullTempBlockable< Real, Node >@finest {
4      innerPoints = [ 512, 512, 512 ]
5      ghostLayers = [ 5, 5, 5 ]
6      duplicateLayers = [ 1, 1, 1 ]
7  }
8  Layout PartTempBlockable< Real, Node >@finest {
9      innerPoints = [ 512, 512, 512 ]
10     ghostLayers = [ 4, 4, 4 ]
11     duplicateLayers = [ 1, 1, 1 ]
12 }
13
14 Field SolutionT< global, FullTempBlockable, 0.0 >[2]@finest
15 Field RHST< global, PartTempBlockable, None >@finest
16
17 Stencil Laplace@finest {
18     [ 0,  0,  0] => 4.8
19     [ 1,  0,  0] => -0.8
20     [-1,  0,  0] => -0.8
21     [ 0,  1,  0] => -0.8
22     [ 0, -1,  0] => -0.8
23     [ 0,  0,  1] => -0.8
24     [ 0,  0, -1] => -0.8
25 }
26
27 Globals {
28 }
29
30 Function LUPs() : Real {
31     Variable dimSize : Integer = 512
32     return(dimSize * dimSize * dimSize)
33 }
34
35 Function SmootherT : Unit {
36     loop over fragments {
```

```
37          loop over SolutionT@finest {
38              SolutionT[nextSlot]@finest = SolutionT[active]@finest + (0.8 / diag(
    Laplace@finest) * (RHST@finest - Laplace@finest * SolutionT[active]@finest))
39          }
40          advance SolutionT@finest
41      }
42  }
43
44  Function InitFields ( ) : Unit {
45      loop over SolutionT@finest sequentially {
46          SolutionT[active]@finest = native('((double)std::rand()/RAND_MAX)')
47      }
48      loop over RHST@finest sequentially {
49          RHST@finest = 0
50      }
51  }
52
53  Function BenchmarkT() : Unit {
54      print('-------------------------------')
55      print('Smoother 3D')
56      print('Cache warmup')
57      repeat 1 times {
58          SmootherT()
59      }
60
61      print('Starting benchmark')
62      startTimer(benchTTimer)
63      repeat 9 times { // marker
64          SmootherT()
65      }
66      stopTimer(benchTTimer)
67
68      Variable time : Real = getTotalFromTimer(benchTTimer)
69      print('Runtime: ', time)
70      print('MLUPs: ', (LUPs() * 9) / time / 1e3)
71  }
72
73  Function Application() : Unit {
74      startTimer(setupWatch)
75      initGlobals()
76      initDomain()
77      InitFields()
78      stopTimer(setupWatch)
79      print('Total time to setup: ', getTotalFromTimer(setupWatch))
80      BenchmarkT()
81      destroyGlobals()
82  }
```

Listing 21: Example program in ExaSlang measuring the performance of a smoother in MLUPs.

D

RUNTIME OF EXEMPLARY MULTIGRID SOLVERS

This appendix provides further runtime results of the experiment discussed in Section 6.2.6. The discussed test cases were generated by the ExaStencils generator with different configurations as described in Table 4.

Figure 38 details the results of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and constant changing thermal conductivity.

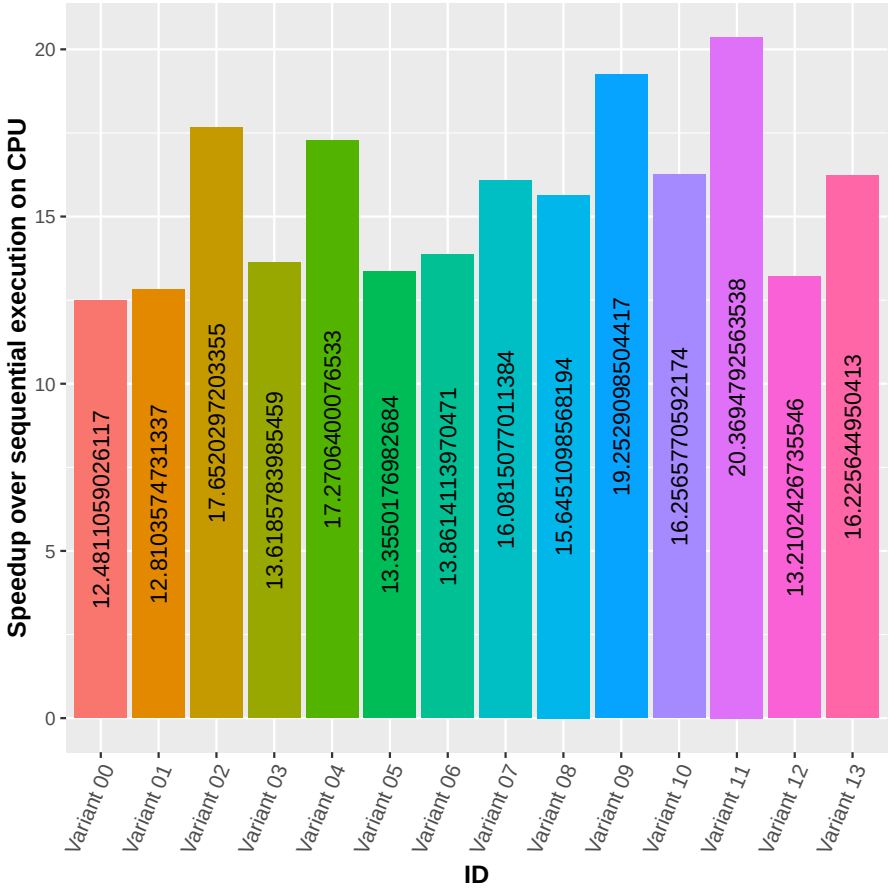


Figure 38: Runtime comparison of variants of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and constant changing thermal conductivity

Figure 39 details the results of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity.

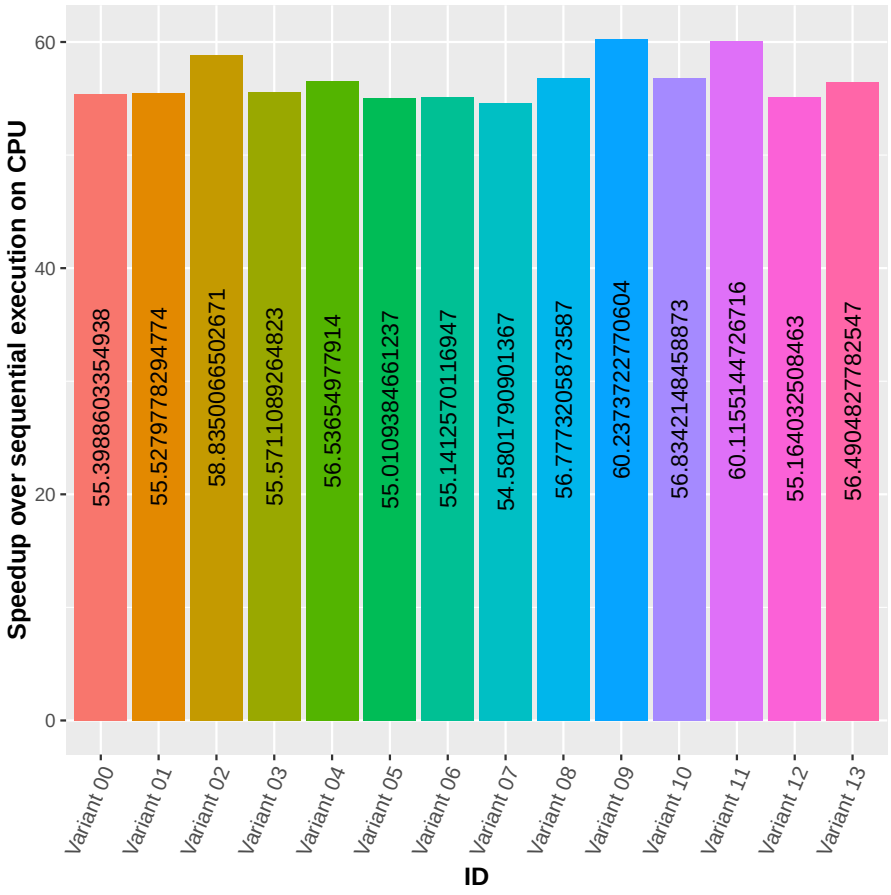


Figure 39: Runtime comparison of variants of the two-dimensional steady-state heat equation with Dirichlet boundary conditions on the unit square and smoothly changing thermal conductivity

E

HYBRID TILING PERFORMANCE RESULTS

This appendix contains the complete performance results of the hybrid tiling experiments presented in Section 6.4. Figure 40 illustrates the execution time of the different test cases. Each plot names the stencil, the problem size, and the performed time steps.

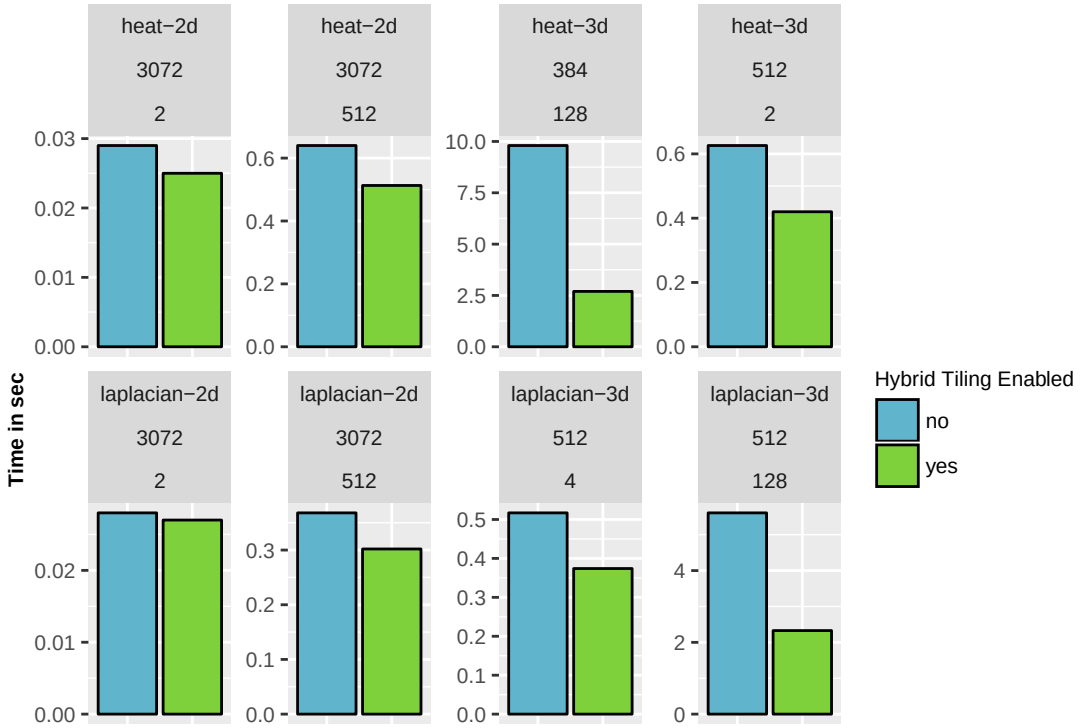


Figure 40: Execution time of different stencil test cases

Table 13 details the performance results of all test cases.

Stencil	Problem Size	Time Steps	Hybrid Tiling Enabled	Time [s]	GFLOPS
laplacian-3d	512	128	no	5.608	24.219884
laplacian-3d	512	128	yes	2.328	58.347458
laplacian-3d	512	4	no	0.517	8.204049
laplacian-3d	512	4	yes	0.374	11.347147
laplacian-2d	3072	512	no	0.368	78.636611
laplacian-2d	3072	512	yes	0.302	95.808356
laplacian-2d	3072	2	no	0.028	4.097634
laplacian-2d	3072	2	yes	0.027	4.248312
heat-3d	384	128	no	9.802	19.653440
heat-3d	384	128	yes	2.698	71.409268
heat-3d	512	2	no	0.626	11.445464
heat-3d	512	2	yes	0.420	17.034886
heat-2d	3072	512	no	0.640	67.911916
heat-2d	3072	512	yes	0.513	84.706147
heat-2d	3072	2	no	0.029	5.908413
heat-2d	3072	2	yes	0.025	6.735032

Table 13: Performance results of hybrid tiling experiments

BIBLIOGRAPHY

[1] Mehdi Amini, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, François-xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4All: From Convex Array Regions to Heterogeneous Computing. In *2nd International Workshop on Polyhedral Compilation Techniques (HiPEAC)*, January 2012.

[2] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Compiler Construction*, Lecture Notes in Computer Science, pages 244–263. Springer, March 2010.

[3] Mark Ebersole. What Is CUDA | NVIDIA Official Blog. `https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/`, 2012.

[4] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, number 1132 in Lecture Notes in Computer Science, pages 79–103. Springer, 1996.

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

[6] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 66:66–66:75, 2014.

[7] Mark Harris. An Easy Introduction to CUDA C and C++. `https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/`, 2012.

[8] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974.

[9] Stefan Kronawitter and Armin Größlinger. Polyhedral Search Space Exploration for Stencil Codes. In *3rd International Workshop on High-Performance Stencil Computations*, January 2016. URL `http://www.exastencils.org/histencils/2016/`.

[10] Stefan Kronawitter and Christian Lengauer. Optimizations Applied by the ExaStencils Code Generator. Technical Report MIP-1502, Faculty of Informatics and Mathematics, University of Passau, January 2015.

[11] Christian Lengauer. Loop parallelization in the polytope model. In *4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Lecture Notes in Computer Science*, pages 398–416. Springer, 1993.

[12] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. ExaStencils: Advanced Stencil-Code Engineering. In *Lecture Notes in Computer Science*, volume 8806, pages 553–564, 2014.

[13] Christian Lengauer, Matthias Bolten, Robert D. Falgout, and Olaf Schenk. Advanced Stencil-Code Engineering (Dagstuhl Seminar 15161). Dagstuhl reports, Dagstuhl Seminar 15161, 2015.

[14] J. David Logan. *Applied Partial Differential Equations*. Undergraduate Texts in Mathematics. Springer, 3 edition, 2015.

[15] Naoya Maruyama and Takayuki Aoki. Optimizing Stencil Computations for NVIDIA Kepler GPUs. In *1st International Workshop on High-Performance Stencil Computations*, 2014.

[16] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-Stream Compiler. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer US, 2011.

[17] Paulius Micikevicius. 3D Finite Difference Computation on GPUs Using CUDA. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, 2009.

[18] NVIDIA. GeForce GTX TITAN Black Edition Grafikkarte | GeForce | NVIDIA. `http://www.nvidia.de/object/geforce-gtx-titan-black-de.html#pdpContent=2`.

[19] NVIDIA. NVIDIA Kepler Compute Architecture Datasheet, May 2012.

[20] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210. 2014.

[21] NVIDIA. CUDA Compiler Driver NVCC, September 2015.

[22] NVIDIA. CUDA C Programming Guide, September 2015.

[23] NVIDIA. Tuning CUDA Applications for Kepler, September 2015.

[24] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *State of the Art Reports*, volume 26, pages 21–51, August 2005.

[25] S. V. Patankar and D. B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15(10):1787 – 1806, 1972.

[26] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers. In *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51, November 2014.

[27] Richard M. Stallman. Using the GNU Compiler Collection. 2015. URL `https://gcc.gnu.org/onlinedocs/gcc/`.

[28] Diego A. Vasco, Nelson O. Moraga, and Gundolf Haase. Parallel Finite Volume Method Simulation of Three-Dimensional Fluid Flow and Convective Heat Transfer for Viscoplastic Non-Newtonian Fluids. *Numerical Heat Transfer, Part A: Applications*, 66(9):990–1019, 2014.

[29] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *3rd International Congress on Mathematical Software (ICMS)*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[30] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54:1–54:23, January 2013.

[31] H. Le Verge. A note on Chernikova's Algorithm. Technical Report 635, IRISA-Rennes, July 1994.

[32] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 579–586. IEEE, July 2009.

DECLARATION

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig 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, Germany, October 2016

Christoph Woller

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Final Version as of October 10, 2016 (`classicthesis` Version 1.0).