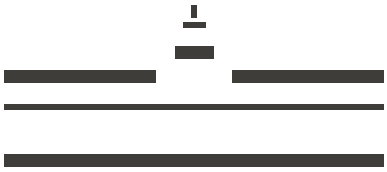

Westfälische
Wilhelms-Universität
Münster

EFFICIENT IMPLEMENTATION AND OPTIMIZATION OF GEOMETRIC MULTIGRID OPERATIONS IN THE LIFT FRAMEWORK

MARTIN LÜCKE

Masterthesis

Computer Science Department
Parallel and Distributed Systems
University of Münster

FIRST SUPERVISOR: Prof. Dr. Sergei Gorlatch
(University of Münster)
SECOND SUPERVISOR: Prof. Dr. Herbert Kuchen
(University of Münster)

July 2018

Martin Lücke: *Efficient Implementation and Optimization of Geometric Multigrid Operations in the Lift Framework,* Masterthesis, © July 2018

"I started the day with some nothin' tea.
Nothin' tea is easy to make.
First, get some hot water, then add nothin'."

— Andy Weir, The Martian

This thesis is dedicated to my former teacher Wilhelm Sternemann.
He initially raised my interest in Computer Science
by introducing me to the beautiful world of fractals.

ABSTRACT

Geometric Multigrid (GMG) is an efficient method to solve partial differential equations. It consists of four operations (*smooth*, *residual*, *restrict* and *prolongate*), which are applied iteratively. All four operations are stencil computations and hence benefit from being executed on many-core architectures like GPUs. Programs executed on a GPU are usually written using low-level programming approaches like OpenCL or CUDA. In order to achieve high performance, expert knowledge is required to manage the hardware details such as the memory and thread hierarchy, exposed to the programmer by these low-level approaches. However, the optimizations that achieve high performance may be different for individual architectures and programs, so tuning for one device often leads to poor performance on other devices. Using the Lift framework is a promising approach for achieving *performance portability*. Computations are expressed using a small set of reusable parallel primitives. Optimizations are encoded in *rewrite rules* to be applied automatically in a rewriting process.
This thesis demonstrates how the four GMG operations can be expressed as a combination of simple 1-dimensional Lift primitives. In addition to that, well-known optimizations for GMG methods are analyzed. Finally, we express some of them in the functional language of Lift and investigate why the other optimizations are currently not expressible.

The process of preparing programs for a digital computer is especially attractive, not only because it can economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.

— Donald E. Knuth [14]

ACKNOWLEDGMENTS

I would first like to thank my supervisor Sergei Gorlatch for giving me the opportunity to write my thesis in his department. I am especially thankful for the opportunity to visit the University of Stuttgart for one week, where I learned a lot and met inpiring people. I thank Bastian Hagedorn for introducing me to the Lift project. The door to his office was always open whenever I ran into trouble and he provided me great advice and guidance. I thank the Lift team for their ongoing interest in my research and inpiring debates during the weekly meeting. I am looking forward to continue collaboration on topics related to the work, described in this thesis. I thank Johannes Lenfers for many fruitful discussions about both our theses and for giving me strength even in the most stressful times. Finally, I thank my friends, my family and my sister for their support and encouragement throughout my years of study and through the process of researching and writing this thesis.

CONTENTS

LIST OF FIGURES

LISTINGS

1 INTRODUCTION AND BACKGROUND

1.1 MOTIVATION

Today, elliptic partial differential equations are used for describing a broad range of problems, which range from fluid dynamics [23], tumor growth [5], up to the evolution of biological species [30]. Solving them efficiently is critical. Geometric Multigrid (GMG) methods have the unique potential to solve these mathematical problems with N unknowns with $\mathcal{O}(N)$ work. In order to execute the methods fast and efficently, parallelization is key.

Nowadays, GPUs are an inherent part of high performance computing for accelerating computations. However, optimizing computations for them remains challenging. Programs executed on a GPU are usually written using low-level programming approaches like OpenCL or CUDA. In order to achieve high performance, expert knowledge is required to manage the hardware details such as memory and thread hierarchy, exposed to the programmer by these low-level approaches. However, the optimizations that achieve high performance may be different for individual architectures and programs, so tuning for one device often leads to poor performance on other devices [19]. In order for a program to run well on more than one architecture, it is necessary to optimize it by hand multiple times.

Using a functional approach for generating high-performance code for GPUs has already been proven to be successful for different applications with the introduction of Lift [26]. The problem of performance portability is solved by automatically applying different optimizations, encoded as semantics preserving rewrite rules. To achieve this, the program is written in the high-level functional language of Lift. Then it gets transformed to an intermediate representation, which encodes OpenCL low-level details. The rewrite system of Lift automatically explores different optimizations and generates many optimized OpenCL kernels for the purpose of finding one that executes well on the given architecture.

The main goal of this thesis is to achieve high performance of Multigrid methods by expressing them in the functional language of Lift and extend it, if additional features are necessary for this. Moreover, we will elaborate if the Lift rewrite system can be extended to support well-known optimizations, applicable to GMG operations, and evaluate their performance.

1.2 BACKGROUND

In this section we introduce all the concepts needed to understand this work. We start by introducing Multigrid methods and their parallelization. Then we will look at heterogenous devices and how to program them using OpenCL. Afterwards we introduce the LIFT framework.

1.2.1 *Multigrid methods*

Multigrid methods solve partial differential equations (PDEs) using a combination of different components. The components will be shown using the example of the 1D-Poisson equation with Dirichlet boundary conditions following the approach of Meister [17]. However, many details will be omitted for the sake of brevity.

$$\begin{aligned} &\text{Given: } \Omega = (0,1) \text{ and } f : \Omega \to \mathbb{R} \\ &\text{Solution: } u : \Omega \to \mathbb{R} \text{ satisfying} \\ -u''(x) &= f(x) && \text{for } x \in \Omega \\ u(x) &= 0 && \text{for } x \in \partial\Omega = \{0,1\} \end{aligned} \tag{1.1}$$

In order to solve this differential equation we have to discretize it on our domain Ω first. For this purpose we define the *step size sequence* $\{h_l\}_{l=0}^{\infty}$ with $h_0 = 1/2$ and $h_l = h_0/2^l$ as well as the *grid sequence*

$$\Omega_l := \Omega_{hl} = \{jh_l | j = 1, ..., 2^{l+1} - 1\} \tag{1.2}$$

With $u_j^l := u(jh_l), j = 1, ..., N_l := 2^{l+1} - 1$ and $f_j^l := f(jh_l), j = 1, ...N_l$ we can approximate the differential equation with the algebraic equation

$$\frac{-u_{j-1}^l + 2u_j^l - u_{j+1}^l}{h^2} = f_j^l \tag{1.3}$$

After eliminating boundary values $u_0^h = 0$ and $u_{N_l+1}^h = 0$ we get the following system of linear equations

$$A_l u^l = f^l \tag{1.4}$$

with $u^l = (u_1^l, ..., u_{N_l}^l)^T$, $f^l = (f_1^l, ..., f_{N_l}^l)^T$ and

$$A_l = \frac{1}{h_l^2} \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{N_l x N_l}. \tag{1.5}$$

This system of linear equations represents our initial problem, so solving yields the solution.

1.2.1.1 *Smoothing*

The first component of the multigrid method is the iterative *Jacobi relaxation method*. This method is a variation of the iterative *Jacobi method*. The method contains an additional parameter ω which can be varied to maximize convergence rate. For $\omega = \frac{1}{2}$ both methods are equivalent. The method with parameter ω for this problem is as follows

$$u_{m+1}^l = \underbrace{(\mathbb{1} - \omega h_l^2 A_l)}_{M_l(\omega):=} u_m^l + \underbrace{\omega h_l^2 \mathbb{1}}_{N_l(\omega):=} f^l \tag{1.6}$$

This method converges for every starting value $u_0 \in \mathbb{R}$ to the solution even without using a relaxation paramater $\omega \neq \frac{1}{2}$. Besides, the convergence rate does not improve when varying ω. This is not a problem, because we do not use Jacobi to solve the original problem. Instead, we examine how the Jacobi method effects the error in iteration step m with different relaxation parameters ω.
It can be shown that A_l and the iteration matrix $M_l(\omega)$ have the same eigenvectors

$$e^{l,j} = \sqrt{2h_l} \begin{pmatrix} \sin(j\pi h_l) \\ \vdots \\ \sin(j\pi N_l h_l) \end{pmatrix} \text{ with } j = 1, ..., N_l \tag{1.7}$$

and respective eigenvalues, depending on the parameter ω.

$$\lambda^{l,j}(\omega) = 1 - 4\omega \sin^2(\frac{j\pi h_l}{2}) \text{ for } j = 1, ..., N_l. \tag{1.8}$$

We can express the error in iteration m as

$$u_m^l - u^{l,*} = \sum_{j=1}^{N_l} \alpha_j \left(\lambda^{l,j}(w)\right)^m e^{l,j} \text{ for } m = 0, 1, ... \text{ and } \alpha_j \in \mathbb{R} \tag{1.9}$$
$$\text{with } u^{l,*} = A_l^{-1} f_l \text{ as exact solution}$$

It is obvious here that the error depends significantly on the eigenvalues. The eigenvalues of this problem are a discrete sampling of the sine wave $\sin(j\pi x)$ with $j = h_l, 2h_l, ..., N_l h_l$. For $1 \leq j \leq \frac{N_l+1}{2}$ we call them *low frequency* errors and for $\frac{N_l+1}{2} \leq j \leq N_l$ we call them *high frequency* errors.

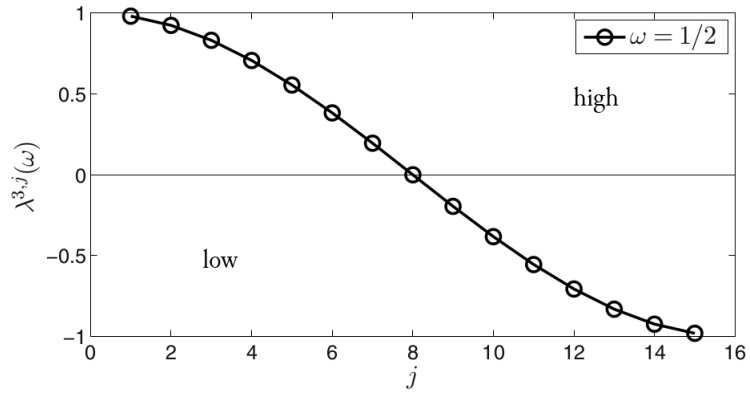

Figure 1.1: Eigenvalue distribution for the Jacobi relaxation method for $\omega = \frac{1}{2}$ [17, p. 111]

Figure 1.1 shows the symmetrical distribution of the eigenvalues around 0. With Equation 1.9 we see that low and high frequency error components persevere, but medium frequency errors are damped. Choosing $\omega = \frac{1}{2}$ yields optimal convergence rate, however for Multigrid methods it is fundamental to reduce high frequency errors as we will see later.

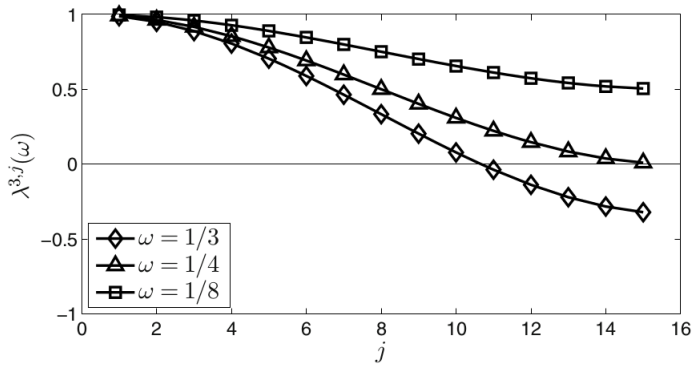

Figure 1.2: Eigenvalue distribution for the Jacobi relaxation method for different ω [17, p. 113]

In Figure 1.2 we see that choosing $\omega = \frac{1}{4}$ yields a non optimal convergence rate in the Jacobi relaxation method but leads to a very good dampening of high frequency errors, which is what we desire.
The total error is a linear combination of all error frequencies. The effect of two iterations of this Jacobi relaxation on the total error vector can be seen in Figure 1.3, with k as the vector-components. First, the error oscillates and has many high frequency parts. After two iterations the error is still high, but much smoother. This means we

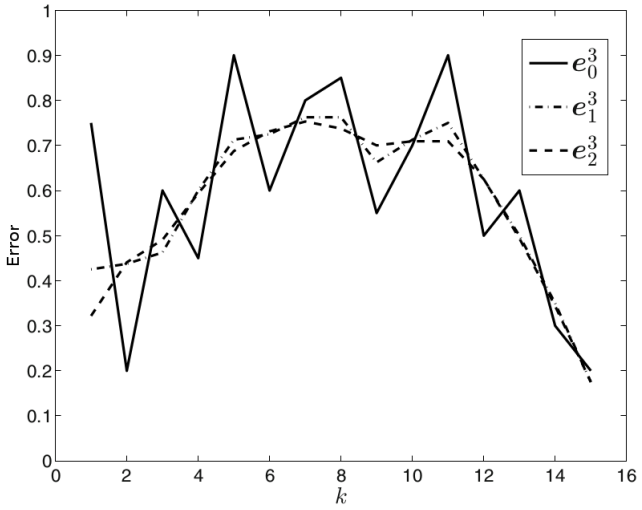

Figure 1.3: Development of the error vector with the Jacobi relaxation method,
$\omega = \frac{1}{4}$, grid level $l = 3$, $N_l = 15$ [17, p. 114]

can use the Jacobi relaxation method to reduce high frequency error components in our multigrid method. After damping high frequency parts of the error it would be beneficial to also reduce low frequency parts to get closer to the solution of our problem. This does not work with the previously regarded Jacobi relaxation method, but we will take advantage of its effects.

1.2.1.2 *Restriction*

The smoothness of the error after the application of the Jacobi relaxation method allows us to approximate the error of grid Ω_l on a coarser grid. To do that we need a transformation $F : \mathbb{R}^{N_l} \to \mathbb{R}^{N_{l-1}}$. We call it restriction from Ω_l to Ω_{l-1} if it is linear and onto. An example which satisfies the requirements is the so called *injection*, where we leave out each second component of u^l.

$$R_l^{l-1} u^l = \begin{pmatrix} u_2^{l-1} \\ u_4^{l-1} \\ \vdots \\ u_{N_{l-1}}^{l-1} \end{pmatrix} \tag{1.10}$$

This means the values at $\frac{\Omega_l}{\Omega_{l-1}}$ are omitted. The loss of information leads to non optimal performance, so instead we use *linear restriction* which uses

$$R_l^{l-1} = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 & & & & & & \\ & & 1 & 2 & 1 & & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & & & & & \ddots & \ddots & \ddots & \\ & & & & & & & 1 & 2 & 1 \end{pmatrix}. \tag{1.11}$$

Here each new value is an interpolation between previously neighboured values, so we do not lose information anymore. The structure of this restriction can be seen in Figure 1.4.

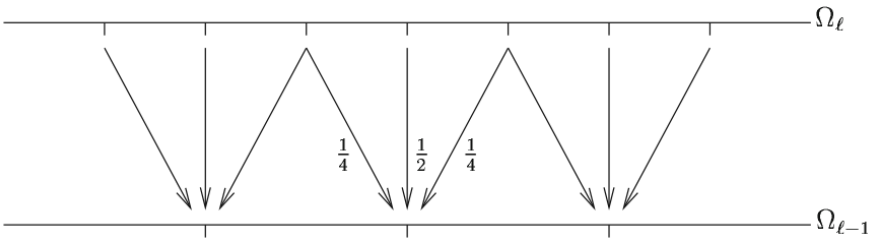

Figure 1.4: linear restriction [17, p. 116]

1.2.1.3 *Prolongation*

After restricting the error to a coarser grid we need a method to propagate the result back to the finer grid. For this we define a transformation $G : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$ as *prolongation* from Ω_{l-1} to Ω_l if it is linear and one-to-one. We do not want to lose any information during the prolongation, so the image has to have the full dimension N_{l-1}, which results in the one-to-one requirement.
An example for this is the *linear prolongation* $G(u) = P_{l-1}^l u$ with

$$P_{l-1}^l = \frac{1}{2}\begin{pmatrix} 1 & & \\ 2 & & \\ 1 & 1 & \\ & 2 & \\ & 1 & \\ & & \ddots \\ & & 1 \\ & & 2 \\ & & 1 \end{pmatrix} \in \mathbb{R}^{N_l x N_{l-1}}. \tag{1.12}$$

The structure of the propagation to the finer grid is shown in the following Figure 1.5.

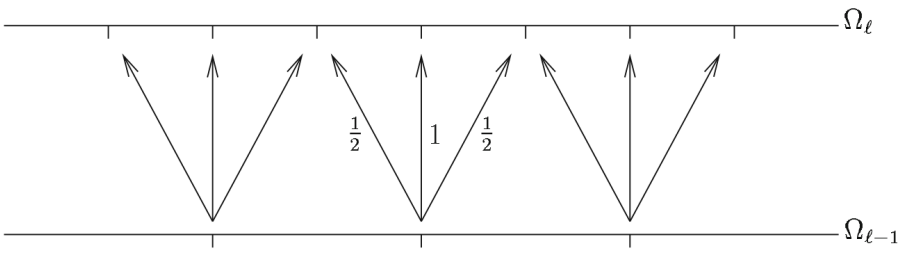

Figure 1.5: linear prolongation [17, p. 117]

1.2.1.4 *Coarse Grid Correction*

With the restriction and the prolongation we can now formulate the coarse grid correction. After applying the Jacobi method with $\omega = \frac{1}{4}$ for m times the error e_m^l is generally smooth. This means we can approxiamte it on the coarser grid Ω_{l-1} with much lower cost.
With the defect

$$d_m^l := A_l u_m^l - f^l \tag{1.13}$$

we have

$$A_l e_m^l = d_m^l \tag{1.14}$$

To approximate e_m^l we examine Equation 1.14 on the grid Ω_{l-1} and solve it precisely and prolongate the resulting vector back to grid Ω_l. So on the coarse grid we have

$$\begin{aligned} &A_{l-1} e_m^{l-1} = d_m^{l-1} \\ &\text{with the restricted defect } d^{l-1} = R_l^{l-1} d_m^l. \end{aligned} \tag{1.15}$$

After solving Equation 1.15 precisely the following Prolongation results in the approximation of the error e_m^l:

$$P_{l-1}^l e^{l-1} = P_{l-1}^l A_{l-1}^{-1} d^{l-1} \tag{1.16}$$

The resulting coarse grid correction is

$$\begin{aligned} &u_m^{l,new} = \phi_l^{CGC}(u_m^l, f^l) \\ &\text{with } \phi_l^{CGC}(u_m^l, f^l) = u_m^l - P_{l-1}^l A_{l-1}^{-1} R_l^{l-1}(A_l u_m^l - f^l) \end{aligned} \tag{1.17}$$

To summarize, it consists of restricting the defect to the coarser grid, solving for e_m^{l-1} precisely and propagating the result back to the fine grid. The coarse grid correction itself cannot be used as an iterative solver, because it is not convergent. However, this is not a problem,

as we do not want to use it as a solver but as a part of the multigrid method. For this we analyse its effect on the eigenvalues and hence the error components.

$$\begin{aligned} e_m^{l,new} &= u_m^{l,new} - u^{l,*} \\ &= u_m^l - u^{l,*} - P_{l-1}^l A_{l-1}^{-1} R_l^{l-1}(A_l u_m^l - f^l) \\ &= e_m^l - P_{l-1}^l A_{l-1}^{-1} R_l^{l-1} A_l e_m^l \\ &= \Psi_l^{CGC}(e_m^l) \end{aligned} \tag{1.18}$$

with

$$\begin{aligned} &\Psi_l^{CGC} : \mathbb{R}^{N_l} \to \mathbb{R}^{N_l} \\ &\qquad e \mapsto \Psi_l^{CGC}(e) = (\mathbb{1} - P_{l-1}^l A_{l-1}^{-1} R_l^{l-1} A_l)e. \end{aligned} \tag{1.19}$$

So the images of the eigenvectors are as follows.

$$\begin{aligned} &\Psi_l^{CGC}(e_j^l) = s_j e^{l,j} + s_j e^{l,\bar{j}} \text{ for } j \in \{1,...,N_{l-1}\} \text{ and } \bar{j} = N_l + 1 - j, \\ &\Psi_l^{CGC}(e_j^l) = e^{l,j} \qquad\qquad \text{for } j = N_{l-1} + 1, \\ &\Psi_l^{CGC}(e_j^l) = c_{\bar{j}} e^{l,j} + c_{\bar{j}} e^{l,\bar{j}} \text{ for } j = N_l + 1 - \bar{j} \text{ and } \bar{j} \in \{1,...,N_{l-1}\} \\ &\qquad \text{with} c_{\bar{j}} = \cos^2(\bar{j}\pi\frac{h_l}{2}) \text{and } s_j = \sin^2(j\pi\frac{h_l}{2}) \end{aligned} \tag{1.20}$$

For large wave numbers j the factor c_j approaches 1, so the coarse grid correction will not have a beneficial effect on high frequency error components. Small wave numbers on the other hand are damped with the factor s_j, which is small for small j. This means we can expect good dampening of low frequency error components which is shown in figure 1.6.

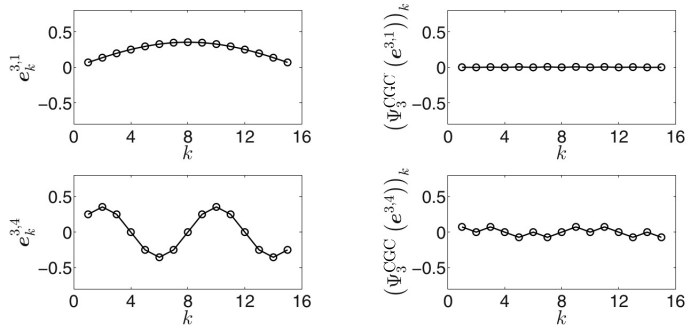

Figure 1.6: Reduction of low frequency error components with the coarse grid correction Ψ_l^{CGC} [17, p. 129]

1.2.1.5 *Multigrid Method*

We have defined and analyzed two different components, which we will use to realize a *two grid method* for the 1D-Poisson problem. The

first one is the Jacobi method, which we call smoother ϕ_l^v with v iterations on the grid level l. The second one is the coarse grid correction ϕ_l^{CGC}. We combine them to a two grid method: $\phi_l^{TGM(v1,v2)} := \phi_l^{v2}\ o\ \phi_l^{CGC}\ o\ \phi_l^{v1}$. First, we apply the smoother $v1$ times to reduce high frequency error components. Now it is possible to apply the coarse grid correction which approximates the error on a coarser grid and corrects the initial guess. Last, we do $v2$ iterations of post smoothing to reduce eventual effects of the prolongation on the high frequency error components. The structure of this can be seen in Figure 1.7.

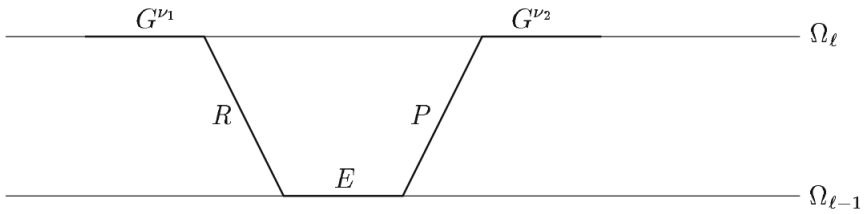

Figure 1.7: Two grid method with smoother G, restriction R, solving E, prolongation P [17, p. 133]

image

This method can be applied iteratively with iteration matrix

$$M_l^{TGM(v1,v2)} = M_l^{v2}(\mathbb{1} - P_{l-1}^l A_{l-1^{-1}} R_l^{l-1} A_l) M_l^{v1}. \tag{1.21}$$

Figure 1.8 shows that one iteration of this reduces the error significantly even with $v2 = 0$.

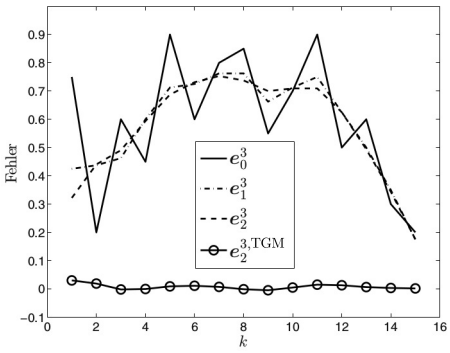

Figure 1.8: Development of the error after three Jacobi smoothings and a subsequent coarse grid correction [17, p. 134]

A problem in practice is the solving of the defect Equation 1.14: $A_l e_m^l = d_m^l$ on the coarse grid. In most applications the coarse grid still has a very large number of unknowns, so the solving is still very expensive. However, prolongating the result e^{l-1} of this to the fine layer only yields an approxamtion of e^l, so it is not necessary to solve for e^{l-1} precisely. The Equation 1.14 has a similar form to our initial Equation 1.4: $A_l u^l = f^l$, so we can use the two grid method to approximate e^{l-1}. Here we have to solve $A_{l-2} e^{l-2} = d^{l-2}$. We can apply this idea

successively and get a multigrid method with $l+1$ grids $\Omega_l, ..., \Omega_0$, where we only have to solve $A_0 e^0 = d^0$. In practice we choose l so that solving for e^0 is simple and has minimal cost. The structure of this multigrid method can be seen in Figure 1.9 and resembles a V-shape, called V-cycle. [17, pp. 106 sqq.]

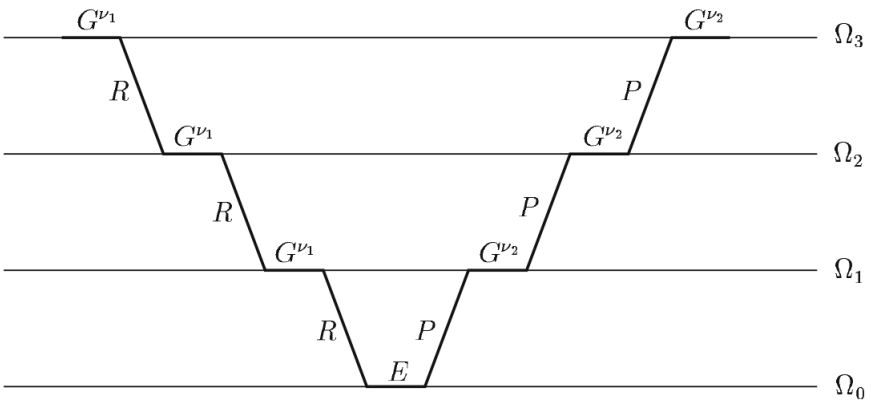

Figure 1.9: V-cycle of the multigrid method with smoother G, restriction R, solving E, prolongation P [17, p. 138]

With the presented components we have constructed an efficient multigrid method for solving the 1D-Poisson equation. after smoothing the error on every layer, we restrict it subsequently across l layers to the coarsest layer. There we have reduced the number of unknowns so much that solving precisely is simple. This solution for the error is then propagated across l layers, using the prolongation, back to the finest layer. After every application of the prolongation the approximation of the error is smoothed again to minimize eventual undesired effects. On the finest level we finally correct our guess with the calculated error and achieve a new better approximation of the correct solution for our initial problem.
However, multigrid methods are not black box solvers. This means for other problems the presented components might not have the same beneficial effect on the error and thus have a lower efficiency. In that case we can analyze and choose for example another smoother like the Gauss-Seidel method to maximize our efficiency. This is very worthwhile if a problem has to be solved many times like the 3D Helmholtz equation in daily weather prediction.

Efficiency

Multigrid methods are highly efficient. The number of arithmetic operations to solve a problem is proportional to the number N of unknowns in the problem. So they are optimal and belong to the class $\mathcal{O}(N)$. Besides that the proportionality constants in this statement are small if the multigrid method is designed well. So they are also very efficient in a practical sense.

Furthermore, multigrid methods are applicable to a wide range of problems with full efficiency. [28, pp. 20,21] For example:

- to general elliptic equations with variable coefficients
- in general domains
- for general boundary conditions
- in 2D, 3D dimensions (trivially also for 1D problems)
- to scalar equations and systems of equations

Adaptivity

In many cases it is not suitable to define a global grid independent of the solution process. Often some features like shocks, singularities, oscillations or turbulent behaviour may not be recognized until the solution process. This means the discretization error is not uniform across the domain, so it is necessary to adapt the grid to the behaviour of the solution. This is especially relevant in 3D applications where the resolution of the grid has to be very fine at crucial points, but using this fine resolution at the whole domain would be unnecessary and expensive. This is easy to realize with multigrid methods. In adaptive multigrid methods finer grids are not constructed globally, but only in regions of the domain where the discretization error is significantly large. All other components of the method remain the same. [28, p. 22]

Parallelism

In order to minimize the time to solve large problems, highly parallel computers are used. It is necessary to achieve a good parallelization to maximize efficiency. For this we have to take many aspects into account, for example minimizing the communication overhead of our operations. We will go into detail about this in Chapter 2.

In summary Multigrid methods are highly relevant in practice because they prove to be efficient for a variety of problems. Parallelization of them is thus a much discussed topic as well.

1.2.2 *Parallel Hardware*

Graphics processing units (GPUs) have traditionally been designed for accelerating complex graphics and 3D-scenes. Today, despite the name, they have been generalized to support a broad range of general computations. They are constructed to leverage the *Single Instruction, Multiple Data* (SIMD) principle, such as that they contain a magnitude of lightweight cores to perform computations in parallel. In contrast,

CPUs only contain a few very capable cores, which for example support instruction prefetching and out-of-order execution. They are optimized for a low instruction latency in tradeoff for lower instruction throughput. The GPU architecture, on the other hand, is optimized to achieve a high instruction throughput at the cost of instruction latency. This design enables the usage of GPUs as accelerators, for example in scientific applications. Compute intensive work, which can be executed highly parallel, can be offloaded to a GPU for computation.
The basic architecture of a GPU is presented by reference to the latest GP104 chip by NVidia [18]. A GPU consists of several *Streaming Multiprocessors* (SM), which itself contain many so called *CUDA Cores*. In comparison to a CPU core, a CUDA core has very low capability. Each SM has its own set of schedulers, registers, instruction pipeline and small caches. Threads are scheduled in groups of 32 called warps to the CUDA cores inside an SM. All CUDA cores of a warp execute the same instruction on different data independently of other warps. Each GPU contains a large but small off-chip memory of several gigabytes analogous to the RAM called *global memory*. It is accessible to all CUDA cores and even to the CPU of the system. Besides that, each SM contains a smaller and faster on-chip memory of several kilobytes, which is only accessible to the cores of this SM and is called *local memory*. Aside from a few transparent cache layers, the programmer is responsible for explicitly managing the memory. In consequence of the difference in latency between the different memory types of roughly 100 times, it is crucial to utilize local memory for exploiting data locality and thus achieving high performance.

1.2.3 *Programming Parallel Devices in OpenCL*

Today, the newest CPUs have multiple cores and use parallelism rather than higher clock speeds to speed up computations. GPUs have done the transformation from being fixed hardware rendering pipelines to becoming general purpose computation devices. Besides that, highly parallel accelerators like the Intel Xeon Phi have emerged and Field Programmable Gate Arrays (FPGAs) can execute parallel code as well. However it is still problematic to write a program that works on all of those devices. They have dramatically different hardware specifications and different specific programming models for programming them. OpenCL[9] is an open programming model developed by the Khronos group in 2008 that supports all of the mentioned devices from a single codebase. The hierarichally organized model will be described in the following.

1.2.3.1 *Platform Model*

The *platform model* consists of one *host* which is connected to one or more *compute devices*. For example the CPU is the host and the GPU of the system is a compute device. Each compute device contains one or more *compute units (CU)*, which in term contain one or more *processing elements (PE)*. In the example of the GPU as compute device, CUs map to SMs and PEs to the cores inside a SM. This hierarchy is shown in figure 1.10.

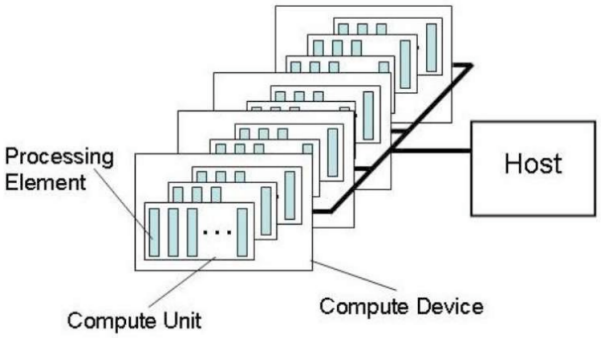

Figure 1.10: OpenCL Platform Model [9, p. 23]

1.2.3.2 *Execution Model*

Execution of an OpenCL program is divided into the *host program* which executes on the host and the *kernels* which execute on the compute devices. The host program is responsible for managing the kernels, e.g. starting them and providing data to them. In the host program it is also determined how the started kernel instances are structured in dimensions 0 to 3. A kernel instance is called *work-item* and is identified by its *global ID* which represents its position in the global 3D array of work-items. Furthermore, work-items can be grouped into work-groups. For identification each work-group has a *work-group ID* and each work-item inside it has a *local ID*, representing their position in the underlying grids. The hierarchical organizations of work-items in work-groups makes it easier to distribute the work between them. All work-items execute the same kernel, however at one point during time not all work-items have to execute the same line of code, so different work-items may take different execution paths in the kernel.

1.2.3.3 *Memory Model*

OpenCL exposes different *address spaces* to the programmer. At the start of an OpenCL programm all data that is needed on the device is copied from the host to the *global memory* and *constant memory* of the device. These are the only address space the host can access, so the

result of the computation has to be stored here at the end or the host would not be able to access it. All work-items have read-access to the constant memory and full access to the global memory. In addition to that there are also address spaces that are only accessible to the compute device: The work-group specific *local memory* and the work-item specific *private memory*. This means each work-group has its own local memory that can only be accessed by the work-items of this work-group and each work-item has its own private memory which is not visible to other work-items. Note the similarity between the GPU memory hierarchy and the OpenCL memory model.

1.2.3.4 *Programming Model*

OpenCL supports a *data parallel programming model* and a *task parallel programming model*. We focus on the former, where a computation is expressed in terms of instructions applied to multiple elements of a large structure. For this multiple kernel instances are started, organized accourding to the explanation above. In the task parallel programming model only a single kernel instance is started. Parallelism can be introduced by vectorization or by launching different kernels concurrently.

OpenCL exposes many low-level hardware details, which the programmer has to manage himself. To achieve high performance, it is important to take advantage of the memory hierarchy by exploiting the local and private memory in order to hide latency of the larger, slower memories. Memory accesses can be performed coalescened if elements are accessed with the right alignment. Besides that kernel code has to include parallelism on work-item and on work-group layer to maximize hardware utilization and avoid work-item idling. Furthermore, the actual sizes and speeds of the address spaces depend heavily on the used hardware. So OpenCL programming in the C-like language has a high capacity for optimizations but is also very error prone. These aspects mean optimizing a programm requires expert knowledge of the used hardware and much experience.

1.2.4 ***The Lift Framework***

Lift is a novel approach for producing highly optimised OpenCL code for parallel execution. For this a computation is expressed in an abstract high-level language which encapsulates the algorithmic structure of the computation as well as specific optimizations. It consists of simple functional primitives which are divided into two groups. *High-level algorithmic primitives* are exposed to the developer and are used as building blocks in a functional high-level program to express the computation, similar to other skeleton libraries like SkelCL [24], Skandium[16] and Skil[6]. Using *rewrite rules*, this high-level

program is rewritten by the LIFT compiler into an intermediate language (IL) consisting of *low-level OpenCL specific primitives*. They expose OpenCL-specific features and enable the expression of optimizations like loop unrolling without losing information or context. A program expressed in *low-level OpenCL specific primitives* is closely related to an OpenCL kernel, such as that imperative OpenCL code can be generated from it. The compilation workflow is shown in Figure 1.11.

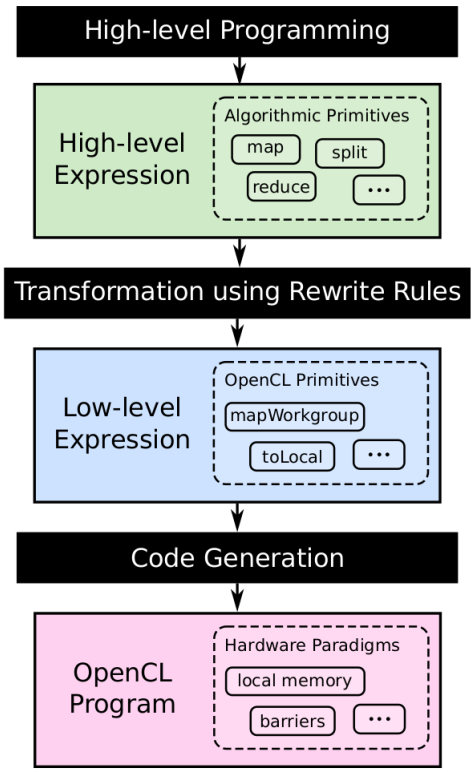

Figure 1.11: LIFT compilation workflow [11]

1.2.4.1 *Primitives*

In LIFT computations are expressed as combinations of basic algorithmic blocks called primitives. The primitives are designed in a functional fashion, so they just describe what is happening to the data and its structure, not exactly how the output is being computed. The most important algorithmic primitives for the applications of this thesis are described in the following with an additional example for each. In order to define how primitives can be combined we specify the type for each primitive: We write $e : t$ to denote that expression e is of type t. For an array with n elements of type T we write $[T]_n$. We denote functions that map elements of type α to elements of type β as $\alpha \to \beta$. Functions are curried, so we write $\alpha \to \beta \to \gamma$ for a function, which takes elements of type α and β and outputs an element of type

γ. For clarity we sometimes write $f(\alpha, \beta) \to \gamma$ for the same function with the name f. With $\circ$ we denote sequential function composition and with \$ function application, whereby $\circ$ binds stronger than \$. So $f\ o\ g$ \$ input $= (f\ o\ g)$ \$ input. In addtion, we denote anonymous functions as x => expr \$ x. A tuple of two elements of type α and *beta* is denoted as $\{\alpha, \beta\}$.

Map : $(f : T \to U) \to in : [T]_n \to [U]_n$

The *Map* primitive is widely known to functional programmers as it is present in many functional languages. It takes a function f and applies it to all elements of an array. The result is an array of the same length. The map primitive is the only primitive in Lift to explicitly express data parallelism.

$$Map(f : (x \to x^2)) \ \$ \ [0,1,2,3,4] \to [0,1,4,9,16] \tag{1.22}$$

Reduce : $(init : U, f : (U, T) \to U) \to in : [T]_n \to [U]_1$

The *Reduce* primitive accumulates all elements of an array to an array with only a single element. For that it uses an initialization value and an accumulation function.

$$\begin{aligned} &Reduce(init : 0.0, f : ((x,y) \to x + y)) \ \$ \ [0,1,2,3,4] \to [10] \\ &Reduce(init : 1.0, f : ((x,y) \to x * y)) \ \$ \ [1,2,3,4] \to [24] \end{aligned} \tag{1.23}$$

Next to the normal *Reduce* primitive there exist variations of it with the same semantics, but a different OpenCL code generation. For instance ***ReduceSeqUnroll*** can be used if the number of elements in the reduction is known at compile time and generates an unrolled version, rather than a `for` loop.

Zip : $(in1 : [U]_n, in2 : [T]_n) \to [\{T, U\}]_n$

The *Zip* primitive takes two arrays of the same size n and combines them to an output array of tuples of size n.

$$Zip([1,2,3],[4,5,6]) \to [\{1,4\},\{2,5\},\{3,6\}] \tag{1.24}$$

Split : $(m : Int) \to in : [T]_n \to [[T]_m]_{\frac{n}{m}}$

The *Split* primitive takes an input array and splits the contained elements into $\frac{n}{m}$ smaller arrays of size m.

$$Split(m : 2) \ \$ \ [1,2,3,4] \to [[1,2],[3,4]] \tag{1.25}$$

Join : $(in : [[T]_n]_m) \to [T]_{n \times m}$

The *Join* primitive takes an array whose elements itself are arrays and flattens this structure. So the resulting array is nested one level less.

This can for example be used to revert the effects of the split primitive.

$$Join([[1,2],[3,4]]) \to [1,2,3,4] \tag{1.26}$$

Pad : $(l : Int, r : Int, h : (i : Int, len : Int) \to Int) \to in : [T]_n \to [T]_{l+n+r}$

The *Pad* primitive increases the size of an array by prepending l and appending r elements. Which elements are to be used here is determined with the function h, which returns an index inside the input array.

$$\begin{aligned} &Pad(l:2, r:1, h:clamp)\ \$\ in:[1,2,3,4] \to [1,1,1,2,3,4,4] \\ &\text{with } clamp(i:Int, len:Int) \to \begin{cases} 0 & \text{for } i < 0 \\ len & \text{for } i > n \end{cases} \end{aligned} \tag{1.27}$$

This is can be used to realize boundary conditions in a computation.

Slide : $(size : Int, step : Int) \to in : [T]_n \to [[T]_{size}]_{\frac{n-size+step}{step}}$

The *Slide* primitive slides a window of extent *size* across the input array with stride *step*. The output array contains each of the windows as array.

$$Slide(size:3, step:1)\ \$\ [1,2,3,4,5] \to [[1,2,3],[2,3,4],[3,4,5]] \tag{1.28}$$

This is very useful if a neighbourhood around a value is needed for computation. This is a common requirement in stencil computations.

Iterate : $(f : [T]_n \to [T]_n, m : Int) \to in : [T]_n \to [T]_n$

The *Iterate* primitive takes a function and applies it m times to the input.

$$\begin{aligned} &Iterate(f : timesTwo, m:3)\ \$\ [1,2,3] \to [8,16,24] \\ &\text{with } timesTwo : [T]_n \to Map(f : (x \to x * 2))\ \$\ [T]_n \end{aligned} \tag{1.29}$$

At : $(i : Cst) \to in : [T]_n \to T$

The *At* primitive returns the element at a specific index i of the input array.

$$At(i:2)\ \$\ [1,2,3,4,5] \to 3 \tag{1.30}$$

Get : $(i : Cst) \to in : T_1, T_2, ... \to T_i$

The *Get* primitive returns the member i of a tuple. It can also be written as $._i$.

$$\begin{aligned} Get(i:1)\ \$\ \{1,2\} &\to 2 \\ \{1,2\}._1 &\to 2 \end{aligned} \tag{1.31}$$

UserFun : $(s1 : ScalarT, s2 : ScalarT', ...) \rightarrow ScalarU$

The primitive *UserFun* can be used to define custom functions for using them in an expression. The functions are written in C and are embedded in the final OpenCL code. The arguments of this primitive are the name of the function to be generated, the input arguments, the body of the function and finally the argument and output types.

$$UserFun("square", "x", "return\ x * x;", Float) \rightarrow Float \tag{1.32}$$

Building on the work of Backus[3] the shown foundational Lift high-level algorithmic primitives can be combined to form more complex primitives to simplify the expression of complicated computations. Examples for this are two-dimensional versions of the *Pad* and *Slide* primitives. The definition of *Pad2D* is shown in Listing 1.1. It can be used to take care of boundary values of 2-dimensional arrays by adding elements on top, bottom, left and right of the structure.

Listing 1.1: simple 9-point stencil Lift expression

```
Pad2D = Map(Pad(left, right, boundary)) o
        Pad(top, bottom, boundary)
```

With the combination of primitives even complex 3D computations can be expressed, which we will show in Chapter 2. All high-level primitives can be used to form a Lift high-level expression. An example for this is given in Listing 1.2. Lift expressions are written in a functional style and have to be read from right to left.

Listing 1.2: simple 9-point stencil Lift expression

```
Join() o
Map(Reduce(+, 0)) o
Slide(9,1) o
Pad(1,1,clamp) $ input
```

1.2.4.2 *Rewriting*

We can express computations with simple algorithmic primitives which encode how we operate on the data. But this is not enough to produce high performance code for arbitrary platforms. We have to be able to apply optimizations. In Lift, optimizations are represented by rewrite rules. This way we can apply them automatically in a rewriting process. Every rewrite rule is proven to be semantics preserving, so it is never changed what will be computed, but the computation can be changed for example to access elements in a different order. An example for a rewrite rule can be seen in Listing 1.3.

Listing 1.3: split-join rule

```
Map(f) $ input → Join() o Map(Map(f)) o split(x) $ input
```

Instead of applying the function f to every element of the input, we split the input into chunks of size x and apply f to every element of one chunk so we apply the divide and conquer paradigm. The parameter x has to divide the size of the input array. This means we still have a very large amout of options for it. The best value for x depends heavily on the architecture the computation will be executed on. The Jacobi example of Listing 1.2 after the application of the *split join* rule with $x = 3$ is shown in Listing 1.4.

Listing 1.4: split-join rule applied to 9-point stencil

```
Join() o Join() o
Map(Map(Reduce(+, 0))) o
Split(3) o
Slide(9,1) o
Pad(1,1,clamp) $ input
```

Each application of a rewrite rule to an expression creates a new expression, which itself can be rewritten again. In addition to that the high-level expressions get transformed into low-level expressions, consisting of *low-level OpenCL specific primitives*, which expose OpenCL specific aspects. So after the rewriting process we have a magnitude of expressions that all compute the same result, but contain different optimizations for doing so.

Parallel Primitives

OpenCL provides a hierarchical organization of threads for distributing work, which is explained in detail in Section 1.2.3.2. We can specify how the work is being distributed with the following four low-level primitives which have exactly the same semantics as the high-level *Map* primitive.

MapSeq $: (f : T \to U, in : [T]_n) \to [U]_n$

The primitive *MapSeq* applies the function f to the elements of $[T]_n$ successively without any parallel computations. So the execution is done by a single thread.

MapGlb $: (dim : Int, f : T \to U, in : [T]_n) \to [U]_n$

The primitive *MapGlb* maps the application of f to the elements of $[T]_n$ to the global threads in dimension *dim*.

MapWrg : $(dim : Int, f : T \to U, in : [T]_n) \to [U]_n$

The primitive *MapWrg* maps each application of f to one work-group in dimension *dim*.

MapLcl : $(dim : Int, f : T \to U, in : [T]_n) \to [U]_n$

The primitive *MapLcl* can only be nested inside the primitive *MapWrg* and maps each application of f to a local thread inside the current work-group.

Adress Space Primitives

OpenCL exposes three different address spaces to the programmer, described in more detail in Section 1.2.3.3. To make use of them in Lift we provide the following primitives which influence where an element will be stored. They store the input element in the memory corresponding to the name.

toGlobal : $(in : T) \to T$
toLocal : $(in : T) \to T$
toPrivate : $(in : T) \to T$

At the end of a computation the result has to be in global memory so that it can be copied back to the host. This can be accomplished with *toGlobal*. The other two can be used to reduce memory transfers from the slower global memory if an element is accessed many times by storing it in the faster local or private memory. [25]
After this transformation we can map our low-level expressions one to one to OpenCL kernels. These kernels have to be evaluated to find out which optimization works best on our target architecure.

To sum it up, once a computation is expressed in the Lift IL it is possible to generate high performance OpenCL kernels for various target hardware architectures, effectively making it performance portable. This means when a new hardware architecture emerges on the market the computation does not need to be hand-tuned by an expert anymore.

2

EXPRESSING GMG OPERATIONS IN LIFT

To achieve high performance of a geometric multigrid algorithm it is feasible to parallelize and optimize the independent operations. This way every individual operation is being executed as fast as possible while still maintaining customizability. If another problem needs to be solved it is still easy to switch to another smoother or restriction method to cause convergence for it without losing efficiency. Hence for every GMG operation a Lift expression was designed. In order to be able to compare our results to a reference implementation of GMG, we do not use exactly the same computation for each of the operations as presented in Chapter 1.2.1. However, this is no confinement of the expressible GMG operations, because other variants of the operations can be expressed with similar Lift expressions.
All introduced operations compute on 3D structured data. However, it is much easier to analyze the computation in a 1D version. After finding the right composition of Lift patterns to express the simple version, it is not hard to extend this expression to 2D and 3D. Building on the 1D basis it is even possible to formulate expressions of arbitrary dimension for the operations.

2.1 RESTRICTION

The restrict operation which we will express in the Lift IR computes an output value as an interpolation between two neighbouring elements in every dimension. Starting with the 1D version, our input is an N-sized array and the output is an $\frac{N}{2}$-sized array. The structure can be seen in Figure 2.1.

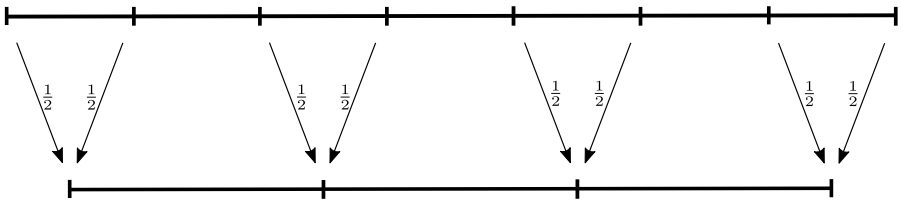

Figure 2.1: weighted 1-dimensional restriction

An output element in the one-dimensional restriction is thus computed from two neighbouring elements. We can use the *Slide* primitive to build all the necessary neighbourhoods. Each consists of two elements and there is no overlapping or left out elements between them. So choosing size = step = 2 for the *Slide* produces the desired neighbourhoods. All of the primitives have to be applied to every

neigbourhood, so we have to use a variation of the *Map* pattern. For this we use *MapGlb(o)*, to distribute the work between all work-items in the first dimension. The two neighbours in each neighbourhood have to be multiplied by $\frac{1}{2}$ and added together. We can change the order of operations to add them first using the *ReduceSeqUnroll* pattern and scale them afterwards, using a C-function defined with the *UserFun* primitive. The structure of this expression is shown in Listing 2.1 and its effects on the data in Figure 2.2.

Listing 2.1: 1-dimensional restriction Lift expression

```
1 divideBy2 = UserFun("divideBy2", "x", "{ return x*0.5; }",
2  Float, Float)
3 input =>
4         MapGlb(0)(
5           MapSeq(toGlobal(divideBy2)) o
6                 ReduceSeqUnroll(add, 0.0f)
7         ) o Slide(2, 2) $ input
```

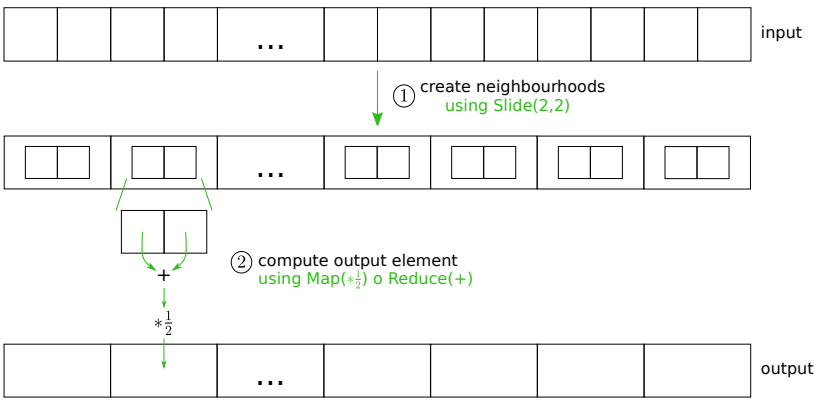

Figure 2.2: 1-dimensional Lift restriction operation

In order to build a one-dimensional expression for the restriction operation we have identified the following components needed in this computation. For implementing each of these components we have used a Lift primitive:

1 building a neighbourhood of input elements for each output element - *Slide*

2 the following computation has to be executed for every neighbourhood - *MapGlb*

3 summing up all items in a neighbourhood - *ReduceSeqUnroll*

4 dividing by the number of elements in a neighbourhood - *UserFun*

The concept of the two-dimensional version of restriction is shown in 2.3. It uses exactly the same components as the one-dimensional

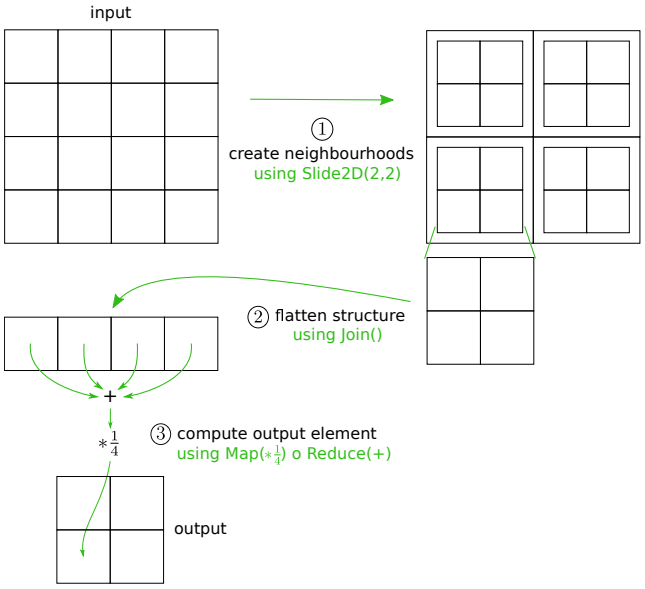

Figure 2.3: 2-dimensional Lift restriction operation

version, but we have to modify the primitives because of the change in the data structure. Instead of normal *Slide* we use *Slide2D*, which builds two-dimensional neighbourhoods. Again, we map all of the following functions to each neigbourhood. But the data is organized in two-dimensions, so using only one *MapGlb* would apply all of the following to each row. Hence we use two nested *MapGlb*s. Each neighbourhood is two-dimensional as consequence of *Slide2D*, for example it looks like this: $[[1,2],[3,4]]$, so we cannot apply *ReduceSeqUnroll* directly. Instead we use *Join* to flatten the neigbourhood and apply *ReduceSeqUnroll* afterwards. At this point we have reduced the neighbourhood to one element exactly like in the one-dimensional case, so the application of *UserFun* stays the same. The resulting expression is shown in Listing 2.2.

Listing 2.2: 2-dimensional restriction Lift expression

```
1  divideBy4 = UserFun("divideBy4", "x", "{ return x*0.25; }",
2   Float, Float)
3  input =>
4        MapGlb(1)(
5          MapGlb(0)(
6            MapSeq(toGlobal(divideBy4)) o
7            ReduceSeqUnroll(add, 0.0f) o
8            Join()
9          )
10       ) o Slide2D(2, 2) $ input
```

In the transformation to a three-dimensional expression we do not have any additional obstacles. The only difference to the two-dimensional version is that we use *Slide3D* for building a three-dimensional neigh-

bourhood of 8 values, use *MapGlb* three times for mapping to each neighbourhood and apply *Join* two times before the *ReduceSeqUnroll* to flatten the three-dimensional neighbourhood. The resulting expression is shown in Listing 2.3.

Listing 2.3: 3-dimensional restriction Lift expression

```
1  divideBy8 = UserFun("divideBy8", "x", "{ return x*0.125; }",
2   Float, Float)
3  input =>
4      MapGlb(2)(
5        MapGlb(1)(
6          MapGlb(0)(
7            MapSeq(toGlobal(divideBy8)) o
8            ReduceSeqUnroll(add, 0.0f) o
9            Join() o Join()
10         )
11       )
12     ) o Slide3D(2, 2) $ input
```

It is possible to use the same strategies to produce expressions which perform restriction in arbitrary dimensions.

2.2 PROLONGATION

Expressing the prolongation in Lift is very interesting, because most computations which have been designed so far preserve or shrink the input size. Here the output size is twice as large as the input size. Each input element contributes to four output elements, where two of them are influenced very strongly and get a minor fraction from neigbouring elements and the other two elements get a smaller portion of this element and a major fraction from a neighbouring element. The structure of this is shown in Figure 2.4.

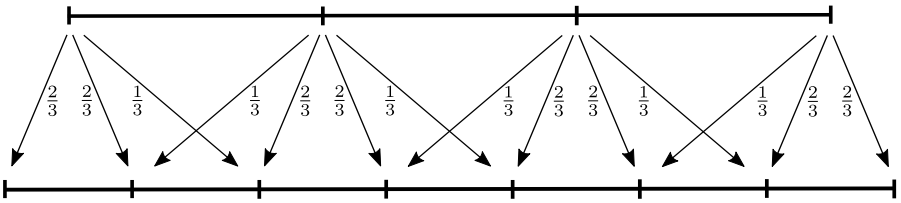

Figure 2.4: weighted prolongation

The expression for this operation on the other hand has to produce double the input size in every dimension as output size. For each two neighboured input elements two output elements are computed, the left one with more influence of the left input element and the right one with more influence of the right input element.
The concept of the designed Lift expression is shown in Figure 2.5. We duplicate the elements on the edge using *Pad* and create the

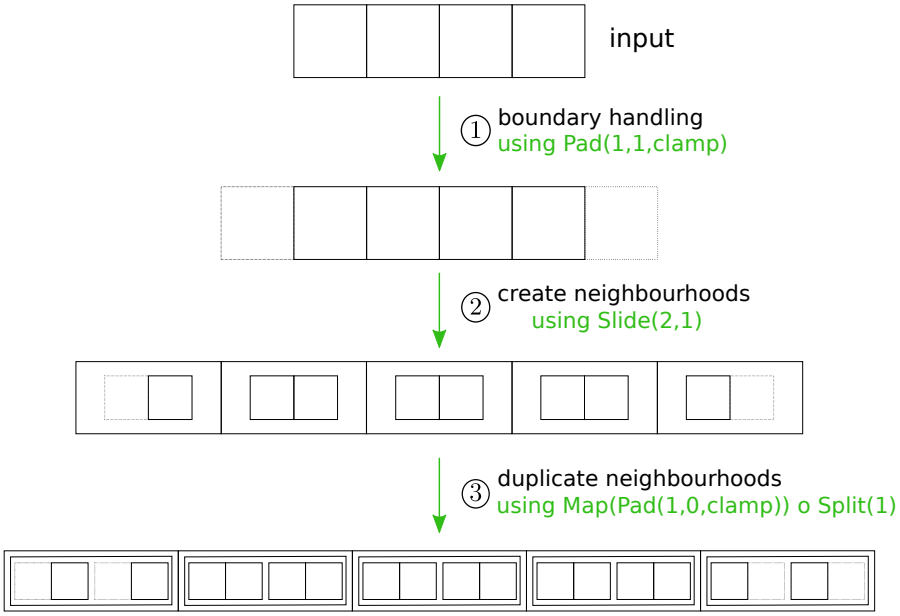

Figure 2.5: LIFT prolongation neighbourhood duplication

needed neigbourhoods with the *Slide* primitive. For duplicating them we first apply the *Split* primitive with parameter 1 to wrap each neighbourhood in an array. An example for the change of the data structure is shown in Equation 2.1.

$$[[1,2],[3,4],[5,6]] \overset{Split(1)}{\rightarrow} [[[1,2]],[[3,4]],[[5,6]]] \tag{2.1}$$

This allows us to use the *Pad* primitive with parameters 1 and 0 nested inside a *Map* primitive to duplicate each of the created arrays which contain a neighbourhood. This is a very unconventional way of using *Pad*. It is intended to be used for padding border values, not for being applied to a large number of elements. Thus it is not optimized for this usecase. However, it is currently the only way to express this enlargement of data in LIFT. It will be evaluated in Chapter 4 whether this has bad effects on the performance.
The number of neighbourhoods is sufficient, however half of them are duplicates. We do not want to compute duplicate output elements, so we have to scale each value in the neighbourhood with a respective weight. The procedure of the LIFT expression is shown in Figure 2.6. Using the weights $[\frac{1}{3}, \frac{2}{3}]$ we get the same scaling as pictured in Figure 2.4 and compute the correct output elements. But applying the weights is not straightforward. Looking at one pair of neighbourhoods the elements should not get identical weights, otherwise the computed output of them would be the same. The weights applied to the second element have to be a mirrored version of the weights applied to the first element. We can achieve this by applying the *Pad* primitive with parameters 2, 0 and the boundary function *mirror*. To

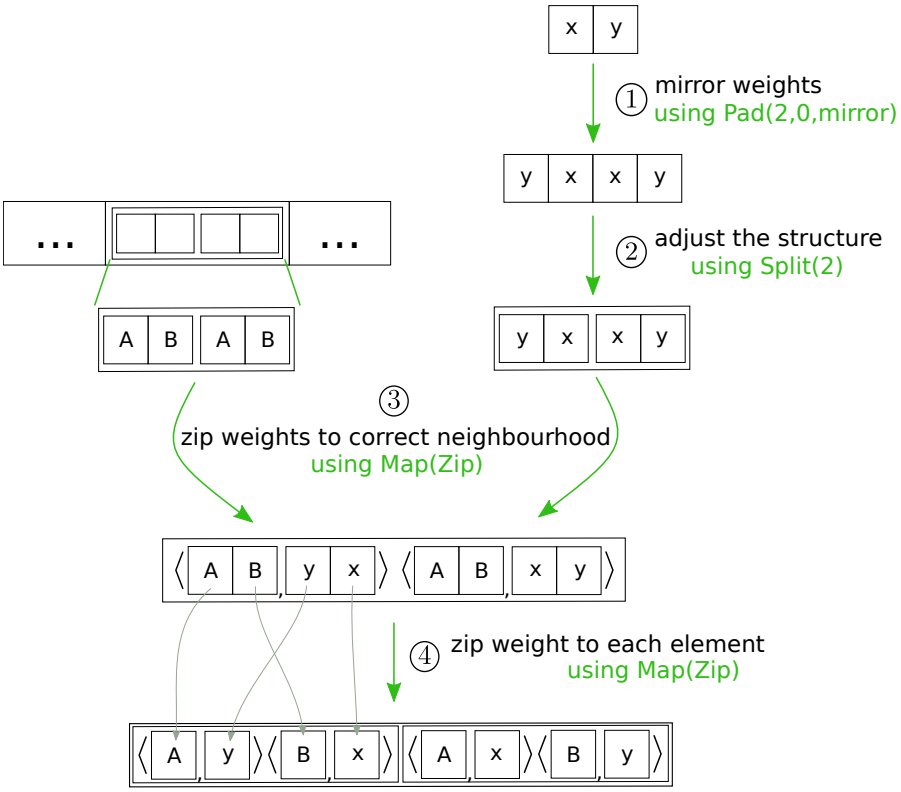

Figure 2.6: LIFT prolongation weight zipping

be able to apply *Zip* to two arrays they have to have the same structure, so we apply *Split(2)* to the weights to adjust the structure. At this point the weights look like this: $[[\frac{1}{3}\ \frac{2}{3}][\frac{1}{3}\ \frac{2}{3}]]$. We can use *Map(Zip)* to build a tuple for each neighbourhood and the corresponding weights. In the next step we use *Zip* one more time to attach each element of the neighbourhood to the respective weight.
After this we do not have to keep track of which neigbourhoods were duplicates, because the weights are already applied. So we apply the *Join* primitive to flatten the structure and have all neighbourhoods in one array. In Figure 2.4 we see that the input elements at the edge only contribute to 3 output elements. This means with our present strategy we compute too many output elements. This can be seen in Figure 2.5, where we have 4 input elements at the beginning and 10 neighbourhoods for computing output elements at the end. So at each edge we have one element too many. Currently we do not have a primitive in LIFT for deleting values in the input. We have the *Pad(l,r,b)* primitive, which does the opposite. It prepends l values and appends r values using a reindexing function b to determine which

values are used. For deleting the unnecessary values we introduce the new primitive *Drop*, built on the foundation of *Pad*.

$$\begin{aligned} &\textbf{Drop} : (l : Int, r : Int, in : [T]_n) \to [T]_{n-l-r} \\ &\text{Drop}(l, r) = \text{Pad}(-l, -r, \text{id}) \end{aligned} \tag{2.2}$$

$$\begin{aligned} &\text{with id} : (i : Int, len : Int) \to Int \\ &\quad \text{id}(i, len) = i \end{aligned}$$

Drop is built on the foundation of the *Pad* primitive to decrease the input $[T]_n$ with negative l and r values. The new indexing function *id* assigns all elements with their correct new index, ensuring that all values stay at the right position. The application of this pattern on a simple array can be seen in the following Example 2.3. The effect on the structure of the data in the prolongation is shown in Figure 2.7.

$$Drop(l : 2, r : 1, in : [1, 2, 3, 4, 5, 6]) \to [3, 4, 5] \tag{2.3}$$

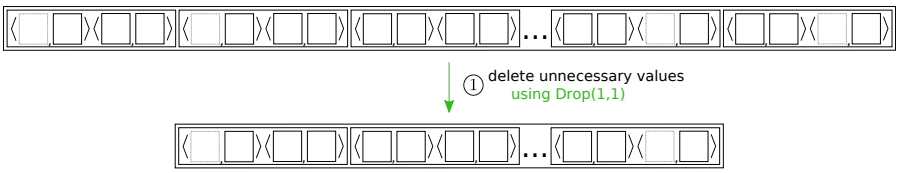

Figure 2.7: Lift prolongation deleting unnecessary values

The resulting data structure contains one neighbourhood for every output element and each element is in a tuple with the respective necessary weight. The only thing left to express is the computation itself. The concept of the following Lift expression is shown in Figure 2.8.

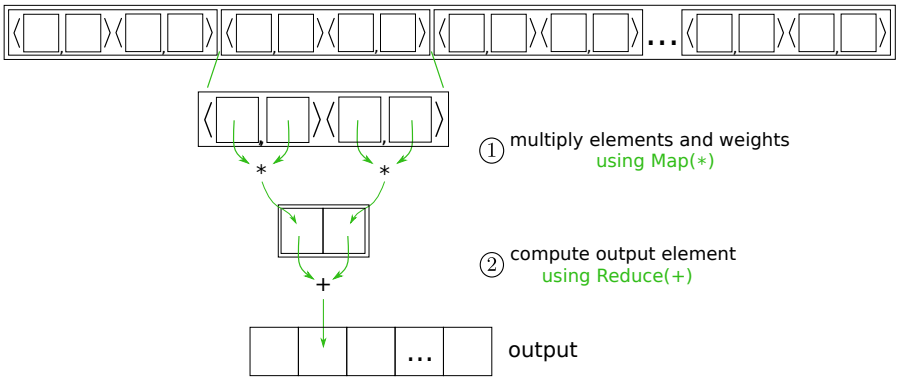

Figure 2.8: Lift prolongation computation

We start by multiplying each element with its respective weight using a custom function defined with the pattern *UserFun*. Then, we add the

two elements in each neighbourhood using *ReduceSeqUnroll* and store the result in global memory using *toGlobal*. All of these primitives are nested inside the primitive *MapGlb* to distribute the work between several threads.
We have identified 5 major components which can be used together to express one-dimensional prolongation as pictured in Figure 2.4:

1 Creating neighbourhoods using *Pad* and *Slide*

2 Duplicating all neighbourhoods using *Map(Pad)* and *Split*

3 Preparing and zipping of weights to the elements using *Pad*, *Split* and *Map(Zip)*

4 Flattening the structure using *Join* and omitting border elements using the new *Drop*

5 Computing the output values using *UserFun*, *ReduceSeqUnroll*, *toGlobal* and *MapGlb*

Listing 2.4: 1-dimensional prolongation LIFT expression

```
(input, weights) =>
      //5
      MapGlb(0)(
        toGlobal(MapSeq(id)) o
        ReduceSeqUnroll(toGlobal(add), 0.0f) o
        toGlobal(MapSeq(tuple => mult(tuple._0, tuple._1)))
      ) o

      //4
      Drop(1, 1) o
      Join() o

      //3
      Map(Map(tuple =>
        Zip(tuple._0, tuple._1)
      )) o
      Map(nbh =>
        Zip(nbh,
          Split(2) o Pad(2,0,Mirror) $ weights)
      ) o

      //2
      Map(Pad(1,0,clamp)) o Split(1) o

      //1
      Slide(2,1) o
      Pad(1,1,Pad.Boundary.Clamp)
    $ input
```

We see that the new *Drop* primitive is used before the final computation, so we prevent unnecessary computations of elements which are

not needed in the result.
The 2- and 3-dimensional expressions use the same components as shown here with a different implementation to account for the change of structure.

1 To build neighbourhoods in the final 3-dimensional expression we simply use the 3D primitives *Pad3D* and *Slide3D* with identical parameters to the 1D version.

2 For duplicating each neighbourhood in every dimension in the 3-dimensional expression we have to apply *Pad3D* to each neighbourhood. *Pad3D* operates on 3D structured data so we have to prepare the data such that each neighbourhood is nested inside a 3-dimensional array, which contains only this neighbourhood. We accomplish this similar to the 1-dimensional expression with the *Split* primitive, which we apply 3 times to each neighbourhood. Afterwards we can apply *Pad3D* to each prepared structure. After this each neighbourhood is present in our data 8 times.

3 The third component which contains the zipping of the respective weights to each element can also be realized in 3 dimensions. The weights are 3-dimensional, so the mirroring described above has to be realized with *Pad3D* as well. In the 1-dimensional expression we use *Split* to align the structure of the elements in the weights array to the neighbourhoods. There is no 3D version of the *Split* primitive in Lift yet and instead of building it, it is feasible to use *Slide3D* to accomplish this. The weights could then be read as a 2x2x2 matrix, containing elements which are different 2x2x2 weights. The 8 different groups of weights are then zipped to each group of 8 identical neighbourhoods using *Zip3D*. Now we have tuples, whose elements are a neighbourhood and the corresponding weight. In order to build tuples which contain an element of a neighbourhood and its corresponding weight we use *Zip* just like in the 1-dimensional expression.

4 In order to change our data structure from a 3D matrix, containing 3D matrices, containing neighbourhoods to a 3D matrix, containing neighbourhoods, we have to use the *Join* primitive to flatten some of the nestings. For that we have to associate the dimensions of the outer 3D matrix with the same dimensions of the inner 3D matrix. We accomplish this with the *Transpose* primitive, which swaps two dimensions of a data structure e.g. to transpose a matrix. We swap the dimensions of our structure such that related dimensions are placed next to each other, in other words nested into each other. This allows us to flatten them with *Join* and achieve the desired structure. This structure

enables the use of the *Drop3D* primitive for removing all boundary values, which are redundant and not needed in the result.

5 The only difference in the final computation is the need to apply *MapGlb* 3 times to distribute all neighbourhoods in the 3D matrix to work-items.

The 3-dimensional prolongation expression resulting from these changes is shown in Listing 2.5.

Listing 2.5: 3-dimensional prolongation Lift expression

```
(matrix,weights) =>
  //5
  MapGlb(0)(
    MapGlb(1)(
      MapGlb(2)(
          toGlobal(MapSeq(id)) o
          ReduceSeqUnroll(add, 0.0f) o
          MapSeq(tuple=> mult(tuple._0, tuple._1)
  )))) o

  //4
  Drop3D(1,1) o

  Map(Map(Join())) o
  Map(Join()) o
  Join() o

  Map(Map(Map(Transpose()))) o
  Map(Transpose()) o
  Map(Map(Transpose())) o

  //3
  Map(Map(Map(Map(Map(Map(tuple =>
      Zip(
          Join() o Join() $ tuple._0,
          Join() o Join() $ tuple._1
      )
    ))))) o
  Map(Map(Map(nbh =>
    Zip3D(
      nbh,
      Slide3D(2,2) o
        Pad3D(2,0,Pad.Boundary.Mirror)
        $ weights
    )))) o


  //2
    Map(Map(Map(Pad3D(1,0,Pad.Boundary.Clamp)))) o
      Map(Map(Split(1) o Split(1) o Split(1))) o

  //1
  Slide3D(2,1) o
  Pad3D(1,1,1,Pad.Boundary.Clamp)
$ matrix
```

2.3 RESIDUAL

For the residual computation we use sequential from Basu et al.[4], which is similar to the approach of the HPGMG benchmark, as reference. We do this, because the original intention was to compare our results to the results of the highly optimized HPGMG benchmark. However, due to compatibility constraints we are not able to present results for this in this work. As can be seen in the sequential code in Listing 2.6 the computation operates on 6 input arrays in contrast to the previous operations, which all used 1 input array.

Listing 2.6: sequential residual operation code [4]

```
for(k=0;k<K;k++)
  for(j=0;j<J;j++)
    for(i=0;i<I;i++)
      res[k][j][i] = rhs[k][j][i]
        - a * alpha[k][j][i] * phi[k][j][i]
        - b*h2inv*(
          beta_i[k][j][i+1]*( phi[k][j][i+1]-phi[k][j][i] )
          -beta_i[k][j][i] *( phi[k][j][i] -phi[k][j][i-1])
          +beta_j[k][j+1][i]*( phi[k][j+1][i]-phi[k][j][i] )
          -beta_j[k][j][i] *( phi[k][j][i] -phi[k][j-1][i])
          +beta_k[k+1][j][i]*( phi[k+1][j][i]-phi[k][j][i] )
          -beta_k[k][j][i] *( phi[k][j][i] -phi[k-1][j][i]));
```

So for computing one output value, we need data from multiple input arrays. To express this in LIFT we start by formulating a 1-dimensional expression, taking two input arrays. First we use the *Pad* and *Slide* primitives on each input array to build the desired neighbourhoods. Afterwards we build tuples of the corresponding neighbourhoods from the different inputs with the *Zip* primitive. Now all the data needed to compute one output element is present in each tuple. We see in lines 4-12 of the sequential residual code that the computation is rather complex and cannot be expressed with the *Reduction* primitive as in the other operations. We use the *UserFun* primitive to define the computation in C. To privide the input values to the function we use a combination of the primitives *get* and *at*. With *get* we specify from which input array the value comes, so the first or the second element of the tuple in this case. With *at* we specify which of the 3 elements in the neighbourhood is the desired value. The structure of this is shown in Figure 2.9. In Listing 2.7 a LIFT expression is shown, which works as previously described and builds the sum of all 3 elements in both neighbourhoods in the C-defined function.

Listing 2.7: 1-dimensional LIFT expression using multiple input arrays

```
    def f = UserFun("f", Array("a1", "a2", "a3", "b1", "b2", "
        b3"),
      "return a1 + a2 + a3 + b1 + b2 + b3;",
```

```
    Seq(Float, Float, Float, Float, Float, Float), Float)

  (A, B) => {

    MapGlb(0)(tuple => {

      val a1 = tuple._0.at(0)
      val a2 = tuple._0.at(1)
      val a3 = tuple._0.at(2)

      val b1 = tuple._1.at(0)
      val b2 = tuple._1.at(1)
      val b3 = tuple._1.at(2)

      toGlobal(x =>
        f(a1, a2, a3, b1, b2, b3)) $ A

    }) $ Zip(
    Slide(3,1) o Pad(1,1,Pad.Boundary.Clamp) $ A,
    Slide(3,1) o Pad(1,1,Pad.Boundary.Clamp) $ B
  )
})
```

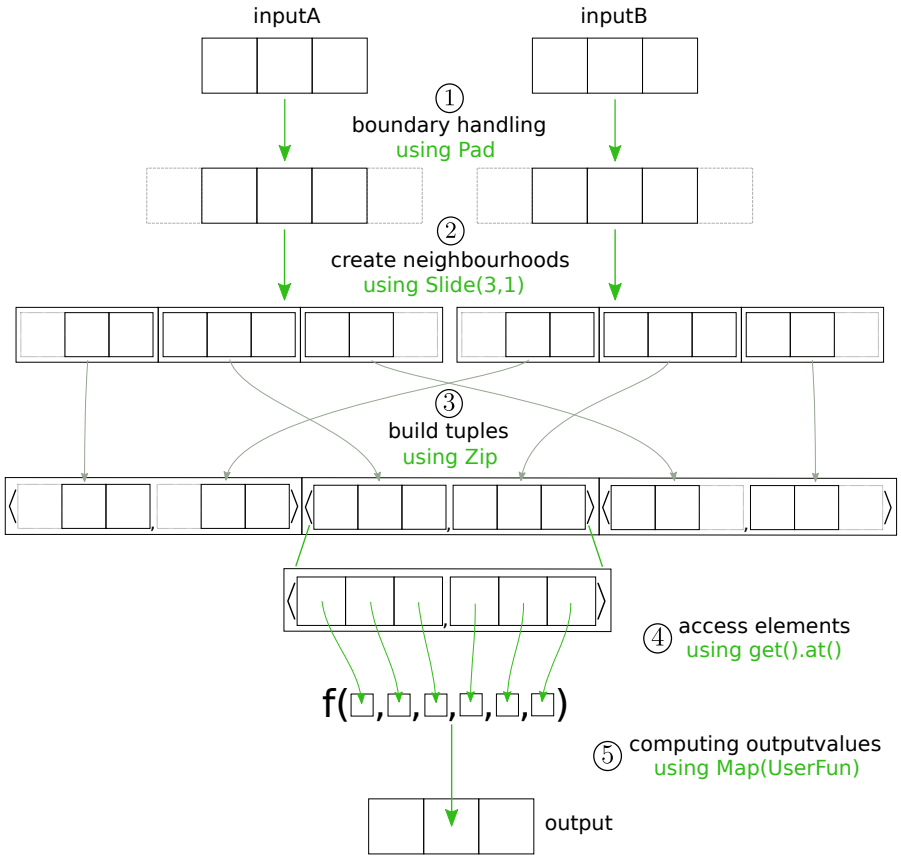

Figure 2.9: Lift multiple inputs example

Scaling this expression to a larger number of input arrays is as easy as adding them to the *Zip* primitive and gathering its values with *get* and *at* as shown before. Scaling this expression to 2, 3 or N dimensional inputs and output is also possible starting by replacing *Pad*, *Slide* and *Zip* with their corresponding 2D, 3D and ND patterns. In the example of a 3-dimensional expression, we build a 3D-matrix of tuples, containing neighbourhoods as elements which are smaller 3D-matrices this way. Providing the necessary values to the C-defined function works with the same primitives as before. First we use *get* with the index of the input array that we want to access to specify the correct element of each tuple. Then we use *at* to get the desired element, but the number of times we have to use it depends on the input dimensions. In order to access the center element of a 3-dimensional neighbourhood created with *Slide3D(3,1)* we use *at(1).at(1).at(1)*. Moreover the computation is nested inside 3 *MapGlb* primitives to distribute the computations on the tuples between the work-items in all 3 dimensions. In Listing 2.8 the final 3-dimensional residual LIFT expression is shown in a slightly shortened version for brevity, which takes 6 input arrays and operates on them as described previously.

Listing 2.8: 3-dimensional residual LIFT expression

```
1   // [X-1][][] = F(ront) [X+1][][] = B(ack)
2   // [][X-1][] = N(orth) [][X+1][] = S(outh)
3   // [][][X-1] = W(est)  [][][X+1] = E(ast)
4   def f = UserFun([...],
5     "float Ax = a * alphaC * inputC - b * h2inv * (beta_iC *
          inputF - inputC + beta_jC * inputN - inputC +
        beta_kC * inputW - inputC + beta_iB * inputB -
        inputC + beta_jS * inputS - inputC + beta_kE *
        inputE - inputC);
6      return rhsC - Ax;",
7     Seq(Float, [...] , Float), Float)
8
9   (a, alpha, b, beta_i, beta_j, beta_k, h2inv, input, rhs)
      => {
10      MapGlb(2)(MapGlb(1)(MapGlb(0)(tuple => {
11
12        val alphaC = tuple._0.at(1).at(1).at(1)
13        [...]
14        val rhsC = tuple._5.at(1).at(1).at(1)
15
16        toGlobal(x =>
17          f(x, alphaC, b, h2inv, beta_iC, beta_jC, beta_kC,
              beta_iB,
18            beta_jS, beta_kE,
19            inputC, inputE, inputW, inputS, inputN, inputB,
                inputF, rhsC)))
20
21    })) $ Zip3D(
22      Slide3D(3,1) $ alpha,
```

```
23          [...]
24          Slide3D(3,1) $ rhs
25        )
26    })})
```

The used C-defined function is very similar to the code from Basu et al., but the used approach makes it possible to apply optimizations encoded in LIFT rewrite rules. So whereas the computation stays the same, e.g. data access patterns can be varied to gain better performance on specific devices.

2.4 SMOOTHING

Similar to the residual operation, the reference for the smoothing operation is the one presented by Basu et al. They use the Gauss-Seidel red black method for smoothing the error. The Jacobi relaxation method, described in Chapter 1.2.1 is easy to parallelize, because only data from the previous time step is used, so all calculations are independent from each other. In the basic Gauss Seidel method, on the other hand, previously computed elements from the same time step are used in the computation. This means there are dependencies among computations of elements in the same time step. In consequence, the computation of single elements cannot simply be distributed between independent work-items. Following the approach of Adams and Ortega [1] we can assign different colors to the elements so as to eliminate dependencies between elements of the same color.

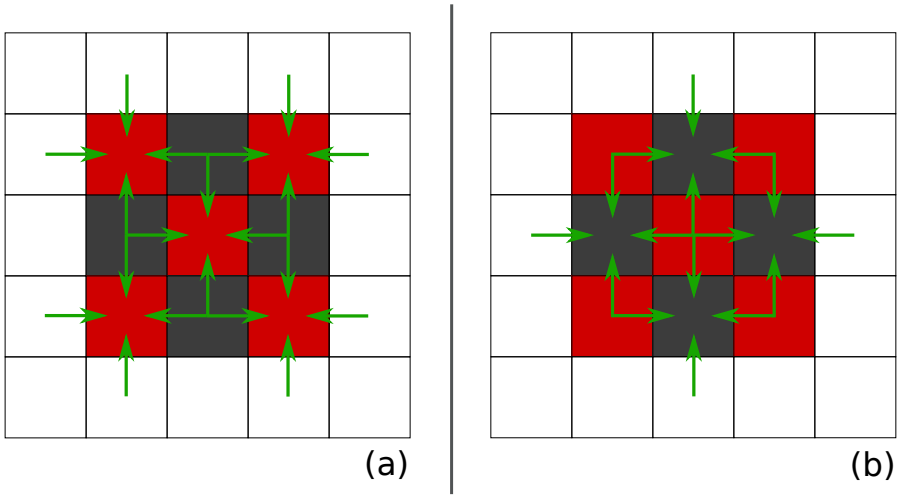

Figure 2.10: Gauss Seidel Red Black dependencies (inspired by [1])

In this case each element depends only on the directly adjacent neighbours, thus two colors suffice. The result of this is a checkerboard pattern, usually with the colors red and black. One iteration is now split

into two steps. In each step only elements of one of the two colors are being updated. The dependencies of the red and black elements are shown in Figure 2.10 (a) and (b). This way the computation of elements in each step is independent, making simple parallelization possible again. The technique is being used in the following sequential code for the smoothing operation by Basu et al. in Listing 2.9.

Listing 2.9: sequential smoothing operation code [4]

```
1
2  /* Laplacian(phi) = b div beta grad phi */
3  for (k=0;j<N;k++)
4    for (j=0;j<N;j++)
5      for (i=0;i<N;i++)
6        /* statement S0 */
7          temp[k][j][i] = b*h2inv*(
8            beta_i[k][j][i+1] * (phi[k][j][i+1]-phi[k][j][i])
9            -beta_i[k][j][i] * (phi[k][j][i]-phi[k][j][i-1])
10           +beta_j[k][j+1][i] * (phi[k][j+1][i]-phi[k][j][i])
11           -beta_j[k][j][i] * (phi[k][j][i]-phi[k][j-1][i])
12           +beta_k[k+1][j][i] * (phi[k+1][j][i]-phi[k][j][i])
13           -beta_k[k][j][i] * (phi[k][j][i]-phi[k-1][j][i])
14         );
15
16 /* Helmholtz(phi) = (a alpha I - laplacian)*phi */
17 for (k=0;j<N;k++)
18   for (j=0;j<N;j++)
19     for (i=0;i<N;i++)
20       /* statement S1 */
21       temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i]
22         - temp[k][j][i];
23
24 /* GSRB relaxation: phi = phi - lambda(helmholtz-rhs) */
25 /* Jacobi smoother is similar but without if-condition */
26 /* and reads, writes separate grids */
27 for (k=0;j<N;k++)
28   for (j=0;j<N;j++)
29     for(i=0;i<N;i++){
30        if((i+j+k+color)%2==0)
31          /* color is 0 for Red pass, 1 for black */
32          /* statement S2 */
33          phi[k][j][i] = phi[k][j][i] - lambda[k][j][i] *
34            (temp[k][j][i]-rhs[k][j][i]);
35     }
```

In this smoothing operation we have 7 3-dimensional input arrays, so it is reasonable to formulate the Lift expression similar to the residual Lift expression according to Figure 2.9. The problem is that we cannot compute the color for an element on the fly similar to the shown code. The C-defined function is executed with values from each of the neighbourhoods and has no context where in the data

structure it operates at a given time due to the functional style of Lift. In consequence we have to prepopulate an array with the information if an element should be updated in this iteration and provide it to the function. This means the Lift expression has to use one input array more and copy more data to the device as well. Whether this makes a difference in performance will be evaluated in the following Chapter 4. A shortened version of the expression is shown in Listing 2.10.

Listing 2.10: 3-dimensional smoothing Lift expression

```
1   def f = UserFun([...],
2     "if (!color) return inputC;
3      float helmholtz = a * alphaC * inputC - b * h2inv * (
           beta_iE * (inputE - inputC) - beta_iC * (inputC -
           inputW) + beta_jS * (inputS - inputC) - beta_jC * (
           inputC - inputN) + beta_kB * (inputB - inputC) -
           beta_kC * (inputC - inputF));
4      return inputC - lambdaC * (helmholtz - rhsC);",
5     Seq(Float, [...], Float), Float)
6
7   (a, alpha, b, beta_i, beta_j, beta_k, h2inv, input, lambda
        , rhs) => {
8     MapGlb(2)(MapGlb(1)(MapGlb(0)(tuple => {
9
10      val alphaC = tuple._0.at(1).at(1).at(1)
11      [...]
12      val rhsC = tuple._6.at(1).at(1).at(1)
13
14      toGlobal(x =>
15        f(x, alphaC, b, h2inv, beta_iE, beta_iC, beta_jS,
              beta_jC, beta_kB, beta_kC,
16          inputC, inputE, inputW, inputS, inputN, inputB,
                inputF, lambdaC, rhsC, color))
17
18    }))) $ Zip3D(
19      Slide3D(3,1) $ alpha,
20      [...]
21      Slide3D(3,1) $ rhs
22    )
23  })
```

The C-defined function here is slightly different from the one used by Basu et al., because the function in the Lift expression always has to return a value. The parameter *color* determines whether an element has to be updated in this iteration, so if its value is *false* the calculation is skipped entirely.

Expressing the GMG operations in Lift posed a number of interesting challenges to be tackled. Almost all of them can be expressed with well known combinations of existing Lift primitives, only the structure of the prolongation is harder to express. In order to accomplish

this we use the *Pad* primitive in an uncommon way. It was intended for prepending and appending boundary values to an input array. So usually we add the same number of elements to each side of the input. In the prolongation expression we use it with the parameters 0 and 1 to duplicate each of the neighbourhoods. In addition to that we introduced the new *Drop* primitive for deleting a number of values from both sides of an array to remove unnecessary neighbourhoods before the final computation. We will evaluate whether the designed expressions perform better in comparison to a reference implementation.

3

OPTIMIZING MULTIGRID OPERATIONS USING LOW-LEVEL PRIMITIVES

In this chapter, we analyze how to optimize GMG operations using different well-known optimizations. If possible we formalize them using the functional approach of Lift to apply them systematically within the rewriting process, rather than ad-hoc. If expressing them is not possible, we explore what capabilties Lift is currently missing and how they can be added.

3.1 SPACIAL TILING

In order to exploit the fast and small on-chip memory of GPUs we divide the input into several *tiles*. The tiles are being distributed to the work-groups and copied into local memory. The local threads of the work-groups can take advantage of the fast memory and execute the computations with reduced memory latency. This significantly reduces the number of global memory accesses in the computation. The classical tiling approach, which is commonly used in matrix multiplication, requires disjoint tiles. This means we cannot use it for operations, in which a non-overlapping partition is impossible.

3.1.1 *Applying tiling the Restriction Operation*

The restriction operation takes N input elements and produces $\frac{N}{2}$ output elements. In the process, each element contributes a fraction to one output element as pictured in Figure 2.1. In addition to that, we see that the Lift expression designed in Chapter 2.1 uses disjoint neighbourhoods for computation. In consequence of this we can apply tiling to the restriction operation. We partition the input into tiles, containing an even number of elements and compute the output elements for each tile. An example for this partitioning with a tile-size of six elements is shown in Figure 3.1.
The used approach is analogous to the application of tiling to matrix multiplication in Lift as shown in [22]. In Figure 3.2 the application of the Lift expression for the tiled restriction operation in one dimension is shown on a simple example.
First, we create tiles of size six by using the *Split* primitive. With six elements in each tile we have to compute three output elements per tile. In order to achieve this, we build neighbourhoods by applying *Slide* to each tile using *Map*. Then we distribute all of the tiles to work-groups using *MapWrg* and the contained neighbourhoods to local threads of

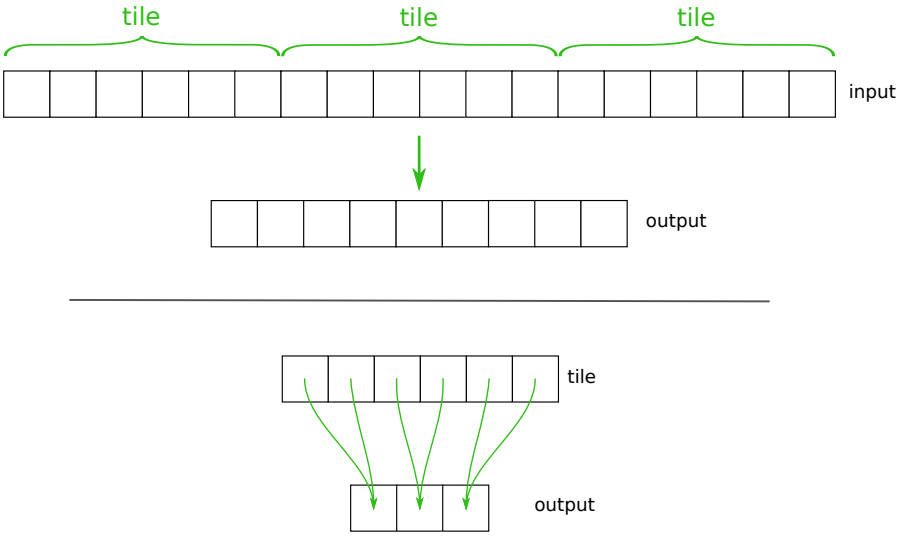

Figure 3.1: tiling of the 1-dimensional restriction operation

the work-group for computation using *MapLcl*. In comparison to the 1-dimensional restriction Lift expression the only difference is the construction of the tiles and the strategy for distributing the work to threads.
All other components of the Lift expression remain the same, as seen in the final computation of the output elements. Not pictured in Figure 3.2 is the required *Join* primitive after the computation, which flattens the tiles back to a simple array of elements. The Lift expression implementing tiling in the 1-dimensional restriction operation is shown in Listing 3.1.

Listing 3.1: tiled 1-dimensional restriction Lift expression

```
1  divideBy2 = UserFun("divideBy2", "x", "{ return x*0.5; }",
2   Float, Float)
3
4  input =>
5      Join() o
6      MapWrg(
7        MapLcl(
8          MapSeq(toGlobal(divideBy2)) o
9          Reduce(add, 0.0f)
10     )) o
11     Map(Slide(2, 2)) o
12     Split(6) $ input
```

As mentioned above, the tiling optimization has already been used in the context of Lift matrix multiplication. In this process it was also expressed as semantics preserving rewrite-rule to be applied automatically within the Lift rewriting. In the three operations besides restriction we use a neighbourhood of elements for computing one output element as well, but the used neighbourhoods are not disjoint.

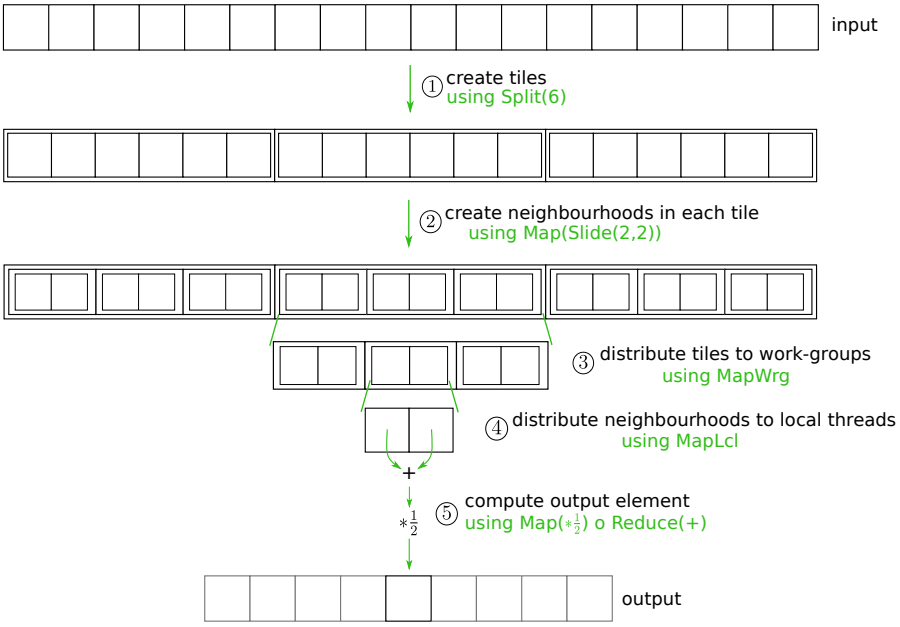

Figure 3.2: LIFT tiled 1-dimensional restriction operation

Neighbouring output elements use overlapping neighbourhoods for computations, so they share some input elements. This means we cannot use the classical tiling approach presented above, where the input is simply split into disjoint tiles.

3.1.2 *Overlapped Tiling*

In order to take advantage of the local memory of the GPU in the other operations as well, we have to adjust the strategy. In consequence of the neighbourhoods, which share elements, the tiles have to be designed to overlap their neighbours. So we have elements which are included in multiple tiles, which are often called ghost or halo elements. The overlapping tiles for a simple 3-point stencil are pictured in Figure 3.3.
This well-known optimization is called *overlapped tiling*[10]. The only difference to normal tiling is the design of the tiles. The local threads of a work-group should compute multiple (e.g. three) elements from a tile. This means each tile has to contain five input elements. We cannot simply partition the input as before, using *Split(5)*, because the tiles have to overlap for correct computation, as shown in 3.3. For building the tiles we can reuse the same primitive we use for building the neighbourhoods. The *Slide* primitive allows us to build the tiles with a particular size and a defined distance from the beginning of one tile to the beginning of the next tile. Using the parameters $size = 5$ and $step = 3$ for this primitive, we create the required tiles, containing five elements and sharing two elements with each neighbour. After creating the tiles this way, the expression works exactly

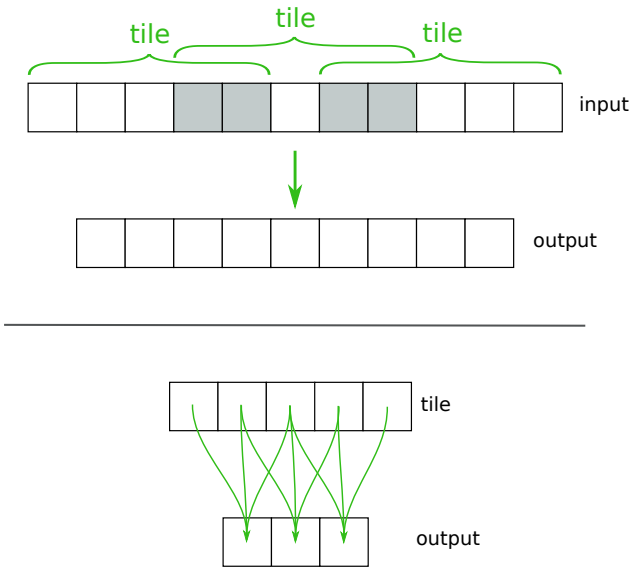

Figure 3.3: 1-dimensional overlapped tiling example

like normal tiling, as presented above. The complete Lift expression for 1-dimensional overlapped tiling of a 3-point stencil can be seen in 3.2. Overlapped tiling has already been expressed and evaluated in Lift for different computations[11]. In addition to that, it is expressed as semantics preserving rewrite rule as well.

Listing 3.2: tiled 1-dimensional restriction Lift expression

```
input =>
    Join() o
    MapWrg(
      MapLcl(
        MapSeq(toGlobal(id)) o
        Reduce(add, 0.0f)
    )) o
    Map(Slide(3, 1)) o
    Slide(5,3) o Pad(1,1,clamp) $ input
```

We have shown the tiling optimizations with an example tile-size of six and five elements. This parameter can easily be varied by changing the parameters of the *Split* or *Slide* primitive. In order to produce a correct kernel we have to consider the constraint, that all elements have to be distributed to tiles and all tiles contain the same number of elements and the first element of the first tile must be the first element of the input whereas the last element of the last tile must also be the last element of the input

The performance of the OpenCL kernel, generated from the Lift expression, depends heavily on the tile-size, so choosing it according to the target architecture is very important. This optimization works in higher dimensions as well. Designing Lift expressions employing

tiling in higher dimensions works similar to scaling of the GMG expressions, shown in Chapter 2.

3.2 TIME-SPACE TILING

As explained in Chapter 1.2.1, GMG methods contain a smoother which is applied iteratively multiple times in a row on each coarsening level. Similar to many other stencil applications the computation is not the most time-consuming part. Most time is used for transferring data from the host to the device and for memory accesses on the device. In terms of the smoothing example, the data movement is shown on the left part of Figure 3.4.

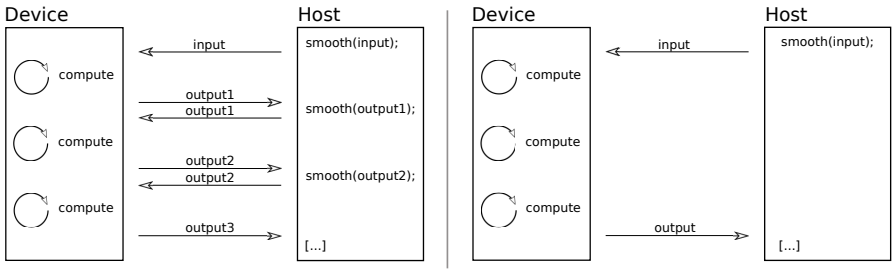

Figure 3.4: smoothing operation data movement

The following happens multiple times in a row: We transfer input data to the device, compute output data on the device and transfer it back to the host. Between the computations the data is not changed on the host. In this combination we have very high data throughput and simple computations, so bandwidth is the limiting factor for performance. Ideally we transfer input data to the device once, compute multiple iterations on the GPU without transferring data between host and device and at the end we transfer the final result back to the host. This is shown on the right part of Figure 3.4.
However, we cannot simply perform the same computation three times in a row in one kernel. The considered iterative operation uses overlapping neighbourhoods for computation, so we have shared halo elements between them. When performing multiple iterations, the halo elements have to be copied back to the global memory after each iteration, because other neighbourhoods depend on them as well. In addition to that we have to synchronize the computations, such as that no halo elements are accessed before they are computed. Previously we started a new kernel instance for each smoothing operation, which is an implicit synchronization point in the computation. All data is copied to and back from the host. By leaving out the implicit synchronization we enable race conditions, because the consecutive computations access the data of the respective predecessor. In consequence we have to employ a strategy for applying this optimization.

3.2.1 *Overlapped Temporal Tiling*

To tackle these issues *overlapped temporal tiling* has been proposed [15] as a technique to reduce the data sharing and synchronization requirements by introducing redundant operations. Instead of synchronizing the computations and exchanging the computed halos, each tile computes its halos redundantly. This technique trades the redundant computation of halo regions for a reduction of global memory accesses and thus allows time-space tiling on GPU targets. Overlapped temporal tiling is sometimes called overlapped tiling as well, but in order to differentiate between the strategy presented in 3.1.2 and this approach, we call it overlapped temporal tiling. We will now discuss how this can be expressed in Lift and evaluate the efficiency of the generated code with this approach. The structure of this optimization is shown in Figure 3.5 on a simple 1-dimensional example of a Jacobi 3-point stencil.

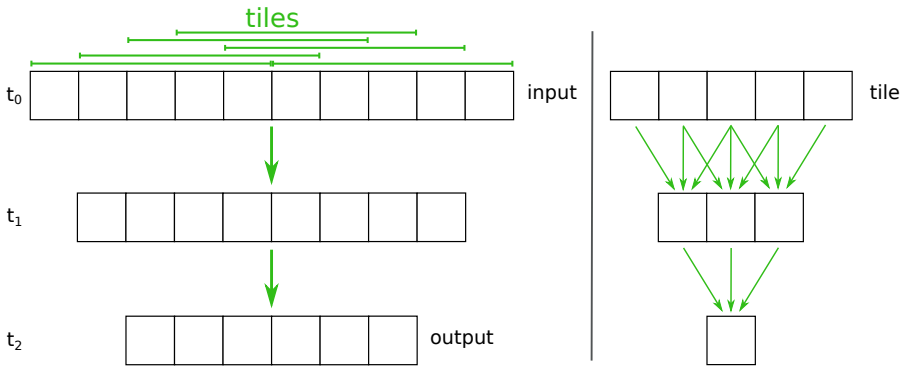

Figure 3.5: 1-dimensional overlapped temporal tiling structure

Boundary handling of the input is omitted here for better understanding. In order to compute one output element of timestep t_2, we need the same element and its two neighbours from t_1. In order to compute each of them we need each element and its neighbours from t_0. This means we need a total of five elements of t_0 to compute one output element of t_2. Traditionally, each element of t_1 would be computed one time, copied to global memory and then distributed to all threads, which need it. In overlapped temporal tiling, we construct a tile for each output element to be computed of t_2. This means each of the six tiles contains five input elements and overlaps the neighbouring tiles by four elements. In each tile we compute the three elements of t_1 and use them to compute the single output element of t_2. After building the tiles and copying them into the local memory of a work-group, no further global memory accesses are needed. In exchange for that we see that most of the elements of t_1 are needed in the computations of multiple output elements of t_2, so they are computed in multiple tiles redundantly. The structure of the Lift expression, performing 1-dimensional overlapped temporal tiling is shown in Figure 3.6.

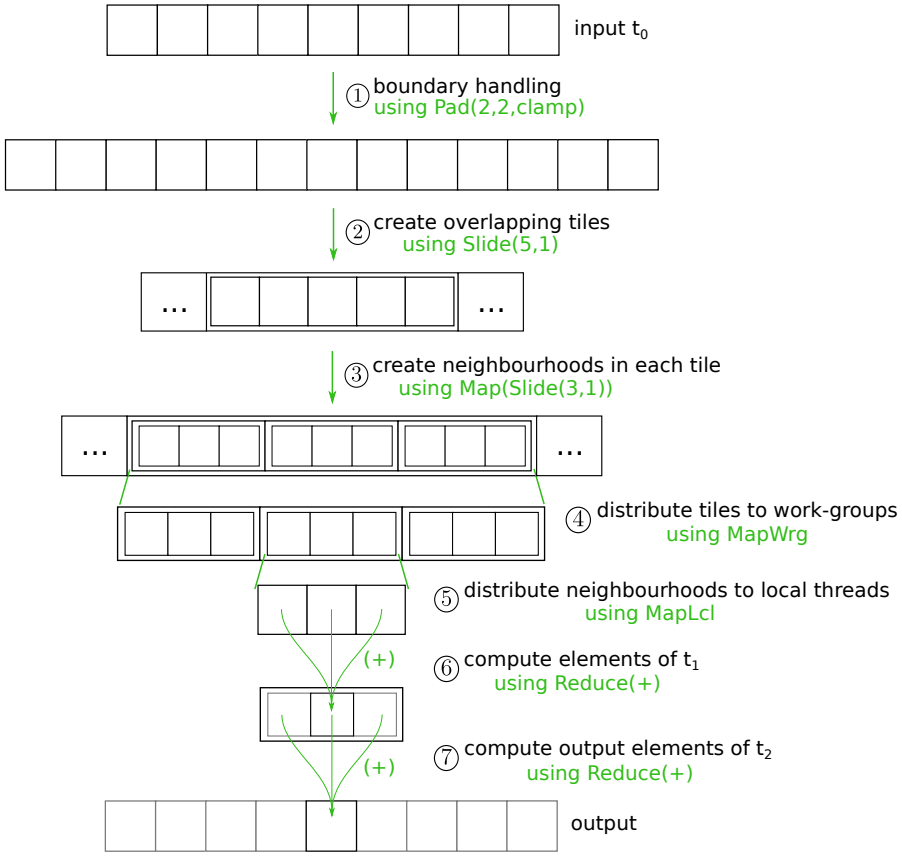

Figure 3.6: LIFT 1-dimensional overlapped temporal tiling

1 For handling the boundaries we have to take into consideration, that we are performing two iterations in a single LIFT expression. This means padding one boundary element on each side of the input is not sufficient. We use *Pad(2,2,clamp)* to prepend the first element two times and append the last element two times. Depending on the boundary conditions, other reindexing functions beside *clamp* can be used as well.

2 The overlapping tiles are created similar to the tiles of overlapped tiling. *Slide(5,1)* builds tiles of size five with an offset of one. Therefore neighbouring tiles overlap each other by four elements.

3 One element of t_1 is computed from three input elements of t_0, so we build neighbourhoods of three values. Similar to the tiles, the neighbourhoods have an offset of one, so we realize them using *Slide(3,1)*. The neighbourhoods have to be created inside each tile, which we accomplish by applying *Map(Slide(3,1))*.

4/5 In order to harness the advantages described above, we divide the work between different threads. We use *MapWrg* to distribute the following computations for each tile to a work-group

and *MapLcl* to distribute the computations for each neighbourhood inside a tile to local threads inside the work-group.

6 We compute the elements of timestep t_1 with *Reduce(+)*. Not pictured is the following *Join*, which flattens the structure in preparation for the next step.

7 In a similar manner we compute the final output elements of timestep t_2 with *Reduce(+)*, copy them into global memory using *toGlobal(id)* and flatten the created tiles to a single array using *Join*.

The complete Lift expression, implementing 1-dimensional overlapped temporal tiling is shown in Listing 3.3.

Listing 3.3: overlapped temporal tiling 1-dimensional Lift expression

```
1 input =>
2     Join() o
3     MapWrg(
4       MapSeq(toGlobal(id)) o
5       Reduce(add, 0.0f) o
6       Join() o
7       MapLcl(Reduce(add, 0.0f)) o
8     Map(Slide(3,1)) o
9     Slide(5,1) o Pad(2,2,clamp) $ input
```

In this expression the computation is being performed exactly as pictured in Figure 3.7(a).

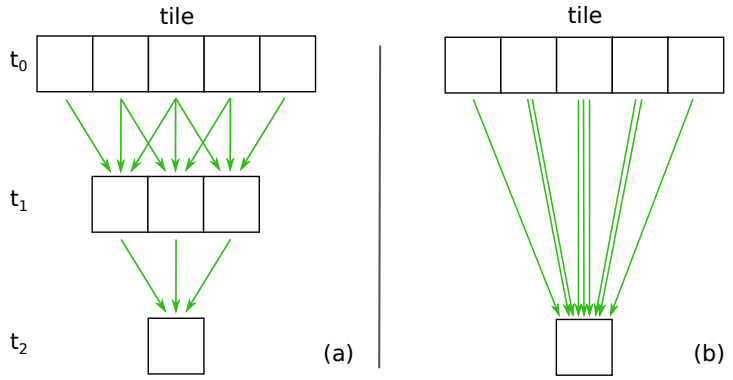

Figure 3.7: 1-dimensional overlapped temporal tiling with optimized computation

This means we have two instances of the *Reduce* primitive. In the generated OpenCL code we perform a total of four reduce operations per tile. Three of them for the neighbourhoods and the last one to sum up the results of them. So in total we sum up all elements in a tile, whereby some elements occur as much as three times in the summation, according to how often they appear in neighbourhoods.

We can optimize the expression to only use one instance of *Reduce* per tile. We flatten the neighbourhoods in each tile after their creation to a simple array. Then we can apply *Reduce* to it and get the same result as before. This structure of this changed computation is shown in Figure 3.7(b). The LIFT expression implementing this optimization is shown in Listing 3.4.

Listing 3.4: overlapped temporal tiling 1-dimensional LIFT expression with optimized computation

```
input =>
    Join() o
    MapWrg(
      MapSeq(toGlobal(id)) o
      Reduce(add, 0.0f)) o
    Map(Join()) o
    Map(Slide(3,1)) o
    Slide(5,1) o Pad(2,2,clamp) $ input
```

The overlapped tiling approach requires redundant computation of the elements in halo regions. The number of redundant operations depends on the stencil shape and the number of iterations inside one kernel. We explore further variants of time-space tiling while avoiding redundant computations to find the best possible optimizations for the GMG operations.

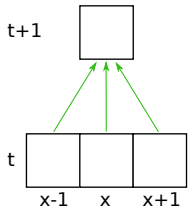

Figure 3.8: 1D 3-point stencil dependencies

It is favourable that a work-item, which computes a particular output element of the first iteration, also performs the computations, which depend on this element in the second iteration. This way the element can stay in the local memory of the work-item and the additional latency of a global memory access is avoided. In order to achieve this, we divide the data into *tiles* along the space and the time dimension. Each tile is assigned to a work-group and copied into its local memory, hence the corresponding work-items can perform the computation of the output elements efficiently. However, the data we operate on have dependencies along the time dimension. Building tiles while ignoring this fact leads to unwanted and unnecessary inter-tile dependencies. In consequence the shape of the tiles depends heavily on the dependencies in the data. For the following examples for time-space tiling we consider a simple 1D 3-point stencil. One element depends on itself and the two neighbouring elements of the previous time-step,

this is pictured in Figure 3.8. First, we present two further approaches to time-space tiling, then we investigate how they can be expressed in the functional language of LIFT.

3.2.2 *Parallelogram Tiling*

In parallelogram tiling the iteration space is divided into parallelogram shaped tiles. Each tile only contains elements whose computations depend on other elements in the same tile or on elements in the left neighbouring tile. The inter-tile dependencies are pictured as arrows in Figure 3.9.

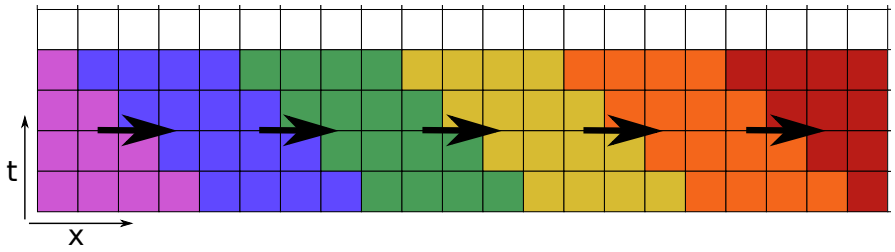

Figure 3.9: parallelogram tiling (inspired by [7])

In consequence of the inter-tile dependencies the different tiles cannot be computed asynchronously in parallel, but have to be computed in a strict order from left to right. Computations within a tile can be computed in parallel, as long as they do not depend on each other. By computing the elements in batches of tiles rather than ordered along the time dimension, we increase the data locality. In consequence we gain the opportunity to reuse the computed data without additional global memory accesses. It is also worth noting, that parallelogram tiling does not require redundant computations, in contrast to overlapped tiling. Parallelogram tiling is a well-known optimization and is also known as *pipeline parallelization* [20].

3.2.3 *Diamond Tiling*

Diamond tiling is a well-known optimization also known as *Split tiling* [8]. To increase data locality, the iteration space is partitioned into upright and inverted diamond shaped tiles which are shown in Figure 3.10.
In contrast to parallelogram tiling the upright diamond tiles have no inter-tile dependencies at all and the inverted diamond tiles have dependencies to both neighbouring tiles. This enables a two part computation pattern in which first all purple colored tiles are computed in parallel and second all blue colored tiles are computed in parallel. This works because no tiles of the same color depend on each other in any way. In consequence we have only one point for synchroniza-

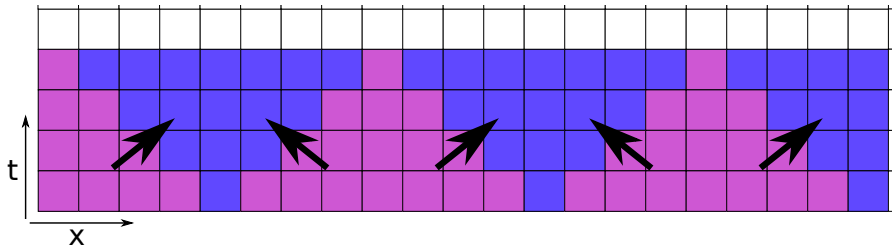

Figure 3.10: diamond tiling (inspired by [7])

tion after the computation of half of the tiles, rather than after each tile like parallelogram tiling. In addition to that, diamond tiling also requires no redundant computations. Accordingly, the increased degree of parallelism and the reduced number of synchronizations lead to better efficiency of the computation.

3.2.4 *Time-Space Tiling in Lift*

Besides overlapped temporal tiling, the presented time-space tiling methods all rely on a partial sequencing of the parallel execution. Not all tiles can be computed asynchronously, but there is a defined order. In order to express parallelism in Lift we have different versions of the *Map* primitive, which are described in 1.2.4.2. All of them distribute work to threads in a different way. However there is no way to specify in which order the parallel parts will be executed. Before adding such functionality to Lift we have to first consider how this can be realized in OpenCL, because we generate OpenCL code from the Lift primitives eventually. In OpenCL there is no possibility to synchronize work-items accross work-groups. The only way to synchronize the computations of the different tiles inside one single kernel would be to include all work-items in a single work-group for synchronization, which does not necessarily lead to good performance and does not utilize the specialized architecture of modern GPUs well. In the Polyhedral Parallel Code Generation[29] (PPCG) compiler the synchronization in diamond tiling is realized on the host side. One kernel is started for computing the upright diamond tiles and another kernel is started with the result of the first kernel for computing the inverted diamond tiles. It is an option to extend Lift with a rewrite rule to split the computation into two kernels. However, the Lift code generation is designed to produce only one kernel per expression, so this would require a significant change. This is outside the scope of this work. In contrast to OpenCL, CUDA 9 introduced a feature called *cooperative groups*, which enables grid-wide and even multi-GPU synchronization on the Pascal and Volta architecture[12]. It is still to be evaluated whether using this yields higher performance than synchronization by starting two different kernels.

We will investigate whether the workaround of writing two separate Lift expressions by hand makes it possible to express the diamond tiling optimization. Building the diamond-shaped tiles in Lift can be achieved with the *Slide* primitive. For the four different time-steps in one iteration of the upright tiles we use *Slide(7,8)*, *Slide(5,8)*, *Slide(3,8)* and *Slide(1,8)* from bottom to the top. In the other computations the *size* parameter is greater than the *step* parameter, to create overlapping tiles. Here it has to be the other way round to create neighbourhoods, which have elements in between, which do not belong to a neighbourhood. This is shown on the example of time-step t_1, using *Slide(5,8)* in Figure 3.11.

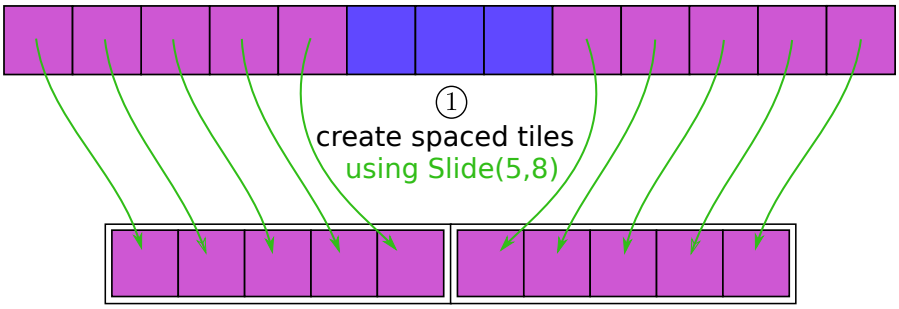

Figure 3.11: creating spaced tiles in Lift

We see, that the three elements, which belong to the inverted blue tile of Figure 3.10, are left out to only create the purple upright diamond tiles. Analyzing the structure of one upright diamond tile, we see that the computation has to be executed in a strict sequence. We have to execute them in the order of time-steps. We cannot compute any items of time-step t_2 without computing the elements of t_1 before. In contrast to the similar problem above, this can be solved by distributing the tiles to work-groups using *MapWrg*, because OpenCL supports synchronization of local threads inside a work-group. Inside, we can use multiple instances of the *MapLcl* primitive, which are synchronized to be executed after another. In the code generation this is achieved with the OpenCL statement `barrier(CLK_GLOBAL_MEM_FENCE);` after the statements of each *MapLcl*. However, Lift currently does not have the capability to save intermediate results and reuse them later. It seems that this could be solved with a similar strategy to the overlapped temporal tiling optimization presented in Section 3.7. This means we compute only the single element at the top of the upright diamond tiles. This is a problem, because in the second part of one iteration we have to compute the elements in the inverted diamond tiles. The elements at their boundaries depend on elements of the upright diamond tiles from different time-steps. In consequence we have to save the intermediate results for all time-steps t_0 to t_3 for correct execution. The fact, that Lift does not support this means, that parallelogram tiling and di-

amond tiling cannot be expressed in LIFT currently. Extending LIFT with this capability requires a drastic change of the compilation flow and is beyond the scope of this thesis.

4 EVALUATION

In this chapter we evaluate the performance of GMG kernels, generated with the functional approach of the Lift framework. Particularly we analyze whether the Lift generated kernels for the individual operations are on par with automatically generated kernels from the PPCG [29] state-of-the-art OpenCL polyhedral compiler. Besides that, we investigate the performance of a GMG V-cycle, which utilizes the operations. Moreover, we inspect whether the optimizations, discussed in Chapter 3, have an impact on the execution time of simple examples.

4.1 HARDWARE AND SOFTWARE SETUP

We used an NVidia GTX 1080 Graphics Card (Compute Capability 6.1) with the proprietary nvidia drivers in version *396.24.02* for conducting the experiments. It contains 20 SM (Streaming Multiprocessors) which each contain 128 CUDA cores for a total of 2560 CUDA cores with a base clock speed of 1607 MHz. Each SM contains 256 KB of register capacity, 96KB of local memory and 48 KB of total L1 cache storage. With 4 Warp Schedulers each of them can schedule 4 warps concurrently. The Graphics Card has a total of 2048 KB of L2 cache and 8 GB of global memory with a bandwidth of 320 GB/s.
The experiments were conducted on the *Ubuntu 16.04 LTS* Operation System with the kernel version *4.13.0-45-generic*. The used OpenCL platform version is *OpenCL 1.2 CUDA 9.2.127*.

4.2 WORKFLOW

Lift still exposes some optimization choices after the rewriting. Reaching back to the example of the split-join rule in Listing 1.3, we see that after the application of this rule we have many choices for the parameter x. The value of it will heavily affect whether the optimization has a positive impact on the performance of the resulting kernel. Thus, parameters of applied rules e.g. tile-sizes, the number of local/global threads and the workload on each thread are still subject to optimization. For finding the best parameter combinations we use the ATF [21] auto-tuning framework, which builds on top of OpenTuner [2]. ATF offers constraint specification of the parameter space and can also take into account OpenCL specific constraints like global thread counts have to be a multiple of local thread counts. Each operation was auto-tuned for a maximum of one hour. PPCG also exposes

thread counts and parameters like tile-sizes as tunable parameters in each dimension. The same auto-tuner is being used here again for a maximum tuning time of one hour for each operation. The time measurement is done with the OpenCL profiling API. Data transfer times to the device are ignored since the focus is on the quality of the generated kernel code.

4.3 EVALUEATION OF SPECIFIC OPTIMIZATIONS

4.3.1 *Spacial Tiling*

In the previous chapter, we showed how classical spacial tiling can be applied to the restriction operation to take advantage of local memory. However, we did not manage to get a speedup over the LIFT generated kernel from the expression shown in Section 2.1 using a tiled approach. Tiling works very well in the context of matrix multiplication, because many input elements are accessed more than once. Hence, keeping tiles in local memory for many computations saves global memory accesses. In the restriction operation, we do not have repeated accesses to the same elements. Each input element only contributes to exactly one output element. In addition to that, the computation of an output element is simply the summation of the corresponding input elements and a subsequent multiplication. Each input element is involved in exactly one computation. In consequence, we do not get better performance by tiling the restriction operation. For further evaluation of this optimization, we increased the computational complexity of the operation by applying a weight to each input element. Thus, the computation has more potential to benefit from the exploitation of local memory. The LIFT expression for this is shown in Listing 4.1.

Listing 4.1: spacial tiling LIFT expression

```
(matrix, weights) =>
  MapWrg(0)(
    MapWrg(1)(tile =>
      MapLcl(0)(
        MapLcl(1)(nbh =>
          MapSeq(toGlobal(id)) o MapSeq(divideBy4) o
            ReduceSeq(fun((acc, tuple) => {
              multAndSumUp.apply(acc, tuple._0, tuple._1)
          }), 0.0f) $ Zip(Join() $ nbh, weights)
      )) o
      //build neighbourhoods
      Slide2D(2, 2)
      //load tile into local memory
      o toLocal(MapLcl(1)(MapLcl(0)(id))) $ tile
  )) o
  //build tiles
```

```
17   Split2D(32, 32) $ matrix
18
19 multAndSumUp = UserFun("multAndSumUp", Array("acc", "l", "r"),
20                   "{ return acc + (l * r); }",
21                   Seq(Float, Float, Float), Float)
```

Tiling is realized as previously described. The input is split into tiles of size 32x32 and the tiles are distributed to work-groups using *MapWrg*. The data is transfered to local memory and computed on by local work-items using *MapLcl*. In consequence of the increased computaional complexity we achieve a speedup of 2.5 over the untiled computation. The runtime comparison is shown in Figure 4.1.

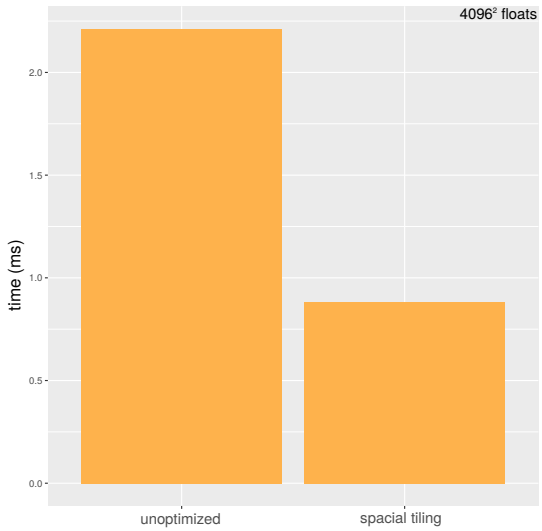

Figure 4.1: modified 2-dimensional restriction operation runtime comparison

The tiling optimization is not only beneficial in matrix multiplication. As long as the considered computation has repeated accesses to input elements or is complex enough to benefit from exploitation of local memory, this optimization has the potential to increase the performance. The specific size of the tiles, which leads to better performance, depends heavily on the considered hardware architecture and is subject to optimization.

A comprehensive evaluation of overlapped tiling in Lift has already been done by Hagedorn [11, pp. 37-45]. In his masterthesis, he achieved a speedup of 1.78 in 2-dimensional convolution by applying overlapped tiling and copying the tiles to local memory. Besides the different approach of building the tiles, the used strategy is similar to the approach presented above.

4.3.2 *Time-Space Tiling*

The *overlapped temporal tiling* optimization as described in Section 3.1.2 allows us to perform two iterations of a stencil in one kernel execution. In order to measure the effectiveness of this optimization, we regard a simple 1-dimensional 7-Point Jacobi stencil. The structure is similar to the overlapped temporal tiling LIFT expression shown in 3.3. Thus, each tile contains 9 elements. We compare an unoptimized version with a temporal overlapped tiling version and a version which implements the optimized overlapped temporal tiling strategy. The runtime comparison is shown in Figure 4.2.

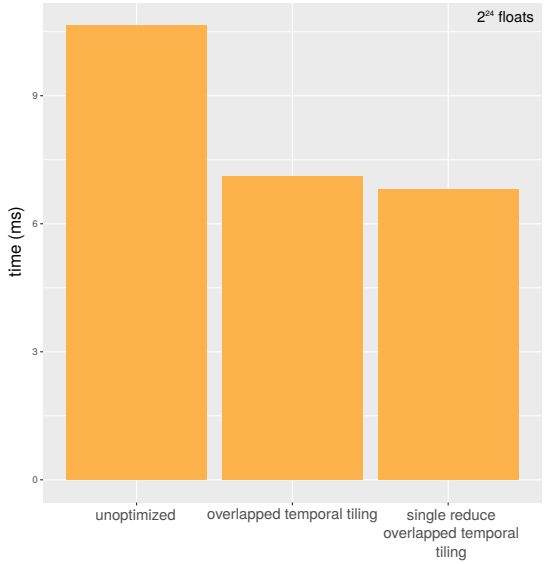

Figure 4.2: overlapped temporal tiling runtime comparison

We observe that, despite the redundant computations, the overlapped temporal tiling version achieves a speedup of 1.5 over the two-time execution of the simple version with omitted additional data transfer. This means this optimization is not only worthwhile because the data movement between host and device is reduced, but also in consequence of efficient local memory usage. The saved overhead of additional data transfers and starting the kernel two times amounts to approximately 130ms in this example and is omitted for the sake of clarity. In addition to that, the optimized version of overlapped temporal tiling is 6% faster than the normal overlapped tiling version.

It is currently not possible to express *parallelogram tiling* and *diamond tiling* in LIFT. It has been shown in several other works that significant speedup can be obtained with them [7, 8, 13, 27]. Furthermore, we achieved high performance in overlapped temporal tiling, thus in the future it is very interesting to investigate how the other presented time-space tiling strategies can be expressed in LIFT.

4.4 GEOMETRIC MULTIGRID OPERATIONS

The LIFT generated kernels for the operations can be executed with arbitrary combinations of global and local thread counts, thus we can tune the kernel invocation, rather than the LIFT compilation. PPCG generates kernels specific to one global and local thread count, hence we have to determine them via auto tuning before the PPCG source to source compilation. This means evaluating one configuration for PPCG takes longer than evaluating a LIFT configuration. This has to be taken into account when comparing the auto-tuning results, because we can evaluate more LIFT configurations than PPCG configurations in the same time. For this reason we have to compare the number of evaluated configurations and possibly perform another auto-tuning run to evaluate more PPCG configurations. Fortunately, we discovered that increasing the number of evaluated PPCG configurations does not yield a faster OpenCL kernel on the target architecture for the considered operations. Another issue regarding auto-tuning the PPCG configuration before source to source compilation is that for large input sizes the PPCG compilation does not terminate. This blocks the auto-tuning run, hence no further configurations are evaluated before the corresponding process is killed, and thus invalidates it. This issue was solved by specifying a timeout for evaluating one PPCG configuration, so when the compilation process hangs, it is killed.

In the following, we present the results of the auto-tuning process in figures with the number of output values on the x-axis and the runtime of the LIFT and PPCG generated kernels on the y-axis, which is scaled logarithmically. Besides that, we show the speedup of the LIFT generated kernel over the PPCG generated kernel for the different sizes. We compared the generated kernels for the output sizes of 32^3, 64^3 and 128^3 elements, because these sizes are commonly used in real GMG applications. The kernels generated by LIFT and PPCG are not designed to be easily read and understood by humans, but we try to work out the reasons for performance differences.
We start by comparing the kernels generated to perform the smoothing operation. In Figure 4.3, the results of the auto-tuning runs are shown.
For the smallest output size of 32^3 elements, the LIFT kernel reached only 85% performance of the PPCG kernel. For the larger input sizes of 63^3 and 128^3 elements, the LIFT kernels achieve a speedup of 1.12 and 1.19 over the PPCG kernels. We observed that the threading strategy of the LIFT generated kernels does not depend on the output size. The structure of the generated kernels only differ in the numbers, related to the output size. In fact, the kernels for the output sizes of 63^3 and 128^3 do not even differ in character count.

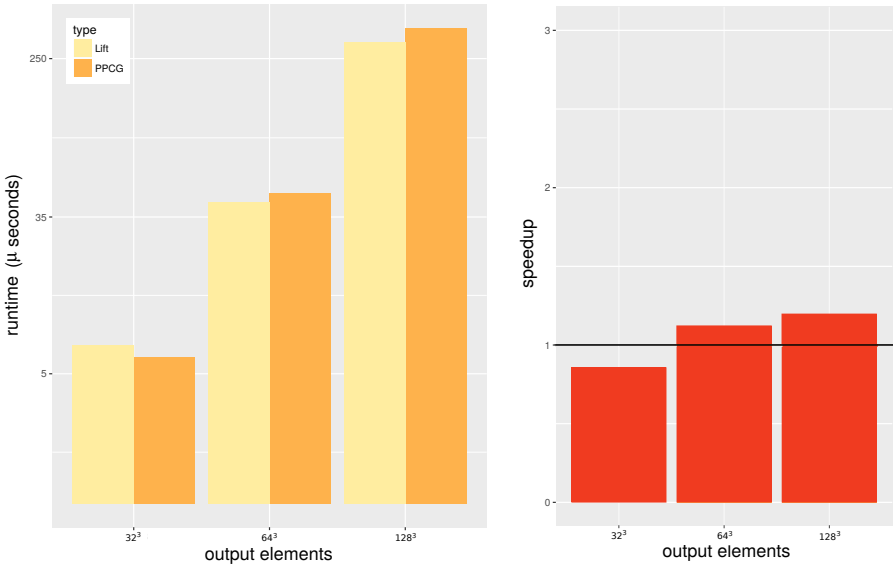

Figure 4.3: auto-tuning results - smoothing operation

This means if the different Lift kernels are started with the same thread counts, each thread always gets the same percentage of input elements. The PPCG kernels, on the other hand, differ a lot for the different sizes. Only the kernel for size 63^3 contains three `for` loops for computation. The other two, each contain two `for` loops, whereby they are not executed by all work-items. Tiling is introduced in all three of them. For reference, all kernels are shown in the Appedix A. PPCG tries to form the loops cosidering the element and thread counts in a way which achieves high performance. However, this proves to be better than the Lift strategy only for the smallest input size. In our comparison, we have to take into account that the smoothing operation is performed iteratively in GMG solvers. This is addressed later in this chapter.

The majority of the noted differences above also concern the residual operation. As explained in Chapter 2.3, the structure of the Lift expressions for the smoothing and the residual operation are very similar. The results of the auto-tuning runs for the residual operation are shown in Figure 4.4.
The kernel implementing the Lift threading strategy again achieves a similar speedup of 1.13 over the PPCG kernel for the largest output size. The kernels for the output size 63^3 achieve very similar performance, which is very interesting, since the kernels have a drastically different structure and were started with completely different configurations. The Lift kernel was started with global thread counts: $(84, 65, 51)$ and local thread counts $(7, 13, 1)$ and the PPCG kernel with global thread counts: $(43, 518, 62)$ and local thread counts $(1, 1, 62)$. The PPCG kernel employs tiling using the element

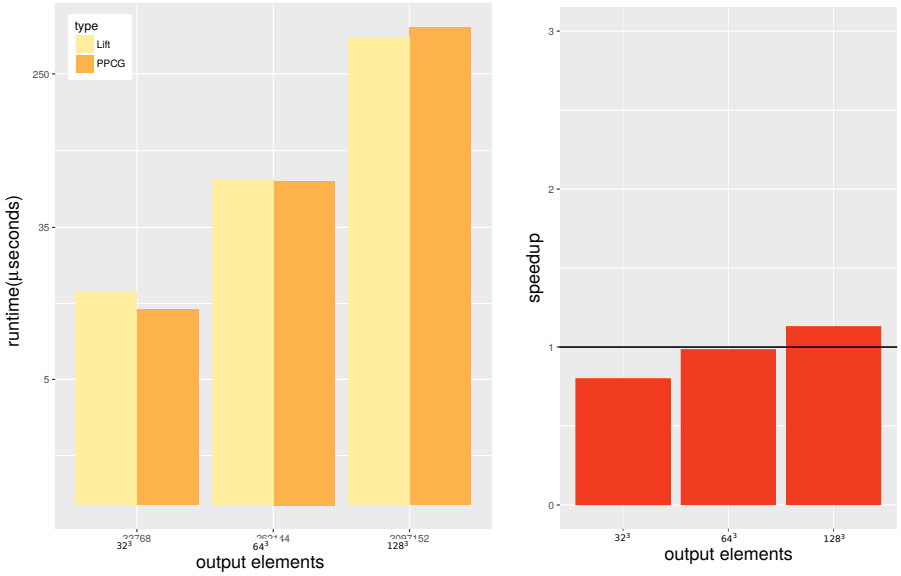

Figure 4.4: auto-tuning results - residual operation

counts $(372, 8, 63)$ for the three dimensions of each tile. Note that these were the best configurations found in the auto-tuning process. PPCG manages to generate a kernel for the smallest output size, which is 20% faster than the Lift generated kernel.

The results of the auto-tuning runs for the restriction operation are shown in Figure 4.5.

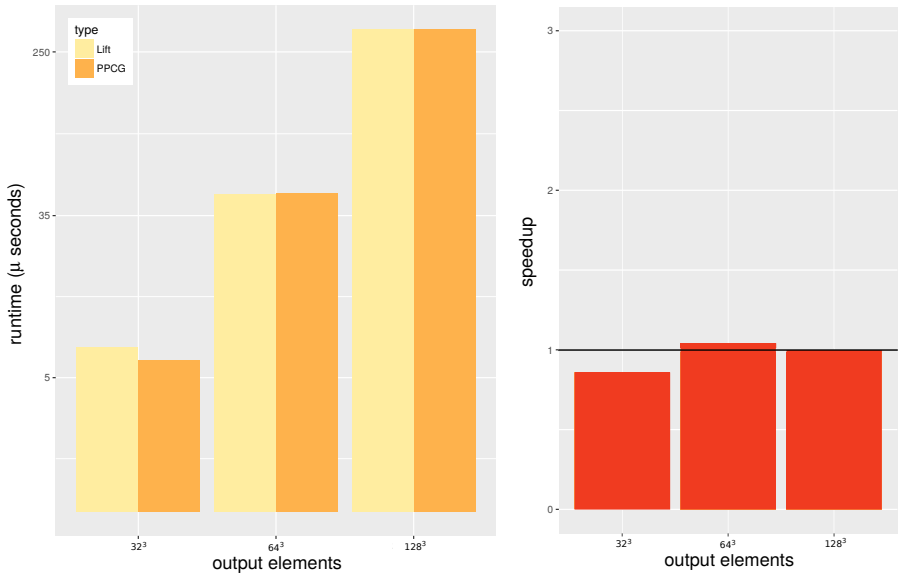

Figure 4.5: auto-tuning results - restriction operation

We observe that the performance of the Lift and PPCG generated kernels are very similar for the two larger output sizes, where they vary only by 1%. For the smallest output size the PPCG kernel is 15%

faster than the Lift kernel. However, an interesting observation is that the absolute difference in execution time is similar for all sizes. The generation of the kernels is as before, such as the Lift kernels are almost identical and the PPCG kernels differ a lot. In addition to that, we observe a trend, that the Lift generated kernel for a specific operation is approximately twice as long as the PPCG generated kernel for the same operation.

The last GMG operation to evaluate is the prolongation. Looking at the results of the auto-tuning runs for the prolongation operation in Figure 4.6, we see that all Lift kernels perform considerably worse than their PPCG counterparts of the same size.

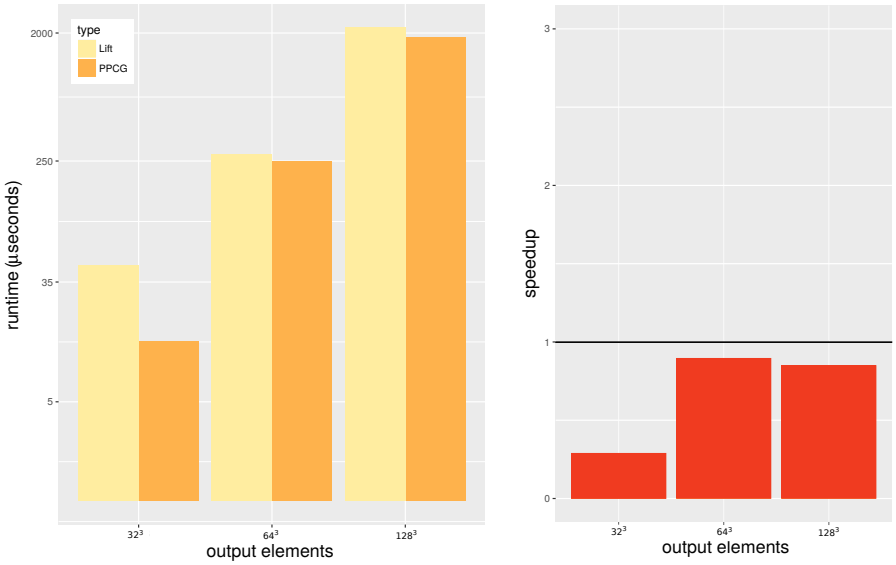

Figure 4.6: auto-tuning results - prolongation operation

The performance differences range from 10% and 15% all the way to 72% for the smallest output size. This is a huge contrast to the other operations, where Lift generated comparatively good performing kernels for the two larger sizes. The PPCG kernels are generated similar to the observations above. In contrast, the Lift kernels are considerably longer and more complicated than those generated for the other operations. A drastically shortened version of the Lift generated kernel for the prolongation operation from 256^3 to 128^3 elements is shown in Listing 4.2.

Listing 4.2: LIFT generated prolongation kernel

```
1  kernel void kernel(const global float* input, [...]) {
2    for (int gl_id_0 = get_global_id(0), [...]){
3      for (int gl_id_1 = get_global_id(1), [...]){
4        for (int gl_id_2 = get_global_id(2), [...]){
5          for([...]){for([...]){for([...]){for([...]){//16 times
6            tmp[...] = mult(input[((32 * ( ((-1 + ((((v_i_71 /
                 2) % 2) + (2 * ((gl_id_2 + (33 * (((input / 4) +
7            //160 lines omitted
8          }}}}
9          for ([...]){ for ([...]){ for ([...]){ //8 times
10           float agg = 0;
11           agg = add(agg, tmp[((16 * v_i_73) + [...])
12                   //6 lines omitted
13           agg = add(agg, v__84[(7 + (8 * v_i_74) + [...])
14
15           output[((4096 * ( ((-1 + v_i_72 + (2 * gl_id_0)) >=
                 0) ? [...] ] = agg;
16         }}}
17   }}}
18 }
```

With a total of more than 13000 characters the kernel is more than five times longer than the PPCG generated kernel. This is due to the fact that the LIFT expression for this, which is shown in 2.5, is very long and complicated in comparison to the expressions for the other operations. We have to use sophisticated combinations of functional LIFT primitives to express 3D prolongation, such as multiple instances of *Zip* or the *Map(Pad))* combination. In order to understand why the generated code is so long, we have to consider the way LIFT generates OpenCL code from LIFT primitives. We can basically distinguish between two different types of primitives in LIFT: Primitives which generate OpenCL statements for computation and primitives which influence how data is read and written. Examples for the first group by reference to Listing 4.2 are

- three instances of *MapGlb* are responsible for the `for` loops in lines *2-4*
- the *mult* function in line *6* originates from the *UserFun* primitive
- the aggregation of values in lines *10-13* is due to the *ReduceSeqUnroll* primitive.

The primitives of the second group do not generate OpenCL statements directly, they rather affect the indices of array accesses. We will illustrate this with aid of the *Pad* primitive. As defined in Section 1.2.4.1, the *Pad* primitive is used to append and prepend a certain number of elements to an array. In order to specify which elements

are padded, a reindexing function is used. An example for the application of *Pad* to a simple array is

$$Pad(l : 1, r : 0, h : clamp)\ \$\ in : [0,1] \rightarrow [0,0,1]. \tag{4.1}$$

With `l = 1` we prepend one element, thus the size of the output array is increased by 1. The *clamp* function is defined in Section 1.27 to return the index of the respective boundary element, hence for prepending the index 0 is returned. In Listing 4.3 we show how this *Pad* primitive affects the generated OpenCL code.

Listing 4.3: Pad code generation example

```
1  //(a)
2    for (i = 0; i < len; i++)
3      output[i] = input[i];
4
5  //(b)
6    int index;
7    for (i = -1; i < len; i++) {
8      if (i > 0) {
9        if (i < len) {
10         index = i;
11       } else {
12         index = len-1;
13       }
14     } else {
15       index = 0;
16     }
17     output[i + 1] = input[index];
18   }
19
20 //(c)
21   for (i = -1; i < len; i++)
22     output[i+1] = input[(i>0) ? ((i<len) ? i ! len-1) ! 0];
```

In *(a)* we have a simple assignment inside a `for`-loop. All elements of the input array are assigned to the respective index in the output array. In *(b)* we see the effect of the *Pad* primitive. The range of the generated `for`-loop is increased by 1, because the output array has a larger size than before. Accessing the input array as before would yield an out of bounds array access and a following segmentation fault or undefined behaviour. Instead, we accomodate for this with a number of *if*-statements to find the correct index for an access, which would be out of bounds otherwise. For easier understanding, we show this with normal *if*-statements in *(b)*. Lift generates the statements inlined, which is shown in *(c)*.
This amendment of the indexing is implemented in Lift with a *View system* [25]. The view of an array specifies how its elements are accessed or being written to. A primitive can alter the view on an array to accomplish its functionality. This means it can change size and

structure of the view of the array. The *Split* primitive for example returns an array of arrays, which is implemented with a change to the view structure as well. For applying more than one primitive to an array, as we have done for all of the operations considered in this thesis, LIFT contains a *ViewStack* for the expression, which contains all changes to the views. During code generation, the views are consumed after another from the *ViewStack*, such as that the view amendmends are executed after another. By designing the code generation like this we are able to combine primitives to perform complex computations. In consequence, if the LIFT expression is complicated, the computations for the correct indices tend to be long and affect the performance of the generated OpenCL kernel negatively. In order to reduce index computation overhead LIFT supports simplifying them to a certain extent. It is very interesting to explore in the future if this can be extended to generate a 3D prolongation OpenCL kernel with less index computations and higher performance.

The 3D prolongation has especially many index computations, because it generates 8 times more output elements than input elements. In a one to one mapping of input to output elements only the structural changes for the actual computation are view amendments. The eightfold increase of elements in the 3D prolongation, on the other hand, can only be accomplished by building additional views for the 7 additional elements besides each 1 existing element. So it is not surprising that the OpenCL kernel generated from this is by far the longest and performs worse than the reference PPCG kernel. In contrast to that, the 3D prolongation operation is much easier to express using imperative programming. For this reason the PPCG generated kernel is so much smaller and has a lesser drawback from index computation.

Overall, besides the prolongation operation, the LIFT generated kernels perform on par or better than the PPCG kernels for the two larger output sizes. For the smallest size of 32^3 it is the other way round. So the LIFT code generation generates high performance OpenCL code from the designed expressions which can keep up with the reference kernels for larger sizes. Note that the PPCG generated kernels implement all possible optimizations supported by PPCG, whereas the LIFT expressions do not implement further optimizations. Finally, we compare the individual operations to each other. We compare the number of computed output elements per second for each operation. So for each operation we show the highest performing kernel of LIFT and PPCG. This is pictured in Figure 4.7. On the x-axis we show the different operations each for LIFT and PPCG, on the y-axis we show the number of computed output elements in 10^6 *elements / sec*.

In this comparison one aspect stands out. Both the LIFT and the PPCG kernel for The prolongation achieve the highest performance compared to the other operations. This is interesting, because we discov-

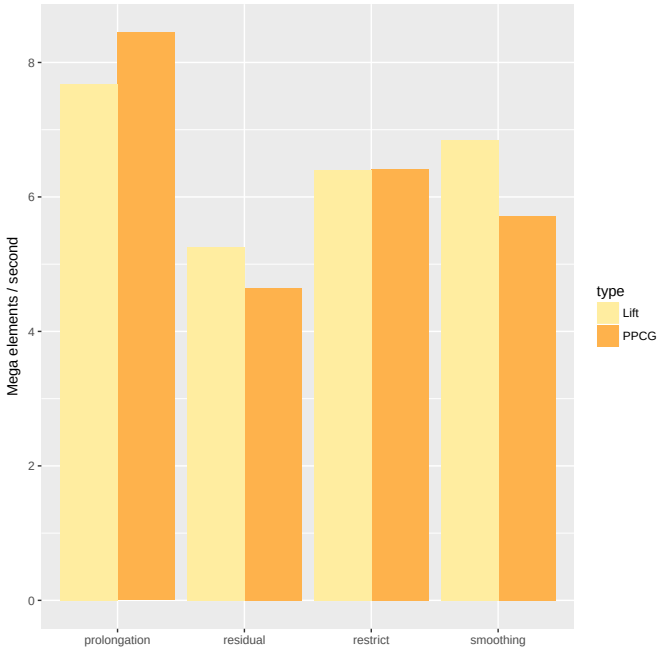

Figure 4.7: Performance comparison of the individual GMG operations

ered the complexity of the operation in form of the eightfold increase in size before. However, this fact, which demands a complex implementation in the functional language of Lift, is also the reason why the prolongation has a performance advantage over the other operations in terms of computed output elements. The input size is only $\frac{1}{8}th$ of the input size of the residual and the smoothing operations and a $\frac{1}{16}th$ of the restriction operation. This does not necessarily result in a reduction of global memory accesses, because the same number of output elements is computed. However, it means that the same elements in global memory are accessed multiple times. In combination with the fact that the considered GPU target architecure is equipped with multiple transparent cache layers per SM, this results in a reduction of memory access delay and thus in more computed elements per second.

4.4.1 *Geometric Multigrid V-Cycle*

In the GMG method all of the operations are used together in a program, hence it is not enough to examine each of them individually. Lift does not support writing entire programs at the moment, so we built a program, implementing the GMG V-cycle as presented in Figure 1.9 in OpenCL. We start on a 64x64x64 grid, which is restricted to 32x32x32 and finally to 16x16x16 during the execution. Before each restriction we compute the residual after applying the smoother 3 times, whereby each application consists of 2 executions of the smoothing

kernel as described in Section 2.4. Before each prolongation we perform 3 post smoothing iterations. We can choose for each operation whether the Lift or PPCG kernel is used in the computation. Each kernel is executed with the best configuration found in the previous auto-tuning process to ensure maximal performance of the individual operations. This way we can investigate if a Lift GMG solver is faster than a PPCG GMG solver. In contrast to the other experiments, this time we include the time for data transfer from and to the device, because in a real world scenario they contribute to the execution time as well. In order to achieve this, we measure the time with the `C time.h` functions. The results of these tests are shown in Figure 4.8.

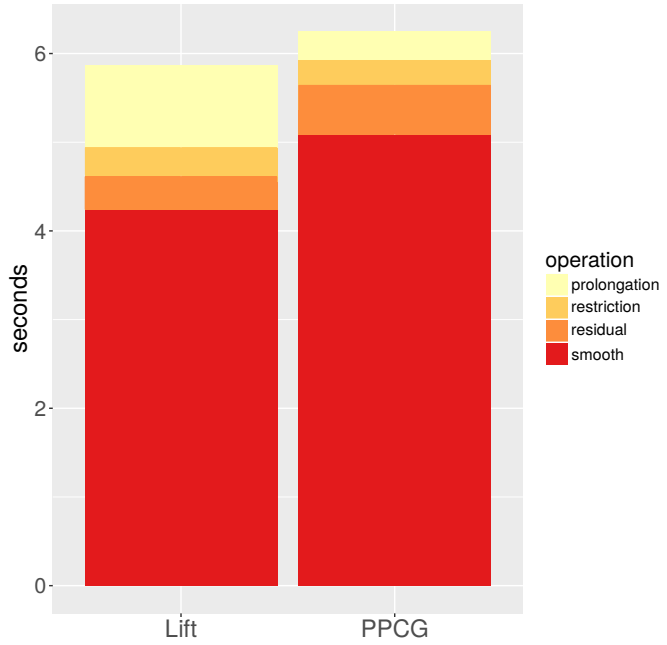

Figure 4.8: GMG solver performance comparison

On the x-axis we show which of the Lift and PPCG solvers are used and on the y-axis we show the execution time in seconds, broken down by operation. In both solvers the smoothing operation dominates the total execution time with 72% in the Lift version and 81% in the PPCG version. This is expected, because the smoothing operation is applied iteratively on each grid, hence it is executed a total of 24 times, whereas the other operations are executed 3 times each. Comparing the Lift and PPCG performance, we observe that the differnces in execution time in the residual and restriction operation are not grave. In the restriction both are on par and Lift is 30% faster in performing the residual operation, but this does not take account for a significant fraction in total execution time. The performance advantage of PPCG over Lift in the prolongation, described above, is reflected in the total prolongation execution time on top of the Figure. The PPCG solver achieves a speedup of 2.98 over the Lift solver

in the prolongation operation. In consequence the prolongation takes up 20% of the total Lift solver execution time, whereas it accounts only for 0.06% of the total PPCG solver execution time. Nevertheless, the total execution time of the Lift solver is still shorter, which is due to the performance advantage in the iterative smoothing operations. They add up to an advantage of 17% in the smoothing operation and a total advantage of 6%. This underlines the importance of analyzing the concerned application to spot key points for optimization. It would be beneficial to further optimize the underperforming Lift prolongation operation and the iteratively applied smoothing operation, where small performance improvements add up to a large runtime advantage.

It was discovered during these tests, that there is a large performance difference according to the structuring of the kernels. Placing each kernel in a different file, rather than having all of them in the same file, improves performance by 58%. This is because we measure not only the runtime of the kernel, but the whole execution. In consequence loading of a large kernel file can negatively impact performance.

5 CONCLUSION

In this thesis, we started by analyzing the four different Geometric Multigrid operations *smooth*, *residual*, *restrict* and *prolongate*. After identifying the main components of each operation, we implemented them as 1-dimensional Lift expression. For expressing the prolongation, we extended Lift with the new *Drop* primitive, which enables the removal of boundary values. With this extension we are capable of expressing all operations in the functional language of Lift. On the foundation of the 1-dimensional expressions we built 2-dimensional and 3-dimensional expressions for all operations. The approach of identifying the main building blocks for each expression in 1 dimensions enables designing Lift expressions for the operations in arbitrary dimensions.
In the subsequent chapter, we analyzed well-known optimizations for GMG operations. We demonstrated how *spacial tiling* optimizations can be applied to Lift expressions for exploiting data locality. The smoothing operation in GMG solvers is executed iteratively, thus the potential for runtime improvement is high. We examined different *time-space tiling* strategies for reducing data movement overhead in this operation. The *overlapped temporal tiling* approach enables utilizing local memory with time-space tiling at the cost of redundant computations. We expressed this optimization in Lift and discovered further opportunities for performance improvement. Last, we analyzed *parallelogram tiling* and *diamond tiling* and investigated why they currently can not be expressed with Lift.
Finally, we evaluated the performance of Lift generated OpenCL kernels for the optimizations and all operations. We could not achieve a speedup for the *restriction* operation using spacial tiling, because no data is reused and too few computations per input element. Repeating the test with an operation of increased computational complexity proved that applying the optimization is worthwile. The implementation of time-space tiling in the form of overlapped tiling yields a substancial speedup over repeated execution of the unoptimized kernel. Besides the prolongation operation, the generated kernels for the operations are on par with the PPCG generated reference kernels. Due to the complexity of the Lift prolongation expression, the generated kernel has an overhead in form of complicated array index computations. Despite this, using the Lift generated kernels in a GMG solver results in better performance than the reference implementation. The reason for this is a small speedup in the iterative Lift smooth operation, which adds up to a substantial runtime improvement.

To summarize, we achieve to generate high-performance OpenCL kernels for GMG operations using the LIFT framework. We emphasized, that focusing on optimizing iterative executed computations is beneficial and had success implementing a form of time-space tiling in LIFT. Further work will investigate, how LIFT can be extended to support other forms of time-space tiling. Expression of time-space tiling optimizations in form of rewrite rules is also desirable for atomatic application in the rewriting process.

A

APPENDIX

A.1 LIFT GENERATED OPENCL KERNELS FOR THE GMG OPERATIONS

All LIFT kernels are pictured for the input size of 32^3 elements. Differences to kernels of other input dimensions are limited to changes in adressing, directly related to the input size. The kernels are modfied for the sake of brevity. The modications include removal of comments and condensing of almost empty lines. The computation is only amended in the prolongation kernel to omit some index computations, in order to keep the waist of paper minimal.

Listing A.1: LIFT generated Gauss Seidel Red Black Smoother OpenCL kernel

```
1  float f(float a, float alphaC, float b, float h2inv, float beta_iE, float
       beta_iC, float beta_jS, float beta_jC, float beta_kB, float beta_kC,
       float inputC, float inputE, float inputW, float inputS, float inputN,
       float inputB, float inputF, float lambdaC, float rhsC, uchar color){
2    if (!color) return inputC;
3    float helmholtz = a * alphaC * inputC - b * h2inv *
4      (beta_iE * (inputE - inputC) -
5       beta_iC * (inputC - inputW) +
6       beta_jS * (inputS - inputC) -
7       beta_jC * (inputC - inputN) +
8       beta_kB * (inputB - inputC) -
9       beta_kC * (inputC - inputF));
10   return inputC - lambdaC * (helmholtz - rhsC);
11 }
12 float id(float x){{ return x; };}}
13
14 kernel void smooth(float v__87, const global float* restrict v__88, float
       v__89, const global float* restrict v__90, const global float* restrict
       v__91, const global float* restrict v__92, float v__93, const global
       float* restrict v__94, const global float* restrict v__95, const global
       float* restrict v__96, const global uchar* restrict v__97, global float*
        v__103){{
15   float v__102;
16   for (int v_gl_id_84 = get_global_id(2); (v_gl_id_84 < 30); v_gl_id_84 = (
         v_gl_id_84 + get_global_size(2))){
17     for (int v_gl_id_85 = get_global_id(1); (v_gl_id_85 < 30); v_gl_id_85 = (
           v_gl_id_85 + get_global_size(1))){
18       for (int v_gl_id_86 = get_global_id(0); (v_gl_id_86 < 30); v_gl_id_86 =
             (v_gl_id_86 + get_global_size(0))){
19         v__102 = f(v__87, v__88[(1057 + v_gl_id_86 + (1024 * v_gl_id_84) +
               (32 * v_gl_id_85))], v__89, v__93, v__90[(1058 + v_gl_id_86 +
               (1024 * v_gl_id_84) + (32 * v_gl_id_85))], v__90[(1057 +
               v_gl_id_86 + (1024 * v_gl_id_84) + (32 * v_gl_id_85))], v__91
               [(1089 + v_gl_id_86 + (1024 * v_gl_id_84) + (32 * v_gl_id_85))],
                v__91[(1057 + v_gl_id_86 + (1024 * v_gl_id_84) + (32 *
               v_gl_id_85))], v__92[(2081 + v_gl_id_86 + (1024 * v_gl_id_84) +
               (32 * v_gl_id_85))], v__92[(1057 + v_gl_id_86 + (1024 *
               v_gl_id_84) + (32 * v_gl_id_85))], v__94[(1057 + v_gl_id_86 +
               (1024 * v_gl_id_84) + (32 * v_gl_id_85))], v__94[(1058 +
```

```
            v_gl_id_86 + (1024 * v_gl_id_84) + (32 * v_gl_id_85))], v__94
            [(1056 + v_gl_id_86 + (32 * v_gl_id_85) + (1024 * v_gl_id_84))],
             v__94[(1089 + v_gl_id_86 + (1024 * v_gl_id_84) + (32 *
            v_gl_id_85))], v__94[(1025 + v_gl_id_86 + (1024 * v_gl_id_84) +
            (32 * v_gl_id_85))], v__94[(2081 + v_gl_id_86 + (1024 *
            v_gl_id_84) + (32 * v_gl_id_85))], v__94[(33 + v_gl_id_86 + (32
            * v_gl_id_85) + (1024 * v_gl_id_84))], v__95[(1057 + v_gl_id_86
            + (1024 * v_gl_id_84) + (32 * v_gl_id_85))], v__96[(1057 +
            v_gl_id_86 + (1024 * v_gl_id_84) + (32 * v_gl_id_85))], v__97
            [(1057 + v_gl_id_86 + (1024 * v_gl_id_84) + (32 * v_gl_id_85))])
            ;
20        v__103[(1057 + v_gl_id_86 + (1024 * v_gl_id_84) + (32 * v_gl_id_85))]
             = id(v__102);
21  }}}}}
```

Listing A.2: Lift generated residual operation OpenCL kernel

```
1
2   float f(float a, float alphaC, float b, float h2inv, float beta_iC, float
        validF, float beta_jC, float validN, float beta_kC, float validW, float
        beta_iB, float validB, float beta_jS, float validS, float beta_kE, float
         validE, float inputC, float inputE, float inputW, float inputS, float
        inputN, float inputB, float inputF, float rhsC){
3     float Ax = a * alphaC * inputC - b * h2inv *
4       (beta_iC * (validF * (inputC + inputF ) - 2.0 * inputC) +
5        beta_jC * (validN * (inputC + inputN ) - 2.0 * inputC) +
6        beta_kC * (validW * (inputC + inputW ) - 2.0 * inputC) +
7        beta_iB * (validB * (inputC + inputB ) - 2.0 * inputC) +
8        beta_jS * (validS * (inputC + inputS ) - 2.0 * inputC) +
9        beta_kE * (validE * (inputC + inputE ) - 2.0 * inputC));
10    return rhsC - Ax;
11  }
12  float id(float x){{ return x; };}
13
14  kernel void residual(float v__78, const global float* restrict v__79, float
        v__80, const global float* restrict v__81, const global float* restrict
        v__82, const global float* restrict v__83, float v__84, const global
        float* restrict v__85, global float* v__93, const global float* restrict
         v__86, const global float* restrict v__87){{
15    float v__92;
16    for (int v_gl_id_75 = get_global_id(2); (v_gl_id_75 < 30); v_gl_id_75 = (
          v_gl_id_75 + get_global_size(2))){
17      for (int v_gl_id_76 = get_global_id(1); (v_gl_id_76 < 30); v_gl_id_76 = (
            v_gl_id_76 + get_global_size(1))){
18        for (int v_gl_id_77 = get_global_id(0); (v_gl_id_77 < 30); v_gl_id_77 =
              (v_gl_id_77 + get_global_size(0))){
19          v__92 = f(v__78, v__79[(1057 + v_gl_id_77 + (1024 * v_gl_id_75) + (32
                 * v_gl_id_76))], v__80, v__84, v__81[(1057 + v_gl_id_77 + (1024
                 * v_gl_id_75) + (32 * v_gl_id_76))], v__87[(33 + v_gl_id_77 +
                (32 * v_gl_id_76) + (1024 * v_gl_id_75))], v__82[(1057 +
                v_gl_id_77 + (1024 * v_gl_id_75) + (32 * v_gl_id_76))], v__87
                [(1025 + v_gl_id_77 + (1024 * v_gl_id_75) + (32 * v_gl_id_76))],
                 v__83[(1057 + v_gl_id_77 + (1024 * v_gl_id_75) + (32 *
                v_gl_id_76))], v__87[(1056 + v_gl_id_77 + (32 * v_gl_id_76) +
                (1024 * v_gl_id_75))], v__81[(2081 + v_gl_id_77 + (1024 *
                v_gl_id_75) + (32 * v_gl_id_76))], v__87[(2081 + v_gl_id_77 +
                (1024 * v_gl_id_75) + (32 * v_gl_id_76))], v__82[(1089 +
                v_gl_id_77 + (1024 * v_gl_id_75) + (32 * v_gl_id_76))], v__87
                [(1089 + v_gl_id_77 + (1024 * v_gl_id_75) + (32 * v_gl_id_76))],
                 v__83[(1058 + v_gl_id_77 + (1024 * v_gl_id_75) + (32 *
                v_gl_id_76))], v__87[(1058 + v_gl_id_77 + (1024 * v_gl_id_75) +
                (32 * v_gl_id_76))], v__85[(1057 + v_gl_id_77 + (1024 *
                v_gl_id_75) + (32 * v_gl_id_76))], v__85[(1058 + v_gl_id_77 +
                (1024 * v_gl_id_75) + (32 * v_gl_id_76))], v__85[(1056 +
                v_gl_id_77 + (32 * v_gl_id_76) + (1024 * v_gl_id_75))], v__85
```

```
            [(1089 + v_gl_id_77 + (1024 * v_gl_id_75) + (32 * v_gl_id_76))],
             v__85[(1025 + v_gl_id_77 + (1024 * v_gl_id_75) + (32 *
            v_gl_id_76))], v__85[(2081 + v_gl_id_77 + (1024 * v_gl_id_75) +
            (32 * v_gl_id_76))], v__85[(33 + v_gl_id_77 + (32 * v_gl_id_76)
            + (1024 * v_gl_id_75))], v__86[(1057 + v_gl_id_77 + (1024 *
            v_gl_id_75) + (32 * v_gl_id_76))]);
        v__93[(1057 + v_gl_id_77 + (1024 * v_gl_id_75) + (32 * v_gl_id_76))]
            = id(v__92);
}}}}}
```

Listing A.3: LIFT generated restriction operation OpenCL kernel

```
float add(float x, float y){{ return x+y; };}
float divideBy8(float x){{ return x*0.125; };}

kernel void restriction(const global float* restrict v__20, global float*
    v__25){{
  float v__21;
  for (int v_gl_id_15 = get_global_id(2); (v_gl_id_15 < 16); v_gl_id_15 = (
      v_gl_id_15 + get_global_size(2))){
    for (int v_gl_id_16 = get_global_id(1); (v_gl_id_16 < 16); v_gl_id_16 = (
        v_gl_id_16 + get_global_size(1))){
      for (int v_gl_id_17 = get_global_id(0); (v_gl_id_17 < 16); v_gl_id_17 =
          (v_gl_id_17 + get_global_size(0))){
        float v_tmp_43 = 0.0f;
        v__21 = v_tmp_43;
        v__21 = add(v__21, v__20[((2048 * v_gl_id_15) + (64 * v_gl_id_16) +
            (2 * v_gl_id_17))]);
        v__21 = add(v__21, v__20[(1 + (64 * v_gl_id_16) + (2048 * v_gl_id_15)
            + (2 * v_gl_id_17))]);
        v__21 = add(v__21, v__20[(32 + (2048 * v_gl_id_15) + (64 * v_gl_id_16
            ) + (2 * v_gl_id_17))]);
        v__21 = add(v__21, v__20[(33 + (64 * v_gl_id_16) + (2048 * v_gl_id_15
            ) + (2 * v_gl_id_17))]);
        v__21 = add(v__21, v__20[(1024 + (64 * v_gl_id_16) + (2048 *
            v_gl_id_15) + (2 * v_gl_id_17))]);
        v__21 = add(v__21, v__20[(1025 + (2048 * v_gl_id_15) + (64 *
            v_gl_id_16) + (2 * v_gl_id_17))]);
        v__21 = add(v__21, v__20[(1056 + (64 * v_gl_id_16) + (2048 *
            v_gl_id_15) + (2 * v_gl_id_17))]);
        v__21 = add(v__21, v__20[(1057 + (2048 * v_gl_id_15) + (64 *
            v_gl_id_16) + (2 * v_gl_id_17))]);
        v__25[(v_gl_id_17 + (256 * v_gl_id_15) + (16 * v_gl_id_16))] =
            divideBy8(v__21);
}}}}}
```

Listing A.4: LIFT generated prolongation operation OpenCL kernel

```
#ifndef Tuple2_float_float_DEFINED
#define Tuple2_float_float_DEFINED
typedef struct __attribute__((aligned(4))){
    float _0;
    float _1;
} Tuple2_float_float;
#endif

float mult(float l, float r){{ return l * r; };}
float add(float x, float y){{ return x+y; };}
float id(float x){{ return x; };}

kernel void prolongation(const global float* restrict v__77, const global
    float* restrict v__78, global float* v__89, global float* v__84){{
  float v__85;
```

```
15   for (int v_gl_id_65 = get_global_id(0); (v_gl_id_65 < 33); v_gl_id_65 = (
          v_gl_id_65 + get_global_size(0))){
16     for (int v_gl_id_66 = get_global_id(1); (v_gl_id_66 < 33); v_gl_id_66 = (
            v_gl_id_66 + get_global_size(1))){
17       for (int v_gl_id_67 = get_global_id(2); (v_gl_id_67 < 33); v_gl_id_67 =
              (v_gl_id_67 + get_global_size(2))){
18         for (int v_i_68 = 0; (v_i_68 < 2); v_i_68 = (1 + v_i_68)){
19           for (int v_i_69 = 0; (v_i_69 < 2); v_i_69 = (1 + v_i_69)){
20             for (int v_i_70 = 0; (v_i_70 < 2); v_i_70 = (1 + v_i_70)){
21               for (int v_i_71 = 0; (v_i_71 < 8); v_i_71 = (1 + v_i_71)){
22                 v__84[(v_i_71 + (8 * v_i_70) + (32 * v_i_68) + (69696 *
                        v_gl_id_65) + (2112 * v_gl_id_66) + (64 * v_gl_id_67) +
                        (16 * v_i_69))] = mult(v__77[index1], v_78[index2]); //
                        index computation omitted for the sake of brevity
23         }}}}
24         for (int v_i_72 = 0; (v_i_72 < 2); v_i_72 = (1 + v_i_72)){
25           for (int v_i_73 = 0; (v_i_73 < 2); v_i_73 = (1 + v_i_73)){
26             for (int v_i_74 = 0; (v_i_74 < 2); v_i_74 = (1 + v_i_74)){
27               float v_tmp_151 = 0.0f;
28               v__85 = v_tmp_151;
29               v__85 = add(v__85, v__84[((16 * v_i_73) + (64 * v_gl_id_67) +
                      (2112 * v_gl_id_66) + (69696 * v_gl_id_65) + (32 * v_i_72)
                       + (8 * v_i_74))]);
30               v__85 = add(v__85, v__84[(1 + (8 * v_i_74) + (32 * v_i_72) +
                      (69696 * v_gl_id_65) + (2112 * v_gl_id_66) + (64 *
                      v_gl_id_67) + (16 * v_i_73))]);
31               v__85 = add(v__85, v__84[(2 + (8 * v_i_74) + (32 * v_i_72) +
                      (69696 * v_gl_id_65) + (2112 * v_gl_id_66) + (64 *
                      v_gl_id_67) + (16 * v_i_73))]);
32               v__85 = add(v__85, v__84[(3 + (8 * v_i_74) + (32 * v_i_72) +
                      (69696 * v_gl_id_65) + (2112 * v_gl_id_66) + (64 *
                      v_gl_id_67) + (16 * v_i_73))]);
33               v__85 = add(v__85, v__84[(4 + (8 * v_i_74) + (32 * v_i_72) +
                      (69696 * v_gl_id_65) + (2112 * v_gl_id_66) + (64 *
                      v_gl_id_67) + (16 * v_i_73))]);
34               v__85 = add(v__85, v__84[(5 + (8 * v_i_74) + (32 * v_i_72) +
                      (69696 * v_gl_id_65) + (2112 * v_gl_id_66) + (64 *
                      v_gl_id_67) + (16 * v_i_73))]);
35               v__85 = add(v__85, v__84[(6 + (8 * v_i_74) + (32 * v_i_72) +
                      (69696 * v_gl_id_65) + (2112 * v_gl_id_66) + (64 *
                      v_gl_id_67) + (16 * v_i_73))]);
36               v__85 = add(v__85, v__84[(7 + (8 * v_i_74) + (32 * v_i_72) +
                      (69696 * v_gl_id_65) + (2112 * v_gl_id_66) + (64 *
                      v_gl_id_67) + (16 * v_i_73))]);
37               int v_i_76 = 0;
38               v__89[((4096 * ( ((-1 + v_i_72 + (2 * v_gl_id_65)) >= 0) ? ( ((
                      v_i_72 + (2 * v_gl_id_65)) <= 64) ? (-1 + v_i_72 + (2 *
                      v_gl_id_65)) : 63 ) : 0 )) + (64 * ( ((-1 + v_i_73 + (2 *
                      v_gl_id_66)) >= 0) ? ( ((v_i_73 + (2 * v_gl_id_66)) <= 64)
                       ? (-1 + v_i_73 + (2 * v_gl_id_66)) : 63 ) : 0 )) + ( ((-1
                       + v_i_74 + (2 * v_gl_id_67)) >= 0) ? ( ((v_i_74 + (2 *
                      v_gl_id_67)) <= 64) ? (-1 + v_i_74 + (2 * v_gl_id_67)) :
                      63 ) : 0 ))] = id(v__85);
39 }}}}}}}}
```

A.2 PPCG GENERATED OPENCL KERNELS FOR THE GMG OPERATIONS

In contrast to the Lift kernels, the PPCG kernels for the individual operations differ a lot depending on the input size. Consequently, we show the generated kernels for the considered input sizes of 32^3, 64^3

and 128^3 elements. Each kernel is pruned, in order to present the differences concisely.

Listing A.5: PPCG generated Gauss Seidel Red Black Smoother OpenCL kernel for inputsize 32x32x32

```
__kernel void smoothing32(float a, __global float *alpha, float b, __global
    float *beta_i, __global float *beta_j, __global float *beta_k, float
    h2inv, __global float *input, __global float *lambda, __global float *
    output, __global float *rhs){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
  float private_helmholtz;

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  if (t2 >= 1)
    for (int c1 = ((b1 + 326) % 342) + 16; c1 <= 480; c1 += 342)
      for (int c3 = ppcg_max(1, (c1 - 1) / 15 - 1); c3 <= ppcg_min(30, (c1 -
          1) / 15); c3 += 1) {
        private_helmholtz = (((a * alpha[(c3 * 32 + (c1 - 15 * c3)) * 32 + t2
            ]) * input[(c3 * 32 + (c1 - 15 * c3)) * 32 + t2] //[...]
            computation omitted for the sake of brevity
         ));
        output[(c3 * 32 + (c1 - 15 * c3)) * 32 + t2] = (input[(c3 * 32 + (c1
            - 15 * c3)) * 32 + t2] - (lambda[(c3 * 32 + (c1 - 15 * c3)) * 32
             + t2] * (private_helmholtz - rhs[(c3 * 32 + (c1 - 15 * c3)) *
            32 + t2])));
}}
```

Listing A.6: PPCG generated Gauss Seidel Red Black Smoother OpenCL kernel for inputsize 64x64x64

```
__kernel void smoothing64(float a, __global float *alpha, float b, __global
    float *beta_i, __global float *beta_j, __global float *beta_k, float
    h2inv, __global float *input, __global float *lambda, __global float *
    output, __global float *rhs){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
  float private_helmholtz;

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  #define ppcg_fdiv_q(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  for (int c2 = 0; c2 <= 62; c2 += 55)
    for (int c3 = ppcg_max(1, 7 * b1 + ppcg_fdiv_q(6 * b1 + t1 - 1, 31) - 1);
         c3 <= ppcg_min(62, 7 * b1 + ppcg_fdiv_q(6 * b1 + t1 - 1, 31)); c3
        += 1)
      for (int c5 = ppcg_max(0, -c2 + 1); c5 <= ppcg_min(54, -c2 + 62); c5 +=
           1) {
        private_helmholtz = (((a * alpha[(c3 * 64 + (223 * b1 + t1 - 31 * c3)
            ) * 64 + (c2 + c5)]) * input[(c3 * 64 + (223 * b1 + t1 - 31 * c3
            )) //[...] computation omitted for the sake of brevity
        output[(c3 * 64 + (223 * b1 + t1 - 31 * c3)) * 64 + (c2 + c5)] = (
            input[(c3 * 64 + (223 * b1 + t1 - 31 * c3)) * 64 + (c2 + c5)] -
            (lambda[(c3 * 64 + (223 * b1 + t1 - 31 * c3)) * 64 + (c2 + c5)]
            * (private_helmholtz - rhs[(c3 * 64 + (223 * b1 + t1 - 31 * c3))
             * 64 + (c2 + c5)])));}}
```

Listing A.7: PPCG generated Gauss Seidel Red Black Smoother OpenCL kernel for inputsize 128x128x128

```
__kernel void smoothing128(float a, __global float *alpha, float b, __global
    float *beta_i, __global float *beta_j, __global float *beta_k, float
    h2inv, __global float *input, __global float *lambda, __global float *
    output, __global float *rhs){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
  float private_helmholtz;

  for (int c1 = ((b1 + 107) % 171) + 64; c1 <= 8064; c1 += 171)
    if (t0 + 3 * ((-63 * t0 + c1 + 62) / 189) >= 1 && t0 + 3 * ((-63 * t0 +
        c1 + 62) / 189) <= 126 && (-63 * t0 + c1 + 62) % 189 >= 63)
      for (int c2 = 0; c2 <= 126; c2 += 27)
        if (t2 + c2 >= 1 && t2 + c2 <= 126) {
          private_helmholtz = (((a * alpha[((t0 + 3 * ((-63 * t0 + c1 - 1) /
              189)) * 128 + (((-63 * t0 + c1 - 1) % 189) + 1)) * 128 + (t2 +
              c2)]) //[...] computation omitted for the sake of brevity
            ));
          output[((t0 + 3 * ((-63 * t0 + c1 - 1) / 189)) * 128 + (((-63 * t0
              + c1 - 1) % 189) + 1)) * 128 + (t2 + c2)] = (input[((t0 + 3 *
              ((-63 * t0 + c1 - 1) / 189)) * 128 + (((-63 * t0 + c1 - 1) %
              189) + 1)) * 128 + (t2 + c2)] - (lambda[((t0 + 3 * ((-63 * t0
              + c1 - 1) / 189)) * 128 + (((-63 * t0 + c1 - 1) % 189) + 1)) *
              128 + (t2 + c2)] * (private_helmholtz - rhs[((t0 + 3 * ((-63
              * t0 + c1 - 1) / 189)) * 128 + (((-63 * t0 + c1 - 1) % 189) +
              1)) * 128 + (t2 + c2)])));
}}
```

Listing A.8: PPCG generated residual operation OpenCL kernel for inputsize 32x32x32

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void residual32(float a, __global float *alpha, float b, __global
    float *beta_i, __global float *beta_j, __global float *beta_k, float
    h2inv, __global float *input, __global float *output, __global float *
    rhs, __global float *valid){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
  double private_Ax;

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  #define ppcg_fdiv_q(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  if (t2 >= 1)
    for (int c0 = ppcg_max(0, 23 * ppcg_fdiv_q(b1 - 16, 345)); c0 <= ppcg_min
        (30, ppcg_fdiv_q(b1 - 1, 15)); c0 += 23)
      for (int c3 = ppcg_max(ppcg_max(0, -c0 + 1), -c0 + (b1 - 1) / 15 - 1);
          c3 <= ppcg_min(ppcg_min(22, -c0 + 30), -c0 + (b1 - 1) / 15); c3 +=
           1) {
        private_Ax = (((a * alpha[((c0 + c3) * 32 + (b1 - 15 * c0 - 15 * c3))
            * 32 + t2]) * input[((c0 + c3) * 32 + (b1 - 15 * c0 - 15 * c3))
            //[...] computation omitted for the sake of brevity
          ));
        output[((c0 + c3) * 32 + (b1 - 15 * c0 - 15 * c3)) * 32 + t2] = (rhs
            [((c0 + c3) * 32 + (b1 - 15 * c0 - 15 * c3)) * 32 + t2] -
            private_Ax);}}
```

Listing A.9: PPCG generated residual operation OpenCL kernel for input-size 64x64x64

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void residual64(float a, __global float *alpha, float b, __global float *beta_i, __global float *beta_j, __global float *beta_k, float h2inv, __global float *input, __global float *output, __global float *rhs, __global float *valid){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
  double private_Ax;

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  #define ppcg_fdiv_q(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  for (int c3 = ppcg_max(1, ppcg_fdiv_q(8 * b1 - 1, 31) - 1); c3 <= ppcg_min(62, (8 * b1 + 6) / 31); c3 += 1)
    for (int c4 = ppcg_max(t1, -((t1 + c3) % 2) - 8 * b1 + 31 * c3 + 2); c4 <= ppcg_min(7, -8 * b1 + 31 * c3 + 62); c4 += 2) {
      private_Ax = (((a * alpha[(c3 * 64 + (8 * b1 - 31 * c3 + c4)) * 64 + (((t2 + 61) % 62) + 1)]) * input[(c3 * 64 + (8 * b1 - 31 * c3 + c4)) * 64 + (((t2 + 61) % 62) + 1)]) //[...] computation omitted for the sake of brevity
       ));
      output[(c3 * 64 + (8 * b1 - 31 * c3 + c4)) * 64 + (((t2 + 61) % 62) + 1)] = (rhs[(c3 * 64 + (8 * b1 - 31 * c3 + c4)) * 64 + (((t2 + 61) % 62) + 1)] - private_Ax);}}
```

Listing A.10: PPCG generated residual operation OpenCL kernel for input-size 128x128x128

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void residual128(float a, __global float *alpha, float b, __global float *beta_i, __global float *beta_j, __global float *beta_k, float h2inv, __global float *input, __global float *output, __global float *rhs, __global float *valid){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
  double private_Ax;

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  #define ppcg_fdiv_q(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  for (int c1 = ((b1 + 393) % 457) + 64; c1 <= 8064; c1 += 457)
    if (t0 + 29 * ppcg_fdiv_q(-63 * t0 + c1 - 127, 1827) >= -28 && t0 + 29 * ppcg_fdiv_q(-63 * t0 + c1 - 127, 1827) <= 97 && c1 >= 63 * t0 + 1827 * ((-63 * t0 + c1 + 1700) / 1827) + 1)
      for (int c2 = 0; c2 <= 126; c2 += 38)
        for (int c5 = ppcg_max(t2, ((t2 + c2 + 32) % 33) - c2 + 1); c5 <= ppcg_min(37, -c2 + 126); c5 += 33) {
          private_Ax = (((a * alpha[((t0 + 29 * ((-63 * t0 + c1 - 1) / 1827)) * 128 + //[...] computation omitted for the sake of brevity
          ));
          output[((t0 + 29 * ((-63 * t0 + c1 - 1) / 1827)) * 128 + (((-63 * t0 + c1 - 1) % 1827) + 1)) * 128 + (c2 + c5)] = (rhs[((t0 + 29 * ((-63 * t0 + c1 - 1) / 1827)) * 128 + (((-63 * t0 + c1 - 1) % 1827) + 1)) * 128 + (c2 + c5)] - private_Ax);
}}
```

Listing A.11: PPCG generated restriction operation OpenCL kernel for inputsize 32x32x32

```
__kernel void restriction32(__global float *input, __global float *output){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  if (b0 >= 1 && t2 >= 1)
    for (int c1 = 0; c1 <= 30; c1 += 4)
      for (int c4 = ppcg_max(0, -c1 + 1); c4 <= ppcg_min(3, -c1 + 30); c4 +=
           1)
        output[(b0 * 32 + (c1 + c4)) * 32 + t2] = (((((((input[(2 * b0 * 64
             + (2 * c1 + 2 * c4)) * 64 + 2 * t2] + input[((2 * b0 + 1) * 64 +
              (2 * c1 + 2 * c4)) * 64 + 2 * t2]) + input[(2 * b0 * 64 + (2 *
             c1 + 2 * c4 + 1)) * 64 + 2 * t2]) + input[((2 * b0 + 1) * 64 +
             (2 * c1 + 2 * c4 + 1)) * 64 + 2 * t2]) + input[(2 * b0 * 64 + (2
              * c1 + 2 * c4)) * 64 + (2 * t2 + 1)]) + input[((2 * b0 + 1) *
             64 + (2 * c1 + 2 * c4)) * 64 + (2 * t2 + 1)]) + input[(2 * b0 *
             64 + (2 * c1 + 2 * c4 + 1)) * 64 + (2 * t2 + 1)]) + input[((2 *
             b0 + 1) * 64 + (2 * c1 + 2 * c4 + 1)) * 64 + (2 * t2 + 1)]) *
             0.125);
}
```

Listing A.12: PPCG generated restriction operation OpenCL kernel for inputsize 64x64x64

```
__kernel void restriction64(__global float *input, __global float *output){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  if (t1 >= 1)
    for (int c2 = 0; c2 <= 62; c2 += 31)
      for (int c3 = 1; c3 <= 62; c3 += 1)
        for (int c5 = ppcg_max(t2, ((t2 + c2 + 27) % 28) - c2 + 1); c5 <=
             ppcg_min(30, -c2 + 62); c5 += 28)
          output[(c3 * 64 + t1) * 64 + (c2 + c5)] = (((((((input[(2 * c3 *
               128 + 2 * t1) * 128 + (2 * c2 + 2 * c5)] + input[((2 * c3 + 1)
                * 128 + 2 * t1) * 128 + (2 * c2 + 2 * c5)]) + input[(2 * c3 *
                128 + (2 * t1 + 1)) * 128 + (2 * c2 + 2 * c5)]) + input[((2 *
                c3 + 1) * 128 + (2 * t1 + 1)) * 128 + (2 * c2 + 2 * c5)]) +
               input[(2 * c3 * 128 + 2 * t1) * 128 + (2 * c2 + 2 * c5 + 1)])
               + input[((2 * c3 + 1) * 128 + 2 * t1) * 128 + (2 * c2 + 2 * c5
                + 1)]) + input[(2 * c3 * 128 + (2 * t1 + 1)) * 128 + (2 * c2
               + 2 * c5 + 1)]) + input[((2 * c3 + 1) * 128 + (2 * t1 + 1)) *
               128 + (2 * c2 + 2 * c5 + 1)]) * 0.125);}
```

Listing A.13: PPCG generated restriction operation OpenCL kernel for inputsize 128x128x128

```
__kernel void restriction128(__global float *input, __global float *output){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  if (b1 >= 1)
    for (int c2 = 0; c2 <= 126; c2 += 32)
      if (t2 + c2 >= 1 && t2 + c2 <= 126)
```

```
10          for (int c3 = ppcg_max(0, -3 * b0 + 1); c3 <= ppcg_min(2, -3 * b0 +
                126); c3 += 1)
11            output[((3 * b0 + c3) * 128 + b1) * 128 + (t2 + c2)] = ((((((((
                  input[((6 * b0 + 2 * c3) * 256 + 2 * b1) //[...] computation
                  omitted for the sake of brevity
12              * 0.125);
13  }
```

Listing A.14: PPCG generated prolongation operation OpenCL kernel for inputsize 32x32x32

```
1  __kernel void prolongation32(__global float *input, __global float *output){
2    int b0 = get_group_id(0), b1 = get_group_id(1);
3    int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
4    int private_delta_k;
5    int private_delta_j;
6    int private_delta_i;
7    __local float shared_input_0[3][16][16];
8
9    #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
10   #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
11   #define ppcg_fdiv_q(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
12   {
13     for (int c0 = ppcg_max(0, -b1 - (b1 + 7) / 15 + 1); c0 <= ppcg_min(2, -b1
           - (b1 + 7) / 15 + 16); c0 += 1)
14       for (int c1 = t1; c1 <= 15; c1 += 5)
15         for (int c2 = t2; c2 <= 15; c2 += 5)
16           shared_input_0[c0][c1][c2] = input[((b1 + c0 + (b1 + 7) / 15 - 1) *
                 16 + c1) * 16 + c2];
17     barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
18     for (int c3 = ppcg_max(1, 2 * b1 + ppcg_fdiv_q(2 * b1 - 1, 15) - 1); c3
           <= ppcg_min(30, 2 * b1 + 2 * b1 / 15 + 2); c3 += 1)
19       for (int c4 = ppcg_max(t1, ((2 * b1 + t1 + 4) % 5) - 32 * b1 + 15 * c3
             + 1); c4 <= ppcg_min(31, -32 * b1 + 15 * c3 + 30); c4 += 5)
20         for (int c5 = ((t2 + 4) % 5) + 1; c5 <= 30; c5 += 5) {
21           private_delta_k = (-1);
22           if ((c5) & 1) {
23             private_delta_k = 1;
24           }
25           private_delta_j = (-1);
26           if ((32 * b1 - 15 * c3 + c4) & 1) {
27             private_delta_j = 1;
28           }
29           private_delta_i = (-1);
30           if ((c3) & 1) {
31             private_delta_i = 1;
32           }
33           output[(c3 * 32 + (32 * b1 - 15 * c3 + c4)) * 32 + c5] =
                 (((((((1.0 * shared_input_0[-b1 - (b1 + 7) //[...]
                 computation omitted for the sake of brevity
34           ));
35 }}}
```

Listing A.15: PPCG generated prolongation operation OpenCL kernel for inputsize 64x64x64

```
1  __kernel void prolongation64(__global float *input, __global float *output){
2    int b0 = get_group_id(0), b1 = get_group_id(1);
3    int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
4    int private_delta_k;
5    int private_delta_j;
6    int private_delta_i;
7    __local float shared_input_0[4][64][6];
```

```

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  for (int c2 = 0; c2 <= 126; c2 += 11) {
    if (t0 <= 3 && 257 * b1 + 126 * t0 >= 64 && 257 * b1 + 126 * t0 <= 8127
         && t2 <= 5 && 2 * t2 + c2 <= 127)
      for (int c4 = 0; c4 <= 63; c4 += 1)
        shared_input_0[t0][c4][t2] = input[((2 * b1 + t0 + (5 * b1 + 62) /
             126 - 1) * 64 + c4) * 64 + (t2 + c2 / 2)];
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
    if (t2 + c2 >= 1 && t2 + c2 <= 126)
      for (int c3 = ppcg_max(((t0 + 4) % 5) + 1, 5 * b1 + t0 - 5 * ((58 * b1
           + 63 * t0 + 441) / 315) + 5); c3 <= ppcg_min(126, 4 * b1 + (5 * b1
            + 3) / 63 + 4); c3 += 5)
        for (int c4 = ppcg_max(0, -257 * b1 + 63 * c3 + 1); c4 <= ppcg_min
             (256, -257 * b1 + 63 * c3 + 126); c4 += 1) {
          private_delta_k = (-1);
          if ((t2 + c2) & 1) {
            private_delta_k = 1;
          }
          private_delta_j = (-1);
          if ((257 * b1 - 63 * c3 + c4) & 1) {
            private_delta_j = 1;
          }
          private_delta_i = (-1);
          if ((c3) & 1) {
            private_delta_i = 1;
          }
          output[(c3 * 128 + (257 * b1 - 63 * c3 + c4)) * 128 + (t2 + c2)] =
               (((((((1.0 * shared_input_0[-2 * b1 - //[...] computation
               omitted for the sake of brevity
          ));
        }
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
}}
```

Listing A.16: PPCG generated prolongation operation OpenCL kernel for inputsize 128x128x128

```
__kernel void prolongation128(__global float *input, __global float *output){
  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1), t2 = get_local_id(2);
  int private_delta_k;
  int private_delta_j;
  int private_delta_i;

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_max(x,y)    ((x) > (y) ? (x) : (y))
  #define ppcg_fdiv_q(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  for (int c2 = 0; c2 <= 254; c2 += 32)
    if (t2 + c2 >= 1 && t2 + c2 <= 254)
      for (int c3 = ppcg_max(ppcg_max(0, -32 * b0 + 1), -32 * b0 + 82 *
           ppcg_fdiv_q(127 * b0 - b1 - 1, 324) + (32 * b1 - 46 * ppcg_fdiv_q
           (127 * b0 - b1 - 1, 324) + 207) / 127 + 79); c3 <= ppcg_min(
           ppcg_min(31, -32 * b0 + 254), -32 * b0 + 82 * ((127 * b0 - b1 +
           323) / 324) + ppcg_fdiv_q(32 * b1 - 46 * ppcg_fdiv_q(127 * b0 - b1
            - 1, 324) - 16, 127)); c3 += 1)
        for (int c4 = ppcg_max(t1, ((t1 + c3 + 3) % 4) + 4064 * b0 - 32 * b1
             + 127 * c3 - 10368 * ppcg_fdiv_q(127 * b0 - b1 - 1, 324) -
             10367); c4 <= ppcg_min(31, 4064 * b0 - 32 * b1 + 127 * c3 -
             10368 * ppcg_fdiv_q(127 * b0 - b1 - 1, 324) - 10114); c4 += 4) {
          private_delta_k = (-1);
          if ((t2 + c2) & 1) {
            private_delta_k = 1;
```

```
18          }
19          private_delta_j = (-1);
20          if ((-(32 * ((127 * b0 - b1 + 323) % 324)) - 127 * c3 + c4 + 10336)
                & 1) {
21            private_delta_j = 1;
22          }
23          private_delta_i = (-1);
24          if ((32 * b0 + c3) & 1) {
25            private_delta_i = 1;
26          }
27          output[((32 * b0 + c3) * 256 + (-(32 * ((127 * b0 - b1 + 323) %
                324)) - //[...] computation omitted for the sake of brevity
28            ));
29 }}
```

BIBLIOGRAPHY

[1] Loyce Adams and J Ortega. "A multi-color SOR method for parallel computation." In: *ICPP*. Citeseer. 1982, pp. 53–56.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "OpenTuner: An Extensible Framework for Program Autotuning." In: *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada, 2014. URL: http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf.

[3] John Backus. "ACM Turing Award Lectures." In: New York, NY, USA: ACM, 2007. Chap. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. ISBN: 978-1-4503-1049-9. DOI: 10.1145/1283920.1283933. URL: http://doi.acm.org/10.1145/1283920.1283933.

[4] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Mary Hall. "Converting Stencils to Accumulations Forcommunication-Avoiding Optimizationin Geometric Multigrid." In: *Proceedings of the Second Workshop on Optimizing Stencil Computations*. WOSC '14. Portland, Oregon, USA: ACM, 2014, pp. 9–16. ISBN: 978-1-4503-2308-6. DOI: 10.1145/2686745.2686749. URL: http://doi.acm.org/10.1145/2686745.2686749.

[5] Borys V. Bazaliy and Avner Friedman. "A Free Boundary Problem for an Elliptic–Parabolic System: Application to a Model of Tumor Growth." In: *Communications in Partial Differential Equations* 28.3-4 (2003), pp. 517–560. DOI: 10.1081/PDE-120020486. URL: https://doi.org/10.1081/PDE-120020486.

[6] G. H. Botorog and H. Kuchen. "Skil: an imperative language with algorithmic skeletons for efficient distributed programming." In: *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*. 1996, pp. 243–252. DOI: 10.1109/HPDC.1996.546194.

[7] Takeshi Fukaya and Takeshi Iwashita. "Time-space tiling with tile-level parallelism for the 3D FDTD method." In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. ACM. 2018, pp. 116–126.

[8] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, Ponuswamy Sadayappan, and Sven Verdoolaege. "Split tiling for GPUs: automatic parallelization using trapezoidal tiles." In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM. 2013, pp. 24–31.

[9] Khronos OpenCL Working Group et al. "The OpenCL Specification, version 1.2, 15 November 2011." In: (2011).

[10] Jia Guo, Ganesh Bikshandi, Basilio B Fraguela, and David Padua. "Writing productive stencil codes with overlapped tiling." In: *Concurrency and Computation: Practice and Experience* 21.1 (2009), pp. 25–39.

[11] Bastian Hagedorn. "An Extension of a Functional Intermediate Language for Parallelizing Stencil Computations and its Optimizing GPU Implementation using OpenCL." MA thesis. Westfälische Wilhelms-Universität Münster, 2016.

[12] Mark Harris. *CUDA 9 Features Revealed: Volta, Cooperative Groups and More*. 2017 (accessed June 23, 2018). URL: https://devblogs.nvidia.com/cuda-9-features-revealed/.

[13] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. "A stencil compiler for short-vector SIMD architectures." In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 13–24.

[14] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, 1997.

[15] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. "Effective Automatic Parallelization of Stencil Computations." In: *SIGPLAN Not.* 42.6 (June 2007), pp. 235–244. ISSN: 0362-1340. DOI: 10.1145/1273442.1250761. URL: http://doi.acm.org/10.1145/1273442.1250761.

[16] M. Leyton and J. M. Piquer. "Skandium: Multi-core Programming with Algorithmic Skeletons." In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 2010, pp. 289–296. DOI: 10.1109/PDP.2010.26.

[17] Andreas Meister. *Numerik linearer Gleichungssysteme*. 5th ed. Wiesbaden, Deutschland: Springer Spektrum, 2015.

[18] NVIDIA Corporation. *NVIDIA GeForce GTX 1080 Gaming Perfected*. 2016. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[19] Nvidia. "Nvidia OpenCL Best Practices Guide." In: (2009).

[20] Shah M Faizur Rahman, Qing Yi, and Apan Qasem. "Understanding stencil code performance on multicore architectures." In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM. 2011, p. 30.

[21] A. Rasch, M. Haidl, and S. Gorlatch. "ATF: A Generic Auto-Tuning Framework." In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2017, pp. 64–71. DOI: 10.1109/HPCC-SmartCity-DSS.2017.9.

[22] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. "Performance portable GPU code generation for matrix multiplication." In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM. 2016, pp. 22–31.

[23] Peter M Sockol. "Multigrid solution of the Navier–Stokes equations on highly stretched grids." In: *International journal for numerical methods in fluids* 17.7 (1993), pp. 543–566.

[24] M. Steuwer, P. Kegel, and S. Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 1176–1182. DOI: 10.1109/IPDPS.2011.269.

[25] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a Functional Data-Parallel IR for High Performance GPU Code Generation." In: *CGO 2017* (2017).

[26] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code." In: *ICFP '15, 2015, Vancouver, BC, Canada* ().

[27] Jan Treibig, Gerhard Wellein, and Georg Hager. "Efficient multicore-aware parallelization strategies for iterative stencil computations." In: *Journal of Computational Science* 2.2 (2011), pp. 130–137.

[28] Ulrich Trottenberg, Cornelius W. Oosterlee, and Anton Schuller. *Multigrid*. London, England: Academic Press, 2000.

[29] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA." In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 54:1–54:23. ISSN: 1544-3566. DOI: 10.1145/2400682.2400713.

[30] Vitaly A Volpert. *Elliptic partial differential equations*. Vol. 1. Springer, 2011.

DECLARATION

I hereby confirm that this thesis on *Efficient Implementation and Optimization of Geometric Multigrid Operations in the Lift Framework* is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited.

I agree to have my thesis checked in order to rule out potential similarities with other works and to have my thesis stored in a database for this purpose.

Münster, July 2018

Martin Lücke

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. 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/

Final Version as of July 20, 2018 (`classicthesis`).