Implementation and Optimization of the Geometric Multigrid Method Using the GPU and OpenCL



Dissertation

Bergische Universität Wuppertal
Fakultät für Mathematik und Naturwissenschaften

eingereicht von
Simon Hoffmann, M. Sc.
zur Erlangung des Grades eines Doktors der Naturwissenschaften

Betreut durch Prof. Dr. Matthias Bolten

Wuppertal, 18.02.2026

Acknowledgments

I would like to thank my supervisor Matthias Bolten for motivating me to study and work on GPU computing in my Master's thesis and extending my interest further for the PhD. Without his guidance and support, this work would not have been possible. It was always a pleasure to have technical discussions with him, gaining interesting perspectives, and receiving his emotional and encouraging support.

I also wish to thank all my current and former colleagues in the Scientific Computing and High Performance Computing group. Working together on otherwise tedious tasks was made much more enjoyable thanks to the consistently relaxed atmosphere and shared sense of humor. I am especially thankful to Marcel for the smooth collaboration in teaching and for his prompt and reliable communication.

I would like to thank my family and friends for their love and constant support. I am particularly grateful to my parents for their continued support in all life circumstances that emerged during my PhD time. I also thank Julian and Linda for proofreading this thesis.

Thank you, Maivi, for all the joy you bring to my life.

Contents

Acknowledgments I

1. Introduction 1

2. Fundamental Elements 3
2.1. Problem Definition and Discretization . . . 3
2.1.1. Stencil Notation . . . 7
2.1.2. Vector-valued Problems . . . 9
2.2. Iterative Methods . . . 11
2.3. Geometric Multigrid Method . . . 16
2.4. High Performance Computing . . . 21
2.5. Other Implementations . . . 24

3. GPU Architecture and OpenCL 25
3.1. OpenCL Overview . . . 25
3.1.1. History of OpenCL . . . 26
3.1.2. OpenCL Programming Model . . . 27
3.1.3. Hardware Implementation / GPU Architecture . . . 30
3.1.4. OpenCL Workflow . . . 39
3.1.5. OpenCL C Kernel Language . . . 41
3.1.6. OpenCL Under the Hood . . . 43
3.2. Best Practices . . . 45
3.2.1. Saturating the Device . . . 46
3.2.2. TLP and ILP . . . 47
3.2.3. Execution Parameters . . . 48
3.2.4. Host-Device Communication and Kernel Streamlining . . . 49
3.2.5. Memory Access Patterns . . . 51
3.2.6. Local Memory Usage . . . 54
3.2.7. Miscellaneous . . . 56

3.3. Profiling Methods 60
3.3.1. Nsight Systems 60
3.3.2. Nsight Compute 61

4. Implementation of Multigrid Components on Single-GPU and its Optimizations 67
4.1. Implementation Overview 68
4.2. Jacobi's Method and Residual Calculation With Varying Coefficients 70
4.2.1. Basic Residual Kernel 72
4.2.2. Coefficients Data Layout 83
4.2.3. Kernel Granularity 85
4.2.4. Exploiting Data Locality 90
4.2.5. Jacobi Split Versus Fused 93
4.2.6. Multiple Jacobi Iterations Without Intermediate Ghost Update 96
4.2.7. Blocking Techniques 98
4.3. Jacobi's Method and Residual Calculation With Fixed Coefficients 106
4.4. Ghost Update 109
4.5. Inter-Grid Transfer 113
4.5.1. Prolongation 114
4.5.2. Restriction 116
4.6. Residual Sum 116
4.7. Coarse Grid Operator 120
4.7.1. Matrix-based Approach (Stencil Arithmetic) 122
4.7.2. Galerkin via Stencil Arithmetic 125
4.7.3. Operating Directly on the Coarse Grid 128
4.8. Miscellaneous 133
4.8.1. V-cycle as a Whole 133
4.8.2. OpenCL Setup 134

5. Multi-GPU Optimizations 137
5.1. Grid Hierarchy 138
5.2. Coarse Grid Operator Computation 139
5.3. Ghost Update 141
5.4. Multiple Jacobi Iterations Without Intermediate Ghost Update Revisited 145
5.5. Overlapped Jacobi's Method 147

6. Blockstencil Implementation 151
6.1. Jacobi's Method and Residual Calculation Using Blockstencils . . 151
6.2. Remaining Multigrid Components 154

7. Evaluation **157**
7.1. Scalar Problems 158
7.1.1. Convergence Tests 159
7.1.2. Varying Technical Parameters of `mgcl` 172
7.2. Vector-valued Problem 176

8. Conclusion and Outlook **181**

A. Appendix **183**
A.1. Data Access Patterns 183
A.2. Residual Kernel Profiling Data 185
A.3. Jacobi Blocking 194
A.4. Residual With Fixed Coefficients 202
A.5. Conversion Between Matrix and Stencil Notations 204
A.6. Overlapped Jacobi's method 207
A.7. Blockstencil Residual 209

List of Figures **211**

List of Tables **215**

Bibliography **220**

Chapter 1

Introduction

Computational science and engineering has become a fundamental discipline across many areas of natural science and engineering, emerging as a simulation-based science that tightly integrates model development, algorithm design, and software implementation. In numerous applications, numerical simulations are a valuable addition to theory and experimentation, often reducing the need for expensive physical testing. Typically, as simulation problems grow in size and complexity, running them efficiently demands increasingly powerful computing resources. In our case of geometric multigrid methods, the same operation is executed on different data in a structured manner, thus parallelization is a crucial aspect in order to gain good performance. Modern CPUs in supercomputing clusters already offer hundreds of cores, enabling substantial parallel processing. However, GPUs can unlock an even higher level of parallelization due to their architecture featuring thousands of threads.

Starting from the late 2000s, using GPUs for general purpose computations became widely accessible and the term GPGPU (general-purpose computing on graphics processing units) emerged. Different vendors have their own libraries for targeting their GPUs: Nvidia developed CUDA, AMD developed ROCm and Intel developed oneAPI. However, as using these libraries restrict the user to use devices manufactured by the respective vendor, a unified platform was developed. OpenCL, maintained by the Khronos Group, offers a unified API that allows for the use of GPUs from different vendors, as well as other hardware architectures, like CPUs, FPGAs and DSPs.

In simulations a common task is to solve a partial differential equation (PDE). Discretizing such an equation results in a linear system of equations, for which numerous numerical methods have been established. While all of those have their own opportunities and limitations, the multigrid method is a fitting choice for the problems that shall be tackled by our implementation. Since it operates on a

structured grid and a highly parallelizable smoother can be plugged in easily, it is well-suited for parallelization using a GPU.

In this thesis, an implementation of a geometric multigrid method is developed and optimized for GPU usage using OpenCL, called `mgcl` [40]. It is mainly restricted to solving 3d elliptic PDEs incorporating spatially varying coefficients, featuring scalar or vector-valued unknowns. Only structured meshes on cuboidal domains are supported.

Chapter 2 gives an overview of the fundamental elements of the problems we focus on, and of multigrid methods with its different ingredients. A brief introduction to high-performance computing and alternative implementations is given.

Chapter 3 explains the general architecture of GPUs and how the OpenCL platform is organized. Furthermore, critical aspects of GPU programming are described, which need to be taken into consideration if meaningful performance gains shall be attained. Lastly, a short insight is given on how kernels can be profiled and analyzed.

Chapter 4 is the heart of this thesis. For single-GPU usage, the implementations of all multigrid components and the optimization opportunities thereof are investigated closely. A particular focus is placed on the computation of the residual and Jacobi's method using spatially varying coefficients, as these are the most expensive parts of the solving phase of the method. In addition, optimizations affecting the whole method as well as OpenCL-specific aspects regarding the setup are examined shortly.

Chapter 5 continues the elaboration of implementation details but focuses on multi-GPU usage. Differences to the single-GPU case are presented and optimization opportunities specific to multi-GPU usage are investigated. The focus is especially on the optimization of the communication between the processes.

In chapter 6, implementation details subject to vector-valued problems are described. Particularly, optimization opportunities in the residual calculation are investigated.

Chapter 7 tests our implementation on different kinds of problems and compares the performance to another pre-existing solution. The impact of varying different technical parameters on the performance of the implementation as a whole is examined.

Finally, chapter 8 provides a summary and an outlook for future work.

Chapter 2

Fundamental Elements

This chapter provides a review of the fundamental elements of simulations, which form the basis of this thesis. The chapter starts with providing model problems as examples. This includes formulating the problem statements using PDEs, discretization methods and stencil notation. After a brief overview of iterative methods, we will take a look at the different components of geometric multigrid methods. Lastly, we will explore some background in high-performance computing, how GPUs are involved, and examine pre-existing solutions.

Please note that the following content is strongly influenced by the way Briggs et al. introduce the foundations of multigrid methods in [11].

2.1. Problem Definition and Discretization

Oftentimes, physical phenomena can be modeled using PDEs, which are mathematical equations that involve unknown functions of multiple variables and their partial derivatives. A wide range of problems, where quantities change over space and time, can be formulated this way, such as heat flow, fluid dynamics and electromagnetic fields.

One simple example is the Helmholtz equation, which can be used to describe the steady-state temperature distribution in a uniform rod.

$$\begin{aligned} -\Delta u(x) + \sigma u(x) &= f(x), \qquad 0 < x < 1, \quad \sigma \geq 0, \\ u(0) &= u(1) = 0. \end{aligned} \tag{2.1}$$

Here, Δ denotes the Laplace operator, which boils down to the second derivative in the 1d case, i.e. $\Delta u(x) = u''(x)$. For higher dimensions, the Laplace operator is

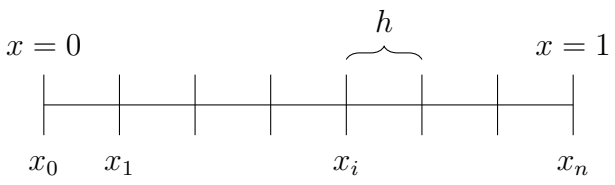


Figure 2.1.: 1d grid on the interval $[0, 1]$ with grid spacing $h = 1/n$ and discrete grid points $x_i,\ i = 0, \ldots, n$.

defined as the second partial derivatives in all directions, e.g. for 3d it is

$$\Delta u(x, y, z) = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) u(x, y, z).$$

In order to solve the problem numerically, we need to discretize the domain $\{x : 0 \leq x \leq 1\}$ into a finite number of points x_i, where $i = 0, \ldots, n$. This creates a grid Ω^h consisting of n subintervals with constant width $h = 1/n$, as shown in figure 2.1.

For each of the $n-1$ interior points, we use a second-order finite difference operator to introduce the approximation v_i to the exact solution $u(x_i)$. Note that, while finite volumes and finite elements are also common methods for the discretization of a PDE, we remain with finite differences in this thesis.

The approximation v_i at the individual points can be aggregated into a vector $\mathbf{v} = (v_1, \ldots, v_{n-1})^T$, where each component satisfies the linear equation

$$\begin{gathered} \frac{-v_{i-1} + 2v_i - v_{i+1}}{h^2} + \sigma v_i = f(x_i), \qquad 1 \leq i \leq n-1, \\ v_0 = v_n = 0. \end{gathered} \tag{2.2}$$

Representing the right-hand side values $f(x_i)$ as a vector $\mathbf{f} = (f_1, \ldots, f_{n-1})^T$, we can write the linear system in matrix form as

$$\mathbf{A}\mathbf{v} = \mathbf{f}, \tag{2.3}$$

where

$$\mathbf{A} = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & \ldots & 0 & 0 \\ -1 & 2 & \ldots & 0 & 0 \\ 0 & -1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & -1 & 2 \\ 0 & 0 & \ldots & 2 & -1 \end{bmatrix} + \sigma \mathbf{I}.$$

With $\sigma = 0$ the term $\sigma\mathbf{I}$ vanishes and equation (2.1) becomes the Poisson equation.

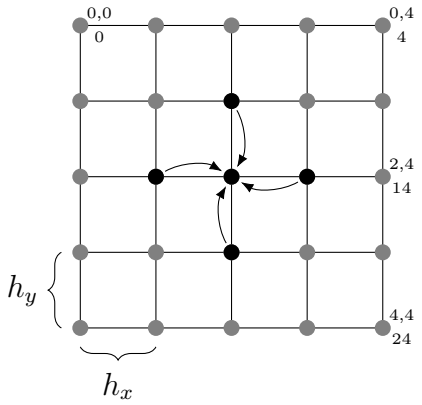


Figure 2.2.: 2d grid on the unit square with grid spacings $h_x = 1/(m-1)$ and $h_y = 1/(n-1)$ and discrete grid points $x_{ij} = x_{in+j},\ i = 0, \ldots, m,\ j = 0, \ldots, n$. The application of the finite difference operator is depicted for the center grid point, taking values from the neighboring points.

We can adjust the problem formulation to model higher dimensional cases, e.g. for 2d (2.1) becomes

$$\begin{aligned} & -\Delta u(x,y) + \sigma u(x,y) = f(x,y), && 0 < x < 1, \quad 0 < y < 1, \quad \sigma \geq 0, \\ \Leftrightarrow \quad & -u_{xx} - u_{yy} + \sigma u = f, && \\ & u = 0 && \text{on } \partial\Omega, \end{aligned} \tag{2.4}$$

where $\partial\Omega$ denotes the boundary of the domain Ω. Discretizing the problem using finite differences again, and having grid point spacings $h_x = 1/m$ and $h_y = 1/n$, we obtain the system of linear equations

$$\begin{aligned} & \frac{-v_{i-1,j} + 2v_{i,j} - v_{i+1,j}}{h_x^2} + \frac{-v_{i,j-1} + 2v_{i,j} - v_{i,j+1}}{h_y^2} + \sigma v_{i,j} = f(x_{i,j}), \\ & v_{0,j} = v_{m,j} = v_{i,0} = v_{i,n} = 0, \quad i = 1, \ldots, m-1,\ j = 1, \ldots, n-1. \end{aligned} \tag{2.5}$$

Figure 2.2 depicts the resulting grid. As we can see, the grid points contributing to the finite differences operator at grid point (i, j) are the directly adjacent neighbors. Moreover, we are using the lexicographic ordering scheme, i.e. the 1d index of a grid point is calculated as $idx_{ij}^{1d} = i \cdot n + j$.

Again, we can formulate the linear system in matrix form as in (2.3), however now the system matrix $\mathbf{A}$ is of size $(m-1)(n-1) \times (m-1)(n-1)$ and can be written

as a block matrix

$$\mathbf{A} = \begin{bmatrix} B & -aI & \dots & 0 & 0 \\ -aI & B & \dots & 0 & 0 \\ 0 & -aI & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -aI & B \\ 0 & 0 & \dots & B & -aI \end{bmatrix}$$

with block dimension $(m-1) \times (m-1)$. Each block B has the same structure as the matrix in (2.1) and is of size $(n-1) \times (n-1)$, while the off-diagonal blocks are identity matrices multiplied by $a = 1/h_x^2$ having size $(n-1) \times (n-1)$.

Such a matrix can easily be constructed using the Kronecker product, e.g. in the 2d case it is

$$\mathbf{A} = \mathbf{I} \otimes \mathbf{D_{xx}} + \mathbf{D_{yy}} \otimes \mathbf{I},$$

where $\mathbf{D_{xx}}$ and $\mathbf{D_{yy}}$ are the system matrices like (2.1) with appropriate number of unknowns.

So far we have only considered the finite differences operator having the same coefficients for each grid point, which restricts the phenomena we can model. In order to extend our modeling capabilities, we can allow the coefficients to change for each grid point. This is the case for e.g. Darcy's law, which models single-phase flow (i.e. the flow of a single liquid) through a porous medium. It can be written as

$$-\nabla \cdot [K(x)\nabla u(x)] = f(x) \quad \text{on } \Omega,$$

where $K(x)$ is the spatially varying permeability of the porous medium. This equation can be seen as the Poisson equation with heterogeneous coefficients [29].

Another important factor of problem statements is the choice of boundary conditions. For our equations (2.1) and (2.4) we used Dirichlet boundary conditions, i.e. having fixed values at the boundary. This means the unknowns at the boundary are no longer unknown and can either be removed completely from the system of equations or kept in but being ignored carefully.

Another example is the presence of periodic boundary conditions, where the domain is repeated at the boundaries. This means, that when for example applying the finite differences operator at a grid point lying at the left boundary, we must attain the value of its imaginary left neighbor from the right boundary of the domain.

There are different ways to handle boundary conditions in the implementation. One is to handle edge cases using branches wherever the boundary condition needs to be considered. This, however, would incur the cost of boundary checks and divergent execution paths, degrading performance under SIMD (Single Instruction Multiple Data) execution. Another approach is to extend the actual grid by one or more layers of ghost points (sometimes called halo), which need to be set to

appropriate values before applying the local operator. While this needs an extra routine to update the ghosts, the actual logic of applying the operator is the same for the entire domain and is thus kept simple and effective.

2.1.1. Stencil Notation

Since the discrete differential operator is a local operator on the well-structured grid, we can make use of stencil notation instead of relying on the matrix representation. This has several advantages compared to storing the full matrix. Firstly, we only need to store the stencil coefficients instead of each matrix entry, of which many are zero as the matrix is sparse, reducing memory consumption and enabling scalability. Secondly, data locality is higher, which can lead to better cache performance. Lastly, and most importantly in our case, stencil code is highly parallelizable since it operates only on the direct neighbors. That means, we do not need information about the entire grid when applying a stencil at a given point and can thus apply the stencil at each point in parallel.

The stencil for the 1d Poisson equation can be written as

$$\mathbf{A} = \frac{1}{h^2}\begin{bmatrix} -1 & 2 & -1 \end{bmatrix},$$

in the 2d case and for $h = h_x = h_y$ it is

$$\mathbf{A} = \frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix},$$

and finally in 3d and for $h = h_x = h_y = h_z$ it is

$$\mathbf{A} = \frac{1}{h^2}\left[\begin{array}{c} \\ -1 \\ \\ \end{array}\right]\left[\begin{array}{ccc} & -1 & \\ -1 & 6 & -1 \\ & -1 & \end{array}\right]\left[\begin{array}{c} \\ -1 \\ \\ \end{array}\right]. \tag{2.6}$$

Applying a stencil on a grid point is equivalent to multiplying a row of the system matrix $\mathbf{A}$ with the vector of approximations $\mathbf{v}$, i.e.

$$s_{ij} = \sum_{j=1}^{n} a_{ij} v_j,$$

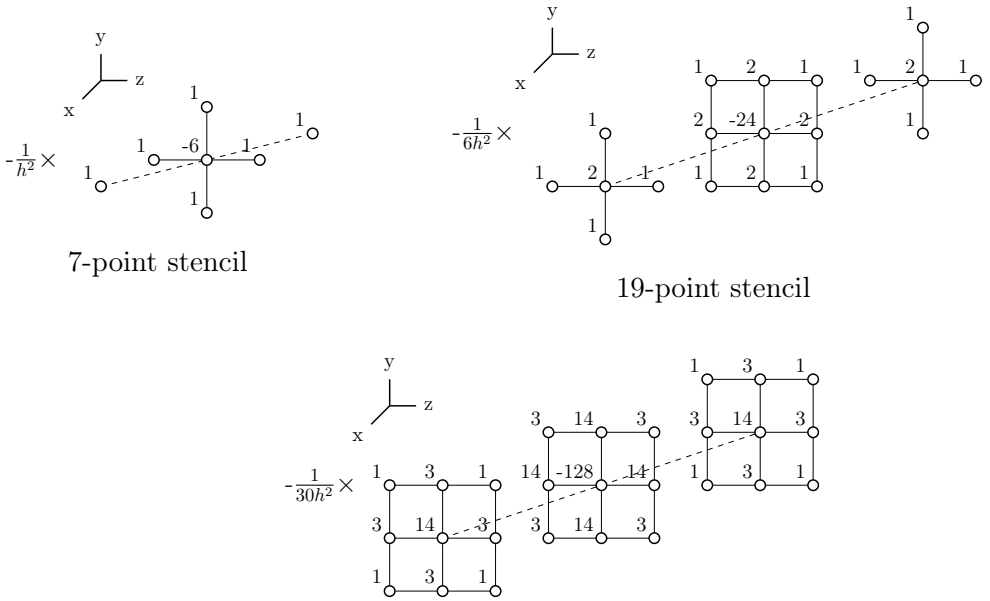
where a_{ij} is the entry of matrix $\mathbf{A}$ at row i and column j. This connection of a matrix to its stencil representation is of importance when constructing the coarse grid operator using stencil notation, which is discussed in section 4.7.

Note that, for actual implementation, we do not need to consider the whole row of $\mathbf{A}$, but only non-zero entries.

The stencil shown in equation (2.6) is called a 7-point stencil (7p stencil in short), since 7 grid points contribute to the stencil sum. This stencil is obtained by summing the 1d second-order central difference formula as shown in equation (2.2) in all three dimensions. However, other stencil shapes can be derived, for example using the high-order compact (HOC) schemes, resulting in 19p and 27p stencils [103, 86]. Compared to the 7p stencil, the 19p stencil has additional values at the edges, while the 27p stencil has additional values at the corners, as shown in the following and as depicted in figure 2.3.

$$\mathbf{A} = \frac{1}{6h^2} \begin{bmatrix} & -1 & \\ -1 & -2 & -1 \\ & -1 & \end{bmatrix} \begin{bmatrix} -1 & -2 & -1 \\ -2 & 24 & -2 \\ -1 & -2 & -1 \end{bmatrix} \begin{bmatrix} & -1 & \\ -1 & -2 & -1 \\ & -1 & \end{bmatrix}$$

$$\mathbf{A} = \frac{1}{30h^2} \begin{bmatrix} 1 & -3 & 1 \\ -3 & -14 & -3 \\ 1 & -3 & 1 \end{bmatrix} \begin{bmatrix} -3 & -14 & -3 \\ -14 & 128 & -14 \\ -3 & -14 & -3 \end{bmatrix} \begin{bmatrix} 1 & -3 & 1 \\ -3 & -14 & -3 \\ 1 & -3 & 1 \end{bmatrix}$$



7-point stencil

19-point stencil

27-point stencil

Figure 2.3.: 3d compact Laplace stencils

Not only can stencils have coefficients mapping to direct neighbors, but also to grid points farther away. While this will occur when exploring stencil arithmetic later, we are generally focusing on compact stencils in this thesis. While the

structure of our stencils is the same for each grid point, the coefficients might vary, as mentioned before. This requires storing not only the coefficients of a single stencil for the entire grid, but the coefficients of one stencil for each grid point.

Note that an alternative to using stencil notation is to use a sparse matrix format like CSR (Compressed Sparse Row) to store the system matrix. In the case of homogeneous coefficients, the stencil notation has the obvious advantage of needing to store only the stencil coefficients once instead of repeating the coefficients for every matrix row. But even in the case of varying coefficients, the stencil notation has some advantages. Firstly, the memory access pattern is more regular, which can lead to better cache performance. Secondly, the implementation of stencil operations is often simpler and more straightforward, as it directly maps to the grid structure. Lastly, stencil operations can be optimized more easily for parallel execution, as they inherently involve local computations that can be performed independently. Each application of a stencil would require a sparse matrix-vector multiplication in the CSR case, which is more complex to implement and optimize. On the other hand, highly optimized libraries for sparse matrix operations exist, especially cuSparse for CUDA, but which would restrict the implementation to Nvidia GPUs only. Another benefit of using sparse matrix formats is the improved flexibility regarding the structure of the grid and the stencils itself. But as we restrict our implementation to regular grids and compact stencils, this is not of concern here.

2.1.2. Vector-valued Problems

Instead of having scalar unknowns at each grid point, we can also formulate a problem with vector-valued unknowns. Please note that the following content is strongly influenced by the introduction of Donea and Huerta to viscous incompressible flows in [19].

Consider the Navier-Stokes equations for modeling the time-dependent flow of a viscous incompressible fluid

$$\left.\begin{aligned} \rho(v_t + (v \cdot \nabla)v) &= \nabla \cdot \sigma + \rho b \\ \nabla \cdot v &= 0 \end{aligned}\right\} \quad \text{in } \Omega \times]0, T[\tag{2.7}$$

with ρ the fluid density, v the fluid velocity, σ the stress tensor, and b the volume force per unit mass of fluid in the domain Ω and in the time interval $]0, T[$. Using Stokes' law

$$\sigma = -p\mathbf{I} + 2\mu\nabla^{\mathrm{S}}v,$$

where $\nabla^{\mathrm{S}} v := \frac{1}{2}(\nabla v + \nabla^{\mathrm{T}} v)$ is the symmetric part of the velocity gradient ∇v and p the kinematic pressure, we can rewrite (2.7) in the velocity-pressure stress-divergence form as

$$v_t + (v \cdot \nabla)v - \nu\nabla^2 v - \nu\nabla(\nabla \cdot v) + \nabla p = b, \tag{2.8}$$

where $\nu = \mu/\rho$ is the fluid kinematic viscosity, μ the fluid dynamic viscosity.

In the case of highly viscous flow, the convective terms in (2.8) can be neglected, leading to the equations of Stokes flow

$$\left.\begin{aligned} v_t - \nu\nabla^2 v + \nabla p &= b \\ \nabla \cdot v &= 0 \end{aligned}\right\} \quad \text{in } \Omega\times]0,T[$$

We can restrict the problem further to model only the steady-state behavior of a viscous incompressible fluid by additionally neglecting the time-dependent term, leading to the steady Stokes problem

$$\left.\begin{aligned} -\nu\nabla^2 v - \nabla p &= b \\ \nabla \cdot v &= 0 \end{aligned}\right\} \quad \text{in } \Omega$$

with the boundary conditions

$$\begin{aligned} v &= v_D && \text{on } \Gamma_D, \\ -pn + \nu(n \cdot \nabla)v &= t && \text{on } \Gamma_N, \end{aligned}$$

where Γ_D is the part of the boundary imposing Dirichlet boundary conditions, Γ_N the part of the boundary imposing Neumann boundary conditions, n the outward normal vector on the boundary, v_D the prescribed velocity on Γ_D and t the "pseudo-traction" on Γ_N.

One way to discretize the Stokes problem is to use the Galerkin formulation, leading to a mixed finite element method. Since a detailed derivation would need the introduction of function spaces and weak formulations, which is out of scope for this thesis, we refer to the literature [19, 51, 21] for further details. The resulting discrete system of equations can be written in block matrix form as

$$\begin{pmatrix} \mathbf{K} & \mathbf{G} \\ \mathbf{G^T} & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{h} \end{pmatrix},$$

where $\mathbf{K}$ is the viscosity matrix, $\mathbf{G}$ is the discrete gradient operator, $\mathbf{G^T}$ is the discrete divergence operator and $\mathbf{u}$ is the auxiliary velocity vector. The vectors $\mathbf{f}$ and $\mathbf{h}$ incorporate the effect of the prescribed velocity v_D on Γ_D and $\mathbf{h}$ additionally includes the effects of the applied body force b and the prescribed traction t on Γ_N. Note that $\mathbf{u}$ has n_{sd} components, where n_{sd} is the dimensionality of the

problem. Furthermore, note the coupling between the velocity and pressure, which is required to model the interactions between the two quantities.

In the sense of the implementation, we no longer have only scalar coefficients in our stencil, but matrix-valued coefficients instead. Thus, the regular scalar multiplication in the stencil sum is replaced by a matrix-vector multiplication. In the 3d case, our vector of unknowns would consist of four components, i.e. $\mathbf{u} = (u_x, u_y, u_z, p)^T$, where u_x, u_y, u_z are the three velocity components and p is the pressure. The stencil would then have a 4×4 matrix at each grid point instead of a scalar coefficient.

2.2. Iterative Methods

After we discretized our problem we obtain a system of linear equations

$$\mathbf{A}\mathbf{u} = \mathbf{f},$$

that we want to solve for $\mathbf{u}$. We use $\mathbf{u}$ to denote the exact solution and $\mathbf{v}$ to denote the approximation. Both are vectors of discrete values residing at the grid points.

There are exact methods to solve such systems of equations, which however are quite costly performance-wise and thus not applicable to large problems. Instead, we resort to iterative methods. The idea of an iterative method is to start with an initial guess $\mathbf{v}^0$ to the solution and successively improve the approximation $\mathbf{v}^k$ until it eventually converges towards the solution.

To assess the quality of our approximation $\mathbf{v}^k$, we introduce two important concepts: the error and the residual.

The error is defined as the difference between the exact solution $\mathbf{u}$ and the current approximation $\mathbf{v}^k$,

$$\mathbf{e}^k = \mathbf{u} - \mathbf{v}^k.$$

It measures how far our approximation is from the true solution. However, since the exact solution $\mathbf{u}$ is unknown, the error cannot be computed directly in practice.

The residual is defined as the difference between the right-hand side $\mathbf{f}$ and the left-hand side evaluated at the current approximation,

$$\mathbf{r}^k = \mathbf{f} - \mathbf{A}\mathbf{v}^k.$$

It measures how well the current approximation satisfies the discrete system. Unlike the error, the residual can be computed at each iteration and is commonly used to monitor the convergence of the iterative method.

The error and the residual are related through the system matrix $\mathbf{A}$,

$$\mathbf{r}^k = \mathbf{A}\mathbf{e}^k.$$

This relationship shows that the error satisfies the same set of equations as the unknown $\mathbf{u}$ if $\mathbf{f}$ is replaced by $\mathbf{r}$. However, a small residual does not necessarily guarantee a small error.

We therefore use the residual as a practical tool to track the progress of the iteration, keeping in mind that it is only an indirect indicator of the actual error.

One of the simplest iterative methods is Jacobi's method. Consider the 1d discrete Poisson problem

$$\begin{aligned} -u_{i-1} + 2u_i - u_{i+1} &= h^2 f_i, \quad 1 \leq i \leq n-1, \\ u_0 &= u_n = 0. \end{aligned}$$

Jacobi's method is then derived as solving the ith equation for u_i and can be written in component form as

$$v_i^{k+1} = \frac{1}{2}\left(v_{i-1}^k + v_{i+1}^k + h^2 f_i\right). \tag{2.9}$$

The new approximation $\mathbf{v}^{k+1}$ relies solely on the previous approximation $\mathbf{v}^k$ and all individual v^{k+1} can be computed independently, which makes Jacobi's method highly parallelizable.

In order to build the connection to our system matrix $\mathbf{A}$, we can express Jacobi's method in matrix form, too. We begin by splitting the system matrix $\mathbf{A}$ into its diagonal part $\mathbf{D}$, the strictly upper part $-\mathbf{U}$ and the strictly lower part $-\mathbf{L}$, such that

$$\mathbf{A} = \mathbf{D} - \mathbf{U} - \mathbf{L}.$$

Substituting this equation in $\mathbf{A}\mathbf{u} = \mathbf{f}$, we obtain

$$\begin{aligned} & (\mathbf{D} - \mathbf{U} - \mathbf{L})\mathbf{u} = \mathbf{f} \\ \Leftrightarrow \quad & \mathbf{D}\mathbf{u} = (\mathbf{L} + \mathbf{U})\mathbf{u} + \mathbf{f} \\ \Leftrightarrow \quad & \mathbf{u} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{u} + \mathbf{D}^{-1}\mathbf{f}. \end{aligned}$$

The last rearrangement of the equation relates to solving the ith equation for u_i, which is exactly what we have seen in the component form earlier.

Defining

$$\mathbf{R}_J = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$$

as the iteration matrix, we can write Jacobi's method in matrix form as

$$\mathbf{v}^{k+1} = \mathbf{R}_J \mathbf{v}^k + \mathbf{D}^{-1}\mathbf{f}. \tag{2.10}$$

We now introduce a simple improvement to Jacobi's method known as weighted (or damped) Jacobi's method. The motivation behind this modification is to accelerate convergence by introducing a relaxation parameter.

In standard Jacobi, the update step from equation (2.10) can be seen as making a full update in the direction suggested by the Jacobi iteration. However, for some problems and properties of the system matrix, it might be beneficial to vary the impact of the update step. This can be done by adding a relaxation parameter $\omega \in (0, 1]$, that is multiplied by the usual update step and added to the previous approximation. Using the component form of standard Jacobi's method (equation (2.9)) as an intermediate result $v_i^* = \frac{1}{2}\left(v_{i-1}^k + v_{i+1}^k + h^2 f_i\right)$, the new approximation for weighted Jacobi's method can be written as

$$v_i^{k+1} = (1-\omega)v_i^k + \omega v_i^*, \quad 1 \leq i \leq n-1.$$

Again, we can write weighted Jacobi's method in matrix form as

$$\mathbf{v}^{k+1} = \mathbf{R}_\omega \mathbf{v}^k + \omega \mathbf{D^{-1}} \mathbf{f},$$

with $\mathbf{R}_\omega$ being the weighted Jacobi's method iteration matrix

$$\mathbf{R}_\omega = (1-\omega)I + \omega \mathbf{R}_J.$$

Additionally, we can express the same iteration using the residual as

$$\mathbf{v}^{k+1} = \mathbf{v}^k + \omega \mathbf{D^{-1}} \mathbf{r}^k.$$

This representation is closer to our actual implementation as we use the residual as part of our termination criterion, too. The optimal choice of ω depends on the properties of the system matrix. It can be found by minimizing the spectral radius of the iteration matrix $\rho(\mathbf{R}_\omega) = \max_i |\lambda_i|$, where λ_i are the eigenvalues of the iteration matrix. This is based on the fact, that the iteration converges if and only if $\rho(\mathbf{R}_\omega) < 1$ [104, 124, 28]. Additionally, the smaller $\rho(\mathbf{R}_\omega)$, the faster is the convergence.

While the convergence behavior of a method can be analyzed using the spectral radius of the iteration matrix, another way is to use matrix norms. A matrix norm is associated with a vector norm. Consider the vector p-norm $||x||_p$ defined by

$$\begin{aligned} ||x||_p &= \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}, \quad 1 \leq p < \infty, \\ ||x||_\infty &= \sup_{1 \leq i \leq n} |x_i|. \end{aligned}$$

The matrix norm induced by the vector norm $||\cdot||_p$ is then given by

$$||\mathbf{A}||_p = \sup_{x \neq 0} \frac{||\mathbf{A}x||_p}{||x||_p}.$$

This leads, among others, to the 2-norm $||\mathbf{A}||_2$, based on the euclidean vector norm $||x||_2$,

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}\ ,$$
$$||\mathbf{A}||_2 = \sqrt{\rho(\mathbf{A}^T\mathbf{A})}\ .$$

For symmetric matrices, the 2-norm is equal to the spectral radius of $\mathbf{A}$,

$$||\mathbf{A}||_2 = \sqrt{\rho(\mathbf{A}^T\mathbf{A})} = \sqrt{\rho(\mathbf{A}^2)} = \rho(\mathbf{A}).$$

There is an important relationship between the error and the norm of the iteration matrix. The error after k iterations is given by

$$\mathbf{e}^k = \mathbf{R}^k\mathbf{e}^0,$$

which leads to the conclusion, that the error vanishes if $||\mathbf{R}|| < 1$ for ongoing iterations. In fact, the spectral radius and the norm of the iteration matrix are related by

$$\rho(\mathbf{R}_\omega) \leq ||\mathbf{R}_\omega||.$$

Consequently, using a vector norm of our choice and its associated matrix norm, the error can be bound by

$$||\mathbf{e}^k|| \leq ||\mathbf{R}||^k||\mathbf{e}^0||.$$

Since the determination of the optimal choice of the relaxation parameter ω is specific to a problem and not always easy to accomplish, in practice near-optimal value is typically used, e.g. $\omega = 0.8$. In addition, Jacobi's method is not used solely but rather as a part of a more sophisticated solver, like the multigrid method.

Numerical methods like Jacobi's method usually only approximate the exact solution up to a certain precision. We need a criterion to determine when our approximation is sufficiently close to the solution. We could either use a fixed amount of iterations, regardless of the achieved accuracy, or we could use a more adaptive approach based on the actual progress of the iteration. A common strategy is to check whether the norm of the residual is below a certain tolerance ϵ, i.e. $||\mathbf{r}^k|| < \epsilon$. In order to scale with the problem size, we can use a relative residual, e.g.

$$\frac{||\mathbf{r}^k||}{||\mathbf{r}^0||} < \epsilon \qquad \text{or} \qquad \frac{||\mathbf{r}^k||}{||\mathbf{f}||} < \epsilon.$$

Note that, as mentioned earlier, a small residual does not necessarily imply a small error. However, in practice it often serves as a sufficient criterion, especially given its ease of computation.

There is much more to convergence and error analysis, that we cannot and will not cover here, as it is not the purpose of this thesis. Further reading can be done in many standard texts like [33, 93, 112].

Until now, we were concerned with only updating one value at a time. In contrast to this *point-wise* scheme, we can adjust the method to update multiple values per step, yielding a *block relaxation* scheme [93]. We agglomerate multiple elements of the vector of unknowns $\mathbf{u}$ and the right-hand side $\mathbf{f}$ into sub-vectors $\hat{\mathbf{u}}_i$ and $\hat{\mathbf{f}}_i$ and divide the system matrix $\mathbf{A}$ into blocks $\mathbf{A}_{ij}$ accordingly,

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1p} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{p1} & \mathbf{A}_{p2} & \cdots & \mathbf{A}_{pp} \end{pmatrix}, \ \mathbf{u} = \begin{pmatrix} \hat{\mathbf{u}}_1 \\ \hat{\mathbf{u}}_2 \\ \vdots \\ \hat{\mathbf{u}}_p \end{pmatrix}, \ \mathbf{f} = \begin{pmatrix} \hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 \\ \vdots \\ \hat{\mathbf{f}}_p \end{pmatrix}.$$

The partitioning of $\mathbf{u}$ and $\mathbf{f}$ must be identical and compatible with the partitioning of $\mathbf{A}$, such that for any vector $\mathbf{x}$

$$(\mathbf{A}\mathbf{x})_i = \sum_{j=1}^{p} \mathbf{A}_{ij} x_j.$$

The purpose of using block relaxation is to overcome strong anisotropy in problems, which can be the result of, for example, a large discrepancy in grid spacings in different directions. Another reason for using block relaxation is to model the coupling between different quantities, e.g. position and charge of a molecule in electrostatics. A common theme is to partition along a whole line of the domain, e.g. aggregating all m unknowns into n vectors for a $m \times n$ grid. This technique is often referred to as *line relaxation*. While the execution of block relaxation methods is more computationally expensive than simple point-wise relaxation, the convergence rate is often higher.

Other standard relaxation schemes include Gauss-Seidel (GS) and SOR (Successive Over-Relaxation). The key difference of GS compared to Jacobi's method is that the most recent values are used in the update step as soon as they are available during the iteration. That is, the computation of $v_i^{(k+1)}$ depends not only on the previous approximation $\mathbf{v}^{(k)}$, but also on the current approximations $v_0^{(k+1)}, v_1^{(k+1)}, \ldots, v_{i-1}^{(k+1)}$. SOR further introduces a relaxation parameter ω that controls the strength of the update step.

While GS and SOR usually have a faster convergence than Jacobi's method, they are not as easy to parallelize, which is a main concern for the implementation optimizations addressed in this thesis. In addition, Jacobi's method is a sufficient method for our multigrid implementation, as it has the appropriate smoothing characteristics, which is discussed in the next section.

2.3. Geometric Multigrid Method

An important characteristic of Jacobi's method is its *smoothing* (or *damping*) property. When applied to problems with error components spanning a wide range of frequencies, Jacobi's method tends to rapidly reduce the high-frequency components of the error within the first few iterations. In contrast, the low-frequency components remain largely unaffected, even after many iterations. This selective smoothing makes Jacobi's method ineffective as a standalone iterative solver, especially for problems requiring convergence across all frequency scales.

However, this limitation can be transformed into a strength when Jacobi's method is used within a multigrid framework. The key idea is to leverage the smoothing property to eliminate high-frequency errors on the fine grid, and then address the remaining low-frequency errors by transferring, or *restricting*, the problem to a coarser grid. On the coarser grid, these low-frequency components appear as higher frequency modes, which are then more effectively reduced using Jacobi's method.

When restricting the problem to a coarser grid, we do not want to relax on the restricted approximation itself, but on the error instead. We are able to do so using the residual equation

$$\mathbf{Ae} = \mathbf{r} = \mathbf{f} - \mathbf{Av}.$$

An argument for relaxing directly on the error is that relaxation on the original equation $\mathbf{Au} = \mathbf{f}$ with an arbitrary initial guess $\mathbf{v}$ is equivalent to relaxing on the residual equation $\mathbf{Ae} = \mathbf{r}$ with the specific initial guess $\mathbf{e} = 0$.

With this in mind, we obtain the following basis for a *correction scheme*.

- Relax on $\mathbf{Au} = \mathbf{f}$ on Ω^h to obtain an approximation $\mathbf{v}^h$.
- Compute the residual $\mathbf{r} = \mathbf{f} - \mathbf{Av}^h$.
- Relax on the residual equation $\mathbf{Ae} = \mathbf{r}$ on Ω^{2h} to obtain an approximation to the error $\mathbf{e}^{2h}$.
- Correct the approximation obtained on Ω^h with the error estimate obtained on Ω^{2h}: $\mathbf{v}^h \leftarrow \mathbf{v}^h + \mathbf{e}^{2h}$.

Here, Ω^h denotes the fine grid with grid spacing h and Ω^{2h} the coarse grid with grid spacing $2h$, i.e. the coarse grid has half the number of grid points as the fine grid per dimension.

With this scheme, there are still two unanswered questions: How can we relax on the error on the coarse grid, and how do we transfer the residual from the fine

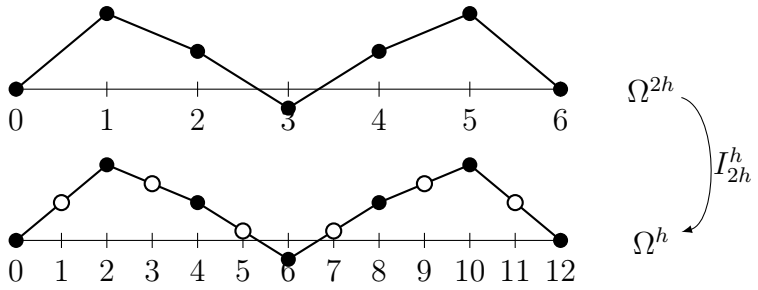


Figure 2.4.: Linear interpolation from coarse grid Ω^{2h} to fine grid Ω^h in 1d

grid to the coarse grid and the error from the coarse grid back to the fine grid? We will tackle the latter question first.

In the multigrid environment it is common for the coarse grids to have half the number of grid points per dimension as the fine grid. Generally, in numerical analysis, the process of finding a point between two other points in space is called *interpolation* or *prolongation*. There are plenty of different interpolation methods, but fortunately, for multigrid, the simple *linear* interpolation is usually sufficient. We will denote the interpolation operator as $P = I_{2h}^h$. Transferring a coarse grid to a fine grid is then denoted by $I_{2h}^h \mathbf{v}^{2h} = \mathbf{v}^h$, where for the 1d case

$$\begin{aligned} v_{2i}^h &= v_i^{2h}, \\ v_{2i+1}^h &= \frac{1}{2}(v_i^{2h} + v_{i+1}^{2h}), \quad 0 \leq i \leq \frac{n}{2} - 1. \end{aligned}$$

Figure 2.4 illustrates the linear interpolation in 1d. Grid points that are present in both grids (even fine grid indices) are simply copied, while grid points that are only present in the fine grid (odd fine grid indices) are computed as the average of their two neighboring coarse grid points.

Interpolation typically works well on smooth functions, but it reaches its limits quickly when the function to be interpolated is highly oscillatory. Figure 2.5 shows how a successful interpolation of a coarse-grid error resembles the exact fine-grid error in contrast to an interpolation of a coarse-grid error that differs heavily from the exact fine-grid error.

Coincidentally, the smoothing property of Jacobi's method ensures that the error after a few iterations is smooth enough for interpolation to work well. Thus, provided that the residual equation on Ω^{2h} can be solved with sufficient accuracy, it is not a matter of concern.

The grid transfer in the opposite direction, i.e. from the fine grid to the coarse grid, is called *restriction*. Again, there are multiple ways to define the restriction

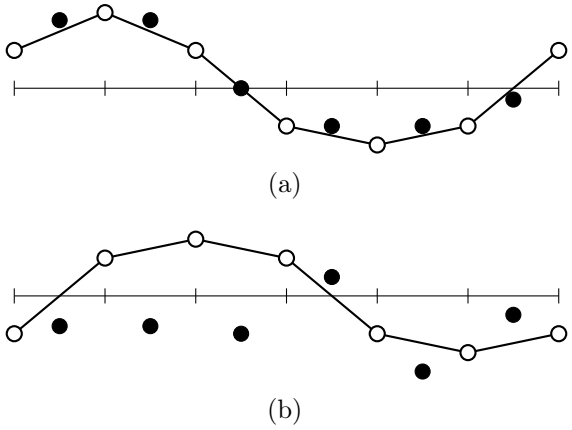


Figure 2.5.: Interpolation of a smooth error (a) and a highly oscillatory error (b)

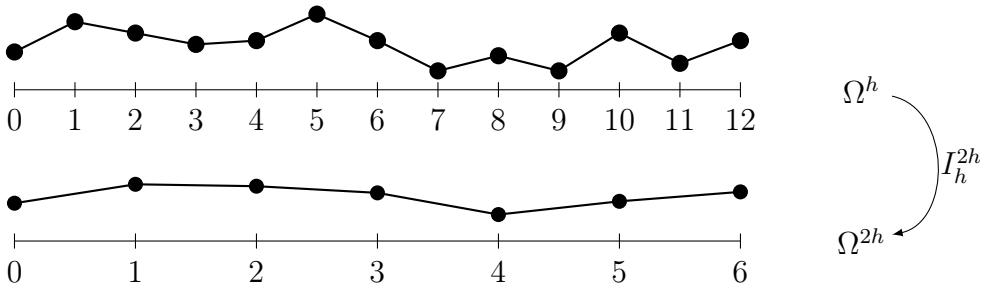


Figure 2.6.: Full-weighted restriction from fine grid Ω^h to coarse grid Ω^{2h} in 1d

operator $R = I_h^{2h}\mathbf{v}^h = \mathbf{v}^{2h}$. A straightforward one is the *full weighted* restriction operator, in 1d defined as

$$v_i^{2h} = \frac{1}{4}(v_{2i-1}^h + 2v_{2i}^h + v_{2i+1}^h), \quad 1 \leq i \leq \frac{n}{2} - 1.$$

Every coarse grid point is computed as the weighted average of the immediate neighborhood of the corresponding fine grid point, which is also illustrated in figure 2.6.

With the choice we made for our restriction operator, the so-called *variational property*

$$I_{2h}^h = c(I_h^{2h})^T, \quad c \in \mathbb{R},$$

is fulfilled.

With our grid transfer operators defined, we can now extend the previously shown correction scheme to the following *two-grid correction scheme.*

- Relax ν_1 times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ on Ω^h with initial guess $\mathbf{v}^h$.

- Compute the fine-grid residual $\mathbf{r}^h = \mathbf{f}^h - \mathbf{A}^h\mathbf{v}^h$ and restrict it to the coarse grid by $\mathbf{r}^{2h} = I_h^{2h}\mathbf{r}^h$.
- Either solve $\mathbf{A}^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h}$ on Ω^{2h} exactly or relax $\nu_1 + \nu_2$ times on $\mathbf{A}^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h}$ with initial guess $\mathbf{e}^{2h} = 0$.
- Interpolate the coarse-grid error to the fine grid by $\mathbf{e}^h = I_{2h}^h\mathbf{e}^{2h}$ and correct the fine-grid approximation by $\mathbf{v}^h \leftarrow \mathbf{v}^h + \mathbf{e}^h$.
- Relax ν_2 times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ on Ω^h with initial guess $\mathbf{v}^h$.

Still, it is unclear, how we can solve the coarse-grid problem $\mathbf{A}^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h}$ effectively. The ability to solve it exactly is not guaranteed, and relaxing on it leaves the question of how to deal with the low frequency modes. Once we realize, that the coarse-grid problem is similarly structured to our fine-grid problem, we might just repeat the process and continue with an even coarser grid Ω^{4h}. In fact, we can define the method recursively, walking over l grids with spacings $h, 2h, \ldots, 2^{l-1}h$. This leads us to the following *multigrid V-cycle scheme*.

$$\mathbf{v}^h \leftarrow V^h(\mathbf{v}^h, \mathbf{f}^h).$$

1. Relax ν_1 times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ with initial guess $\mathbf{v}^h$.
2. $\mathbf{f}^{2h} \leftarrow I_h^{2h}(\mathbf{f}^h - \mathbf{A}^h\mathbf{v}^h)$
3. If Ω^{2h} is the coarsest grid, relax $\nu_1 + \nu_2$ times on $\mathbf{A}^{2h}\mathbf{v}^{2h} = \mathbf{f}^{2h}$ with initial guess $\mathbf{v}^{2h} = 0$,
   Else call $v^h \leftarrow V^{2h}(\mathbf{v}^{2h}, \mathbf{f}^{2h})$.
4. Correct $\mathbf{v}^h \leftarrow \mathbf{v}^h + I_{2h}^h\mathbf{v}^{2h}$.
5. Relax ν_2 times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ with initial guess $\mathbf{v}^h$.

The V-cycle scheme is not the only possible scheme we can think of in multigrid methods. Other common schemes include the W-cycle and the Full Multigrid (FMG) scheme, as depicted in Figure 2.7. While the W-cycle belongs to the same group of schemes as the V-cycle, the so-called μ-cycle methods, the FMG scheme stems from the idea of *nested iterations*. A key difference of FMG compared to the μ-cycle methods is that it starts on the coarsest grid and interpolates the solution to the next finer grid, where it is then improved using a V-cycle. This process is repeated until the finest grid is reached. The FMG scheme is particularly efficient when a good initial guess is not available, as it quickly generates a reasonable approximation on the finest grid, typically in a single iteration, while the V-cycle might require multiple iterations to achieve a good result. On the other hand, implementing FMG is more complex than the V-cycle. Since the single components stay the same for both schemes, our implementation only features the V-cycle, but could be extended to support the other schemes as well.

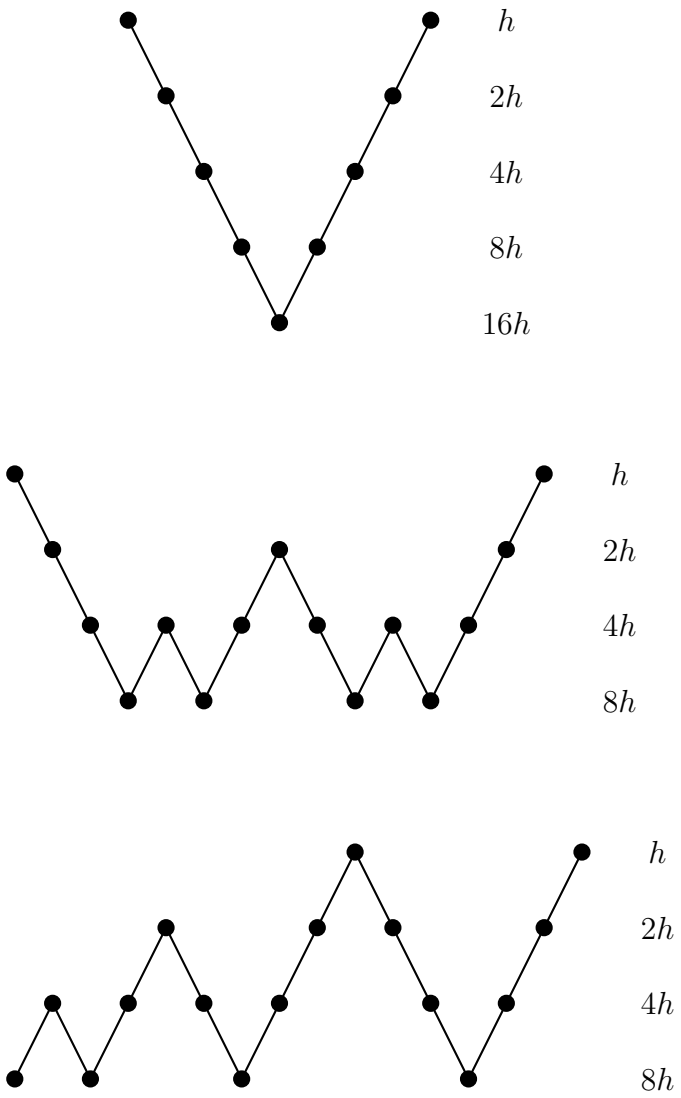


Figure 2.7.: Multigrid schemes: V-cycle (top), W-cycle (middle) and Full Multigrid (FMG) (bottom)

The last essential part of the method we need to discuss is the definition of the coarse grid operator $\mathbf{A}^{2h}$. The first possibility is to rediscretize the original problem on the coarse grid. This is particularly easy for simple problems like the Laplace equation, where the stencil coefficients are fixed and where we only need to adjust the stencil factor according to the respective grid spacing h. However, for more complex problems, e.g. with spatially varying coefficients, we are more flexible using the *Galerkin approach*, which defines the coarse grid operator as a projection of the fine grid operator onto the coarse grid using the restriction and interpolation operators defined earlier,

$$\mathbf{A}^{2h} = I_h^{2h} \mathbf{A}^h I_{2h}^h. \tag{2.11}$$

In this work, we mainly focus on the full-coarsening approach, i.e. the coarse grid Ω^{2h} has half the number of grid points than the fine grid Ω^h *in each dimension*. This is what is implemented in our code as well. However, there are other coarsening strategies, like *semi-coarsening*, where only one dimension is coarsened at a time. This can be beneficial for problems with strong anisotropies, e.g. when the grid spacing in one dimension is significantly larger than in the other dimensions, especially when paired with line or plane smoothers. However, semi-coarsening is more computationally expensive when dealing with isotropic problems, as it requires more levels to reach the coarsest grid.

2.4. High Performance Computing

Traditionally, involved computations occurring e.g. in solving large scale modeling problems are done in clusters – collections of interconnected nodes that work together as a single system. Each node typically contains multiple processors (CPUs), memory, and storage, and the nodes are connected via high-speed networks. Clusters allow parallel execution of tasks, where a large problem is divided into smaller sub-problems that are solved concurrently, greatly speeding up the computation.

In recent years, Graphics Processing Units (GPUs) have become central to high performance computing (HPC). Originally designed for rendering images and video, GPUs excel at parallel processing due to their thousands of smaller cores. The rise of CUDA and OpenCL has made it easier for developers to harness GPU power for general-purpose computing, known as GPGPU (General-Purpose computing on GPUs).

While a CPU is capable of executing a wide variety of commands, serving as a general purpose processor, a GPU has a narrower and more specific instruction set. While a single GPU core is typically slower than a CPU core on its own, the higher amount of cores in the GPUs allows for massive parallelization and can thus execute the same task on a large data set more efficient than a CPU.

Beyond CPUs and GPUs, Field-Programmable Gate Arrays (FPGAs) are another important architecture in HPC. FPGAs are reconfigurable chips that can be programmed at the hardware level to execute specific tasks with extremely high efficiency and low power consumption. Usually FPGAs require more specialized knowledge to program, using hardware description languages like VHDL or Verilog, however since we are using OpenCL for our implementation, we could target these devices as well.

Over time, different frameworks and platforms with different abstraction levels have emerged for targeting GPUs.

Developed by Nvidia, CUDA [78] is the most widely adopted GPU programming framework. It provides a rich set of libraries, tools, and APIs tailored specifically for Nvidia GPUs. The main drawbacks are a) the dependency on Nvidia and b) the restriction to only use Nvidia GPUs. As the core of the library has a low abstraction level, the user has fine-grained control over the execution of kernels and data management. However, mechanisms exist that allow for writing more abstract code and letting the framework handle the details. One example is Unified Memory, with which the user does not need to distinguish between host and device memory. All necessary data movement is handled automatically. While this might improve application development speed, certain optimizations requiring custom data management cannot be done.

Created by AMD, ROCm [3] is an open-source alternative to CUDA designed for AMD GPUs, particularly in HPC and data center environments. While programs written using ROCm can target only AMD GPUs, AMD has released the HIP (Heterogeneous-compute Interface for Portability) API, a C++ runtime API and kernel language that allows developers to create portable applications running in heterogeneous systems, using CPUs and AMD GPUs or Nvidia GPUs from a single source code. HIP provides a simple marshalling language to access either the AMD ROCm backend, or Nvidia CUDA backend, to build and run application kernels.

For Intel GPUs, Intel has created oneAPI [47], which aims to define a common unified and open multi-architecture multivendor software platform. So one cannot just target Intel GPUs but also other vendors and device types. On the one hand, it offers high level library interfaces that are commonly used in a variety of workflows. For example, oneAPI Math Kernel Library (oneMKL) [46] is an open, stable API that enables users to leverage common, parallelizable, mathematical operations. On the other hand, it provides libraries for low-level programming. On the lowest end, we have Level Zero [45], where the user must manually manage devices, command queues, memory management etc. One step higher there is SYCL (provided by the Khronos Group) [108], which allows the user to define kernels running on different devices using C++ lambda expressions. While SYCL

itself defines the interface, the Khronos Group does not provide a compiler. Intel's answer to that is DPC++ [44], which is not only a SYCL compiler, but also an extension to the default SYCL interface, designed to specifically target Intel devices.

Lastly, maintained by the Khronos Group, there is OpenCL, which we use in our implementation [106]. OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse hardware, including CPUs, GPUs, FPGAs, and other accelerators. Unlike CUDA or ROCm, which are tied to specific vendors, OpenCL is designed for portability and interoperability. Fundamentally, OpenCL offers a low-level C-API, giving the user the ability to control all data movement, memory management and kernel executions in detail. While using this API can become cumbersome and error-prone in a larger application, it can also be used from C++ source code, allowing to use object-oriented programming paradigms, like RAII (Resource Acquisition is Initialization), where appropriate. Furthermore, there is an official C++ API that acts as a wrapper for the C-API. Many other languages like Python or Julia also have packages that attach to the OpenCL API, providing bindings to OpenCL. We take a closer look into the ecosystem of OpenCL in section 3.1.

We briefly covered the mostly low-level frameworks from the different GPU manufacturers Nvidia, AMD and Intel. Note that there are other libraries built on top of these frameworks, allowing the user to focus more on the application itself and less about the management of the underlying devices. While not further relevant in this thesis, we want to at least mention Kokkos by Sandia National Laboratories [13], RAJA by Lawrence Livermore National Laboratory (LLNL) [6] and Thrust by Nvidia [7].

When computing solutions to large problems, we typically do not use just one GPU, but several GPUs on several nodes. For both intra- and inter-node communication, MPI (Message Passing Interface) is the de-facto standard, which we also use in our implementation. In order to send data between nodes, we need to transfer the data from our GPU to the host, to the other node and back to the GPU again. While with OpenCL we need to do this manually, CUDA has a feature called CUDA-aware MPI [59], that manages this process automatically, as well as enabling the usage of Remote Direct Memory Access (RDMA) for inter-node communication and GPUDirect Peer-to-Peer (P2P) for intra-node communication. Its performance depends on the size of data that is being sent [89] and its layout, e.g. strided memory accesses might lead to packing each strided element individually, resulting in high amounts of data movement between host and device [111].

2.5. Other Implementations

Before diving into the details of our implementation, we want to mention other existing solutions. While many multigrid implementations employ the GPU, most of them operate on fixed coefficient stencils only. As we are mainly interested in solving linear systems including stencils with varying coefficients, we limit our scope to implementations that can manage these varying coefficients.

Probably most notably, there is HYPRE, developed by Lawrence Livermore National Laboratory [23]. HYPRE is a library of different high performance parallel multigrid methods for the solution of large, sparse linear systems of equations. While being famous for its algebraic multigrid method implementation BoomerAMG, it also offers geometric multigrid method solvers operating on structured or semi-structured grids. Specifically, PFMG [5, 22] is a parallel semi-coarsening multigrid solver using point-wise smoothing, suitable for linear systems arising from the discretization of the diffusion equation

$$\nabla \cdot (D \nabla u) + \sigma u = f.$$

This closely resembles our implementation, except for the semi-coarsening aspect, as we employ full coarsening instead. Thus, it makes a good candidate for comparison regarding the performance of our implementation later. Note that, while HYPRE allows the definition of domains using different boxes of varying sizes, which gives the user more flexibility, we restrict ourselves to rectangular domains.

Another implementation was part of a version of the framework Walberla (Widely applicable Lattice-Boltzmann from Erlangen), but which has never been published. The only remnant of this implementation is a publication from 2013 [57]. Thus, unfortunately, we cannot compare our implementation with this one.

Lastly, there is HPGMG [38] developed by LLNL, a benchmark solving variable-coefficient elliptic problems using finite volume method and Full Multigrid (i.e. F-cycle) aiming for ranking machines on the Top500 list. While it solves the same kind of problem as our implementation, the discretization method and choice of cycle-type are too far away from our implementation to make a fair comparison. In addition, we were unable to successfully run the version of HPGMG that enables the usage of GPUs using CUDA (hpgmg-cuda [96, 95]).

Chapter 3

GPU Architecture and OpenCL

In this chapter, we investigate the architecture of GPUs and the OpenCL ecosystem. We begin with an overview of GPU hardware design, highlighting the fundamental parallelism structures and memory hierarchies that make GPUs well-suited for high-throughput computation. Then, we explore the OpenCL programming framework, including its platform, execution, and memory models, which provide a unified interface for heterogeneous computing. Finally, we discuss best practices for optimizing OpenCL applications on GPU devices and outline analysis tools for profiling and performance tuning.

3.1. OpenCL Overview

OpenCL (Open Computing Language) is an open, royalty-free standard designed for cross-platform parallel programming across a wide range of accelerators, including those in supercomputers, cloud servers, PCs, mobile devices, and embedded systems. Developed and maintained by the Khronos Group, OpenCL boosts application performance by leveraging the parallel processing capabilities of modern hardware.

OpenCL is broadly adopted across the tech industry. Many semiconductor manufacturers include OpenCL support with their chips. Thus, although we are mainly interested in optimizing our application for the GPU, OpenCL enables us to maintain flexibility should we choose to use alternative hardware in the future, e.g. DSPs or FPGAs. However, usually different architectures require different kinds of optimizations to achieve good performance, so the code would still require adaption to the new hardware in order to fully exploit its capabilities.

3.1.1. History of OpenCL

There is extensive literature available on introductions to OpenCL. Munshi et al. [73] provide a very thorough guide for programming with OpenCL 1.1, covering all the features available up to this version, as well as case studies.

Kaeli et al. [52] delve into different hardware architectures and their approaches to parallelism, before tackling OpenCL itself. The features introduced in OpenCL 2.0 are explored in detail, along with a comparison of how they differ in use from those available in version 1.2.

McIntosh-Smith and Deakin [66] provide a comprehensive, practical approach with their two-day lecture course *Hands On OpenCL*, in which concepts and exercises are introduced progressively using OpenCL 1.2.

Lastly, coming directly from the Khronos Group, the *OpenCL Guide* [107] serves as a quick-start reference to get started with OpenCL programming. It covers the essential concepts and offers links for further reading, especially for features introduced in newer versions.

Version 1.0 of the specification was initially released in 2008 and implemented by many vendors. Version 1.2 brought important improvements and additions, such as

- **Device Partitioning**, allowing to split a single physical device into multiple logical sub-devices, enabling improved workload management and finer-grained scheduling control,
- **Separate Compilation and Program Linking**, mirroring the way C and C++ programs are typically developed – by allowing modular compilation and linking of kernel code, and
- **Zero Copy**, allowing a host buffer and a device buffer to share the same physical memory, eliminating extra copies during execution. This is especially useful for applications using a CPU or an integrated GPU as accelerator, which do not have distinct memory spaces [61].

OpenCL 2.0 introduced several new features [42], aiming to simplify programming and improve performance, such as

- **Shared Virtual Memory (SVM)**, which allows the host and device to share pointers and memory addresses, removing the need for explicit memory transfers between host and device (much like CUDA's Unified Memory feature [37]),

- **Device-side command queues**, enabling a child kernel to be directly enqueued by a parent kernel running on a device. This enables nested parallelism – a parallel programming paradigm allowing one thread to launch several additional threads. This is particularly useful for hierarchical algorithms where the number of threads to be launched is not known at the start of execution.

- **Generic Address Space**, which allows functions to operate on any address space, leveraging the reusability of code.

- **C11 Atomics**, allowing a series of operations to be executed safely, ensuring that no other thread can modify the data in between.

While the features introduced in OpenCL 2.0 are very useful in certain scenarios, they got poorly adopted by vendors, especially by Nvidia. Since Nvidia dominated the GPGPU market for a long time, it was impractical to develop an application targeting only OpenCL 2.0, as this would either exclude all Nvidia devices or require relying on Nvidia to implement the missing features at an unspecified time in the future. If the applications was built to run mainly on GPUs, the more reasonable approach was to remain with OpenCL 1.2. This applies to our implementation, too.

One reason for the lack of adoption of OpenCL 2.0 features was the growing diversity of architectures implementing the OpenCL standard, making it difficult to guarantee feature completeness across all devices. This issue was solved with the release of OpenCL 3.0 in 2020, building up on version 1.2 and making all features of version 2.x optional. This allows vendors to concentrate on developer-requested functionality relevant to their target markets. Whether a feature is supported by a device can be queried at runtime. Having a common baseline and the ability to extend on top of it was a welcoming situation for Nvidia, finally conforming to OpenCL 3.0 [72].

3.1.2. OpenCL Programming Model

OpenCL's programming model can be divided into three parts: the platform model, the execution model, and the memory model.

The **Platform Model** defines how OpenCL views the compute resources available in a system and how they are connected from a topological perspective. A typical setup consists of a host that interfaces with one or more *compute devices*. These devices are further broken down into *compute units*, which in turn are composed of multiple *processing elements*, see figure 3.1.

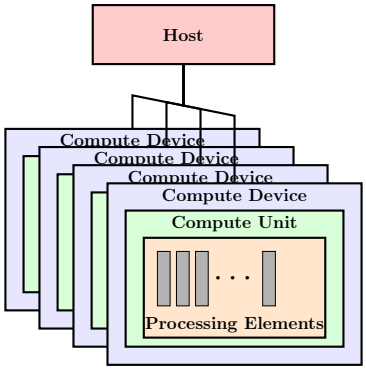


Figure 3.1.: OpenCL Platform Model. *Source: Based on the image from:* `https://github.com/KhronosGroup/OpenCL-Guide/blob/main/images/platform_model.jpg`

Code execution takes place on the processing elements and typically follows either the SIMD (Single Instruction, Multiple Data) or SPMD (Single Program, Multiple Data) execution models.

In context of GPUs, a compute device corresponds to a GPU. The compute units map to the streaming multiprocessors (SMs) of the GPU, while the processing elements correspond to the individual streaming processors (SPs) within each SM. This grouping is an implementation choice aimed at efficiency. By organizing processing elements into compute units, vendors can share instruction dispatch logic and memory resources among them, while also enabling faster communication between processors on the same unit.

The **Execution Model** focuses around the command `clEnqueueNDRangeKernel`, which allows a kernel to be launched in parallel over an N-dimensional range of data. For example, when working with a two-dimensional grid, the `ndrange` corresponds to the grid dimensions, and each grid point is processed by a separate *work-item* – an individual instance of the kernel running on a single processing element.

As described earlier, processing elements are typically organized into compute units to enhance execution efficiency. When invoking `clEnqueueNDRangeKernel`, the developer also specifies a *work-group* size, which defines how work-items within the `ndrange` are grouped together. Work-items within the same work-group share local memory, synchronize through work-group barriers, and take advantage of cooperative functions like `async_work_group_copy`, which are only available for work-items in the same work-group. The concept of the execution model is visualized in figure 3.2.

The **Memory Model** is a hierarchy consisting of different memory types (see figure 3.3):

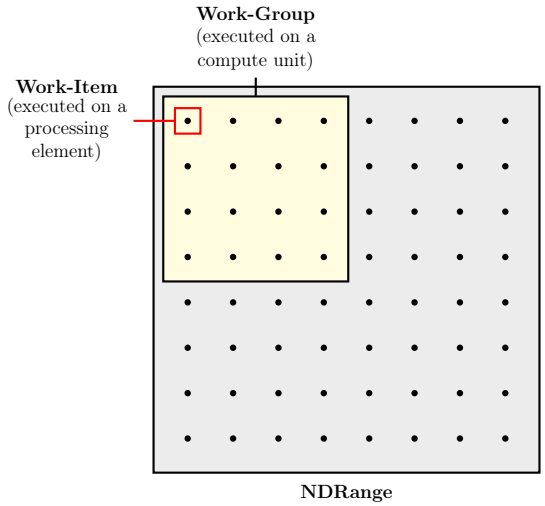


Figure 3.2.: OpenCL Execution Model. *Source: Based on the image from:* https://github.com/KhronosGroup/OpenCL-Guide/blob/main/images/ndrange.jpg

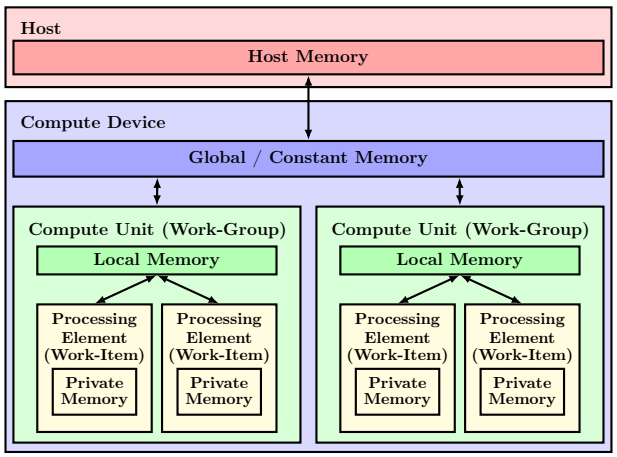


Figure 3.3.: OpenCL Memory Model. *Source: Based on the image from:* https://github.com/KhronosGroup/OpenCL-Guide/blob/main/images/memory_model.jpg

- Host memory refers to the memory accessible by the host CPU, e.g. the main system RAM in consumer computers.
- Global/Constant memory, available to all work-items across all compute units, is typically the largest memory space of a device. Sometimes referred to as *off-chip memory*.
- Local memory is restricted to the work-items of one work-group only. Often referred to as *on-chip memory*, on GPUs it is located directly on a SM and thus has a lower latency, but also a smaller capacity than global memory.
- Private memory is only accessible by a single work-item. It typically consists of a small set of registers, which are physically shared among the whole SM, but are logically mapped to only a single work-item at the same time. This is the fastest available memory type, used for intermediate results and local variables.

Traditionally, OpenCL memory management is explicit. None of the memory types are automatically synchronized, so the application must explicitly manage data transfers between them as needed. This behavior changes when using the aforementioned SVM, which automatically copies memory between the host and device as needed.

3.1.3. Hardware Implementation / GPU Architecture

Now that we have set the scene of the fundamental OpenCL environment, we will investigate, how the abstract concepts map to real hardware. This is essential, as maximum performance needs full exhibition of the hardware capabilities, which in turn requires at least a basic understanding of the hardware design.

In the following we are going to focus on Nvidia GPUs, as we use Nvidia's hardware and profiling tools to test and analyze our implementation. However, the fundamental concepts apply to other vendors' hardware, too. More specifically, we use a Quadro T2000 GPU (Turing architecture, CC 7.5) and A100 GPUs (Ampere architecture, CC 8.0). In order to avoid confusion, table 3.1 compares the terminology used in OpenCL and CUDA. Since we are using OpenCL for our implementation, we primarily use OpenCL terminology in the following sections. An exception is made when describing hardware details, where Nvidia's terms are more appropriate. Additionally, due to the ambiguity of the term local memory, whenever we use it, OpenCL local memory is meant, i.e. CUDA shared memory. In cases where CUDA local memory is meant, we will precede it with CUDA. The term "shared memory" will be used only in the context of Nvidia's GPU architecture.

OpenCL	Nvidia CUDA
Compute Unit (CU)	Streaming Multiprocessor (SM)
Work-Item	Thread
Work-Group	Block
`ndrange`	Grid
Command Queue	Stream
Global Memory	Global Memory
Constant Memory	Constant Memory
Local Memory	Shared Memory
Private Memory	Registers / Local Memory

Table 3.1.: Comparison of OpenCL and Nvidia CUDA terminology

Nvidia GPUs are hierarchically categorized. The **architecture** represents the top level and defines the underlying design of the GPU. Architectures include e.g. Volta (2017), Turing (2018), Ampere (2020) etc. Each architecture has one or more **GPU chips** (dies) targeting different markets. For example for the Ampere architecture, GA100 targets data centers, GA102 targets high-end gaming and workstations, GA104 targets mid-range consumer setups and GA10B targets the embedded landscape. The chip determines the amount of compute units (SMs), presence of tensor cores, power and performance envelope etc. Based on the same chip, multiple **product lines** are released, varying in configuration (e.g. disabled SMs or lower clocks). Products based on the GA100 for example include the A100 40GB and the A100 80GB, both either with PCIe or SXM4 interface. Finally, each architecture has an associated **compute capability** (CC) version number of the form X.Y. GPUs having the same major revision number X share the same underlying fundamental architecture. The minor revision number Y is used to determine incremental improvements to the core architecture, including new features. For example, the Turing architecture has CC 7.5 and is based on the Volta architecture with CC 7.0. The Ampere architecture has CC 8.0. Table 3.2 shows an excerpt of the device attributes for the A100 and Quadro T2000 GPUs.

3.1.3.1. General GPU Parts

According to their whitepapers [81, 85], both the Turing and Ampere architectures consist of multiple Graphics Processing Clusters (GPCs). Each GPC includes multiple Texture Processing Clusters (TPCs) and each TPC includes multiple Streaming Multiprocessors (SMs, i.e. Compute Units). Each SM contains 64 Cuda Cores (i.e. Processing Elements) and further parts, that will be described in the following section. Figure 3.4 shows a full view of the Ampere GPU architecture as an example.

Device	A100	Quadro T2000
Architecture	Ampere	Turing
Chip	GA100	TU117
DRAM	40 GB	4 GB
CC	8.0	7.5
SMs	108	16
L1 Cache per SM	192 KB	64 KB
L2 Cache	40 MB	1 MB
Cores per SM	64	64
Cores Total	6912	1024
float/double ratio	1:2	1:32
Schedulers per SM	4	4
Warps per scheduler	16	8
Blocks per SM	32	16
Registers per thread	255	255
Registers per block	65536	65536
Registers per SM	65536	65536
Shared Mem per block	48 KB	48 KB

Table 3.2.: Excerpt of device attributes for A100 and Quadro T2000 GPUs

The GPU is connected to DRAM (multiple stacks of HBM2 memory for the A100, GDDR5 for the Quadro T2000) and multiple memory controllers. Each SM shares a global L2 cache, that is unified for data, instructions and constant memory.

3.1.3.2. Streaming Multiprocessor Parts

Each SM is partitioned into 4 sub partitions (sometimes called processing blocks) [80]. Each sub partition contains the following units:

- One warp scheduler
- Register file segment
- A dispatch unit
- Execution Units, Pipelines and Cores
  - Integer Execution Units
  - Floating Point Execution Units
  - Memory Load/Store Units
  - Special Function Unit

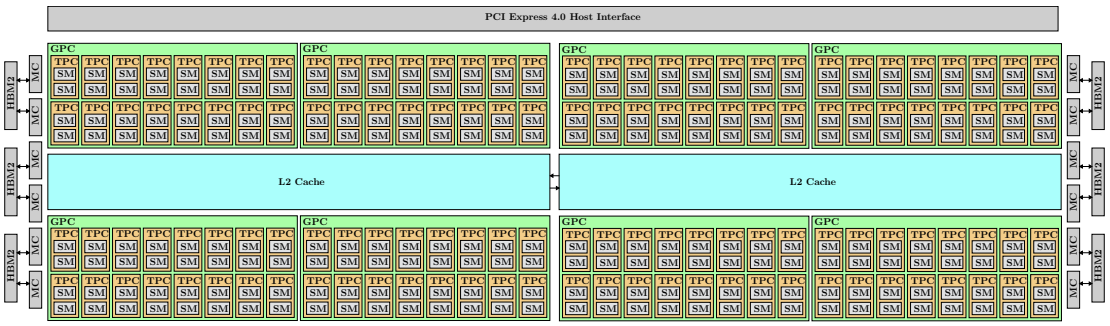


Figure 3.4.: Full view of Nvidia's GA100 GPU architecture (Ampere) [81]. MC: Memory Controller

– Tensor Cores

All four sub partitions share a unified L1 data cache, texture units and ray-tracing cores. Since the Volta architecture, which is the predecessor of Turing, the L1 cache can function as an automatic cache for memory accesses, as well as storage for local memory. Figure 3.5 shows a full view of one SM of the Ampere GPU architecture as an example.

3.1.3.3. Compute Model

When a kernel is launched, a work-group (also called Cooperative Thread Array (CTA) in context of Nvidia's architectures) is assigned to a single SM, i.e. it is guaranteed, that each work-item of a work-group will execute on the same SM. The work-items of a work-group can share memory using the shared memory space and synchronize using barriers. A work-group is further partitioned into one or multiple *warps*, which are groups of 32 work-items. If a work-group cannot be divided by 32, the last warp contains the remaining work-items. Note that, on AMD devices, a warp is called *wavefront* and typically consists of 64 work-items.

The number of work-groups that fit onto a single SM is limited by the physical resources of the architecture. Thus, it differs for different architectures. More specifically, limiting factors are

- the amount of warps per SM,
- the amount of work-groups per SM,
- the amount of registers per SM and
- the amount of shared memory per SM.

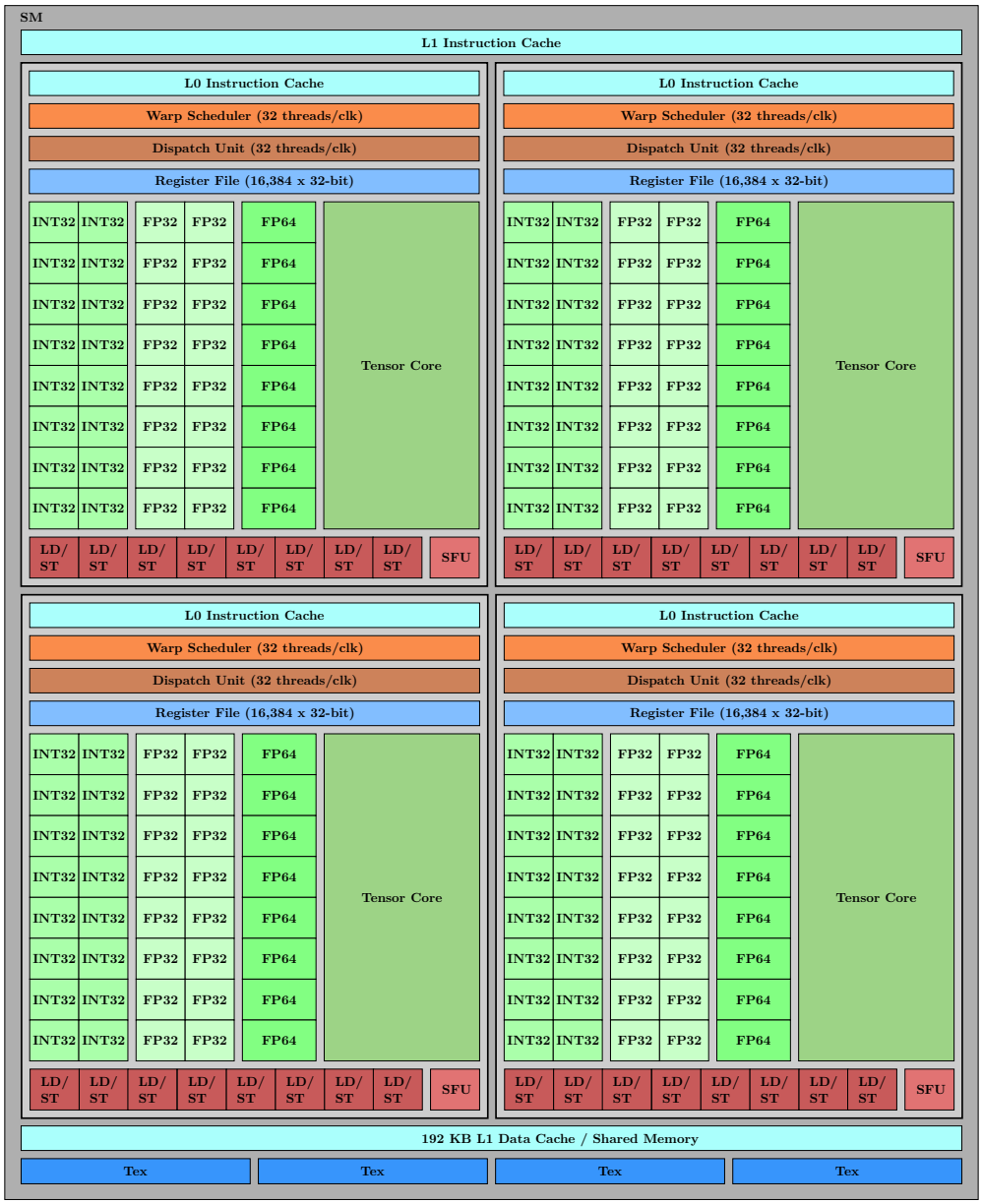


Figure 3.5.: View of one streaming multiprocessor (SM) of Nvidia's GA100 GPU architecture (Ampere) [81]

The ratio between active warps to maximum warps per SM is called *Occupancy*. While a high occupancy does not necessarily lead to high performance, a very low occupancy typically also leads to poor performance. Thus, when choosing the launch parameters for a kernel, the limiters listed above should always be kept in mind. This is further elaborated in section 3.2.1.

While work-items inside a work-group can be cooperative, work-items across multiple work-groups cannot. There is no guarantee in the order of execution, thus each work-group must be able to work independently of the others.

The total number of work-groups that can be run concurrently on a GPU is called a *wave*. If there is less than one full wave launched for a kernel, it indicates that the problem is too small for the GPU to be utilized fully. If there is a very low number of waves combined with partial waves, a kernel might suffer from the *tail effect*, which is shortly discussed in section 3.2.7.

3.1.3.4. Warp Scheduling

Multiple SMs together form a MIMD system according to Flynn's taxonomy [26], i.e. each SM executes its own instruction stream on different data independently.

The work-items are executed following the *SIMT (Single-Instruction, Multiple-Thread)* execution model, which allows each work-item to take a unique execution path albeit still being executed as part of a warp. While this means, that branching at the work-item level is possible, a warp is still executed as a unit, meaning all work-items not taking the same path are stalled. Thus, branch diversion should be minimized to achieve high performance. Alternatively, an instruction might be predicated, meaning it will only execute on work-items where the predicate is true. The choice between predication and branching is usually made by the compiler.

Since each thread has its own instruction address counter and register state, a warp scheduler can quickly switch between active warps, depending on the state of each warp. This feature is essential to enable *latency hiding*: While CPUs rely on out-of-order execution and branch prediction to maximize performance, GPUs hide memory latencies by executing eligible warps while others are stalled. This is the reason, why a GPU excels at problems with huge data sets, but the benefits diminish if a problem cannot be divided to a sufficiently fine granularity.

A warp is assigned to a sub partition, and thus to a warp scheduler, for its whole lifetime, presumably following the simple rule $\text{scheduler_id} = \text{warp_id} \mod 4$ [49]. A warp is considered *active* when it is resident on a SM and is *eligible*, when it is ready to issue an instruction, i.e. when the instruction is decoded and all input dependencies are resolved. Typical stall reasons that prevent a warp from being

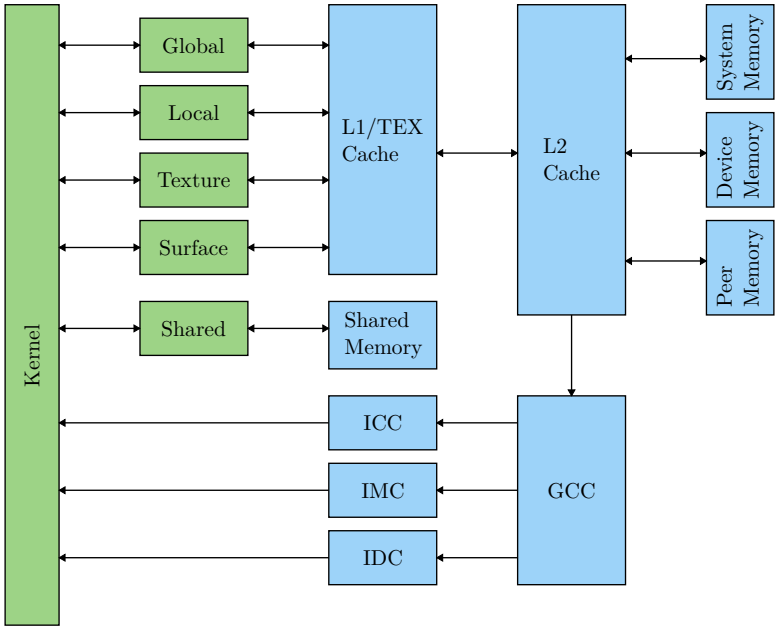


Figure 3.6.: Turing GPU memory overview (CC 7.5) as shown in Nsight Compute. Logical units are displayed in green, physical units in blue. GCC: GPC Constant Cache, ICC: Instruction Constant Cache, IMC: Immediate Constant Cache, ICC: Indexed Constant Cache.

eligible include the warp waiting on an instruction fetch, a memory dependency, an execution dependency or a synchronization barrier.

3.1.3.5. Memory and Cache

Figure 3.6 gives an overview of the hardware implementation of the different kinds of logical memory spaces, that were already introduced in the introduction to OpenCL (section 3.1.2).

Global Memory. On Nvidia GPUs, global memory is a 49-bit virtual memory space, that maps to the physical memory on the device (GDDR6 DRAM in our case), pinned system memory (i.e. memory on the host) or peer memory (i.e. memory of other connected compute devices). Any request to global memory goes through L1 cache and L2 cache before accessing the device memory. Although the diagram separates L1 cache and shared memory, both share the underlying physical memory in the Volta and later architectures. Data in the L1 cache is organized as 128 byte cache lines, each composed of four 32 byte sectors. Each cache line has a tag that is checked when a request arrives, in order to determine if it is a hit or a miss. Furthermore, each individual sector has its own status

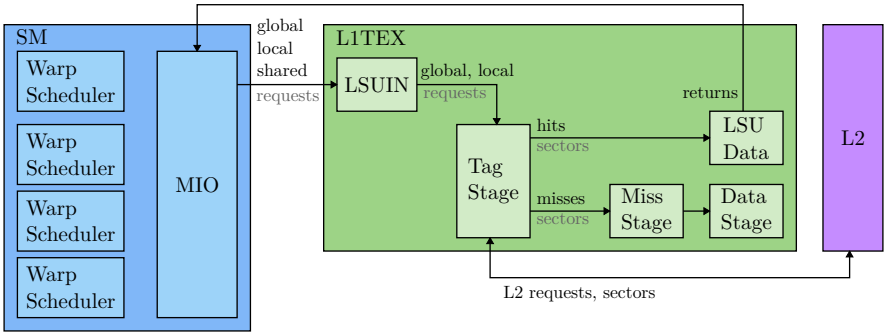


Figure 3.7.: Workflow of the L1 cache on a global load on GA100

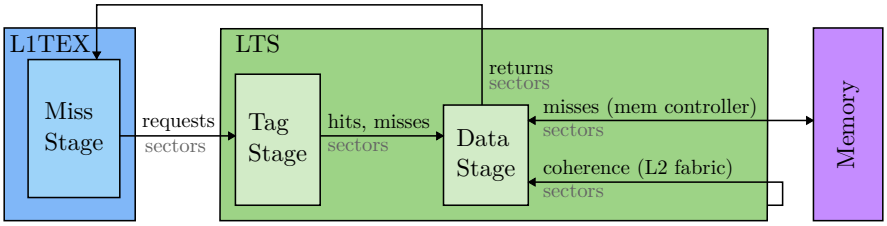


Figure 3.8.: Workflow of the L2 cache on a global load on GA100

bits to indicate whether the sector is considered valid, dirty etc [105, 90, 116]. Note that the details of how the cache exactly works is kept private by Nvidia, since it can vary between architectures and compute capabilities, thus some of the information must be taken from forum posts, preferably by Nvidia employees, or papers in which certain architectures are dissected, e.g. [12, 68]. It appears, that the L1 cache is realized as a set-associative LRU cache for older architectures and a proprietary implementation of an LRU-like cache for recent architectures.

The typical workflow of the L1 cache pipeline is shown in figure 3.7. When a SM issues a **global load** instruction, the memory-input/output (MIO) unit sends a request to the LSUIN (Load Store Unit Input) unit, which forwards the request to the tag stage. All affected sectors inside cache lines whose tag match, i.e. a cache hit, can be returned immediately by the LSU Data unit (part of the Data Stage). If it is a cache miss, the request is sent to the L2 cache via the Miss Stage.

The L2 cache workflow is shown in figure 3.8. Just like with the L1 cache, in the L2 cache, a request passes through a Tag Stage and a Data Stage, which either returns valid sectors back to the L1 cache or forwards the request to an associated memory controller to fetch the data from DRAM. In addition to the caching functionality, L2 also provides mechanics to perform compression and global atomics. An underlying L2 fabric ensures coherence across all SMs on the GPU.

An important quantity for L1 is the *wavefront*. It is the maximum unit that can pass through a pipeline stage per cycle and thus indicates the number of cycles needed to satisfy a request. If not all sectors can be accessed in a single wavefront, multiple wavefronts are created and processed sequentially. Another way to describe a wavefront, as done in [80], is to view it as a unique work-package that is generated for a single request. All work that is part of a single wavefront is performed in parallel, all remaining work is performed in further wavefronts. Thus, the overall work can be viewed as being performed in one or several sequential "waves". This is, however, not to be confused with the *wave* concept introduced in section 3.1.3.3, which uses the same metaphor, but for describing the total number of work-groups that can be run concurrently on a GPU.

Another key metric is the ratio of sectors to requests. It gives a perception of how efficient the memory access pattern is, which will be investigated thoroughly in section 3.2.5.

CUDA Local Memory. Each thread has its own private memory space called local memory in CUDA. This is not to be confused with the the private memory concept of OpenCL, which also functions on a per-thread level. When a thread uses private memory, e.g. in the form of local variables, the driver tries to satisfy the request using registers. If, however, there is more private memory requested than registers available per thread, the memory access is *spilled* to global memory. In CUDA landscape, these register spills are mapped to CUDA local memory, which resides in the DRAM, just like global memory, too. Thus, when CUDA local memory is requested, the address is translated to global virtual address. Furthermore, accesses to local and global memory share the same latency. One key difference is that CUDA local memory is organized in 32-bit words, such that consecutive threads access consecutive memory locations. This leads to full coalescing, if all threads in a warp access the same relative address, e.g. the same index of an array or the same member of a struct.

Shared memory. Shared memory (OpenCL local memory) partially shares its physical memory with the L1 cache, i.e. it is often referred to be located *on chip*. Thus, it has a much higher bandwidth and much lower latency than global memory. It can be used by all work-items across a work-group through the usage of a local barrier (`__syncthreads()` in CUDA, `barrier(CLK_LOCAL_MEM_FENCE)` in OpenCL).

Shared memory is organized as 32 banks, such that successive 32-bits words can be served simultaneously. If multiple threads in a warp access different data in the same bank, a bank conflict occurs and the access has to be serialized. If, however, multiple threads access the same 32-bit word in the same bank, the data is broadcasted to each of the accessing threads. The concept of bank conflicts and how to avoid them is described in more detail in section 3.2.6.

Although the usage of shared memory can still be beneficial with modern GPUs, its importance for achieving high performance has decreased since it has been fused with the L1 cache in the Volta architecture. Many kernels yield a good performance by just relying on the automatic caching mechanisms instead of manually caching data in shared memory. According to a test of Nvidia, in which shared memory loads were replaced by global memory loads for a suite of programs, using global memory only results in a performance drop of only 7 %, while with the prior architecture Pascal it was 30 % [84]. Using shared memory in a sophisticated way adds remarkable complexity to the code, which is why it should be weighed up carefully, whether the benefits outweigh the additional implementation costs.

3.1.4. OpenCL Workflow

The OpenCL framework can be dissected into two APIs: The *Platform Layer API* and the *Runtime API*.

The Platform Layer API is used to initialize the OpenCL runtime and to query the available devices and its extensions. It enables execution on different types of devices, even inside the same program. Each device belongs to one *platform*, which usually corresponds to the backend installed on the system. A platform can have multiple devices. For example, a system might show an Intel platform having the Intel CPU as compute device as well as an Nvidia platform having two Nvidia GPUs as compute devices.

After a device has been selected, the Runtime API is used to compile and enqueue kernels to be run on the device, allocate device memory buffers, synchronize between host and device etc. At first, a *context* must be created, to which further components are assigned to, see figure 3.9. A context can be shared across multiple devices, such that each device gains access to all buffers that belong to the context.

Next, the kernels are compiled into one or more *program* objects, which are needed to run the kernels on the device. The creation of a program object might either happen from scratch, i.e. JIT-compiling source code, or from a precompiled binary. While compiling from scratch takes more time than building the program from a precompiled binary, the latter one can only be used for the specific device that it was compiled for. Therefore, it is important to offer end users the option to compile from source code, provided that sharing such information is permitted within the given environment.

Furthermore, certain flags can be set when compiling a program. These include e.g. the usage of faster but less precise math operations (using `-cl-fast-relaxed-math`) and preprocessor directives. The latter ones enable us to dynamically set variables, that behave like constants inside the program, which might improve performance

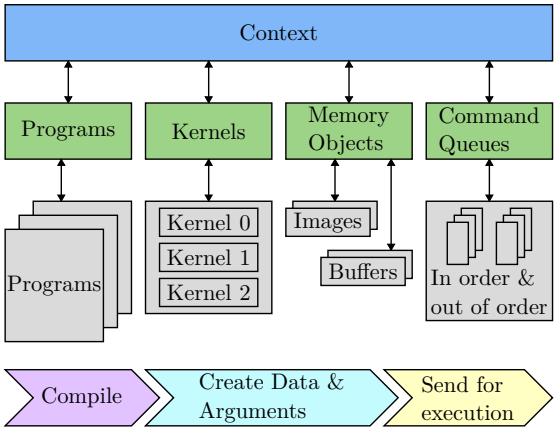


Figure 3.9.: Overview of OpenCL runtime components. *Source: Based on the image from:* `https://github.com/KhronosGroup/OpenCL-Guide/blob/main/images/executing_programs.jpg`

in certain scenarios due to compiler optimizations. For each kernel of a program that shall be run, a kernel object must be created based on its name.

When creating device buffers, there are multiple options to be considered. We can a) choose between buffers being read-write, read- or write-only by a kernel and b) choose whether a host buffer shall be reused or a new buffer shall be created and whether existing data shall be copied. Of course, data can also be written to the created buffer in a separate step. Buffers are typically specified with the `__global` identifier in the kernel parameter list. When a kernel is about to be enqueued, its arguments must be set, i.e. the global and local memory buffers and scalar variables.

Lastly, for each device inside a context, a *command queue* can be created (*stream* in CUDA). The command queue is used to enqueue kernels to be executed on the device as well as for memory transfer operations between host and device. It can be configured with properties, such as in-order or out-of-order queue, enabling OpenCL's profiling feature and whether it should reside on the device (since OpenCL 2.0). Multiple queues can be created for a single device, such that multiple kernels can be enqueued concurrently. We use this feature for overlapping computation and data exchange in a multi-GPU setting in section 5.5.

It should be noted, that since we are using Nvidia GPUs via the OpenCL API, not all of CUDA's features are available to us. These include [77]:

- CUDA-aware MPI: Enables direct GPU-to-GPU communication without copying through host memory. Useful in a multi-GPU scenario.

- Cooperative Groups: Allows flexible synchronization and communication between threads beyond standard block scope, as well as dynamic division of thread blocks.
- Thread block clusters: Physically groups multiple thread blocks to share data and synchronize directly, running on the same GPC.
- Warp Reduce / Warp Shuffle: Enables fast intra-warp data exchange and reductions without shared memory.
- CUDA Graphs: Captures and replays GPU workloads as a graph, minimizing kernel launch overhead.
- Unified Memory: Provides a single memory space accessible by CPU and GPU without manual transfers.
- Set preferred shared memory carveout from L1 (see `cudaSharedCarveout` in [79] section 6.36)
- Dynamic Parallelism: Control program flow directly from the GPU, i.e. allows kernels to launch other kernels.
- Asynchronous Copy from Global Memory to Shared Memory: Allows for overlapping memory transfer from global to shared memory and computation, whilst avoiding intermediate register storage, alleviating register pressure.

3.1.5. OpenCL C Kernel Language

The OpenCL C kernel language ([54] chapter 6) is based on C99 with extensions and restrictions. This fact makes it quite easy for developers to get into the language. Key restrictions include:

- No standard library I/O (`printf` is supported via extension)
- No recursion or function pointers
- No dynamic memory allocation with `malloc`/`free`
- No `static` or global variables with non-constant initializers
- No variable length arrays
- No variadic functions, except for `printf`

The different memory spaces are defined by using designated address space qualifiers:

- `__global`: global memory accessible by all work-items

Function	Description
`get_global_id(dim)`	Returns the global work-item index in dimension `dim`
`get_global_size(dim)`	Returns the `ndrange` size in dimension `dim`
`get_local_size(dim)`	Returns the work-group size in dimension `dim`
`get_local_id(dim)`	Returns the work-item index within its work-group in dimension `dim`
`get_num_groups(dim)`	Returns the number of work-groups in dimension `dim`
`get_group_id(dim)`	Returns the work-group index in dimension `dim`

Table 3.3.: Excerpt of work-item functions in OpenCL C

- `__local`: local memory, shared by work-items of the same work-group
- `__private`: memory private to a single work-item
- `__constant`: constant memory, read-only, potentially optimized to be broadcasted to all work-items

A kernel, i.e. a device function that is callable by the host, is identified using the `__kernel` keyword. A pointer argument might be qualified using the `restrict` keyword, expressing that the data pointed to is only accessed via this single pointer. Consequently, the data regions must not overlap and must be distinct for different pointers. This helps the compiler to limit effects of pointer aliasing.

Most of the usual data types of C99 are also supported in OpenCL C, e.g. `float`, `double`, `int`, `size_t` etc. Additionally, `half` can be used to declare half-precision floating point data types. Furthermore, there are built-in vector data types like `float4`, `double8` and `int16` and image data types.

There are many built-in functions for different purposes. While some of them are tailored utility to the newly introduced datatypes or memory spaces, others are general purpose, e.g. `ceil` or `abs`. Probably most notable are the work-item functions, part of which get used in every kernel. They can be used to query information about the launch configuration. Table 3.3 shows a short overview of the ones of most interest to us.

Just like C, OpenCL C supports preprocessor directives. This allows us to search-and-replace constant variable values in the kernel source code prior to just-in-time compilation, effectively facilitating variable length arrays and compiler optimizations specific to constants. Furthermore, unrolling loops is supported using the `#pragma unroll` directive, if a suitable extension is present (e.g.

`cl_nv_pragma_unroll` for Nvidia GPUs). OpenCL 2.0 introduces native support for loop unrolling using the `__attribute__((opencl_unroll_hint))` specifier.

Note that there is also C++ for OpenCL, a kernel language built on top of OpenCL C and C++17. It supports many, but not all, features of C++17. However, since most of our kernels would not profit from the additional language features, and due to OpenCL C's simplicity and thus clarity, we adhere to OpenCL C. Another reason for this choice is that many resources tend to use OpenCL C instead of C++ for OpenCL, suggesting a higher popularity of OpenCL C, which makes it easier to find information and collaborate with others.

Listing 3.1 shows a side-by-side comparison of CUDA and OpenCL C code. While the host code exhibits larger differences, the kernel code is largely comparable. For someone who is familiar with either of the two languages, understanding code written in the other language should be no complex task. Main differences include:

- Keywords and special functions, e.g. synchronization barriers, are different.
- In CUDA, the execution grid is defined by the size and number of blocks (work-groups in OpenCL), whereas in OpenCL, the work-group size and total work-item count is given.
- CUDA uses a special `<<< ... >>>` syntax, in OpenCL enqueuing a kernel is a regular C function.
- The global ID is given by `get_global_id` in OpenCL, whereas in CUDA it is calculated as `blockIdx.x * blockDim.x + threadIdx.x`.
- OpenCL typically involves more boilerplate code on the host side, which is due to its design focusing on portability.
- CUDA code typically resides side by side with the host code and is compiled together, while OpenCL code is often separated from the host code and compiled just-in-time.

3.1.6. OpenCL Under the Hood

When installing OpenCL on a machine, besides the required headers, an *Installable Client Driver (ICD) Loader*, i.e. an OpenCL backend, must be installed, too. It provides a mechanism for multiple OpenCL implementations to coexist on the same system – for example, Intel's CPU OpenCL runtime and Nvidia's OpenCL driver. The ICD Loader acts as a thin dispatch layer that routes OpenCL API calls to the correct vendor implementation based on the device being used. Each vendor provides an ICD library implementing the OpenCL API for their hardware.

```cuda
#include <cuda_runtime.h>

__device__ float square(float x) {
    return x * x;
}

__global__ void kernel(float *data) {
    extern __shared__ float s[];
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    s[threadIdx.x] = square(data[i]);
    __syncthreads();
    data[i] = s[threadIdx.x];
}

// host side
int N = 1024;
dim3 block(256);
dim3 grid(N / block.x);

float *d;
cudaMalloc(&d, N * sizeof(float));

// shared memory size
size_t smem = block.x * sizeof(float);
kernel<<<grid, block, smem>>>(d);

cudaFree(d);
```

```opencl
inline float square(float x) {
    return x * x;
}

__kernel void kernel(__global float *data,
                     __local float *s) {
    int i = get_global_id(0);
    int li = get_local_id(0);
    s[li] = square(data[i]);
    barrier(CLK_LOCAL_MEM_FENCE);
    data[i] = s[li];
}

// host side
size_t N = 1024;
size_t lws = 256;
size_t gws = N;

cl_context ctx; // must be initialized
cl_command_queue q; // must be initialized
cl_kernel kernel; // must be initialized

cl_mem bufData = clCreateBuffer(ctx,
    ↪ CL_MEM_READ_WRITE, N * sizeof(float),
    ↪ NULL, NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem),
    ↪ &bufData);
// argument 1: local memory size in bytes
clSetKernelArg(kernel, 1, lws * sizeof(float),
    ↪ NULL);

clEnqueueNDRangeKernel(q, kernel,
                       1, NULL, &gws, &lws,
                       0, NULL, NULL);
```

Listing 3.1: Example of CUDA (left) and OpenCL code

This way, an OpenCL program must not be recompiled and linked against vendor-specific binaries.

When a kernel is compiled, there are multiple possible intermediate representations (IR) before actually being translated into machine code. One possibility is to use the LLVM toolchain to compile into LLVM IR, which then can be further translated into various hardware-specific codes or the Standard Portable Intermediate Representation (SPIR) or its successor SPIR-V. The latter approach endorses a high level of portability without the need of distributing the high level kernel source code.

When compiling directly from source code using a Nvidia backend, the kernels are compiled into PTX (Parallel Thread Execution) code first. This is Nvidia's proprietary intermediate representation for CUDA and OpenCL. It gets further compiled into the actual GPU machine code (SASS). The intermediate step allows Nvidia's driver to perform device-specific optimizations dynamically, i.e. ensuring high performance for the present architecture (and thus CC), while maintaining the ability to run on a variety of architectures.

Unfortunately, there is no easy way to modify the SASS code directly in order to incorporate low-level optimizations. We tried to reorder some PTX instructions in certain scenarios, however without an effect to the resulting SASS code.

When a kernel is compiled using a Nvidia backend, the resulting binary is automatically cached for faster subsequent execution.

A further alternate OpenCL implementation is *PoCL (Portable Computing Language)* [48]. Its goal is to offer a highly portable implementation across a wide range of devices in a centrally orchestrated platform. The main benefit in our case is the Nvidia GPU support via a CUDA backend, which is realized using the LLVM NVPTX backend and the CUDA driver API. Since it uses the very same API as usual CUDA programs do, the CUDA driver perceives it just as such – a usual CUDA program. Consequently, we are able to use all the tools Nvidia provides for CUDA, especially emphasizing the two profiling tools Nsight Compute and Nsight Systems. Although this backend is in an experimental state, we found that most of the features we require are supported and that the difference in performance is not large enough, if noticeable at all, to prevent us from using it for profiling purposes.

3.2. Best Practices

In this section we elaborate on the main concerns for achieving high performance on GPUs. While we focus on Nvidia GPUs, the basic concepts transfer to GPUs of other vendors, too.

The optimal usage of a GPU might be divided into two parts: First, general best practices for optimal hardware usage and second, optimization techniques tailored to the specific algorithm being implemented. Thus, although general best practices should be applied whenever possible, achieving maximum performance always remains algorithm-dependent.

Hijma et al. have conducted a large meta study on GPU optimization techniques based on over 450 studies published in a span of 14 years [39]. They explicitly include studies addressing generally underrepresented AMD GPUs and OpenCL code. The proportion of optimization techniques per publication is particularly interesting, as it indicates which techniques are considered most important by the community. As not all of the techniques are relevant to us, we will refer to those taken into account in this work in the following chapter.

Typically, a kernel can be classified into three different categories: *compute bound*, *bandwidth bound* and *latency bound* kernels. Usually, *memory bound* is used as an umbrella term for both, bandwidth bound and latency bound kernels [14]. This categorization is nothing that was newly invented for GPUs, but already used for architectures prior to GPUs, albeit with different naming conventions (e.g. I/O-bound instead of memory bound and CPU-bound instead of compute bound [50, 1]).

Compute bound kernels are limited by the number of arithmetic operations that can be performed per time unit, i.e. the compute throughput of the device. Sometimes it is also referred to as instruction throughput bound.

On the other hand, bandwidth bound kernels are limited by the memory bandwidth, i.e. the amount of data that can be transferred between device memory and the compute units per time unit. Usually, the memory bus is close to being saturated for a bandwidth bound kernel.

Lastly, a kernel is considered latency bound, if neither of the above two limits apply. This is usually the case when there are not enough parallel work-items to hide memory access latencies, leaving many of the work-items being idle for most of the total cycles spent, waiting for data to arrive from memory. In other words, the problem might be too small, or not divided into enough workloads, to fully utilize the device.

We will encounter this classification again in section 3.3 in the context of roofline models.

For several years, there has been a trend of arithmetic capabilities increasing faster than memory bandwidths, i.e. machines are hitting the so-called *memory wall*. This has already been observed for architectures prior to GPUs [121, 67], but holds up for current GPUs [114, 27]. This means, that it is more likely for a kernel to be memory bound than compute bound, such that a large part of the optimization work focuses on improving memory access patterns, reducing excessive memory transactions, hiding memory access latencies etc.

3.2.1. Saturating the Device

A basic requirement for any program to run efficiently in parallel, is that a large part of the problem that is to be solved can be parallelized at all. The potential speedup over a serial version is limited by that fraction, which is described by the well-known Amdahl's law. Fortunately, the local operations in geometric multigrid methods we are concerned with, i.e. especially in Jacobi's method, are very well parallelizable, thus we will assume this constraint holds.

Since a GPU is focused on throughput, in order to fully exploit the device's capabilities, the problem must be divided into sufficiently small and preferably independent sub-problems. The size of the sub-problems, i.e. the amount of work that one work-item performs, can be varied by alternating the kernel design and launch configuration. For example, consider the application of a 27p-stencil onto each grid point of a grid with N^3 grid points. We could launch a kernel with one work-item per grid point, each performing one whole stencil application for the grid point, the work-item is mapped to. Other approaches include distributing the

application of a stencil across multiple work-items, with each work-item computing only a portion of the stencil, or assigning a single work-item to apply multiple stencils at different grid points. The optimal approach may depend on the problem size, i.e. grid and stencil size, device capabilities (e.g. amount of available compute units, cache capacities etc.) and implementation details, such as the choice of memory spaces and branch diversion.

One reason for high device saturation obviously is, that there is work for every physical compute unit, i.e. all the compute units are actually in use. This means the number of work-groups should be at least equal to the number of compute units available. Furthermore, many kernels, especially latency bound ones, profit from launching more work-items than physically supported, since a huge part of a GPUs performance comes from the concept of *latency hiding*. The basic concept is simple: While one warp waits on its operands to be ready for the next instruction, e.g. waiting for data to arrive from global memory, another warp can be scheduled to execute its next instruction on the same compute unit, effectively hiding the latency of the first warp. This is enabled by the SIMT paradigm, i.e. multiple warps can be resident on the same compute unit at the same time. In order to allow each compute unit to have enough warps to switch between, device restrictions must be considered carefully. For example, the A100 can allocate up to 48 KB of local memory per block and per compute unit (when using OpenCL). If we use all of the capacity for a single work-group, only this one work-group can be resident on the compute unit at a time. Even if that work-group is composed of multiple warps, when the work-group hits a local memory barrier statement, the whole work-group has to stall. This time could be used for another work-group to be scheduled, if not all the local memory capacity was used for the first work-group. i.e. the main goal in this context is to keep the GPU as busy as possible.

One indicator of how well the device is saturated is the occupancy of the kernel (as introduced in section 3.1.3.3), i.e. the ratio of active warps to the maximum number of possible warps. However, a high occupancy does not necessarily lead to high performance. At lower occupancy, a kernel has more registers available per thread, which might lead to less register spills to global memory (CUDA local memory). Especially kernels incorporating a high instruction-level-parallelism (ILP) might be able to hide latencies well at low occupancy [113].

3.2.2. TLP and ILP

Volkov [113] gives a profound explanation of TLP (thread-level parallelism) and ILP (instruction-level parallelism) and shows, how GPU kernels can profit from increasing ILP at low occupancies. Although the tests in the reference are made with the Tesla (2006) and Fermi (2010) architectures, the relevance of the two concepts and validity to choose one over the other still holds for modern architectures.

TLP describes the parallelism that is achieved by having multiple work-items (threads) running concurrently. Increasing the TLP is typically achieved by launching more work-items. On the other hand, ILP describes the parallelism that is achieved within a single work-item, i.e. how many independent instructions can be issued concurrently. A common way to increase ILP is to unroll loops and restructure code to have more independent instructions in sequence. Listing 3.2 presents an example of increasing ILP by varying the number of independent operations per iteration, using fused multiply–add operations as an illustration. In older architectures, e.g. CC 2.x, warp schedulers are able to *dual-issue* instructions, i.e. two independent instructions can be issued in one cycle per scheduler. Thus, ILP must be used to attain maximum performance.

```
// ILP = 1
#pragma unroll UNROLL
for(int i = 0; i < N; i++) {
    a = a * b + c;
}

// ILP = 2
#pragma unroll UNROLL
for(int i = 0; i < N/2; i++) {
    a = a * b + c;
    d = d * b + c;
}

// ILP = 4
#pragma unroll UNROLL
for(int i = 0; i < N/4; i++) {
    a = a * b + c;
    d = d * b + c;
    e = e * b + c;
    f = f * b + c;
}
```

Listing 3.2: In this example, adding independent instructions to the loop results in higher ILP and lower TLP.

3.2.3. Execution Parameters

There are some aspects to consider when launching a kernel. First, even when dealing with multidimensional data, a kernel might be launched as a 1d kernel instead of a 3d kernel for example. If e.g. one work-item is mapped to one grid point of a grid of size $m \times n \times o$, listing 3.3 shows the conversion of the 1d work-item index to the 3d grid point index.

```
// 3d kernel
int i = get_global_id(2);
int j = get_global_id(1);
int k = get_global_id(0);

// 1d kernel
int idx = get_global_id(0);
int no = ngh * ogh;
int i = idx / no;
int j = (idx - i * no) / ogh;
int k = idx % ogh;
```

Listing 3.3: Conversion of 1d work-item index to 3d grid point index

Since the total amount of work-items must be a multiple of the work-group size per dimension and the best pick of a work-group size is a multiple of the warp size, the total amount of work-items may need to be padded to meet that constraint. Applying padding in all three dimensions would result in many redundant work-items. When the kernel is launched in a 1d fashion instead, only the last work-group might contain some redundant work-items, which should be negligible compared to the total amount of work-items.

A 2d or 3d kernel is required, whenever 2d or 3d local memory blocks are used. This is quite common for stencil kernels with fixed coefficients, as data is reused by exploiting the multidimensional structure of the stencil and the data.

When launching a 3d kernel, diligent mapping of the work-item index to the memory layout is required to ensure optimal memory access patterns. For Nvidia architectures, the work-items are laid out as

$$\begin{aligned}\texttt{idx} = \;&\texttt{get_global_id(0)}\\ &+ \texttt{get_global_id(0)} * \texttt{get_grid_size(1)}\\ &+ \texttt{get_global_id(2)} * \texttt{get_grid_size(1)} * \texttt{get_grid_size(2)}.\end{aligned}$$

Thus, if one work-item maps to one grid-point and one grid point index is calculated as $i \cdot n \cdot o + j \cdot o + k$, the indices should be assigned as shown in listing 3.4.

```
int i = get_global_id(2);
int j = get_global_id(1);
int k = get_global_id(0);
```

Listing 3.4: Optimal assignment of 3d work-item indices to 3d grid point indices

3.2.4. Host-Device Communication and Kernel Streamlining

Compared to the memory transfer rates within the GPU, the costs of data transfer between host and device, e.g. via PCIe, are very high. Thus, it should be a main concern to minimize such transfers and their sizes as much as possible. There are multiple aspects to this.

First, the amount of operational intensity should justify the amount of data transferred to the device. If only a small set of work-items operate on a large data set, performing only basic operations, using the GPU might not result in a benefit at all (see section 3.2 of [76]).

Second, the longer the data can reside on the device and be reused by multiple kernels, the lower the performance loss due to data transfer between host and device. Keeping data on the device not only diminishes data transfer costs, but

also allows streamlining kernel execution. If the resulting data of kernel (a) is used as input for kernel (b) and data can reside on the device in between, kernels (a) and (b) can be enqueued back-to-back, without the need of waiting for kernel (a) to finish.

This leads to our last point in this section: We aim for the kernel queue to be as long as possible. This way, the driver can schedule the kernels more efficiently, hiding kernel launch overheads and other kernel preparation steps like setting the arguments. While sometimes it can be better to fuse two kernels (*kernel fusion*, section 6.1.8 of [39]), other times it might be beneficial to keep them separate or even divide into further kernels (*kernel fission*, section 6.2.4 of [39]). Dividing a kernel into multiple kernels is always necessary, when global data must be synchronized for all work-items. While CUDA offers cooperative groups and dynamic parallelism to alleviate this issue, these features are not available in Nvidia's OpenCL implementation.

A common approach to achieve higher device saturation, is to launch multiple kernels concurrently, when the individual kernels are too small or when the algorithm does not allow the device to be fully exploited. In our multigrid application we are not able to launch multiple kernels in parallel, as the individual steps depend on each other. However, in the single-GPU case, after the setup phase, in which we allocate all the data on the device, we only need to send data to the host for checking our termination criterion, i.e. the residual norm. This, in turn, can be chosen to be turned off by the user and have a fixed amount of v-cycle iterations performed instead. On the other hand, in the multi-GPU case we must exchange ghost layer data between neighboring processes at certain points, which requires the data to be sent to the host before forwarding it via MPI. However, we attempt to limit the impact of data transfer and kernel synchronization by reducing the overall amount of data transferred via data transformation (section 5.3) and by overlapping computation and data transfer (section 5.5).

Note that both OpenCL and CUDA have a feature to unify the address spaces of host and device, i.e. *unified memory* in CUDA and *shared virtual memory (SVM)* in OpenCL 2.0. It enables the user to allocate memory that is accessible by both host and device, using the same pointer, without the need of explicit data transfer operations. While this might ease the programming effort, care must be taken whenever the memory is accessed, as memory transfer operations are implicitly initiated. Achieving good performance can be quite challenging and depends on the behavior of the architecture, as shown in [37, 97]. Typically, some sort of data initialization kernels and prefetching is required.

Another, older feature is *zero copy* memory, which allows the device to directly access pinned host memory, i.e. page-locked memory. As the data does not reside in device memory and avoids caching, using this feature is only valuable if the

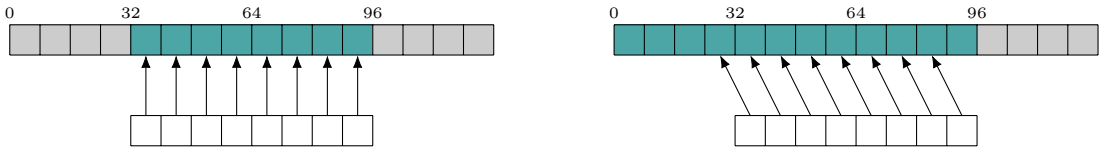


Figure 3.10.: Left: Subsequent work-items access consecutive memory addresses aligned to 32 bytes, resulting in coalesced memory access. Two 32 byte sectors need to be fetched. Right: Subsequent work-items access consecutive memory addresses not aligned to 32 bytes, resulting in misaligned memory access. Three 32 byte sectors need to be fetched.

data is only accessed once by a single kernel on a dedicated device, or if host and device share the same physical memory, which is the case for integrated GPUs.

3.2.5. Memory Access Patterns

Since most kernels are memory bound, optimizing the memory access is one of the most important factors for high performance. The main goal is to minimize redundant memory transactions. As in the beginning, all data resides in global or constant memory, we focus on these memory spaces at first.

A key metric to look at regarding global memory access patterns is the ratio of sectors to requests (section 2.10.2 of [80]). A sector is a 32-byte aligned chunk of memory, thus if each work-item of a warp accesses an individual single-precision value (4 bytes), four 32-bytes sectors must be fetched to satisfy the request, i.e. having a sectors-to-requests ratio of 4. For double-precision values (8 bytes), 256 bytes of data would need to be fetched, thus the optimal ratio lies at 8 sectors per request. This optimal scenario can be achieved, when all work-items of a warp access consecutive memory addresses, i.e. when the memory access is *coalesced*, see figure 3.10. Listing 3.5 provides a simple test to verify the numbers. The kernels `warp_load_coalesced_x` are launched with a single work-group of 32 work-items, i.e. a single warp, for $x \in \{1, 2, 4\}$. Each work-item accesses x double-precision values having a warp-sized stride in between, guaranteeing coalesced access. The sectors-to-requests ratio in Nsight Compute reveals that x requests are made and $8x$ sectors are fetched, leading to the optimal ratio of 8 sectors per request for double-precision values.

Although a lower number indicates better memory coalescing, we can go below the mentioned optimal ratios. For example, consider the kernels in listing 3.6, where only the first half or quarter of a data vector is accessed. This is illustrated in figure 3.11. Still, there is only a single memory request, but it can be served by 4 and 2 sectors, respectively, leading to sectors-to-requests ratios of 4 and 2. Results can be seen in the appendix in table A.1.

```
// Kernel: one warp (block with 32 threads) will collectively load x doubles.
// Each thread loads x/32 elements. Access pattern:
// 1st iteration: threads 0..31 load src[0..31]  (coalesced)
// 2nd iteration: threads 0..31 load src[32..63] (coalesced)
// etc.
__device__ void warp_load_coalesced(
    const double * __restrict__ src, double * __restrict__ dst, int x) {
    const int warp_size = 32;
    int tid = threadIdx.x;
    int per_thread = x / warp_size;

    for (int k = 0; k < per_thread; ++k) {
        int idx = k * warp_size + tid;
        double v = src[idx];
        dst[idx] = v;
    }
}

__global__ void warp_load_coalesced_1(
    const double * __restrict__ src, double * __restrict__ dst) {
    warp_load_coalesced(src, dst, 32);
}

__global__ void warp_load_coalesced_2(
    const double * __restrict__ src, double * __restrict__ dst) {
    warp_load_coalesced(src, dst, 64);
}

__global__ void warp_load_coalesced_4(
    const double * __restrict__ src, double * __restrict__ dst) {
    warp_load_coalesced(src, dst, 128);
}
```

Listing 3.5: In these CUDA kernels, one work-item loads one or more double-precision values that are separated by a warp-sized stride, resulting in coalesced memory access.

```
// only 16 values loaded per warp
// threads 0-15 load elements 0..15
// threads 16-31 load the same 0..15
__global__ void warp_load_16_per_warp(const double * __restrict__ src,
                                      double * __restrict__ dst)
{
    int tid = threadIdx.x;
    int idx = tid % 16;    // both halves of warp map to 0..15
    double v = src[idx];
    dst[tid] = v;
}

// only 8 values loaded per warp
// threads 0-7 load elements 0..8
// threads 8-15 load the same 0..8
// threads 15-23 load the same 0..8
// threads 24-31 load the same 0..8
__global__ void warp_load_8_per_warp(const double * __restrict__ src,
                                     double * __restrict__ dst)
{
    int tid = threadIdx.x;
    int idx = tid % 8;     // each quarter of the warp map to 0..7
    double v = src[idx];
    dst[tid] = v;
}
```

Listing 3.6: In these CUDA kernels, each 16 or 8 consecutive work-items of a warp load the first 16 or 8 double-precision values, respectively. Although the access is coalesced, a lower-than-optimal ratio of sectors to requests is achieved.

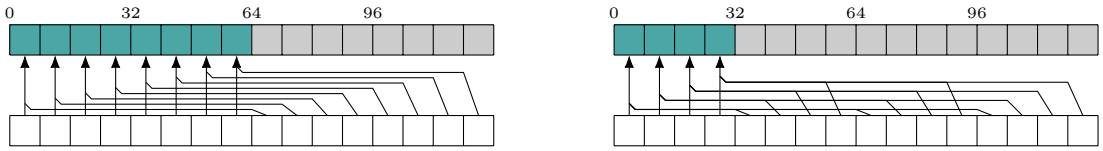


Figure 3.11.: Left: Both halves of the warp access the same memory addresses. Each data element is read by two work-items.
Right: Each quarter of the warp access the same memory addresses. Each data element is read by four work-items.

Note that caching also has an impact on the sectors-to-requests ratio, e.g. if loads from multiple requests are overlapped within a cache line.

Another scenario is misaligned access, i.e. the addresses requested by the warp are not aligned to a multiple of 32 bytes, but still adjacent, see figure 3.10 once again. In section 10.2.1.3 of [76], a simple `offsetCopy`-kernel is used to show, that bandwidth is higher for offsets that are multiples of 8 (tested on a Tesla V100 GPU, CC 7.0), see listing 3.7. For those values, the addresses are aligned to 32 bytes and thus can be satisfied by sending four 32-byte sectors (for single-precision values). For the offset values in between, an additional sector must be fetched. If we run the test on a Quadro T2000 GPU (CC 7.5), we do not see any noticeable difference in kernel runtime for any offset. However, for any offset other than zero, we see that the L1 cache is heavily involved. In particular, it is highest for offsets that are not multiples of 8, slightly lower for multiples of 8 and zero when there is no offset. We conduct, that the L1 cache is able to mitigate the negative effects of misaligned access on this architecture. Again, results can be seen in the appendix in table A.2.

```
__global__ void offsetCopy(float *odata, float *idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

Listing 3.7: `offsetCopy`-kernel with varying offset to analyze misaligned data access pattern.

Another typical access pattern is strided access, i.e. the memory elements accessed by consecutive work-items of the warp have gaps between each other. This is illustrated with a stride of two elements in figure 3.12. In this case, only half of the data fetched is actually used by the warp. Generally, as the stride size increases, the ratio of data used to data fetched decreases. It often occurs for example in matrix operations, when accessing rows of a column-major stored matrix or vice versa or when dealing with multidimensional data. Whenever a stride could not be avoided by simple global loads, local memory might be used to alleviate this issue, as we will see in the following section.

Note that, for older architectures like Fermi, there is a differentiation between

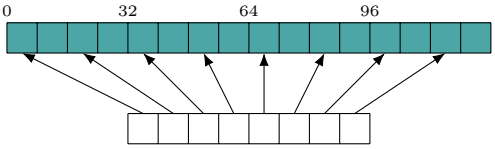


Figure 3.12.: Subsequent work-items access memory addresses separated by a stride of two elements, resulting in a strided memory access pattern. Four 32 byte sectors need to be fetched.

cached loads and uncached loads, i.e. the L1 cache could optionally be bypassed [115, 82]. In CUDA, it can be set using the `dlcm` compiler flag. However, this differentiation does not appear in the CUDA programming guide anymore, indicating its insignificance for modern architectures.

Furthermore, memory instructions can be vectorized, when using the appropriate vector data types or fitting structures [64]. The compiler then replaces `LDG.E` and `STG.E` SASS instructions with their vectorized counterparts `LDG.E.{64,128}` and `STG.E.{64,128}` for loading and storing 64-bit and 128-bit values, respectively. This reduces total number of instructions and might improve bandwidth utilization. However, it also changes how work-items are mapped to memory addresses. For example, if we would start a kernel with one work-item per grid point and each work-item fetches the current approximation v for that grid point, using a vectorized load would lead to one work-item loading values of multiple grid points. This would only make sense if a single work-item handled the work for these multiple grid points, which would in turn reduce overall parallelism. Furthermore, vectorized loads increase register pressure and needs data to be aligned properly. All in all, this optimization technique adds a noticeable amount of complexity to a kernel for only a marginal potential benefit.

3.2.6. Local Memory Usage

Local memory is located on-chip and provides lower latencies than global memory. It can be used for

- exchanging data between work-items inside a work-group,
- enabling coalesced loading from global memory,
- avoiding or minimizing multiple global memory accesses, i.e. user-managed cache.

Data exchange inside a work-group is useful in reduction scenarios, e.g. building the global sum over a vector, as shown in section 4.6. Each work-group calculates a fraction of the sum. The intermediate results are cascaded inside the work-group

until the final partial sum is held by a single work-item. Final summation then either happens on the host or in subsequent kernel executions on the device.

Enabling coalesced load from global memory into local memory is useful, whenever the algorithm does not allow global data to be fittingly structured according to the layout of the work-items. This is the case for example in matrix-matrix multiplication (gemm) $A \cdot B = C$, where accessing columns of B leads to strided access, if B is stored in a row-major order. In fact, gemm is a prime example for local memory usage and can be thoroughly investigated by the reader in other literature sources, e.g. section 10.2.3.2 in [76], chapter 21 in [73] or chapter 5.3 in [43]. Essentially, instead of accessing B directly through global memory, it is loaded into local memory once in a coalesced manner. Further usages do not need to access global memory again, but can resort to faster local memory instead.

Lastly, local memory can be used as a user-managed cache, enhancing data reusage inside a work-group. This occurs for example in stencil codes, when typically one work-item applies a stencil onto one grid point. For this, the values of multiple surrounding grid points are required. Additionally, results might be needed for multiple iterations, such that the result of the i-th iteration can be stored in local memory and loaded from there again for the (i+1)-th iteration. The related concepts *spatial* and *temporal blocking* are covered in section 4.3.

There are mainly two hurdles to keep in mind when using local memory: occupancy and bank conflicts.

Since local memory usage is tightly coupled to the size and layout of work-groups, choosing these parameters is an act of experimentation and balancing. It is essential that the layout of the local memory fits to the size and dimensionality of the data used in an algorithm in the first place. If we stay with the stencil code example, we might choose to store one value per grid point and one or more layers of ghost points in local memory, i.e. the local memory block must have the same size as the work-group plus ghost points (if one work-item of the work-group maps to one real grid point). Choosing a small work-group size reduces the ratio of real points to ghost points, increasing redundant memory and arithmetic operations. Choosing a larger work-group on the other hand might limit occupancy, if one work-group uses a majority of the local memory of a single compute unit. Furthermore, it reduces the capacity of the automatic L1 cache.

If local memory usage is justified and a beneficial strike between size and occupancy is found, care has to be taken regarding bank conflicts. As noted earlier, for Nvidia GPUs local memory is structured in 32 banks, each capable of holding 32-bit (i.e. single-precision) values. Separate banks can serve requests simultaneously. However, requests to different data in the same bank result in a serialized access. This situation is called bank conflict. A common example is the matrix transposition. If a `8x8` matrix A is stored in row-major order in global memory, it is read in a

coalesced fashion into local memory. If we use the same layout to store it there and want to access a column of A by individual work-items, all elements of the column are located at different addresses in the same bank, resulting in an 8-way bank conflict, see figure 3.13 (for the sake of simplicity, in this example we assume there are only 8 memory banks instead of 32). There are two common solutions to this issue. First, we can reorder the elements in local memory, i.e. "swizzle" them, such that the elements of one column are located in different memory banks, see section 10.29.3.1 of [77]. While this is a memory storage efficient solution, it has the overhead of calculating the swizzle pattern. Second, we can pad the inner dimension of the local memory array by one element, i.e. resize it as `8x9`. This induces no overhead in indexing logic, but introduces a larger memory footprint.

Note that there are other ways to avoid bank conflicts, e.g. by transposing the matrix in local memory or use a diagonal mapping of rows [53].

If double precision (i.e. 64-bit data) is used, one element is stored in two consecutive memory banks. This might lead to the assumption, that there is always a 2-way bank conflict when an entire warp accesses consecutive data in local memory, as the i-th and $(i + 16)$-th elements are stored in the same bank for $0 \leq i \leq 15$. In the example shown in figure 3.14, one warp consists of 8 work-items and there are 8 memory banks. Although the elements (0,0) and (1,0) are located in the same bank, the memory controller splits requests into 128-byte transactions. Thus, to deliver the entire 256 bit of data, two requests have to be issued, not considering this as a bank conflict [15].

3.2.7. Miscellaneous

This section covers some miscellaneous optimization strategies, that might be taken into consideration when trying to achieve the last few percentages of performance gains.

Prefetching. Prefetching is a technique to load data before it is actually used to hide memory latencies. It is not unique to GPUs but also common to CPUs. One of the easiest implementations of prefetching is loop unrolling. This works, whenever the statements inside one iteration do not depend on those of other iterations. If the iteration count is too large for the loop to be completely unrolled, unrolling can be performed in batches. As demonstrated in [117], values can be fetched into registers or local memory. Furthermore, an alternative to fetching batches every n-th iteration is to pre-fetch n batches before the first iteration and then fetch the next element in each iteration, evicting the oldest element from the batch. If done asynchronously, and using padding to avoid bank conflicts, this might yield a notable performance improvement.

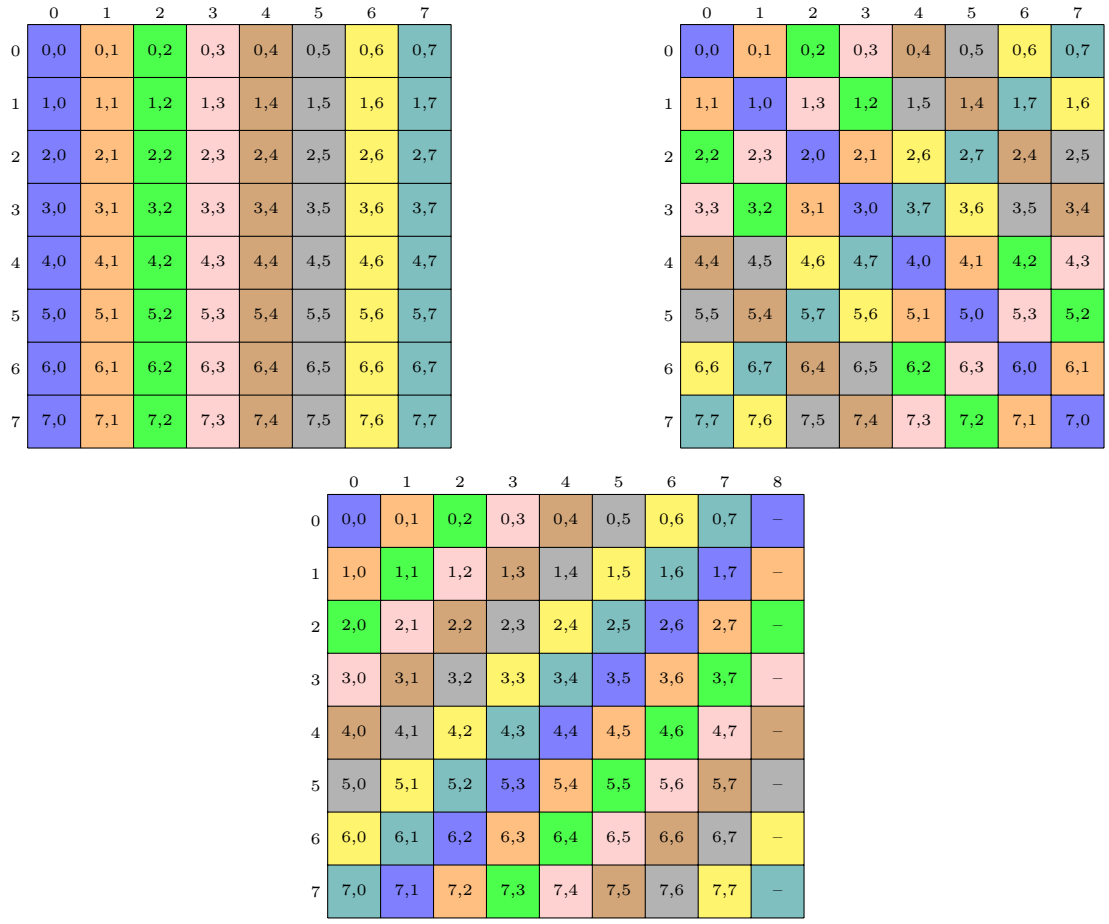


Figure 3.13.: Writing into the transposed of a `8x8` matrix stored in row-major order. The tick labels denote the local memory array indices, the cell contents denote the index in global memory and the color denotes the memory bank. In the case at the top-left, there is an 8-way memory bank conflict. It is solved by swizzling in the top-right and by padding in the bottom grids.

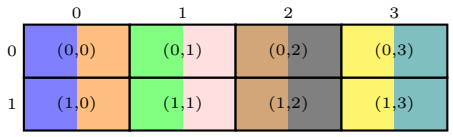


Figure 3.14.: For 64-bit data, one element spans two memory banks. Although elements (0,0) and (1,0) are located in the same bank, two memory requests need to be issued to deliver the entire data set, not considering this as a bank conflict. For this example, we assume a warp to consist of 8 work-items and the presence of 8 memory banks.

Code Generators. Since GPU code can be compiled during runtime of the host program, optimal code for the present architecture can be generated. This might include setting constants, setting loop unrolling factors, setting tiling sizes for local memory usage, removing conditional statements etc. Especially for stencil codes with fixed coefficients there are multiple code generators available. They are investigated deeply in section 4.3.

Tuning. Typically, different architectures require different approaches to a kernel design. This can vary from just adjusting work-group sizes to choosing a completely different parallelization granularity. In order to achieve good results on a wide variety of architectures, a way to dynamically choose a specific kernel that accomplishes a given task is necessary. Additionally, the launch parameters should be tunable, e.g. by performing small micro-benchmarks before executing the program on a large problem. There are tools to automate the search for optimal launch parameters, e.g. *CLTune*, presented by Nugteren and Codreanu in [75].

Grid-Stride Loop. A common practice to ensure the flexibility of a kernel in terms of launch parameters is to use a *grid-stride loop* [34]. Consider the `axpy`-kernels (double-precision a times x plus y) in listing 3.8. The regular `axpy`-kernel must be launched with exactly one work-item per element, i.e. the global work-size must be equal to the number of elements in the vectors. If this is not the case, some elements will not be processed at all, or some work-items will access out-of-bounds memory regions. Using a grid-stride loop as done in the `axpy_gsl`-kernel, however, allows us to launch the kernel with arbitrary global work-sizes. Each work-item processes multiple elements by iterating over the data in steps of the total number of work-items, i.e. with a grid-stride. This way, even if there are less work-items than elements, all elements will be processed correctly. This allows us to launch the kernel, even if the number of elements is larger than the allowed maximum global work-size. Furthermore, we gain the ability to restrict the number of work-groups to a multiple of the number of compute units, which might improve performance due to workload balancing and enhance thread reusage. Another benefit is that we can launch the kernel with only a single work-item for debugging purposes, effectively emulating serial execution.

Double Buffering. Sometimes, using multiple buffers for the same quantity might be beneficial in order to overlap computation and communication. For two buffers, this is often called *double buffering*. It might also be required when fusing multiple kernels, where later kernels operate on the output of the earlier kernels. We use this strategy in the implementation of Jacobi's method in section 4.2.5.

Avoiding Branches. One strategy to eliminate branches is to multiply the operations of the branches with the conditions. Since in OpenCL C `true` evaluates to 1 and `false` to 0, the result is not changed in the end. This is demonstrated

```
// must be launched with one wi per element
__kernel void axpy(
    __global double* restrict v_in,
    __global double* restrict f,
    __global double* restrict r,
    int n
)
{
    int i = get_global_id(0);
    if (i < n)
        y[i] = a * x[i] + y[i];
}

// able to launch with less than one wi per element
__kernel void axpy_gsl(
    __global double* restrict v_in,
    __global double* restrict f,
    __global double* restrict r,
    int n
)
{
    for(int i = get_global_id(0); i < n; i += get_global_size(0))
        y[i] = a * x[i] + y[i];
}
```

Listing 3.8: Comparison of a kernel that requires the launch of one work-item per element vs. a kernel that uses a grid-stride loop. The latter therefore allows a single work-item to handle multiple elements.

in listing 3.9. However, since this means, that all instructions must be evaluated, performance increases might be marginal if existent at all. Commonly, a better approach, if applicable, is to adjust the kernel layout, such that different branches are taken by different warps, i.e. all work-items of one warp take the same branch.

```
// These conditionals ...
if (a < 0)
  x = a + b;
else if (a < 10)
  x = a - b;
else
  x = a * b;

// ... are converted to the following expression.
x = (x < 0) * (a + b)
    + (a < 10) * (a - b)
    + (a >= 10) * (a * b);
```

Listing 3.9: Eliminating branches by multiplication with the condition result (1 or 0)

Tail Effect. The work a GPU performs can be measured in waves, as stated in section 3.1.3.3. A full wave corresponds to the number of work-groups that run concurrently on the GPU, which is determined by the number of work-groups per compute unit and the number of compute units. A kernel launch consists of one or multiple waves, plus a potential partial wave, i.e. the tail. The *tail effect* describes the performance degradation that occurs, when the last wave is not fully occupied, i.e. not all compute units are busy all the time. This is, however, only an issue, when the kernel is executed in only a few waves [18]. In our case of multigrid methods, this might especially occur for coarser grids. As they contribute only a small fraction to the overall runtime, the tail effect is typically negligible for us.

3.3. Profiling Methods

There are multiple ways of profiling an OpenCL or CUDA application, possibly in combination with MPI. Probably the most important ones when working with Nvidia architecture are Nsight Compute and Nsight Systems, which can also be used for OpenCL applications when using PoCL as the backend, as noted in section 3.1.6. Alternatives include, among others, TAU (Tuning and Analysis Utilities) [101], Score-P [56] with Vampire [55] or Cube [99], callgrind with kcachegrind, linux perftools with a flame graph generator [30] and the built-in OpenCL profiling events.

An important step in optimizing an application is finding hotspots that largely contribute to the overall runtime. There is a large variety of options how to accomplish this task, from simply inserting timestamp output instructions at certain points in the code to using a sophisticated profiling tool. To get a first impression, profiling an application with the linux perftools and visualizing the result as a flamegraph might be an option that is easy to set up. However, for GPU applications, there is no differentiation in host and device code, which might distort the results. It is important to investigate both, the isolated kernels and the stream of kernels including the interaction between host and device.

3.3.1. Nsight Systems

Nsight Systems can be used for profiling an entire application. Its timeline view can show the activities of the different device queues (streams in CUDA), i.e. which kernel gets called when and how long it takes to finish. The CPU timeline is shown side-by-side with the device's timeline, such that the interplay can be analyzed closely. Detailed tables are available for showing detailed information of kernel runtimes, memory transfer sizes and times, general CUDA API calls and MPI events.

In our case it is especially useful for parts of the code where multiple subsequent kernel calls are necessary, but options are given to fuse some kernels (e.g. in the implementation of Jacobi's method), or host-device communication is irrefutable, e.g. the ghost update in the multi-GPU case.

Figure 3.15 for example shows the comparison of the implementation of Jacobi's method with and without intermediate ghost update for three iterations in the multi-GPU case. In the upper timeline we see, that there are three separated kernel calls (in blue), followed by a memory transfer from device to host (in red), MPI calls (in grey) and a memory transfer back from host to device (in green). In the lower timeline, the three kernel calls are back-to-back, without any memory transfer or MPI calls between the calls, since the ghost update is only performed

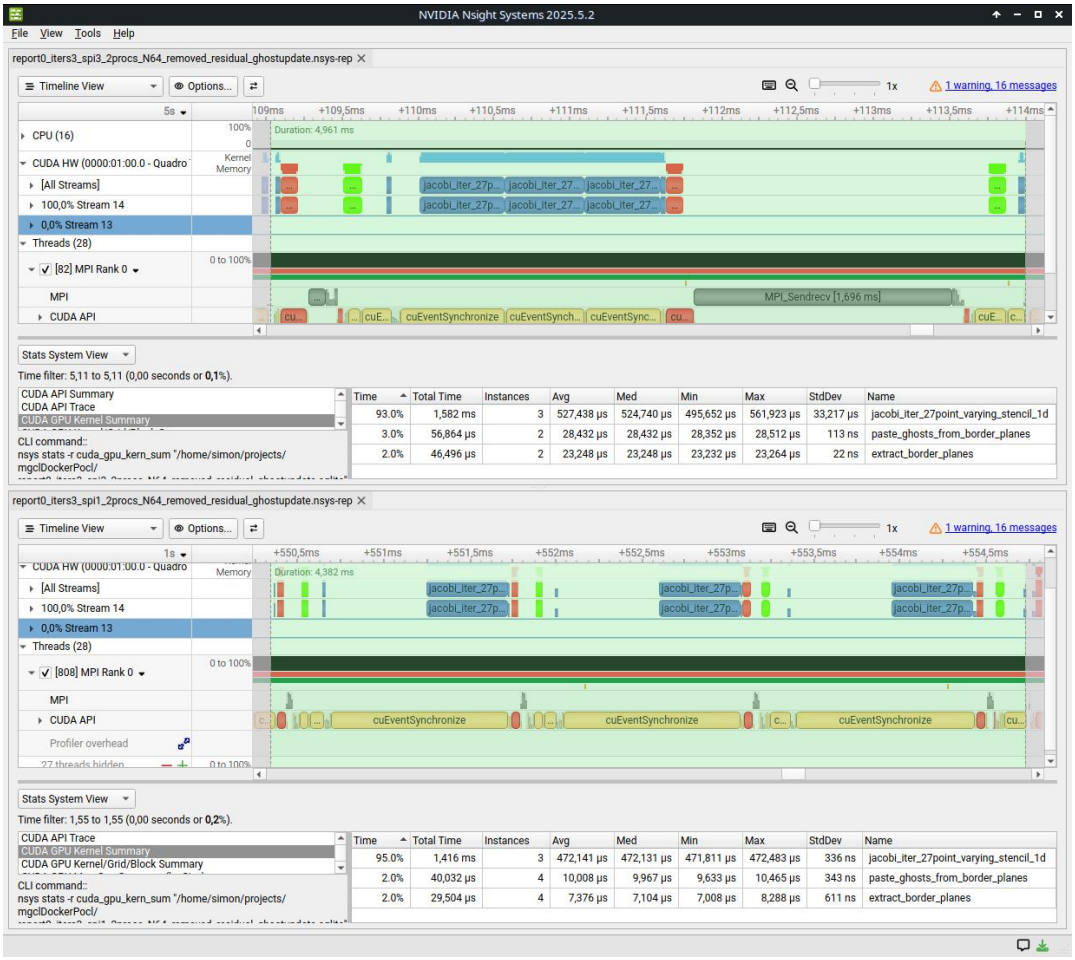
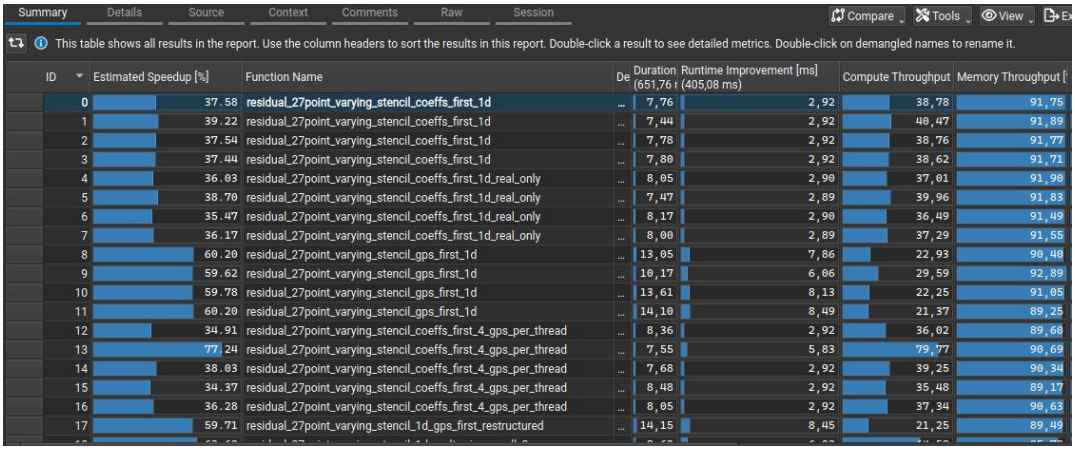


Figure 3.15.: Example of Nsight Systems showing the comparison of the implementation of Jacobi's method with and without intermediate ghost update for three iterations total in the multi-GPU case.

once before the three kernel calls. Each kernel call reflects one Jacobi iteration. The tables below the timelines show that there are actually two more kernels involved for extracting and pasting data, but which need only a fraction of the time that the Jacobi kernels need. Of course, more kernels are involved in the whole program, however, Nsight Systems allows us to filter the timeline for a given time slice, which is especially useful to investigate aggregate values in the tables.

3.3.2. Nsight Compute

Nsight Compute is a tool for in-depth analysis of the behavior of individual kernels. It collects a variety of metrics during multiple runs, and presents the results



ID	Estimated Speedup [%]	Function Name	De	Duration (651,76 r	Runtime Improvement [ms] (405,08 ms)	Compute Throughput	Memory Throughput [
0	37.58	residual_27point_varying_stencil_coeffs_first_1d	...	7,76	2,92	38,78	91,75
1	39.22	residual_27point_varying_stencil_coeffs_first_1d	...	7,44	2,92	40,47	91,89
2	37.54	residual_27point_varying_stencil_coeffs_first_1d	...	7,78	2,92	38,76	91,77
3	37.44	residual_27point_varying_stencil_coeffs_first_1d	...	7,80	2,92	38,62	91,71
4	36.03	residual_27point_varying_stencil_coeffs_first_1d_real_only	...	8,05	2,90	37,01	91,90
5	38.70	residual_27point_varying_stencil_coeffs_first_1d_real_only	...	7,47	2,89	39,96	91,83
6	35.47	residual_27point_varying_stencil_coeffs_first_1d_real_only	...	8,17	2,90	36,49	91,49
7	36.17	residual_27point_varying_stencil_coeffs_first_1d_real_only	...	8,00	2,89	37,29	91,55
8	60.20	residual_27point_varying_stencil_gps_first_1d	...	13,05	7,86	22,93	90,40
9	59.62	residual_27point_varying_stencil_gps_first_1d	...	10,17	6,06	29,59	92,89
10	59.78	residual_27point_varying_stencil_gps_first_1d	...	13,61	8,13	22,25	91,05
11	60.20	residual_27point_varying_stencil_gps_first_1d	...	14,10	8,49	21,37	89,25
12	34.91	residual_27point_varying_stencil_coeffs_first_4_gps_per_thread	...	8,36	2,92	36,02	89,60
13	77.24	residual_27point_varying_stencil_coeffs_first_4_gps_per_thread	...	7,55	5,83	79,77	90,69
14	38.03	residual_27point_varying_stencil_coeffs_first_4_gps_per_thread	...	7,68	2,92	39,25	90,34
15	34.37	residual_27point_varying_stencil_coeffs_first_4_gps_per_thread	...	8,48	2,92	35,48	89,17
16	36.28	residual_27point_varying_stencil_coeffs_first_4_gps_per_thread	...	8,05	2,92	37,34	90,63
17	59.71	residual_27point_varying_stencil_1d_gps_first_restructured	...	14,15	8,45	21,25	89,49

Figure 3.16.: The summary view of Nsight Compute provides an overview of the profiled kernels.

visually in different sections. In the following we aim to quickly iterate through the sections most important to us and give guidance on which information can be retrieved.

Summary View. Figure 3.16 shows the summary view of Nsight Compute. It provides an overview of the profiled kernels, which is especially useful, when comparing different variants of the same kernel. Besides the kernel name and the kernel runtime, it also shows the achieved compute throughput (CT) and memory throughput (MT) as percentages of the peak values of the device. Furthermore, an estimated speedup is given, which could potentially be reached when the primary performance issues are fixed.

Details View. As the name suggests, the details view offers insights into a specific kernel. It is further divided into several subcategories.

The **Speed of Light** section gives a first impression of the kernel's performance. It shows throughput metrics for different memory parts of the device, i.e. DRAM and caches, and aggregated compute and memory throughputs. Those results are visualized using a simple bar chart (see figure 3.17) as well as roofline model plots.

The roofline model was invented as a visual tool, that is easy to understand, for parallel software performing floating point computations [118]. The attainable compute performance of a kernel in FLOP/s is plotted on the y-axis, the arithmetic intensity in FLOP/byte on the x-axis. Usually, the number of bytes transmitted between caches and DRAM is used for the arithmetic intensity, however there are also hierarchical roofline models, that differentiate between the memory hierarchy levels and combine them in a single plot [122]. The arithmetic intensity indicates the DRAM bandwidth needed by a kernel.

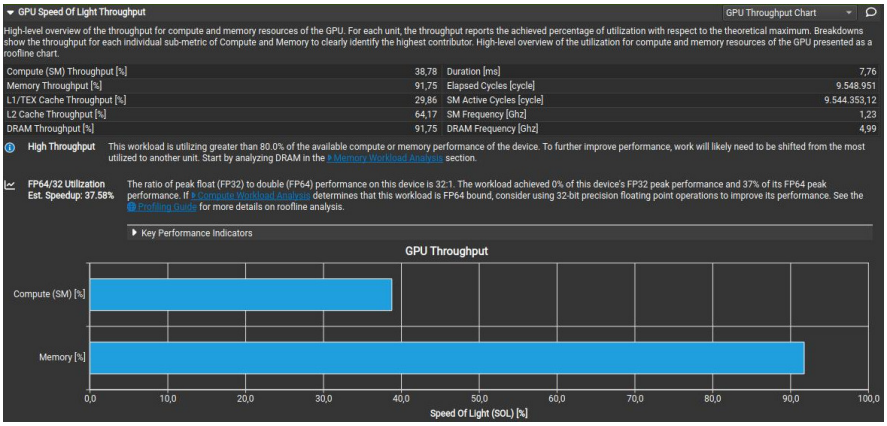


Figure 3.17.: The bar chart in the Speed of Light section shows compute and memory throughputs.

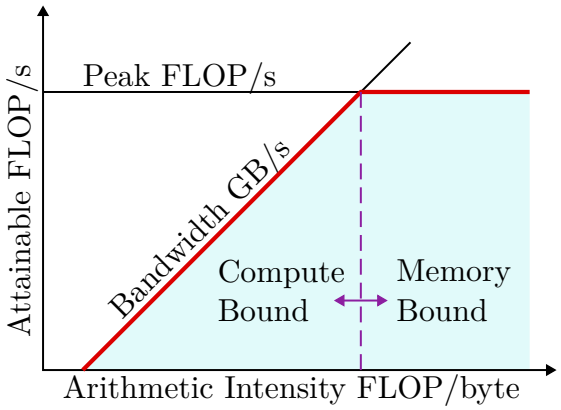


Figure 3.18.: Regions of the roofline model

The eponymous roofline consists of a horizontal line, representing the peak floating point performance and a slanted line, representing the memory bandwidth of the device. The two lines intersect at the point of peak computational performance and peak memory bandwidth.

The performance of a kernel is plotted as a point below the roofline. We can draw a vertical line from this point towards the ceiling, which gives a hint about the limiter of the kernel. If the vertical line intersects with the slanted roofline, the kernel is memory bound. If the vertical line intersects with the horizontal roofline, the kernel is compute bound. If the point is located exactly on the roofline, the kernel utilizes peak memory and/or compute capabilities of the device. This is illustrated in figure 3.18.

Nsight Compute differentiates between a general roofline model, showing different

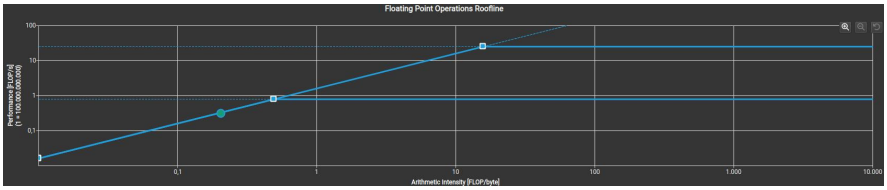


Figure 3.19.: General roofline model in the Speed of Light section. In this example, the kernel is clearly memory bound.

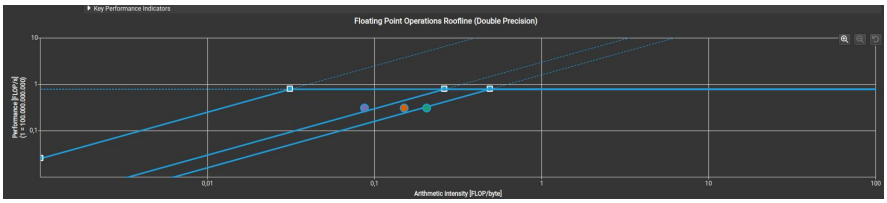


Figure 3.20.: Double-precision roofline model in the Speed of Light section. In this example, the kernel is clearly memory bound.

performances like single- and double-precision in one diagram (see figure 3.19), and hierarchical roofline models per precision used (half, single and double). Figure 3.20 shows the roofline model for a double-precision kernel, containing one point for L1, L2 and DRAM throughputs, respectively.

The **Compute Workload Analysis** section shows how well the device is kept busy by showing the instruction throughput and the amount of executed instructions per cycle. Additionally, charts are displayed that show the utilization levels of different pipes, e.g. double-precision operations (FP64), single-precision and integer multiply operations, logical and integer instructions, and load and store instructions (LSU).

The **Memory Workload Analysis** section shows the utilization of the different memory levels. Metrics like memory throughput, cache hit rates, sectors per request ratios can be viewed in a chart and in a table. Additionally, bank conflicts can be identified here.

In the **Scheduler Statistics** section we can see how many warps are active per scheduler compared to the theoretical maximum at full occupancy. Furthermore, it is stated how many warps are eligible to issue an instruction and how many warps actually issued an instruction per cycle. If a warp is not eligible to issue an instruction, it means that it is stalled for a dependency.

The **Warp State Statistics** section reveals how many cycles a warp is resident per instruction issued. In particular, it breaks down the cycles spent across different warp states, revealing the causes of stalls. For instance, a memory-bound kernel

usually has two primary sources of stalls: Stall Long Scoreboard, which indicates that the warp is waiting for data to arrive from memory, and Stall LG Throttle, which indicates that the warp is waiting for space to become available in the load/store instruction queue. While the former might indicate a poor memory access pattern, the latter occurs when memory requests are extremely frequent.

Furthermore, it is stated how many work-items are active per warp on average. This provides insight into the number of redundant work-items per warp that do not participate in computation.

In the **Occupancy** section we see the theoretical maximum occupancy of our kernel, which is limited by the amounts of registers, shared memory and work-items (block size). Additionally, graphs are displayed that show the effect on occupancy when varying the aforementioned limiters. The actual achieved occupancy is calculated as the ratio of average number of active warps per cycle to the maximum number of possible active warps per compute unit. Large discrepancies between the theoretical maximum and the achieved occupancy indicate imbalances in the workload.

The **Source Counters** section gives insight into the branch divergence level of the kernel. Furthermore, it highlights the instructions that contribute most to stalls. For memory bound kernels, particularly the L2 Theoretical Sectors Global Excessive metric shows how many excessive sectors are requested in L2 from global memory instructions due to uncoalesced access per source instruction. Nsight Compute allows to navigate directly to the respective instruction in the source view, which will be covered next.

Source view. The source view provides information about the SASS code. We can determine, how our OpenCL C or PTX code is translated into SASS, which reveals whether changes in the kernel source code have effects on the final machine code. For example, usually reordering independent statements in the OpenCL C code does not change the produced SASS code. For CUDA programs, we can pass the `-lineinfo` option to the compiler `nvcc`, such that it is closely trackable, which CUDA statement translates into which SASS instructions. Unfortunately, this is not possible with OpenCL, not even with PoCL.

Furthermore, different useful metrics are displayed, e.g. register usage and dependencies, scoreboard dependencies (i.e. global memory dependencies), excessive global sectors and warp stall reasons. It can be used to identify hotspots inside the kernel.

Raw view. While the aforementioned sections present the results in a visually appealing way and calculate aggregate values, sometimes it is useful to take a look at the raw data. All collected metrics are listed in a table for each profiled kernel in the raw view. The table can be filtered for specific metrics, which is especially

useful to compare values between multiple kernels. Lastly, the raw data can be exported to a CSV file.

Chapter 4

Implementation of Multigrid Components on Single-GPU and its Optimizations

In this chapter, we will discuss our implementation of the multigrid method, focusing on GPU usage. The implementation can be dissected into several, mostly independent parts. For each part we will start with a naive implementation and then successively discuss some optimization opportunities. Each optimization is evaluated immediately, such that we can compare the performance of the different implementations and the sources of the performance differences.

Note that the optimizations are mainly tested on two different Nvidia GPUs: Quadro T2000 Max-Q – a smaller Laptop GPU – and an A100 on the Pleiades cluster [8], which is a high-end GPU operating in dedicated GPU clusters and supercomputers. While certain optimizations work well on GPUs, optimizing for other device types may require different strategies. These will, however, not be discussed here, as we focus on optimizing specifically for the GPU.

Although the optimizations are mainly focused on the GPU kernels, writing efficient host code is also important. Sometimes, especially for coarse grids, running parts of the solver is even faster executed sequentially on the CPU due to kernel launch overheads and/or an insufficient parallelization degree. Since we are using C++ for the host code, we can make use of object-oriented programming features to enhance important software engineering principles, such as a clear code organization, modularity and reusability. Main benefits of using classes with constructors and destructors are, in our case, using RAII (Resource Acquisition is Initialization) for host and device buffer management, the management of the OpenCL environment, using smart pointers etc. On the other side, performance penalties can easily be introduced in the object-oriented context, for example when passing a copy of an object instead of using the move semantics or references.

```
class Cuboid {
  std::vector<double> field_1d;
  double*** field_3d;
  // ...
  inline double** operator[](int index) { return field_3d[index]; }
};

mgcl::Cuboid c(2,4,8);
c[1][2][3] = 5;
assert(c.field1d()[1*4*8 + 2*8 + 3] == 5);
```

Listing 4.1: Accessing elements using the Cuboid class in 3d fashion can only be inlined in different compilation units when using LTO.

A meaningful technology, which does not affect our code directly, is Link-Time-Optimization (LTO). Consider our class `Cuboid`, which models a 3d array. It allocates a consecutive block of memory and creates a 3d array of pointers to the memory block, enabling access to data in a 3d fashion, by overloading the `operator[]` method (see listing 4.1). While this greatly enhances usability of the class, without LTO, the compiler would not be able to inline the call to the `operator[]` method in different compilation units, decreasing performance significantly. Fortunately, using modern CMake features, automatically testing for and enabling LTO support is a trivial task.

Since we do not want to completely rely on LTO to be available, in performance critical parts of our implementation, like the calculation of the residual, we extract the raw pointer to the data inside the `Cuboid` and use it directly. This way, we circumvent the `operator[]` method entirely.

4.1. Implementation Overview

Before delving into the optimization approaches in detail, we first want to give a quick overview of the different parts of our implementation and discuss their purpose and their relevance regarding optimization.

Overall, we aim to optimize the individual parts mainly for large grids, as operations on such have the biggest impact on overall performance. For example, solving a problem on a grid of size 64^3 only takes about 21 % of the time of a problem on a grid of size 128^3 using our implementation on a single A100 GPU. But still, the coarser grids shall not be neglected, as not only the single kernels are important, but the general program flow as well.

Our implementation can be split into the following parts:

- Ghost update
- Jacobi's method

- Residual calculation (and its norm)
- Restriction, prolongation and error correction
- Constructing the coarse grid operator using the Galerkin approach (setup only)

Ghost update. A core operation of the multigrid method lies in the application of stencils onto the grid points. For this, the surrounding grid points are needed. While this is straight forward for inner grid points, there are generally two approaches to handle grid points at the border of the domain. We can either detect the position of a grid point, i.e. determine whether it lies on the boundary or not, and then conditionally calculate the index of the neighboring grid points. While this has the advantage of no redundant memory overhead, the implementation is slightly more complicated than the alternative. More importantly, introducing branching in SIMT or SIMD always results in a performance penalty. Plus, for the calculation of indices of grid points from the opposite border (when using periodic boundary conditions), we need the modulo operator, which is expensive by nature. Additionally, we would need to include the choice of the boundary condition in the stencil application logic.

The alternative is to introduce a layer of so-called ghost points at each border. These ghost points are set to appropriate values, depending on the chosen boundary condition, before applying a stencil. The logic of the stencil application is then quite simple, as it does not change for inner grid points and those at the boundary. The drawback obviously is that we need more memory for each ghost layer we introduce. However, for large grids, the ratio of real grid points to ghost points is usually small, such that this is not a major issue. For example, for a grid of size 128^3, adding a ghost layer of width 1 would result in a grid of size 130^3. Thus, only $100\,\% - 128^3/130^3 = 4.5\,\%$ of the grid points are redundant ghost points.

When using fixed boundary values, which appear for example for Dirichlet boundary conditions, we can set the values of the ghost points once at the start of the method and do not need to change them afterwards (in the single-GPU case). However, when using periodic boundary conditions or in the multi-GPU case, the values of the ghost points need to be updated before each application of the stencil. This is handled by the `updateGhosts` routine. The details of the implementation will be discussed in section 4.4.

Residual calculation. The calculation of the residual is mainly part of Jacobi's method. This is where the stencil is applied onto the grid points, which form the most expensive step of the solving process. There is plenty literature of optimizing stencil based code, both for traditional CPU plus MPI code and modern GPU code. Most of this literature, however, focuses solely on fixed coefficient stencil applications, leaving the variable coefficient case quite unexplored. Because of

this and the fact, that using varying coefficient yields the opportunity of solving a broader field of problems, we primarily focus on the variable coefficient case. However, studies and main optimization techniques for fixed coefficient problems shall be presented as well.

Another part of the residual step is the calculation of the residual norm. `mgcl` supports the calculation of the infinity-norm and the L2-norm. The interesting part of this is building the global sum of the global residual array in parallel. While being a fairly simple operation at glance, some effective optimization techniques can be applied.

Jacobi's method. In the sense of the implementation, Jacobi's method adds operations to the residual calculation and performs these for multiple iterations. While the residual calculation optimization focuses mainly on the isolated kernel itself, for Jacobi's method the program flow, i.e. the chain of different kernels, is interesting to investigate. Consequently, key factors influencing performance might change when switching from single-GPU to multi-GPU.

Coarse grid operator calculation. The calculation of the coarse grid operator using the Galerkin approach is part of the setup phase. Depending on the specialization of the implementation, the impact on the total runtime can vary drastically. Since the naive solution comes with severe limitations to both memory consumption and runtime performance, optimization by tailoring the implementation to our needs is essential.

Restriction, prolongation and error correction. The inter-grid transfer operations and the error correction are quite straight-forward in the sense of implementation. Since they contribute to the overall runtime only by a small fraction, only general best practices are considered.

4.2. Jacobi's Method and Residual Calculation With Varying Coefficients

Jacobi's method is the most expensive part of our solver (excluding the setup phase). Thus, most of our effort is dedicated to optimizing the implementation of Jacobi's method and the residual calculation.

It makes sense to address the residual calculation and Jacobi's method together, as the latter includes the residual calculation. A basic sequential implementation, applying a 27-point stencil with varying coefficients, might look like in listing 4.2. Note that the code in the listing is simplified for the sake of readability and does not include details like different loop boundaries for the different grid arrays,

provides no differentiation in the type of residual norm and boundary conditions and does not employ error checking.

```
void residual(/* ... */) {
    double stencilsum = 0;
    for (int i = gh; i < m + gh; i++)
    for (int j = gh; j < n + gh; j++)
    for (int k = gh; k < o + gh; k++) {
        // A*v
        stencilsum = coeffs[1][1][1][i][j][k] * v[i][j][k]
            + coeffs[1][1][0][i][j][k] * v[ i ][ j ][k-1] + coeffs[1][1][2][i][j][k] * v[ i ][ j ][k+1]
            + coeffs[1][0][1][i][j][k] * v[ i ][j-1][ k ] + coeffs[1][2][1][i][j][k] * v[ i ][j+1][ k ]
            + coeffs[0][1][1][i][j][k] * v[i-1][ j ][ k ] + coeffs[2][1][1][i][j][k] * v[i+1][ j ][ k ]

            + coeffs[1][0][0][i][j][k] * v[ i ][j-1][k-1] + coeffs[1][0][2][i][j][k] * v[ i ][j-1][k+1]
            + coeffs[1][2][0][i][j][k] * v[ i ][j+1][k-1] + coeffs[1][2][2][i][j][k] * v[ i ][j+1][k+1]
            + coeffs[0][1][0][i][j][k] * v[i-1][ j ][k-1] + coeffs[0][1][2][i][j][k] * v[i-1][ j ][k+1]
            + coeffs[2][1][0][i][j][k] * v[i+1][ j ][k-1] + coeffs[2][1][2][i][j][k] * v[i+1][ j ][k+1]
            + coeffs[0][0][1][i][j][k] * v[i-1][j-1][ k ] + coeffs[0][2][1][i][j][k] * v[i-1][j+1][ k ]
            + coeffs[2][0][1][i][j][k] * v[i+1][j-1][ k ] + coeffs[2][2][1][i][j][k] * v[i+1][j+1][ k ]

            + coeffs[0][0][0][i][j][k] * v[i-1][j-1][k-1] + coeffs[0][0][2][i][j][k] * v[i-1][j-1][k+1]
            + coeffs[0][2][0][i][j][k] * v[i-1][j+1][k-1] + coeffs[0][2][2][i][j][k] * v[i-1][j+1][k+1]
            + coeffs[2][0][0][i][j][k] * v[i+1][j-1][k-1] + coeffs[2][0][2][i][j][k] * v[i+1][j-1][k+1]
            + coeffs[2][2][0][i][j][k] * v[i+1][j+1][k-1] + coeffs[2][2][2][i][j][k] * v[i+1][j+1][k+1];

        // r = f - A*v
        r[i][j][k] = f[i][j][k] - stencilsum;
    }
}

void jacobi(/* ... */) {
    for (int iter = 0; iter < maxiter; iter++) {
        updateGhosts(v);
        residual(f, v, r);

        for (int i = gh; i < m + gh; i++)
            for (int j = gh; j < n + gh; j++)
                for (int k = gh; k < o + gh; k++) {
                    v[i][j][k] = v[i][j][k] + omega * (1.0 / coeffs[1][1][1][i][j][k]) * r[i][j][k];
                }
    }
}
```

Listing 4.2: Naive sequential implementation of Jacobi's method and residual calculation.

Note that `v`, `f` and `r` are instances of the `Cuboid` class, representing the different quantities on the 3d grid, i.e. `v` is the current approximation, `r` stores the current residual and `f` is the right-hand side of our equation. `coeffs` is an instance of our `StencilValues` class, which wraps a 6d array for storing a 27-point stencil with varying coefficients for each grid point. The layout is coefficients-first, i.e. we select the coefficient using the first three indices and the grid point using the last three indices. The choice of coefficient data layout is discussed in section 4.2.2.

Furthermore, note that we already include an optimization by specialization: Instead of looping over the coefficients, we restrict our code to handle only 27-point stencils. By writing out the equation explicitly, we are giving the compiler more optimization opportunities.

In the following subsections, different implementation strategies are presented and evaluated. Since there are different variables that can be changed, each of which affect the overall performance, it is challenging to establish a clear optimization order without impairing the comparability of the different variants. Thus, we will presume some fundamental optimization choices and see, how varying other

aspects affects the performance. All metrics collected during profiling can be viewed in the appendix. Tables A.3 and A.4 in the appendix serve as a reference for important metrics.

4.2.1. Basic Residual Kernel

4.2.1.1. One Work-item Per Grid Point (Automatic Choice of Work-group Size)

We begin by considering only the residual calculation. There are different approaches to parallelize the code shown before. An intuitive one is to remove the 3d loop over the grid points and thus assign one work-item to one grid point. A first kernel might look like in listing 4.3. As arguments, we obviously need all the relevant grids, i.e. v, f and the coefficients `stencilValues` as input and r as output. Furthermore, since we can only access an array in OpenCL C using the 1d index, we also need the information about the grid size and the number of ghosts. We assume that v, f and r all have the same grid extent $m \times n \times o$ and number of ghosts gh, however the number of ghosts can be different for the coefficients array, depending on the way the Galerkin operator is calculated. An extend of the ghosted grid is denoted by $d_{gh} = d + 2gh$ for $d \in \{m, n, o\}$.

The index of the grid point that a work-item is responsible for is simply attained using the built-in function `get_global_id`. Since we are using a 3d `ndrange`, we can directly fetch the 3d indices of the grid point. The 1d index is then calculated using the grid dimensions as $idx_{1d} = ino + jo + k$. The 1d index of the coefficients array is calculated similarly, respecting the possibly different number of ghosts.

Albeit the 1d indexing, the application of the stencil is quite similar to the sequential version. In order to find the neighboring grid points, we add or subtract offsets to the 1d index. For v, f and r, the offsets have the sizes $\pm no$, $\pm o$ and ± 1 in x, y and z-direction, respectively.

We use a coefficients-first layout for our coefficients here, e.g. the center coefficient for grid point (i, j, k) is stored at the index $(1, 1, 1, i, j, k)$. Thus, the offsets of the coefficients include the multiplication of 9, 3 and 1 with the entire ghosted grid size $m_{gh} \times n_{gh} \times o_{gh}$ in x, y and z-direction, respectively.

Lastly, the kernel must be called with exactly $m_{gh} \times n_{gh} \times o_{gh}$ work-items. The `if`-statement ensures that only work-items that correspond to real grid points (i.e. not ghost points) perform calculations. Additionally, it prevents redundant work-items, that stem from padding, from accessing out-of-bounds memory.

```
__kernel void residual_27point_varying_stencil_coeffs_first_3d_m0(
    __global double* restrict v_in,
    __global double* restrict f,
    __global double* restrict r,
    __global double* restrict stencilValues,
    const int mgh, const int ngh, const int ogh,
    const int svmgh, const int svngh, const int svogh,
    const int ghosts, const int ghosts_sv,
    const int svGridSize)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    int k = get_global_id(2);

    if (i >= ghosts && j >= ghosts && k >= ghosts && i < mgh - ghosts && j < ngh - ghosts && k < ogh - ghosts)
    {
        int ioff = ngh * ogh;
        int joff = ogh;
        int koff = 1;
        int index = i * ioff + j * ogh + k;

        int svno = svngh * svogh;
        int index_sv = (i - ghosts + ghosts_sv) * svno + (j - ghosts + ghosts_sv) * svogh + (k - ghosts +
            ↪ ghosts_sv);

        double stencilsum = stencilValues[index_sv + (9 + 3 + 1) * svGridSize] * v_in[index]
            + stencilValues[index_sv + (9 + 3) * svGridSize]      * v_in[index - 1]
            + stencilValues[index_sv + (9 + 3 + 2) * svGridSize]  * v_in[index + 1]
            + stencilValues[index_sv + (9 + 1) * svGridSize]      * v_in[index - joff]
            + stencilValues[index_sv + (9 + 6 + 1) * svGridSize]  * v_in[index + joff]
            + stencilValues[index_sv + (3 + 1) * svGridSize]      * v_in[index - ioff]
            + stencilValues[index_sv + (18 + 3 + 1) * svGridSize] * v_in[index + ioff]

            + stencilValues[index_sv + (9) * svGridSize]          * v_in[index - joff - koff]
            + stencilValues[index_sv + (9 + 2) * svGridSize]      * v_in[index - joff + koff]
            + stencilValues[index_sv + (9 + 6) * svGridSize]      * v_in[index + joff - koff]
            + stencilValues[index_sv + (9 + 6 + 2) * svGridSize]  * v_in[index + joff + koff]
            + stencilValues[svGridSize * 3 + index_sv]            * v_in[index - ioff - koff]
            + stencilValues[index_sv + (3 + 2) * svGridSize]      * v_in[index - ioff + koff]
            + stencilValues[index_sv + (18 + 3) * svGridSize]     * v_in[index + ioff - koff]
            + stencilValues[index_sv + (18 + 3 + 2) * svGridSize] * v_in[index + ioff + koff]
            + stencilValues[svGridSize + index_sv]                * v_in[index - ioff - joff]
            + stencilValues[index_sv + (6 + 1) * svGridSize]      * v_in[index - ioff + joff]
            + stencilValues[index_sv + (18 + 1) * svGridSize]     * v_in[index + ioff - joff]
            + stencilValues[index_sv + (18 + 6 + 1) * svGridSize] * v_in[index + ioff + joff]

            + stencilValues[index_sv]                           * v_in[index - ioff - joff - koff]
            + stencilValues[svGridSize * 2 + index_sv]            * v_in[index - ioff - joff + koff]
            + stencilValues[index_sv + (6) * svGridSize]          * v_in[index - ioff + joff - koff]
            + stencilValues[index_sv + (6 + 2) * svGridSize]      * v_in[index - ioff + joff + koff]
            + stencilValues[index_sv + (18) * svGridSize]         * v_in[index + ioff - joff - koff]
            + stencilValues[index_sv + (18 + 2) * svGridSize]     * v_in[index + ioff - joff + koff]
            + stencilValues[index_sv + (18 + 6) * svGridSize]     * v_in[index + ioff + joff - koff]
            + stencilValues[index_sv + (18 + 6 + 2) * svGridSize] * v_in[index + ioff + joff + koff];

        r[index] = f[index] - stencilsum;
    }
}
```

Listing 4.3: Basic 3d residual kernel applying a 27-point spatially varying stencil

Assuming the kernel object is created and the arguments are set up, listing 4.4 shows how the kernel can be enqueued. Currently, the work-group size is chosen automatically and thus, no padding is applied.

```
size_t global[3] = {mgh, ngh, ogh};
clEnqueueNDRangeKernel(commands, kernel, 3, NULL, global, NULL, 0, NULL, NULL);
```

Listing 4.4: Enqueuing the 3d residual kernel without padding and with automatic choice of work-group size

Running on the Quadro T2000 GPU, this kernel takes around 108 ms to complete for a real grid size of 128^3 and $gh = 1$. We measure only the best time from start to end of multiple sample runs using Nsight Compute (NCU), i.e. all the overhead of building the kernel object, setting up the arguments and enqueuing the kernel is excluded.

We can check the automatically chosen work-group size either from inside the kernel using `get_local_size` or by analyzing the kernel using NCU. The kernel is launched with a work-group size of `130x1x1`, i.e. `1x130x130`, work-groups are used. NCU shows that this kernel only achieves 1.73 % compute throughput and 41.21 % memory throughput. The SM Busy metric is only at 1.73 %, showing that the device is heavily underutilized.

There are multiple issues with this kernel and how it is launched. First, the work-group size is not a multiple of the warp size, which is 32 work-items for Nvidia GPUs. So for each work-group, there are $(\lfloor 130/32 \rfloor + 1) \cdot 32 - 130 = 160 - 130 = 30$ work-items that are simply masked off by the hardware. Second, the mapping of work-item indices to the memory layout is not optimal, resulting in a poor memory access pattern. In the following, we will successively address the issues and observe how they affect performance.

4.2.1.2. One Work-item Per Grid Point (Manual Choice of Work-group Size)

Fortunately, reducing redundant work-items is easy to fix. We do not let the OpenCL runtime choose the work-group size automatically, but instead set it manually to a multiple of the warp size. Thinking about the 3d nature of the code, a work-group size of `4x4x4`, i.e. 64 work-items per work-group, intuitively might be a good choice. In order to make sure that all work-items in the work-group are used, we need to pad each `ndrange` dimension to a multiple of the corresponding work-group size dimension. This is shown in listing 4.5.

Now the kernel only takes around 44 ms to complete, without changing anything at the kernel itself. While this change has an exceptional impact in this case, in

```
global[3] = {mgh, ngh, ogh};
local[3] = {4, 4, 4};

for (int i = 0; i < 3; i++)
    if (global[i] % local[i] != 0)
        global[i] += local[i] - (global[i] % local[i]);

clEnqueueNDRangeKernel(commands, kernel, 3, NULL, global, local, 0, NULL, NULL);
```

Listing 4.5: Enqueuing the 3d residual kernel with padding and with a manually chosen work-group size of `4x4x4`

other cases the automatic choice might work just as well as picking it manually. However, as it is a simple change to implement and assures a certain level of control and clarity, we recommend to manually pick work-group sizes whenever possible.

NCU now shows a compute throughput of 3.7 % and a memory throughput of 39.41 %. The SM Busy metric is at 3.68 %, which is a 113 % increase compared to our previous attempt, but still a low value.

We see that from a theoretically achievable occupancy of 100 %, only 63.66 % is achieved, i.e. there are many inactive warps. The Warp State Statistics section indicates that, on average, each warp spends 746 cycles being stalled waiting for slots in the L1 instruction queue for global memory access to become available (Stall LG Throttle). This occurs for very frequent memory accesses.

Before considering the memory access pattern in more detail, we aim to tackle the second aforementioned issue, addressing the mapping between 3d work-group index and 1d work-item inside a warp. As mentioned in section 3.2.3 and as shown in section 5.2 of [77], for a work-group size of (D_x, D_y, D_z) the work-item index of a work-item with index (x, y, z) is calculated as $x + yD_x + zD_xD_y$. Translated to OpenCL terminology, this means we should use `get_global_id(0)` for the fastest varying index of memory arrays and `get_global_id(2)` for the slowest varying index. Taking a look at Listing 4.3 again, we assign our first index i to `get_global_id(0)`, second index j to `get_global_id(1)` and last index k to `get_global_id(2)`. It is a simple change to swap the indices, see listing 4.6.

```
// before
int i = get_global_id(0);
int j = get_global_id(1);
int k = get_global_id(2);

// after
int i = get_global_id(2);
int j = get_global_id(1);
int k = get_global_id(0);
```

Listing 4.6: Swapping the work-item ids to match memory layout

With this change, the kernel now only takes around 14 ms to complete, which is a 314 % increase in performance compared to the previous version, see table 4.1.

Kernel Variant	**Time [ms]**	**SoL CT [%]**	**SoL MT [%]**
Auto-choice of wg-size	108	1.7	41
3d kernel, `4x4x4` wg, suboptimal wi-indexing	44	3.7	39
3d kernel, `4x4x4` wg, optimal wi-indexing	14	11	47

Table 4.1.: Performance of different 3d residual kernel variants on a single Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$. CT: Compute Throughput, MT: Memory Throughput, SoL: Speed of Light, i.e. theoretical maximum, wg: work-group.

Both the compute and the memory throughput are increased to 11.0 % and 47 %, the SM Busy metric is at 11 %, which is a 205 % increase compared to the previous version. The most important difference can be seen in the warp state statistics. We now only spend 282 cycles on Stall LG Throttle and 85 cycles on Stall Long Scoreboard. The full result set can be found in the appendix in table A.5.

These improvements stem from an improved memory access pattern. With the suboptimal work-item indexing, the layout does not match the memory layout of v, resulting in a typical strided access pattern. For a grid of size `128x128x128`, there is a memory gap of 128^2 elements between each of the first four work-items (indices $0, 0, 0$ to $0, 0, 3$) of the warp, see figure 4.1. The memory element accessed by the fifth work-item (index $0, 1, 0 = 4$) is located 128 elements away from the first work-item. The seventeenth work-item (index $1, 0, 0 = 16$) accesses an element in memory that is located right beneath the memory element accessed by the first work-item. All in all, out of the 16 double-precision elements that can be fetched with a single 128-byte transaction, only two work-items (e.g. $0, 0, 0$ and $1, 0, 0$ or $2, 1, 1$ and $3, 1, 1$) of one of the two warps, that the work-group is composed of, can be used. Thus, in order to satisfy all work-items in the warp, $32/2 = 16$ memory transactions are needed.

In contrast, when using the improved work-item indexing, it fits to the indexing of v, see figure 4.2. Four consecutive work-items of a warp can be satisfied by a single memory transaction. Moreover, the memory access executed by those work-items happens in a coalesced manner, i.e. work-items and elements in memory are direct neighbors.

Still, we need $32/4 = 8$ memory transactions to fetch one value of v per work-item for one warp, so there is still room for improvement.

In the previous memory analysis we saw that memory access is fastest, when the memory region being fetched by a warp is contiguous. Continuing this line of thought, we can try to choose a work-group size that is more elongated in one dimension, e.g. `64x1x1`. This way, a higher coalescing of memory access can be achieved, reducing the amount of redundant memory transactions.

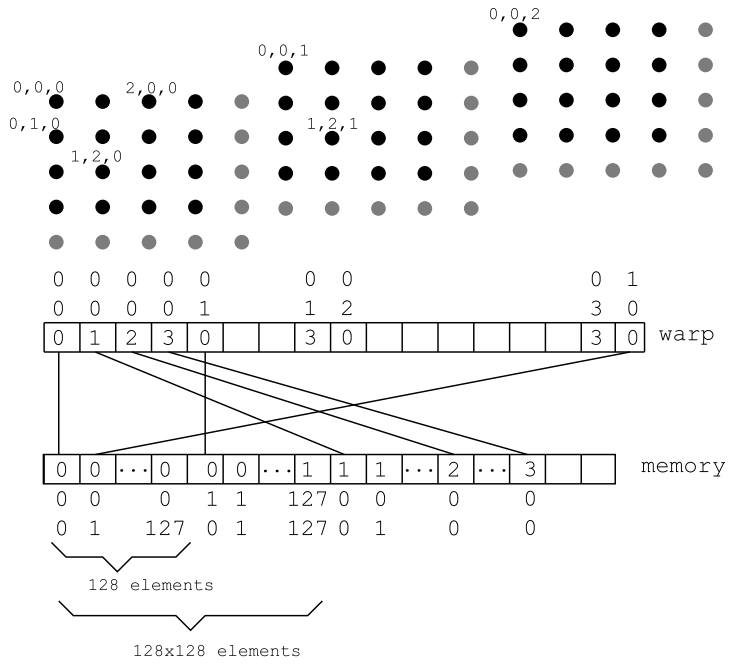


Figure 4.1.: Memory access pattern of v in the 3d residual kernel with wg-size `4x4x4` and suboptimal work-item indexing

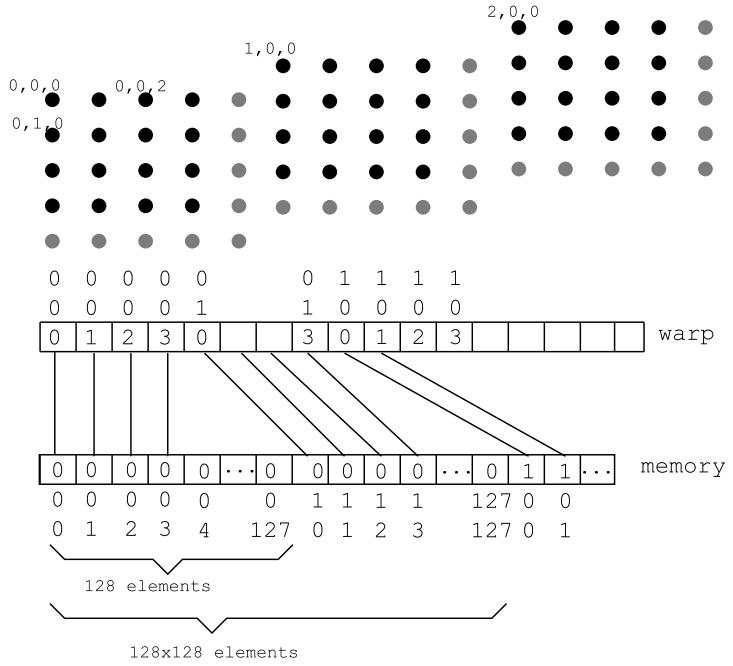


Figure 4.2.: Memory access pattern of v in the 3d residual kernel with wg-size `4x4x4` and improved work-item indexing

Kernel Variant	**Time [ms]**	**SoL CT [%]**	**SoL MT [%]**
Auto-choice of wg-size	108	1.7	41
3d kernel, `4x4x4` wg, suboptimal wi-indexing	44	3.7	39
3d kernel, `4x4x4` wg, optimal wi-indexing	14	11	47
3d kernel, `8x1x1` wg	6.70	95	51
3d kernel, `16x1x1` wg	3.77	89	91
3d kernel, `32x1x1` wg	**3.75**	50	92
3d kernel, `64x1x1` wg	4.00	47	91
3d kernel, `96x1x1` wg	3.97	47	91
3d kernel, `128x1x1` wg	3.94	47	91
3d kernel, `256x1x1` wg	3.82	49	92

Table 4.2.: Performance of different 3d residual kernel variants on a single Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$. CT: Compute Throughput, MT: Memory Throughput, SoL: Speed of Light, i.e. theoretical maximum, wg: work-group.

There are different work-group sizes we want to investigate. Following the best practices guidelines, we want to choose a work-group size that is a multiple of the warp size, i.e. 32 for our Nvidia GPUs. However, since our kernel is very memory-intensive, we might also try using smaller work-group sizes to give the scheduler more flexibility for hiding long-lasting memory instructions. We want to test the work-group consisting of $8, 16, 32, 64, 128$ and 256 work-items. Table 4.2 shows the kernel duration and achieved compute and memory throughput for each work-group size. Note that, in order to keep the optimal work-item indexing inside the warp, the first work-group dimension must be elongated instead of the last one.

As we can see, all one-dimensional work-groups perform significantly better than the previously attempted 3d work-group. The best performance is achieved for a work-group size of `32x1x1`, closely followed by `16x1x1`. The difference to the other larger work-groups is not substantial either.

The only work-group size that falls behind is `8x1x1`. While this variant exhibits the highest compute throughput (95 %), the memory throughput is remarkably low (51 %), compared to all other variants. While the scheduler has enough warps to switch in when other warps are stalled on memory instructions, half of the data that is being fetched by one warp, i.e. only 8 of 16 possible double-precision values in a single 128-byte transaction, is not used. This results in almost twice the overall amount of required memory transactions compared to work-group size `16x1x1`: The metric `l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum` in NCU shows, that there are 16.18 million vs. 8.57 million memory requests from global to L1 for the `8x1x1` and `16x1x1` work-group size, respectively. It can also be seen in the `derived__memory_l2_theoretical_sectors_global_excessive`

metric: There are 254 thousand vs. 123 thousand excessive theoretical number of L2 sectors for work-group sizes `8x1x1` and `16x1x1`, respectively. Furthermore, only around a quarter of the warp is active for `8x1x1` and half for `16x1x1`, which can be seen in the metric `smsp__thread_inst_executed_per_inst_executed.ratio`, which is at 7.64 and 14.63 work-items for `8x1x1` and `16x1x1`, respectively.

The comparison between the work-group sizes `16x1x1` and `32x1x1` is interesting as well. While the memory throughput is almost identical, the compute throughput of the smaller work-group size has almost doubled. This stems from the fact that for a work-group size of `16x1x1`, only half of a warp is active at any time. Thus, there are more warps available to switch in when other warps are stalled on memory instructions. This can be seen when comparing the number of warps being issued per scheduler per cycle (metric `smsp__issue_active.avg.per_cycle_active`): 0.12 vs. 0.07. Also, as expected, the two most impactful stall reasons are less significant for the smaller work-group size: 53 cycles vs. 32 cycles (Stall Long Scoreboard) and 6.39 cycles vs. 0.07 cycles (Stall LG Throttle) for `32x1x1` and `16x1x1`, respectively. All in all however, this does not compensate for the highly increased amount of instructions needed.

To wrap up the memory access pattern analysis, table 4.3 shows the ratio of sectors per memory request (per warp) for the different work-group sizes, which gives an indication of how well memory access is coalesced. `4x4x4` with suboptimal work-item indexing has the expected ratio of 16 sectors per request. The ratio for `4x4x4` with optimal work-item indexing lies around 12 sectors per request, which we actually rather expected to be around 8, since the 4 elements requested by neighboring work-items do fit into a single 32-byte memory transaction. One reason for this deviation might be misalignment: If one of those 4 elements is spread over two 32-byte-aligned memory chunks, two sectors must be fetched instead of one. If this misalignment occurs only partially over the entire warp, a fractional value for the sectors-to-requests ratio may be the result. Since we must not only obtain the value of v for the grid point that the work-item is mapped to, but all the values for the 27-point stencil, not every chunk of consecutive four elements is aligned to 32 bytes.

The ratios 2 for `8x1x1` and 4 for `16x1x1` fit to our expectations, since $8 \cdot 8/32 = 2$ sectors for `8x1x1` and $16 \cdot 8/32 = 4$ sectors for `16x1x1` need to be fetched per warp for a single value of v.

The ratio of sectors to requests is between 6.35 and 6.7 for 1d work-groups with at least 32 work-items. If every work-item needed to get a distinct value of v, we would expect a value of 8 if coalescing was perfect. However, since one work-item also gets the value of surrounding grid points, i.e. some values can be reused by other work-items of the same warp, the ratio is a bit lower for 1d work-groups having at least 32 work-items.

Variant	$\frac{\texttt{l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum}}{\texttt{l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum}}$
`4x4x4` suboptimal wg-indexing	4.108.797/67.619.968 = 16.46
`4x4x4` optimal wg-indexing	4.108.797/48.582.101 = 11.82
`8x1x1`	16.180.940/39.825.327 = 2.46
`16x1x1`	8.566.380/34.111.577 = 3.98
`32x1x1`	4.759.100/31.875.372 = 6.70
`64x1x1`	4.809.800/31.577.984 = 6.57
`96x1x1`	4.809.800/31.472.573 = 6.54
`128x1x1`	4.911.200/31.454.886 = 6.40
`256x1x1`	4.911.200/31.199.044 = 6.35

Table 4.3.: Sectors per request for different 3d residual kernel variants on a single Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$.

Work-group size	**16x1x1**	**32x1x1**	**64x1x1**	**128x1x1**	**256x1x1**
Theoretical Occupancy [%]	50	50	100	100	100
Theoretical Active Warps per SM [warps]	16	16	32	32	32
Achieved Occupancy [%]	48.98	49.46	81.75	82.71	80.93
Achieved Active Warps per SM [warps]	15.67	15.83	26.16	26.47	25.90

Table 4.4.: Occupancies of different 3d residual kernel variants on a single Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$.

One remaining question is, why does the kernel perform better for `32x1x1` than for larger work-group sizes? Considering the occupancy, we note that `32x1x1` is limited to 50 % theoretical occupancy, while the larger variants can theoretically achieve 100 % occupancy on our Quadro T2000 GPU, which might suggest that the larger work-groups should actually perform better. The limit stems from a combination of the maximum active warps per compute unit (SM), which is 32 for CC 7.5, and the maximum resident blocks per SM, which is 16 here. Then, the maximum possible number of warps per SM is

$$
\begin{aligned}
&\text{Maximum possible Warps per SM} \\
&= \text{Upper limit of active work-groups per SM} \cdot \text{Number of warps per block}, \\
&= 16\frac{\text{work-groups}}{\text{SM}} \cdot 1\frac{\text{warp}}{\text{work-group}} = 16\frac{\text{warps}}{\text{SM}} \text{ for work-group size 32 and} \\
&= 16\frac{\text{work-groups}}{\text{SM}} \cdot 2\frac{\text{warps}}{\text{work-group}} = 32\frac{\text{warps}}{\text{SM}} \text{ for work-group size 64.}
\end{aligned}
$$

An overview of theoretical and achieved occupancies for the different work-group sizes can be seen in table 4.4. While the theoretical occupancy is at 100 % for

`64x1x1`, `128x1x1` and `256x1x1`, the achieved occupancy is only between 80 % and 83 %. In contrast, with 49.46 % occupancy, `32x1x1` almost achieves the theoretical maximum of 50 %.

A large discrepancy lies in the stalled warp states, see table 4.5. While there is no significant difference between work-group sizes for Stall Long Scoreboard (`256x1x1` being the exception), there is a significant jump in Stall LG Throttle from `32x1x1` to `64x1x1`. This suggests, that the memory request pipeline is already saturated with the 16 theoretically active warps per SM in case of our residual kernel. More active warps lead to more concurrent memory requests, such that the queue is filled up. Since the memory throughput is already above 90 % for `32x1x1` and our kernel is memory bound, there is no benefit in increasing the number of active warps per SM, i.e. achieving a higher occupancy.

Furthermore, the drop of Stall LG Throttle from `64x1x1` to `128x1x1` and then to `256x1x1` is remarkable, especially in combination with the increased Stall Long Scoreboard for `256x1x1`. The ratio of sectors per request suggests, that coalescing is slightly better for the larger work-group sizes. However, also the total number of sectors transferred is larger. For work-groups with at least 64 work-items, the relative differences between the amount of data transferred via DRAM and the total kernel execution time closely match, indicating that the slightly improved memory access pattern of larger work-groups does indeed reduce overall execution time.

Finally, we want to explain the change in stalls for `128x1x1` and `256x1x1`. Since we extend our 128^3-sized grid by one ghost layer, we actually start a kernel for 130^3 grid points. One line of grid points then must be handled by two `128x1x1` work-groups, where the second one is responsible for only the two last grid points, leaving 126 work-items of that work-group unused. For `256x1x1` almost half of the work-groups are unused, since, with our 3d work-item indexing, we do not wrap the remaining work-items to the next row of the grid. In combination with the work-group limits of `128x1x1` and `256x1x1` being 8 and 4, respectively, due to the limit of 1024 warps per SM, we practically have fewer warps that can be swapped in while others are stalled due to memory requests. On the other side, this circumstance seems to help with caching slightly, as the L1 cache hit rates increase from `64x1x1` to larger work-groups.

On a side note, we want to mention, that the kernel with automatic choice of the work-group size (`130x1x1`) needs 3.91 ms with the optimal work-item indexing, which is very comparable to the manually set work-group size of `128x1x1` (3.94 ms). However, it still is not fully on par with the optimal work-group size of `32x1x1`.

All used NCU metrics for the 3d `ndrange` residual kernel with 1d work-group sizes are given in table A.6 in the appendix.

Work-group size	16x1x1	32x1x1	64x1x1	128x1x1	256x1x1
Stall LG Throttle [cycles]	0.07	6.39	37.01	28.25	15.59
Stall Long Scoreboard [cycles]	32.28	53.14	56.69	57.56	70.80
DRAM sectors read	16.5M	16.6M	17.7M	17.5M	16.9M
L1 hit rate [%]	33.78	27.09	27.78	30.29	32.63

Table 4.5.: Warp stalls for the residual kernel with varying coefficients on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$.

Work-group size	16x1x1	32x1x1	64x1x1	128x1x1	256x1x1
Time [ms]	3.77	3.75	4	3.94	3.82
CT [%]	88.98	49.68	46.57	47.45	48.72
MT [%]	90.61	91.52	91.04	91.47	91.7
L1 requests	8.6M	4.8M	4.8M	4.9M	4.9M
L1 sectors	34.1M	31.9M	31.6M	31.5M	31.2M
L1 sectors / requests	3.98	6.70	6.57	6.40	6.35
L1 hit rate [%]	33.78	27.09	27.78	30.29	32.63
L2 hit rate [%]	39.26	34.74	31.79	30.8	30.17
Stall LG Throttle	0.07	6.39	37.01	28.25	15.59
Stall Long Scoreboard	32.28	53.14	56.69	57.56	70.8
SM Busy [%]	89	49.75	46.78	47.44	48.92

Table 4.6.: Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using one work-item for one grid point

Instead of launching the kernel on a 3d `ndrange`, we can also use a 1d `ndrange` and calculate the 3d grid point from the 1d work-item index. While the kernel runtime remains unaffected by this change, it has two potential benefits. First, the risk of confusion regarding the mapping between grid points and work-items is reduced. Second, the work-items of larger work-groups are automatically wrapped to the next row of grid points. Since we are using 1d work-groups for both variants, there are no differences in padding we need to be concerned about.

A summary of the most important metrics for the 3d residual kernel for different work-group sizes on a grid of size 128^3 with $gh = 1$ for the Quadro T2000 is given in table 4.6. The largest grid we are able to execute using the Quadro T2000 is of size `128x128x256`, for which the tendencies of the results compared between different work-group sizes are equal to the 128^3 grid.

Table 4.7 shows the results for the same kernel and on the same grid on the A100 GPU. In all cases, the compute throughput and the SM busy metric is fairly low compared to the Quadro T2000. This is expected as our kernel is memory bound and the float to double ratio of the Quadro T2000 is 1:32, where for the A100 it is at 1:2. Furthermore, the A100 has overall higher peak double-precision performance

Work-group size	16x1x1	32x1x1	64x1x1	128x1x1	256x1x1
Time [ms]	0.43	0.38	0.38	0.38	0.38
CT [%]	18.24	11.55	11.7	11.79	11.77
MT [%]	76.51	86.4	86.4	86.18	86.63
L1 requests	8.1M	4.5M	4.5M	4.5M	4.5M
L1 sectors	34M	31.8M	30.7M	30.6M	30.5M
L1 sectors / requests	4.19	7.07	6.81	6.79	6.77
L1 hit rate [%]	30.83	28.05	34.34	34.97	36.61
L2 hit rate [%]	50.77	45.75	41.74	41.2	40.35
Stall LG Throttle	0.83	2.04	4.16	3.03	2.01
Stall Long Scoreboard	45.31	74.2	76.19	68.72	68.47
SM Busy [%]	15.35	9.63	10.04	10.76	10.82
Theoretical Occ. [%]	50	50	62.5	62.5	62.5
Achieved Occ. [%]	48.1	48.38	53.69	51.71	51.04

Table 4.7.: Metrics for the varying stencil residual kernel on a A100 GPU for a real grid size of 128^3 and $gh = 1$ using one work-item for one grid point

than the Quadro T2000 (9.7 TFLOPS vs. 89 GFLOPS). The memory throughput is much more comparable. Furthermore, we see, that the kernel performs equally well for any work-group size that is a power of two between 32 and 256. A notable difference is, that the theoretical occupancy is only at 62.5 % for work-groups equal to or larger than `64x1x1`. This is due to register usage of the kernel. For each work-item, 48 registers are allocated. For the limit of 65536 registers per compute unit and 32 work-items per warp, this results in only $65536/(32 \cdot 48) \approx 42$ active warps per compute unit. NCU rounds it further down to 40 warps per compute unit. The A100 (CC 8.0) is capable of launching 64 warps per compute unit, resulting in a theoretical occupancy of 62.5 %. Reducing the amount of registers needed is not a simple task, as the compiler optimizes the balance between register usage and memory latency hiding by itself. One way is to increase TLP, which we attempt in section 4.2.3.

The largest grid we were able to test the kernel on the A100 is of size `256x256x512`. The results are shown in table 4.8. Interestingly, the work-group size that performs best is `256x1x1` here. However, still all work-group sizes from `32x1x1` onwards show comparable results.

4.2.2. Coefficients Data Layout

So far we used a coefficients-first data layout to store the coefficients. We could also choose to store it the other way around, i.e. grid-first. This would change how we calculate the indices of the coefficients and the offsets. More impactful, it changes the memory access pattern. Since a whole warp collectively steps through the multiplications of the coefficients with the approximation, we can

Work-group size	16x1x1	32x1x1	64x1x1	128x1x1	256x1x1	512x1x1
Time [ms]	6.63	6.21	6.15	6.11	6.09	6.14
CT [%]	17.54	9.63	10.17	10.17	10.18	9.99
MT [%]	81.51	85.68	86.78	87.29	87.7	87.14
L1 requests	118M	61M	61M	61M	61M	61M
L1 sectors	538M	501M	483M	481M	478M	477M
L1 sectors / requests	4.52	8.18	7.88	7.85	7.82	7.80
L1 hit rate [%]	28.31	24.84	31.11	32.58	34.56	36.98
L2 hit rate [%]	50.9	46.28	42.56	41.48	40.37	39.19
Stall LG Throttle	1.15	3.34	10.06	7.75	6.28	4.74
Stall Long Scoreboard	48.78	91.16	97.16	93.61	81.92	61.91
SM Busy [%]	14.51	7.98	8.14	8.37	8.78	9.44
Theoretical Occ. [%]	50	50	62.5	62.5	62.5	50
Achieved Occ. [%]	49	49.36	58.35	56.35	51.31	42.69

Table 4.8.: Metrics for the varying stencil residual kernel on a A100 GPU for a real grid size of `256x256x512` and $gh = 1$ using one work-item for one grid point

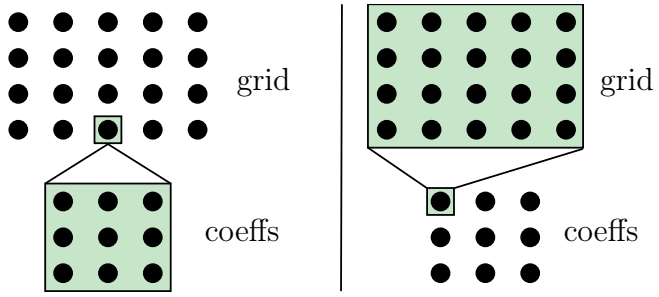


Figure 4.3.: Grid-first (left) and coefficients-first (right) data layout. Grid-first is indexed as `coeffs[gpx][gpy][gpz][cx][cy][cz]`. Coefficients-first is indexed as `coeffs[cx][cy][cz][gpx][gpy][gpz]`.

look at only a single statement for the sake of memory access pattern. If we store the coefficients as coefficients-first, the same coefficients for different grid points are stored contiguously in memory, making consecutive warps access them in a coalesced manner. On the other hand, when we store the coefficients as grid-points-first, all coefficients of a single grid point are stored contiguously in memory. For 27 coefficients and a warp size of 32 work-items, this would result in at most two work-items of the warp being able to use the data received in a single transaction, as there is a stride of 27 elements between the same coefficient for each grid point. The layout difference is illustrated in figure 4.3.

In table 4.9 we observe, that the kernel using grid-first layout performs worse than the previous attempts on the Quadro T2000 GPU. The ratio of sectors to requests is approximately three times higher than for the coefficients-first layout. Consequently, the memory stalls take more cycles, too. The A100 shows similar tendencies – results can be viewed in the appendix in the tables A.7 and A.8.

Work-group size	32x1x1	64x1x1	128x1x1	256x1x1
Time [ms]	5.1	6.77	6.53	6.95
CT [%]	29.86	22.46	23.3	21.98
MT [%]	92.26	91.82	91.15	90.16
L1 requests	3736497	3736498	3736500	3736504
L1 sectors	70869228	70600185	70632094	69997457
L1 sectors / requests	18.97	18.89	18.90	18.73
L1 hit rate [%]	41.54	37.61	38.08	36.29
L2 hit rate [%]	71.94	63.17	64.03	63.65
Stall LG Throttle	62.22	176.38	176.4	180.32
Stall Long Scoreboard	35.74	46.39	44.68	37.73

Table 4.9.: Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using grid-first layout

4.2.3. Kernel Granularity

So far we mapped one work-item to one grid point. We could also increase ILP by mapping one work-item to multiple grid points, or increase TLP by assigning more work-items to one grid point.

When mapping one work-item to multiple grid points, we can easily iterate through the grid points inside the kernel by launching the kernel on a 1d `ndrange` and increment the 1d index of the current grid point by the size of the `ndrange`, see listing 4.7. Of course the size of the `ndrange` is reduced by a factor equal to the amount of grid points per work-item. Basically, we are using the grid-stride loop concept here. This ensures, that neighboring work-items inside a warp are mapped to neighboring grid points, enabling coalesced memory access. No further adjustments are required to the kernel. The mapping is illustrated in figure 4.4.

```
__kernel void residual_27point_varying_stencil_coeffs_first_4_gps_per_thread(/* arguments */) {
    int no = ngh * ogh;

    // E.g. gps_per_wi = 4 for four grid points per work-item. Loop could additionally be unrolled.
    for (int idx = get_global_id(0); idx < gps_per_wi; idx += get_global_size(0)) {
        int i = idx / no;
        int j = (idx - i * no) / ogh;
        int k = idx % ogh;
        // Calculate residual and store in global memory ...
    }
}
```

Listing 4.7: Residual kernel version calculating multiple grid points per work-item.

The results in table 4.10 show, that the kernel performs slightly worse for four grid points per work-item than the kernel using one work-item per grid point. The ratio of sectors to requests is worse for every work-group size, the memory stall reasons as well. Overall, the GPU is kept busy a lower amount of time than for one work-item per grid point. The A100 performs equally well or worse on this kernel type, too. Results are shown in the appendix in tables A.9 and A.10.

Work-group size	16x1x1	32x1x1	64x1x1	128x1x1	256x1x1
Time [ms]	3.83	3.89	4.33	4.24	4.08
CT [%]	79.19	39	35.32	35.85	37.16
MT [%]	89.41	89.47	88.58	89	89.81
L1 requests	7362969	3685005	3685006	3685008	3685008
L1 sectors	31856669	30708220	30469771	30323459	30028340
L1 sectors / requests	4.33	8.33	8.27	8.23	8.15
L1 hit rate [%]	33.72	25.27	23.48	24.4	32.25
L2 hit rate [%]	25.97	28.96	24.68	24.52	18.49
Stall LG Throttle	0.03	9.65	74.59	74.03	39.45
Stall Long Scoreboard	36.43	67.57	66.53	67.36	78.94
SM Busy [%]	79.37	39.04	35.41	35.75	37.21

Table 4.10.: Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using one work-item for four grid points

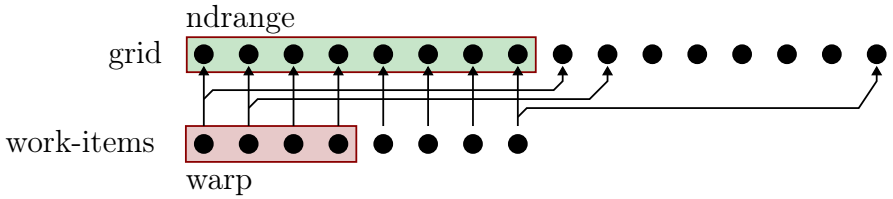


Figure 4.4.: Mapping of work-items to grid points for having one work-item for two grid points. In this example, the `ndrange` consists of 8 grid points, the entire grid has 16 grid points and one warp consists of 4 work-items.

If we want to increase the TLP by mapping multiple work-items to one grid point, we divide the work of applying a 27-point stencil onto these work-items. If for example we use four work-items per grid point, the `ndrange` must consist of $mgh \times ngh \times 4ogh$ work-items. We can use a conditional statement to check, which work-items shall apply which part of the stencil. This can be done in two ways. First, we can use `get_local_id(0) % 4 == {0,1,2,3}`, which would map four neighboring work-items to one grid point, each applying a different part of the stencil to the same grid point. The partial result is then stored in local memory using `partials[get_local_id(0)] = stencilsum`. The first of the four work-items then sums up the different partial sums from local memory and writes the result back to global memory. Since the work-items taking different branches are adjacent to each other, this approach suffers from branch divergence inside one warp. Furthermore, both halves of the warp store the values in the same local memory banks, resulting in 2-way bank conflicts.

The other solution is to spread out the work-items calculating the partial stencil for the same grid point, much like we do it by using a grid-stride loop. We can calculate the amount of grid points per work-group by dividing the work-group size by the amount of work-items per grid point and then check in which section of the work-group the work-item is located. Implementation sketches of the two versions are shown in listing 4.8.

If we choose a work-group size large enough, we can ensure that no branch divergence occurs inside a warp. For the case of four work-items per grid-point, we would need a work-group size of $32 \cdot 4 = 128$ work-items.

Table 4.11 shows the results on the Quadro T2000 GPU for four consecutive work-items per grid point. We can see a branch efficiency of 50 % for all tested work-groups and the presence of bank conflicts. Table 4.12 shows the results for the version, where the work-items mapped to the same grid point are spread out evenly across the work-group. For a work-group size of at least `128x1x1`, branch efficiency is at 100 % and no bank conflicts occur. Table 4.13 shows the same version, but using only two work-items per grid point. There are no bank conflicts here either, but a smaller work-group size (`64x1x1`) is sufficient to eliminate branch divergence. The durations of two and four work-items per grid point for the spread out variants are quite similar.

This attempt helped to reduce the kernel's duration slightly (3.73 ms vs. 3.75 ms). However, generally, this difference is not significant enough to accept the loss of flexibility and portability that the usage of local memory causes, but it should be kept in mind and reevaluated on future architectures.

Using the A100 GPU, unfortunately we do not achieve meaningful results when using NCU with PoCL. But we can run the benchmark without NCU and record the kernel durations using the OpenCL profiling API. The results are shown

```
__kernel void residual_27point_varying_stencil_1d_mult_wi_per_cell_4_sv_spread(/* arguments */) {
    int idx = get_global_id(0);
    int idx_gp = idx / wiPerGridPoint; // 1d grid point index
    int no = n * o;
    int i = idx_gp / no;
    int j = (idx_gp - i * no) / o;
    int k = idx_gp % o;

    double stencilsum = 0.0;
    if (get_local_id(0) % 4 == 0) {
        stencilsum = ... /* stencil entries 0-7 */
    } else if (get_local_id(0) % 4 == 1) {
        stencilsum = ... /* stencil entries 8-14 */
        partials[get_local_id(0)] = stencilsum;
    } else if (get_local_id(0) % 4 == 2) {
        stencilsum = ... /* stencil entries 15-21 */
        partials[get_local_id(0)] = stencilsum;
    } else {
        stencilsum = ... /* stencil entries 22-26 */
        partials[get_local_id(0)] = stencilsum;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
    if (get_local_id(0) % 4 == 0)
    {
        stencilsum += partials[get_local_id(0) + 1];
        stencilsum += partials[get_local_id(0) + 2];
        stencilsum += partials[get_local_id(0) + 3];
        r[index] = f[index] - stencilsum;
    }
}

__kernel void residual_27point_varying_stencil_1d_mult_wi_per_cell_4_sv_spread_shmem_spread(/* arguments */) {
    int idx = get_global_id(0);
    int gridPointsPerBlock = get_local_size(0) / wiPerGridPoint;
    int idx_gp = (idx % gridPointsPerBlock) + get_group_id(0) * gridPointsPerBlock; // 1d grid point index
    int no = n * o;
    int i = idx_gp / no;
    int j = (idx_gp - i * no) / o;
    int k = idx_gp % o;

    double stencilsum = 0.0;
    if (get_local_id(0) < gridPointsPerBlock) {
        stencilsum = ... /* stencil entries 0-7 */
    } else if (get_local_id(0) < gridPointsPerBlock * 2) {
        stencilsum = ... /* stencil entries 8-14 */
        partials[get_local_id(0)] = stencilsum;
    } else if (get_local_id(0) < gridPointsPerBlock * 3) {
        stencilsum = ... /* stencil entries 15-21 */
        partials[get_local_id(0)] = stencilsum;
    } else {
        stencilsum = ... /* stencil entries 22-26 */
        partials[get_local_id(0)] = stencilsum;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
    if (get_local_id(0) < gridPointsPerBlock)
    {
        stencilsum += partials[get_local_id(0) + gridPointsPerBlock];
        stencilsum += partials[get_local_id(0) + gridPointsPerBlock * 2];
        stencilsum += partials[get_local_id(0) + gridPointsPerBlock * 3];
        r[index] = f[index] - stencilsum;
    }
}
```

Listing 4.8: Residual kernel versions having four work-items per grid point. In the first version, the work-items of the group are direct neighbors. In the second version, the work-items are spread out evenly across the work-group.

Work-group size	32x1x1	64x1x1	128x1x1	256x1x1	512x1x1
Time [ms]	7.33	7.3	7.31	7.33	7.36
CT [%]	91.6	91.96	91.88	91.66	91.16
MT [%]	46.64	46.85	46.79	46.65	46.4
L1 requests	14924865	14924866	14924868	14924872	14924880
L1 sectors	35783705	35563922	35578006	35679472	35730980
L1 sectors / requests	2.40	2.38	2.38	2.39	2.39
L1 hit rate [%]	33.23	40.2	40.64	41.99	43.82
L2 hit rate [%]	35.02	28.37	27.83	26.03	23.62
Stall LG Throttle	0.05	0.13	0.13	0.57	1.21
Stall Long Scoreboard	20.43	33.85	33.95	33.69	30.21
SM Busy [%]	91.62	91.98	91.9	91.67	91.2
Theoretical Occ. [%]	50	100	100	100	100
Achieved Occ. [%]	49.08	93.75	98.27	98.04	97.28
Branch efficiency [%]	50	50	50	50	50
Load bank conflicts	798720	798720	798720	798720	798720

Table 4.11.: Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using four consecutive work-items for one grid point

Work-group size	32x1x1	64x1x1	128x1x1	256x1x1	512x1x1
Time [ms]	7.35	3.75	3.73	3.73	3.75
CT [%]	91.31	89.62	45.11	45.09	44.97
MT [%]	46.49	91.26	91.58	91.65	91.34
L1 requests	14924865	7603266	3942468	3942472	3942480
L1 sectors	35810430	32249816	30707848	30207367	29869281
L1 sectors / requests	2.40	4.24	7.79	7.66	7.58
L1 hit rate [%]	36.56	30.23	35.81	36.35	36.93
L2 hit rate [%]	31.65	30.11	19.95	19.85	19.67
Stall LG Throttle	0.04	0.07	1.59	1	1.66
Stall Long Scoreboard	17.89	22.14	25.45	22.3	19.8
SM Busy [%]	91.35	89.69	45.12	45.16	44.99
Theoretical Occ. [%]	50	100	100	100	100
Achieved Occ. [%]	49.11	87.99	86.79	76.19	76.92
Branch efficiency [%]	50	75	100	100	100
Load bank conflicts	0	0	0	0	0

Table 4.12.: Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using four spread out work-items for one grid point

Work-group size	32x1x1	64x1x1	128x1x1	256x1x1	512x1x1
Time [ms]	3.78	3.73	3.74	3.74	3.73
CT [%]	83.24	42.14	42.17	42.11	42.14
MT [%]	90.5	91.63	91.7	91.65	91.67
L1 requests	7465953	3805154	3805156	3805160	3805168
L1 sectors	32046707	30650006	30093525	29824892	29814947
L1 sectors / requests	4.29	8.05	7.91	7.84	7.84
L1 hit rate [%]	37.15	35.4	36.35	36.82	41.88
L2 hit rate [%]	23.03	20.07	19.44	19.22	12.86
Stall LG Throttle	0.19	5.53	6	8.48	12.73
Stall Long Scoreboard	20.97	39.02	40.21	37.38	31.74
SM Busy [%]	83.27	42.17	42.16	42.18	42.19
Theoretical Occ. [%]	50	100	100	100	100
Achieved Occ. [%]	49.19	84.14	87.84	86.64	86.12
Branch efficiency [%]	66.67	100	100	100	100
Load bank conflicts	0	0	0	0	0

Table 4.13.: Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using two spread out work-items for one grid point

Work-group size	4 consecutive work-items time [ms]	4 spread work-items time [ms]	2 spread work-items time [ms]
32x1x1	0.66	0.63	0.51
64x1x1	0.53	0.42	0.41
128x1x1	0.52	0.41	0.40
256x1x1	0.51	0.40	0.40
512x1x1	0.51	0.40	0.40

Table 4.14.: Durations for the varying stencil residual kernel on a A100 GPU for a real grid size of 128^3 and $gh = 1$ using multiple work-items for one grid point

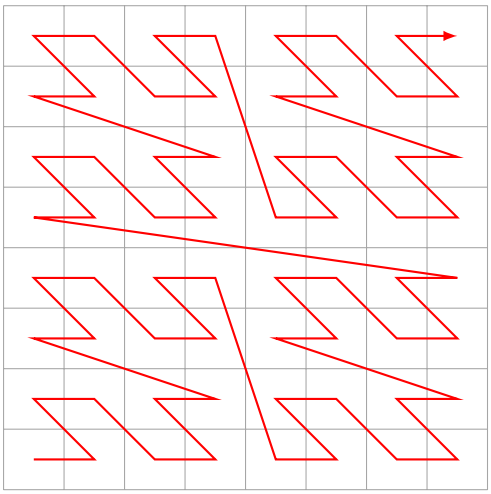
in table 4.14. No version matches the performance of the kernel mapping one work-item to one grid point, although the differences are small.

4.2.4. Exploiting Data Locality

Since the same approximation data is reused in multiple stencil applications, it is intuitive to exploit the high amount of data locality that is present in stencil computations. There are classical approaches of spatial tiling involving the usage of local memory to share data among nearby grid points, i.e. work-items, if one work-item is mapped to one grid point. In the case of varying coefficients, however, the coefficients only get used once, as they can differ for each grid point. This makes the coefficients not a viable candidate for storing in local memory for reuse purposes. Furthermore, since the size of the coefficient data clearly outweighs the

42	43	46	47	58	59	62	63
40	41	44	45	56	57	60	61
34	35	38	39	50	51	54	55
32	33	36	37	48	49	52	53
10	11	14	15	26	27	30	31
8	9	12	13	24	25	28	29
2	3	6	7	18	19	22	23
0	1	4	5	16	17	20	21

(a) Morton index in each cell



(b) Morton Z-order traversal

Figure 4.5.: Visualization of a Morton (Z-order) index grid and traversal

amount of data for the current approximation, implementing this optimization only for the approximation has little to no benefit. Depending on the implementation, it might even hurt performance. This is also due to the improved automatic caching on current GPU architectures. Still, the tiling techniques are investigated closely in section 4.2.7.

Thus, instead of focusing on manual caching using local memory here, we might try to improve data locality to help the automatic cache. This could be done using a space filling curve like the z-curve, i.e. Morton codes [71]. Instead of storing the approximation in a linear fashion, i.e. row-by-row, using Morton codes, nearby grid points would be stored contiguously in memory, see figure 4.5. This is achieved by interleaving the bits of the 3d index parts of a grid point, i.e. using so-called dilated integers. For example, the grid point with 3d index (2,11,10) has the following binary representation.

$$\begin{aligned} i = 2_{10} &= 0010 \\ j = 11_{10} &= 1011 \\ k = 10_{10} &= 1010 \\ \texttt{morton}(i,j,k) &= 011000111010 \end{aligned}$$

Since three integers are combined into one number, the data type for storing the Morton code must be capable of storing at least three times the number of bits of the 3d index. This directly limits the grid sizes our solver can handle.

The conversion of a 3d index to a Morton code can be implemented by using a simple for-loop containing bit-shift operators. While this approach is easy to

implement and understand, it severely lacks performance. There is a more efficient way involving bitwise operations with magic numbers, using lookup tables or using intrinsics [4]. Analogously, increments and decrements of individual grid points coordinates can be performed directly on the Morton code using magic numbers, without the need of converting back and forth. This is also called tesseral arithmetic in the context of quadtrees.

In scientific computing, Morton codes are usually used in the context of domain decomposition and octrees. However, there is research on how Morton codes perform compared to row-major and column-major storage layouts applied to different algorithms, including 2d Jacobi's method [109]. The tests were performed sequentially on a single CPU. The results for Jacobi's method show, that the row-major format is superior in all cases.

A highly optimized version using Morton codes requires a substantial effort, which, however, is not our focus here. Our intermediate prototype implementations performed severely worse than the basic variant from section 4.2.1.2. Nevertheless, we want to briefly point out potential GPU implementation challenges.

First, we cannot use the Morton codes for indexing directly. If we would do it for storing v, the values might be scattered across the entire 32-bit range (when using 32-bit integers for indexing), i.e. it is not guaranteed that all values are located in the range of $[0, ..., mgh \times ngh \times ogh]$, like it is the case for the usual row-major format. This especially occurs for grids that do not have the same lengths in all dimensions. For example for a grid of size `2x2x4`, $\texttt{morton}(1, 1, 1) = 7_{10}$ and $\texttt{morton}(0, 0, 2) = 32_{10}$. Since we would also need to store the coefficients in Morton order to ensure an efficient global memory access pattern, we could no longer add the grid size to the grid point index to retrieve the correct index. A remedy to this issue is to sort the computed Morton codes and assign individual ranks to each of them. Then, the indexing would be done using the ranks instead of the Morton codes directly, ensuring all values of v for one grid are stored in a consecutive chunk in memory.

Second, a balance between precomputing indices, offsets and ranks and memory consumption must be found. For example for a grid of size 130^3, we would need to store $2 \cdot 130^3$ Morton ranks, i.e. around 11 MB, in a lookup table for v and the coefficients. This size is clearly too large for private and local storage and therefore would have to reside in global (constant) memory. Since our residual kernel is already memory bound, exerting more pressure on global memory is likely to degrade performance.

Lastly, we would sacrifice flexibility and scalability due to the limitations of the indexing. However, there are ways to alleviate this issue, e.g. by disassembling the grid into sub-grids, called microcells in [88].

4.2.5. Jacobi Split Versus Fused

Now that we have the residual calculation covered, we want to address the implementation of Jacobi's method. We use the residual kernel version, that maps one work-item to one grid-point.

A draft of a sequential version of Jacobi's method can be found in listing 4.9. For a fixed number of iterations ν, we update the ghost points (if using periodic boundary conditions), calculate the residual and use it to update the approximation.

```
void jacobi(/* ... */) {
    for (int iter = 0; iter < maxiter; iter++) {
        updateGhosts(v);
        residual(f, v, r);

        for (int i = gh; i < m + gh; i++)
            for (int j = gh; j < n + gh; j++)
                for (int k = gh; k < o + gh; k++) {
                    v[i][j][k] = v[i][j][k] + omega * (1.0 / coeffs[1][1][1][i][j][k]) * r[i][j][k];
                }
    }
}
```

Listing 4.9: Draft of a sequential Jacobi's method implementation.

One option is to add a kernel, that only performs the Jacobi update step after the residual has been calculated. A version that gets called with a 1d `ndrange` is shown in listing 4.10. With this approach, synchronization of global memory is ensured between the residual and Jacobi update kernel calls. While this attempt is easy to implement, there is a way to bypass the memory synchronization and reduce the total kernel launch overhead. That is, we can fuse the residual and Jacobi update kernels into a single kernel, effectively cutting the number of required kernel calls in half. Since the residual calculation depends on the current approximation, we cannot just paste the Jacobi update code below the residual calculation and store the update in the array `v`. An alleviation to this issue is to use a second, mirrored array `v_out`. We then use `v = v_in` solely as our input array and `v_out` as the output array. In the next iteration, we swap the roles of the arrays, i.e. using `v_out` as input and `v_in` as output. We can do this by resetting the two corresponding OpenCL kernel arguments. Additionally, we need to update the ghosts of the correct input array before the next iteration. Furthermore, there is no need to store the residual in the array `r` after every iteration, as it is computed on the fly, which saves us one global load and one global store for each but the last iteration. This kernel version is shown in listing 4.11.

Table 4.15 and figure 4.6 show the durations for the two versions. The difference in time increases with the iteration count, the fused version having a consistent edge over the split version. Thus, the fused version is what is implemented in `mgcl`.

```
__kernel void jacobi_iter_27point_varying_stencil_1d_update_step_only(
    __global double* restrict v, __global double* restrict f,
    __global double* restrict r, __global double* restrict stencilValues,
    const double omega, const int mgh, const int ngh, const int ogh,
    const int svmgh, const int svngh, const int svogh,
    const int ghosts, const int ghosts_sv, const int svGridSize, const int idx_start)
{
    int idx = get_global_id(0);
    int no = ngh * ogh;
    int i = idx / no;
    int j = (idx - i * no) / ogh;
    int k = idx % ogh;

    if (i >= idx_start && j >= idx_start && k >= idx_start &&
        i < mgh - idx_start && j < ngh - idx_start && k < ogh - idx_start)
    {
        int index = i * ngh * ogh + j * ogh + k;
        int index_sv = (i - ghosts + ghosts_sv) * svngh * svogh +
                       (j - ghosts + ghosts_sv) * svogh + (k - ghosts + ghosts_sv);
        double sv_self = stencilValues[index_sv + (9 + 3 + 1) * svGridSize];
        v[index] = v[index] + omega * (1.0 / sv_self) * r[index];
    }
}
```

Listing 4.10: OpenCL kernel applying the Jacobi update step.

```
__kernel void jacobi_iter_27point_varying_stencil_1d(
    __global double* restrict v_in, __global double* restrict v_out,
    __global double* restrict f, __global double* restrict r,
    __global double* restrict stencilValues,
    const double omega, const int mgh, const int ngh, const int ogh,
    const int svmgh, const int svngh, const int svogh,
    const int ghosts, const int ghosts_sv, const int svGridSize,
    const int idx_start, const int store_residual)
{
    int idx = get_global_id(0);
    int no = ngh * ogh;
    int i = idx / no;
    int j = (idx - i * no) / ogh;
    int k = idx % ogh;

    if (i >= idx_start && j >= idx_start && k >= idx_start && i < mgh - idx_start && j < ngh - idx_start && k
        < ogh - idx_start)
    {
        int ioff = ngh * ogh;
        int joff = ogh;
        int koff = 1;
        int index = i * ioff + j * ogh + k;

        int svno = svngh * svogh;
        int index_sv = (i - ghosts + ghosts_sv) * svno + (j - ghosts + ghosts_sv) * svogh + (k - ghosts +
            ghosts_sv);

        double res;
        double v_in_index = v_in[index];
        double sv_self = stencilValues[index_sv + (9 + 3 + 1) * svGridSize];

        double stencilsum = sv_self * v_in[index]
            + stencilValues[index_sv + (9 + 3) * svGridSize]      * v_in[index - 1]
            /* ... */
            + stencilValues[index_sv + (18 + 6 + 2) * svGridSize] * v_in[index + ioff + joff + koff];
        res = f[index] - stencilsum;
        v_out[index] = v_in_index + omega * (1.0 / sv_self) * res;

        if (store_residual)
            r[index] = res;
    }
}
```

Listing 4.11: Fused OpenCL kernel for calculating the residual and applying the Jacobi update step. Some lines that are part of the residual calculation are omitted for brevity.

	Time [ms]			
	Quadro T2000		A100	
ν	Fused	Split	Fused	Split
1	3.93	4.39	0.46	0.53
2	7.81	8.73	0.88	1.03
3	11.68	13.05	1.31	1.53
4	15.54	17.37	1.74	2.02
5	19.4	21.71	2.16	2.53
6	23.28	26.03	2.51	2.87
7	27.13	30.34	2.93	3.34
8	31.01	34.65	3.35	3.81
9	34.87	38.99	3.76	4.29
10	38.73	43.31	4.17	4.76

Table 4.15.: Durations for Jacobi's method having residual and update step kernels split vs. fused. Executed on Quadro T2000 and A100 on a grid of size 128^3 and $gh = 1$.

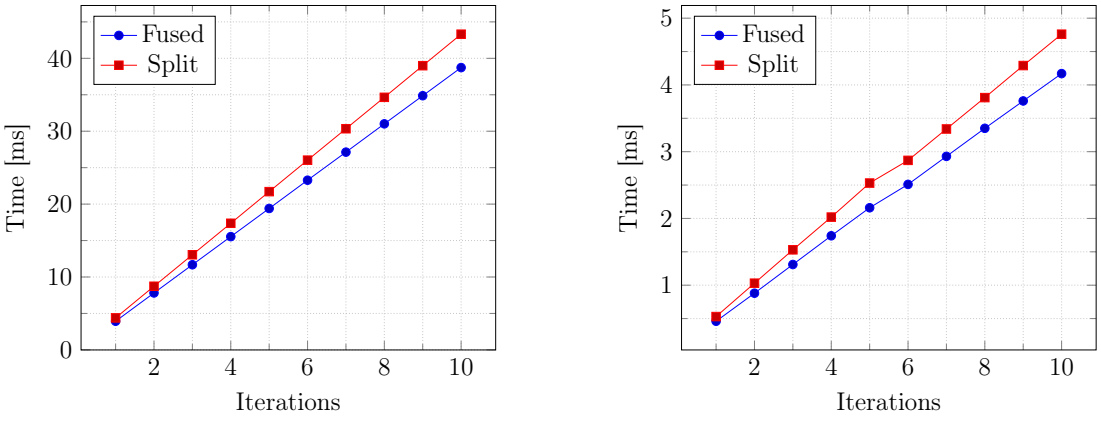


Figure 4.6.: Durations for Jacobi's method having residual and update step kernels split vs. fused. Executed on Quadro T2000 (left) and A100 (right) on a real grid of size 128^3 and $gh = 1$.

4.2.6. Multiple Jacobi Iterations Without Intermediate Ghost Update

For periodic boundary conditions, we need to update the ghosts of the input in each iteration. In the single-GPU case, the ghost update only needs a fraction of the time that the Jacobi kernel does. Since this changes in the multi-GPU case, we still aim to evaluate a method to reduce the number of ghost updates by adding redundant calculations at the border of the grid. We denote the number of iterations without ghost updates in between as spi (`stepsPerIter`). For this to work, we need to enlarge the grid by more ghost layers, such that the number of ghost layers is equal to the iterations that shall be computed in one kernel call. More precisely, $gh_v = \text{spi}$ and $gh_r = gh_f = \text{spi} - 1$. The ghost update routine must be updated accordingly, which is described in section 4.4. An additional inner loop is added to the host code as shown in the listing 4.12. The amount of boundary layers for which values need to be calculated decreases with increasing inner iteration count k_{inner}, i.e. $i_{\text{start}} = gh - (\min(\text{spi} - k_{\text{inner}}, \nu - k_{\text{global}}) - 1)$.

```
int globalIter = 0;
while (globalIter < maxiter) {
  // update ghost of v_in if globalIter % 2 == 0, else update ghosts of v_out
  for (int innerIter = 0; innerIter < stepsPerIter && globalIter < maxiter; innerIter++, globalIter++) {
    // set kernel arg for v_in and v_out
    // calculate idx_start = ghosts - (min(stepsPerIter - innerIter, maxiter - globalIter) - 1);
    // enqueue kernel
  }
}
```

Listing 4.12: Added inner loop for multiple Jacobi iterations without intermediate ghost update

In table 4.16 we see, that on the Quadro T2000, this approach does not result in a lower execution time. For the A100, executing two iterations without intermediate ghost update seems to be slightly more efficient than the alternatives, however the difference is negligible.

Table 4.17 compares the durations of the individual kernel calls for 5 Jacobi iterations. The sum of the individual kernel durations lies slightly below the total durations measured on the host from start to end of Jacobi's method. For example, for spi = 1, 5 Jacobi kernel calls and 5 ghost update kernel calls are executed. For spi = 2, 5 Jacobi kernel calls and only 3 ghost update kernel calls are executed. This makes $5 \cdot 387\,\mu\text{s} + 5 \cdot 25\,\mu\text{s} = 2060\,\mu\text{s}$ for spi = 1 and $5 \cdot 396\,\mu\text{s} + 3 \cdot 25\,\mu\text{s} = 2055\,\mu\text{s}$ for spi = 2 using an A100 GPU. The slight differences to the measured total durations might be the result of the overhead of creating the kernel objects, setting kernel arguments and enqueuing the kernels.

Since the benefit is negligible, we continue to use the regular version with ghost updates between each iteration for the single-GPU case. The multi-GPU case is analyzed in chapter 5.4.

		Time [ms]	
ν	spi	Quadro T2000	A100
1	1	3.94	0.45
2	1	7.81	0.88
	2	7.96	0.87
3	1	11.67	1.31
	2	11.87	1.30
	3	12.29	1.34
4	1	15.52	1.73
	2	15.86	1.71
	3	16.29	1.78
	4	16.79	1.80
5	1	19.41	2.16
	2	19.74	2.14
	3	20.32	2.20
	4	20.86	2.23
	5	21.54	2.32

Table 4.16.: Durations for Jacobi's method for various iteration counts without intermediate ghost update for a grid of size 128^3 and $gh = 1$. spi: Iteration count without intermediate ghost update.

	Time [µs]			
	Quadro T2000		A100	
spi	Jacobi	Ghost Update	Jacobi	Ghost Update
1	3,758	66	387	25
2	3,834	70	396	25
3	3,906	88	407	27
4	3,942	96	407	27
5	3,971	112	417	28

Table 4.17.: Kernel durations for Jacobi's method for 5 iterations with varying amount of ghost updates for a grid of size 128^3 and $gh = 1$. spi: Iteration count without intermediate ghost update.

4.2.7. Blocking Techniques

Instead of enlarging the grid in global memory, we can use local memory to share data that is used by multiple neighboring grid points and to store inter-iteration results. The techniques commonly used for this are called *spatial blocking* (or spatial tiling) and *temporal blocking* (or temporal tiling). While these techniques are well examined in the literature, the majority focuses on boundary conditions with fixed values, e.g. Dirichlet boundary conditions, and/or fixed coefficients. Although the memory transfer size in the case of varying coefficients mainly comes from the coefficients itself instead of the actual grid point values, we might still benefit from fusing multiple iterations in one kernel launch.

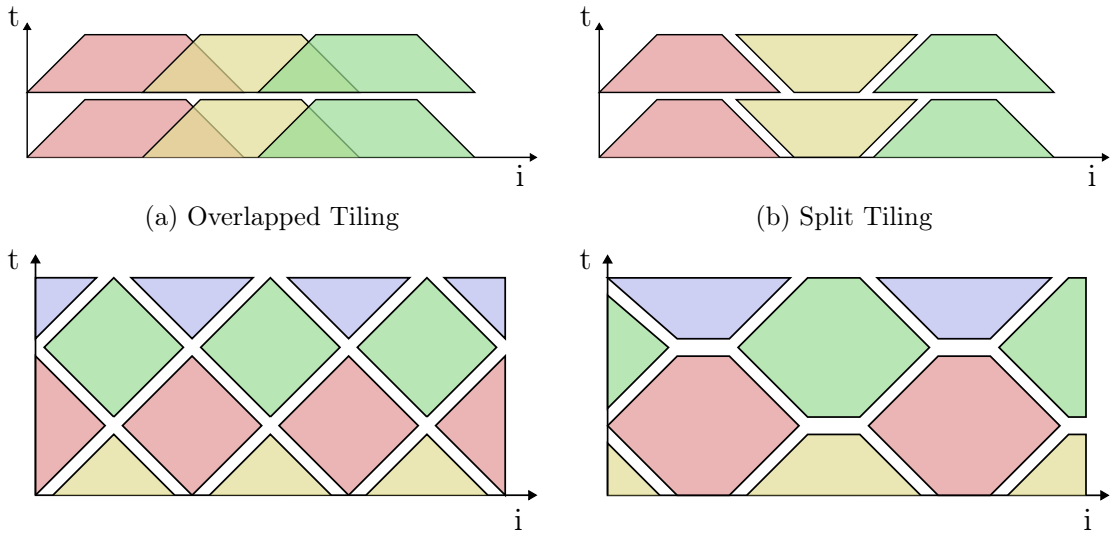
4.2.7.1. Background

Before examining how well spatial and temporal blocking can be applied to our use case involving varying coefficients and periodic boundary conditions, we first want to illuminate the background of these techniques as presented in the literature.

The idea is based on common loop tiling techniques, that were present already before GPUs were used for general purpose computations [120, 119, 2, 60]. Nested iterative loops define an *iteration space*, which is a finite and discrete Cartesian space whose dimensionality corresponds to the depth of the loop nest. Simple loops, where the computations and loop boundaries do not share dependencies, can easily be divided in smaller, rectangular blocks. If there are dependencies involved, e.g. the time dimensional dependence occurring in many stencil computations, more sophisticated tiling strategies, resulting in other shapes in the iteration space, must be investigated.

There are different approaches to realize spatial blocking. With spatial blocking, we are only looking at the stencil computation at a fixed time step. The goal is to achieve higher data reuse using local memory instead of global memory. The naive approach is to load an entire 3d tile plus ghost points into local memory. Since the local memory is restricted in size, the tiles would have to be so small, that this is not a practical approach. Instead, a common solution is to use 2d tiles, i.e. loading 2d planes into local memory, and streaming along the third dimension [70, 16].

Most publications focus on temporal blocking instead of just spatial blocking. It can be seen as an extension to spatial blocking, where the results of previous iterations are kept in local memory to be reused in the next iteration. Depending on the method and how intra-tile data is kept in local memory, different shapes occur in the iteration space, see figure 4.7.



(a) Overlapped Tiling (b) Split Tiling

(c) Diamond Tiling (d) Hexagon Tiling

Figure 4.7.: Illustration of different tile shapes

Overlapped tiling essentially resembles the approach we used in the previous section. Each tile has ν layer of ghost points, where ν is the number of iterations. For each iteration, redundant calculations are performed inside the halo zone. A GPU implementation of this approach is presented by Holewinski et al. in [41]. Nguyen et al. [74] combine it with streaming along the third dimension and call it 3.5D blocking.

Split tiling eliminates the necessity of redundant calculations by dividing each tile into two kinds of sub-tiles – one being dependent on the grid points of neighboring tiles and the other one having no dependencies on other tiles, thus being executable concurrently. Algorithms then consist of two phases: First, the independent sub-tiles are computed concurrently and exchange their results with neighboring tiles. Second, the remaining sub-tiles are executed concurrently, using the results of the independent sub-tiles. A GPU implementation can be found for example in [32].

Diamond tiling [9] and hexagonal tiling [31] extend on the idea of split tiling and tackle its start-up phase and load imbalance issues.
A further attempt is to use a dynamic hybrid tiling strategy, which adopts static tiling for some dimensions and provides dynamic tile shapes that can change with time steps for other dimensions [62]. Another strategy called *Semi-stencil* partially applies stencils in multiple sweeps [17, 94].

Since attaining optimal performance using these techniques is strongly dependent on the limits of the present architecture, there is a variety of stencil code generators, e.g. STENCILGEN [91], AN5D [65], DRStencil [123], Artemis [92]. All of these generators aim to saturate the device by achieving a high occupancy, which limits

the amount of local memory and register count per compute unit. EBISU [125] instead tries to find the lowest amount of occupancy that still exhibits a sufficient level of ILP. This is done by increasing the size of the individual tiles, until a single tile can saturate the entire device, called *device tiling*. Each device tile is further divided into smaller tiles saturating a single compute unit, called *SM tiling*. The different device tiles are computed sequentially. This concept is realized using CUDA's grid level barriers, which allows synchronization of data across multiple work-groups. Unfortunately, this is not possible with Nvidia's OpenCL backend implementation.

Shan and Araya-Polo [100] compare different strategies at the example of an acoustic isotropic approximation of the wave equation. They use a fixed-coefficients 25-point 3d stencil (8-th order in space) and dissect the domain into two parts, whereas the paper concentrates primarily on the computational interior region. A main focus is placed on full 3d blocking using global memory only, full 3d blocking using local memory, 2.5d streaming using local memory and the Semi-stencil approach. A regular version, that synchronizes inter-work-group data using global memory and separate kernel calls, is compared to a version using CUDA's thread block clusters to communicate data between work-groups without the need of global synchronization. Overall, no benefit in using thread block clusters could be seen.

The following publications concentrate on stencil computations using varying coefficients on GPUs. However, such literature is quite rare.

Fang et al. [24] investigate the implementation of the Explicit Time Evolution (ETE) method on GPUs, involving an 8-th order 3d stencil with spatially varying coefficients. A 3.5d spatial and temporal blocking technique is implemented in tandem with stencil decomposition. The coefficients are stored in a 2d matrix and indexed using a 3d coefficients index array. Using a Tesla K20 GPU (CC 3.5), a ca. 5x speedup is observed using the optimizations compared to a naive implementation using global memory directly. Note that this GPU architecture does not include automatic L1 caching when local memory is not used. In a another publication, Fang et al. [25] further investigate on the same topic, introducing a differentiation between the *multi-to-one* and *one-to-multi* schemes. That is, instead of applying a whole stencil to the current grid point, needing all coefficients to be resident in local memory or registers at the same time, only a single xy-plane is held in local memory and the stencil result is accumulated step-by-step in registers. Using a Titan X GPU (CC 6.1), 3.5x - 6.5x speedups of the optimized version compared to naive version for different stencil sizes are observed.

Einkemmer and Ostermann [20] implement stencil code in the context of exponential integrators with position-dependent diffusion coefficients. Two versions are presented: A naive version (using global memory only) and an optimized version

involving spatial blocking using local memory. The optimized version performs slightly worse on the two devices under test for a grid of size 256^3. Boundary conditions are either neglected or implemented using branching inside the kernel. The coefficients are computed on-the-fly, which makes the kernel compute bound. Hence, this is not a suitable comparison for us.

4.2.7.2. Applicability to Varying Coefficients and Periodic Boundary Conditions

In this section, we want to discuss the applicability of spatial and temporal blocking to problems with varying coefficients and periodic boundary conditions. We provide a prototype implementation based on the 3.5D blocking technique for v. There are multiple considerations to make.

- *Do we choose a 1d, 2d or 3d kernel layout?* While a 1d kernel layout allows for a very low portion of inactive work-items, it is challenging to map the work-items to the grid points that belong to the same work-group. Plus, there is less locality than in the higher dimensional cases.

  A 3d kernel layout suffers from high local memory storage requirements in tandem with low work-group sizes per dimension, which has a negative impact on global memory access.

  A 2d kernel layout plus streaming along the outermost dimension allows for a good balance between workload per work-item and local memory requirements.

- *How many work-items should a work-group consist of?* The bigger a work-group the less redundant memory accesses and computations are necessary. However, with growing work-group size, local memory requirements are growing, too. It strongly depends on the hardware used.

- *How do we respect periodic boundary conditions?* This is one of the more distinct differences opposed to the literature, as often times Dirichlet-like boundary conditions are used, which do not require an update of ghost points between iterations.

  The simplest approach is to have multiple layers of ghosts in global memory and start one work-item per grid point. This is what is presented in section 4.2.6.

  A more sophisticated approach involves using local memory. The amount of ghost layers in local memory must match the iteration count that shall be performed within a single kernel launch. Real grid points in local memory can be populated easily. Ghost points must be handled separately. We

can assign specific work-items of the work-group to fill certain ghost cells. Essentially, we perform an ad-hoc ghost update in local memory, which removes the necessity of updating ghosts before the kernel launch.

- *Do we map work-items to real grid points only, or to ghost points, too?* The advantage of the latter is that each work-item applies one stencil, i.e. an almost uniform workload distribution is achieved. On the other hand, if we map one work-item to each real grid point only, we can perfectly divide a 2^n-sized grid into work-groups of sizes that are a power of two, thus not having redundant work-items at the boundaries.

  In our implementation, we chose to map one work-item to each real grid point of a yz-plane. The x dimension is the slowest varying one in our case. There might be multiple work-items in x-direction, too, i.e. one work-item might operate on only a subset of all yz-planes. For our prototype, we restrict ourselves to perform only two iterations in one kernel launch, such that there is no need to change the amount of ghost layers of our coefficients array, as it is at one layer per default. Therefore, the two outer layers of a work-group load multiple grid points (self and ghosts) and the outermost layer of the work-group applies the stencil multiple times (self and for ghost layer).

Orienting to the 3.5D blocking scheme presented by Nguyen et al., our algorithm consists of a prologue and a rolling phase. In the prologue, we populate the local memory with the last 2 and first 3 real yz-planes of a fixed slice of yz-planes. Stencils in the first four yz-planes are applied, such that the first final result can be written back to global memory. After that, the rolling phase begins, where repeatedly the next plane is loaded, a stencil at $t = 0$ is applied and the result is then used for the next plane at $t = 1$. We use a rolling buffer (i.e. circular queue) keeping five yz-planes in local memory, i.e. when the end of the queue is reached, the next plane is stored at the beginning. The streaming concept is illustrated in figure 4.8.

The prototype implementation is tested for multiple grid sizes using the Quadro T2000 and the A100 GPUs. The tile sizes, and how many yz-planes are streamed through, i.e. the degree of grid division in x-direction, are varied. For each grid size, the blocking version is slower than our previous implementation using global memory only. The lowest durations per combination of tile size and number of yz-planes per grid size are shown in table 4.18.

On the 8^3 grid, using the maximum possible tiles size of `8x8` and streaming along one yz-plane is fastest. The device is already underutilized due to the small grid size, i.e. it only records 0.12 waves in this case. Streaming over more planes only reduces device saturation. Using a `8x8` tile is faster than e.g. a `4x4` tile, since with a `4x4` tile size, only 16 out of 32 work-items of a warp are active, leaving half of

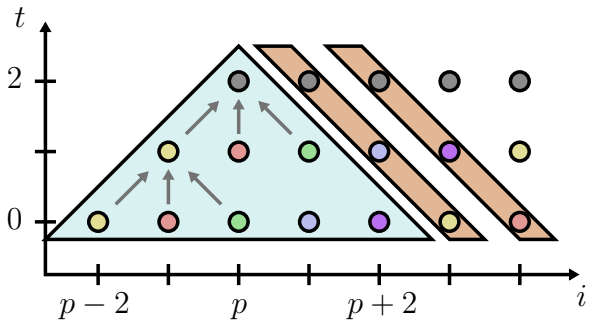


Figure 4.8.: Streaming along the outermost dimension. Each circle represents a yz-plane. Same colors represent same local memory buffers. The teal pyramid forms the prologue phase, the brown parallelograms the iterations in the rolling phase. Results at $t = 2$ are written to global memory.

Grid Size	#yz-planes	Tile Size	Time [ms]
	Quadro T2000		
8^3	1	`8x8`	0.09
16^3	2	`8x16`	0.14
32^3	8	`4x32`	0.31
64^3	64	`4x32`	1.59
128^3	128	`2x64`	12.95
	A100		
8^3	1	`8x8`	0.07
16^3	2	`8x16`	0.08
32^3	4	`2x32`	0.09
64^3	16	`2x32`	0.23
128^3	16	`2x64`	1.2

Table 4.18.: Durations for Jacobi's method using our prototype implementation using 3.5D blocking for 2 iterations

the work-items unused. Furthermore, in general, the larger the tile size, the fewer redundant computations are executed.

On the 32^3 grid, using a `8x16` tile and streaming over 8 yz-planes is the fastest variation. Theoretical occupancy is at 25 % on the Quadro T2000 for all amounts of yz-planes due to register usage. The achieved occupancy is noticeably higher for 8 yz-planes (24.9 %) than for 16 and 32 yz-planes (12.5 %). While when using 8 yz-planes, the kernel is stalled for global memory requests for more cycles (18 cycles) than when using 16 or 32 yz-planes (12 and 11 cycles), a higher degree of latency hiding can be achieved, as the overall device utilization is higher. This is also reflected by the number of waves, which is at 0.25, 0.5 and 1 waves per compute unit for 8, 16 and 32 yz-planes, respectively.

For the 64^3 grid, tile sizes `2x64` and `4x32` combined with streaming along all 64 yz-planes are fastest. When streaming along all yz-planes, there are only 2 redundant planes (one at start and one at end), i.e. in total 64+2 = 66 planes. When streaming along e.g. 8 chunks, there are $64[\text{real}] + 8 \cdot 2[\text{ghosts of each chunk}] = 64 + 16 = 80$ planes, for which the stencil needs to be applied. Loading happens for $64 + 4 = 68$ and $64 + 8 \cdot 4 = 96$ planes, respectively. So, for stencil application and loading there is an increase of $80/66 - 1 = 21.2\,\%$ and $96/68 - 1 = 41\,\%$, respectively. Compared to this, the time increase is at $2.59\,\text{ms}/1.91\,\text{ms} - 1 = 35.6\,\%$. Executed instructions are at 13,317,632 (8 yz-planes) and 10,232,256 (all yz-planes), a 30 % increase. Ultimately, the device is already saturated with streaming along all yz-planes (executing exactly one wave with high achieved occupancy). Hence, dividing the x-dimension in chunks only adds overhead.

Why is a tile size of `2x64` faster than e.g. `8x8` for the 64^3 grid? With `8x8`, there are $10 \cdot 10 = 100$ grid points that the stencil is applied on for each work-group, and there are $8 \cdot 8 = 64$ blocks, thus $64 \cdot 100 = 6400$ grid points with stencil application in total for one plane. With `2x64`, there are $4 \cdot 66 = 264$ grid points with stencil application per work-group and there are only 32 blocks, thus $32 \cdot 264 = 8448$ grid points with stencil applications in total for one plane. Why would `2x64` be faster than `8x8`, if there are more redundant stencil applications? The amount of instructions executed is at 13.7M vs. 10.2M for `8x8` and `2x64`, respectively. `8x8` has about 30 % more branch instructions than `2x64` (971k vs. 746k). Branching is much more severe in `8x8` case, as both, the routines to load a plane and to apply a stencil onto an entire plane, have branching instructions for handling ghost grid points. For plane loading, the two outermost work-items of the work-group branch off, while the inner ones do not. For the stencil application, it is one layer of work-items at the border. In table 4.19 we see, that for `8x8` a higher ratio of work-items of the work-group take different branches than in the `2x64` case. The most prevalent case is the ratio of work-items taking the branch only in y-direction when applying the stencil: For `2x64`, 97 % of the work-items in the work-group (and of the warp, due to work-group shape) take the same branch, while for `8x8`, it

		Load Plane				Apply Stencil			
	Tile	none	y-dir	z-dir	yz-dirs	none	y-dir	z-dir	yz-dirs
abs	`8x8`	16	16	16	16	36	12	12	4
abs	`2x64`	0	120	0	8	0	124	0	4
rel	`8x8`	0.25	0.25	0.25	0.25	0.5625	0.1875	0.1875	0.0625
rel	`2x64`	0	0.9375	0	0.0625	0	0.96875	0	0.03125

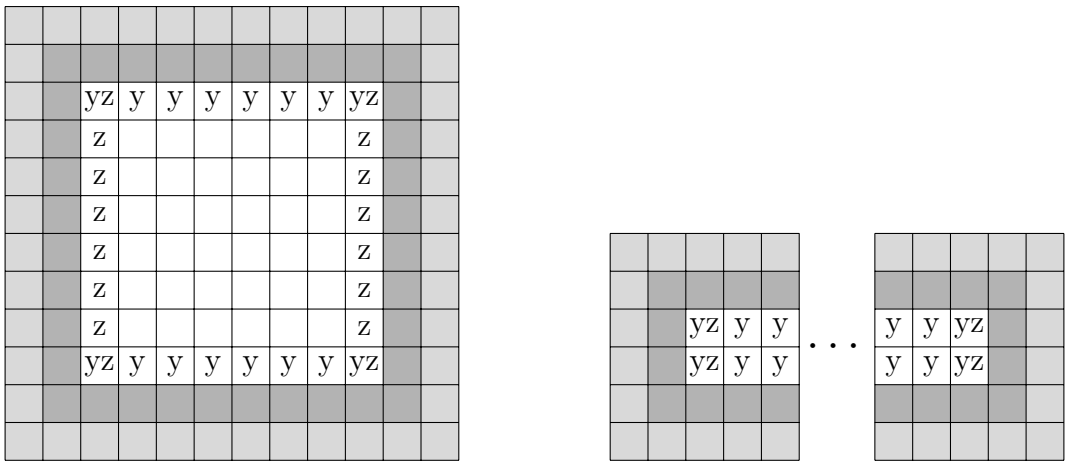
Table 4.19.: Amount of work-items choosing different branches in our prototype blocking implementation for a grid of size 64^3. abs is the absolute number of work-items, rel is the number of work-items relative to the total number of work-items in the tile. Branches are differed in none (no branch taken), y-dir (branch in y-direction taken), z-dir (branch in z-direction taken), yz-dirs (both branches in y- and z-directions taken).

is only 19 %. Looking at figure 4.9, we can furthermore see, that for `2x64`, almost all the work-items of a warp are active due to not having a diverged execution path, while for `8x8`, many work-items of the warp are idle while waiting for the few work-items taking branches. Lastly, access to global memory is not coalesced for `8x8`.

NCU metrics from runs using the Quadro T2000 for grid sizes 8^3, 32^3 and 64^3 can be seen in the appendix in tables A.12 to A.17.

In general, it must be noted, that this is only a prototype implementation, i.e. it is not well optimized. Surely, the performance can be increased e.g. by reducing register usage or by launching one work-item per grid point including the inner ghost layer, thus reducing the amount of wrap-around logic. However, its potential benefit stands in no comparison to the amount of work that would be needed to even get on par with the naive global memory version. Our prototype already consumes 286 lines of code, whereas the naive version without blocking needs only 78 lines of code. Furthermore, as already noted in the previous section, attaining high performance using blocking techniques requires intensive parameter tuning, which depends on the specific architecture. As this comes with highly complex setup needs and reduced flexibility and maintainability of the code base, we choose to not further pursue blocking implementations in `mgcl`.

Another promising technique, however, is the one-to-multi scheme presented by Fang et al., as it severely reduces local memory requirements. This might also enable storing the coefficients in local memory, which do not change between iterations. If the register pressure could be alleviated, too, the coefficients might even be stored in registers. Generally, blocking is more beneficial for a higher iteration count. However, as with the multigrid method, often a low iteration count (e.g. ≤ 3) is sufficient, the overall benefit of blocking might be negligible.



(a) `8x8` tile (b) `2x64` tile

Figure 4.9.: Branching in `8x8` and `2x64` tiles for our prototype blocking implementation. yz means the work-item takes branches in both, y- and z-directions. y and z mean, the work-item takes the branch in y- and z-direction, respectively. An empty cell means, the work-item takes no branch and thus is idle.

4.3. Jacobi's Method and Residual Calculation With Fixed Coefficients

Although our main focus lies on problems involving spatially varying coefficients, the step towards fixed coefficients is straightforward. Since a blocking technique would be out of the scope of this work, we discuss some other implementation concerns that do not rely on local memory. This also induces, that only the residual calculation itself changes, not the surrounding implementation of Jacobi's method. We therefore only look at the residual calculation here.

The naive version is similar to the one having varying coefficients, with the exception, that the 6d coefficients array is replaced by a `3x3x3` 3d coefficients array. The coefficient indices do not depend on the grid point index the work-item is mapped to, but only on the relative positions of the neighboring grid points.

We test the following kernel variants.

- `gmem`: Coefficients are stored in a `3x3x3` array in global memory.
- `const_arg`: Coefficients are passed as constant arguments to the kernel.
- `preproc`: Coefficients are set during kernel compile time as preprocessor constants.

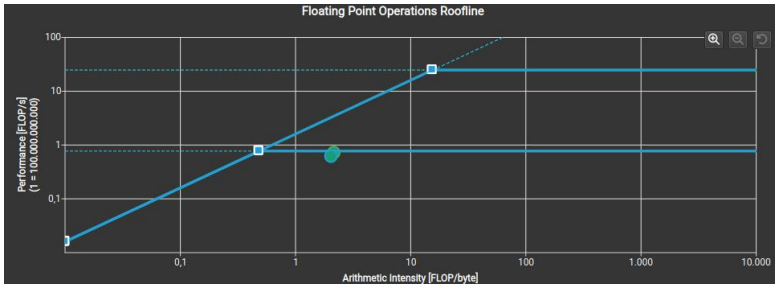


Figure 4.10.: Roofline from NCU for `gmem` and `preproc` variants of the residual kernel with fixed coefficients, executed on the Quadro T2000 GPU

- `const_mem`: Coefficients are stored in constant memory private to each work-item.

Using Nsight Compute, we obtain the roofline model for the Quadro T2000 as shown in figure 4.10. Clearly, the kernel is compute bound, as it hits the double precision ceiling.

Table 4.20 shows, that for a grid of size 128^3 on the Quadro T2000 the version using global memory is the fastest. This might be rather surprising, as additional global memory loads occur in this version. Table 4.21 reveals, that other variants inhibit a noticeably larger amount of tex throttle stalls. The reason this stall occurs without us invoking any texture or surface operation is, based on a discussion on the Nvidia developer forum[1], that for smaller GPUs often times the double-precision and tex units share the same issue and return paths. Furthermore, when looking at the SASS code produced for the different kernel variants, we see that the compiler groups most of the double-precision instructions (especially `DFMA`) together towards the end of the kernel, while these instructions are interleaved with global memory loads in the `gmem` version. Therefore, the FP64 pipe is used more evenly throughout the execution of the `gmem` kernel. The instructions are similarly ordered for all other variants. Since `const_arg` additionally needs to load data from constant memory compared to the remaining two versions, it is the slowest of all. Excerpts of the SASS codes are given in the appendix in figure A.1.

The roofline in figure 4.11 shows, that the kernels are memory bound when run on the A100. Compared to the Quadro T2000 GPU, this is expected, since the ratio of single-precision to double-precision performance is 1:2 on the A100, whereas it is at 1:32 on the Quadro T2000. As a consequence, the `gmem` variant is slower than the others. The remaining versions do not differ significantly in duration.

[1] `https://forums.developer.nvidia.com/t/better-kernel-performance-when-using-more-global-memory-reads-help-needed-with-analysis/316900/4`

Kernel Variant	Time [µs] Quadro T2000	A100
`gmem`	1620	84.96
`const_arg`	1850	57.92
`preproc`	1780	55.87
`const_mem`	1780	56.1

Table 4.20.: Durations for residual calculation with fixed coefficients for a grid of size 128^3

Kernel Variant	`gmem`	`const_arg`	`preproc`	`const_mem`
SM Busy	95.71	84	84.01	84.01
LSU Pipe Active [%]	24.81	11.65	11.69	11.69
FP64 Pipe Active [%]	95.71	84	84.01	84.01
Stall Long Scoreboard	43.82	58.4	33.17	33.05
Stall Tex Throttle	13.71	23.06	41.78	44.79

Table 4.21.: NCU metrics for residual calculation with fixed coefficients for a grid of size 128^3 on a Quadro T2000

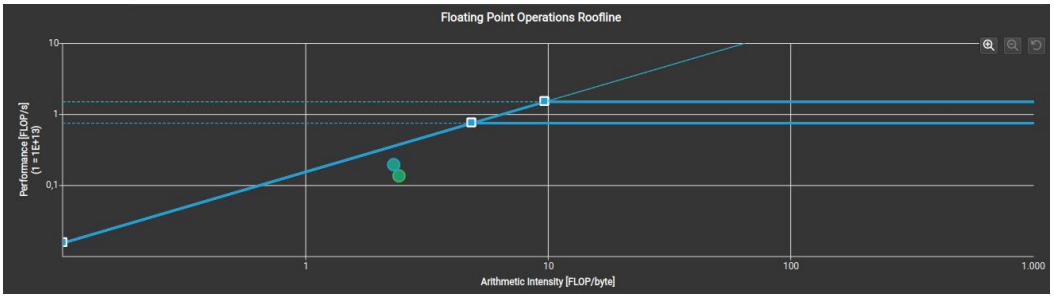


Figure 4.11.: Roofline from NCU for `gmem` and `preproc` variants of the residual kernel with fixed coefficients, executed on the A100 GPU

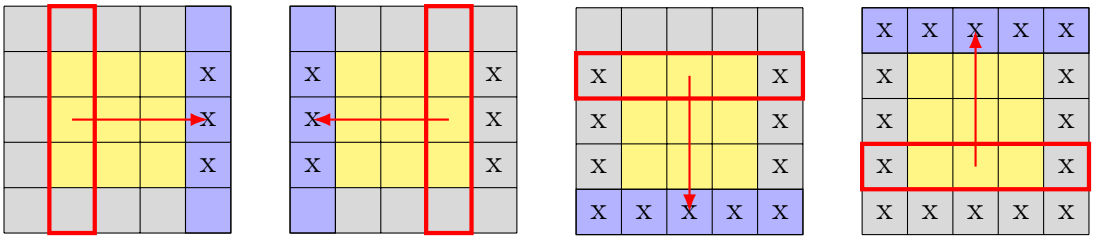


Figure 4.12.: Sequential ghost update sequence on a 2d grid. In each phase, values in the red rectangle are read and written to the blue cells. Cells containing the letter x are up-to-date.

4.4. Ghost Update

When using periodic boundary conditions, the ghost update routine reads data from the outermost layers of the real grid and writes it to the ghost points at the opposite direction of the grid. While the operation itself is not computationally expensive, the ghost update is a very frequent operation in the whole multigrid method implementation.

A sequential implementation is simple. For all three dimensions, we iterate over gh layers of the entire ghosted plane, where gh is the amount of ghost layers, and write data to the opposite side. Since this way, we are also iterating over the corners, they happen to be updated automatically after iterating over the last dimension. The procedure is illustrated in figure 4.12.

When parallelizing the ghost update for the GPU, we might launch a kernel with one work-item per grid point. We cannot just use the presented sequential procedure, as shifting values to the next dimension depends on ghost cells updated by the previous shifting direction. Instead, we can calculate the index of the real grid point based on the ghost point's index. For $gh \leq m$,

$$\begin{aligned} i_{\text{real}} &= i_{\text{gh}} + m, \quad 0 \leq i_{\text{gh}} < m, \\ i_{\text{real}} &= i_{\text{gh}} - m, \quad m + gh \leq i_{\text{gh}} < m + 2gh, \end{aligned}$$

where m is the number of grid points per dimension, i_{real} is the index of the real point, i_{gh} is the index of the ghost point and gh is the ghost layer width. For $gh > m$, which might occur when enlarging the grid to perform several Jacobi iterations without intermediate ghost update on coarse grids, we adjust the calculation slightly. First, we divide the ghost layers into blocks of size m and find the index of the block the ghost point is located in using $\text{gh_block_idx} = \lfloor (gh - 1 - i_{\text{gh}})/m \rfloor + 1$. This can then be multiplied with the real grid size m and added to the ghosted point's index to find the real point's

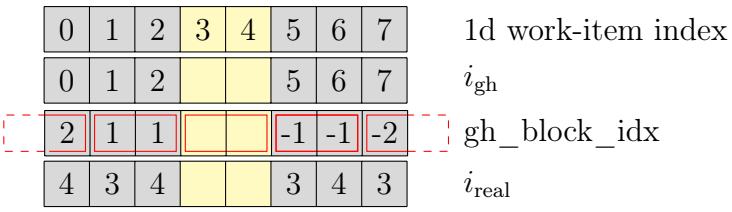


Figure 4.13.: Calculation of the real point index based on the ghost point index for $gh \geq m$. i_{real} is the index of the real point from which the ghost point with index i_{gh} gets its value. With ghost layers divided into blocks of size m, gh_block_idx is the index of the block the ghost point is located in.

index, i.e.

$$\begin{aligned} i_{\text{real}} &= \text{gh_block_idx} \cdot m + i_{\text{gh}} \\ &= (\lfloor (gh - 1 - i_{\text{gh}})/m \rfloor + 1) \cdot m + i_{\text{gh}}. \end{aligned}$$

This is illustrated in figure 4.13.

An implementation of the kernel is given in listing 4.13. The kernel is launched with one work-item per grid point. Work-items mapping to real grid points are just masked off. The implementation of the calculation of the real point index requires double precision arithmetic, which is costly, especially on smaller GPUs. We can instead use an approach involving pure integer arithmetic.

```
__kernel void update_ghosts(
    __global double* restrict c,
    int m, int n, int o,
    int ghm, int ghn, int gho)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    int k = get_global_id(2);

    int mgh = m + 2 * ghm;
    int ngh = n + 2 * ghn;
    int ogh = o + 2 * gho;

    if ((i < ghm || j < ghn || k < gho ||
         i >= ghm + m || j >= ghn + n || k >= gho + o) &&
        (i < mgh && j < ngh && k < ogh))
    {
        int ireal = i + floor(((double)(ghm - 1 - i)) / m + 1) * m;
        int jreal = j + floor(((double)(ghn - 1 - j)) / n + 1) * n;
        int kreal = k + floor(((double)(gho - 1 - k)) / o + 1) * o;

        int idx_gh_cell = i * ngh * ogh + j * ogh + k;
        int idx_real_cell = ireal * ngh * ogh + jreal * ogh + kreal;

        c[idx_gh_cell] = c[idx_real_cell];
    }
}
```

Listing 4.13: OpenCL ghost update kernel launching one work-item per ghost point.

We can formulate the real point index as

$$i_{\text{real}} = (i_{\text{gh}} - gh) \mod m + gh.$$

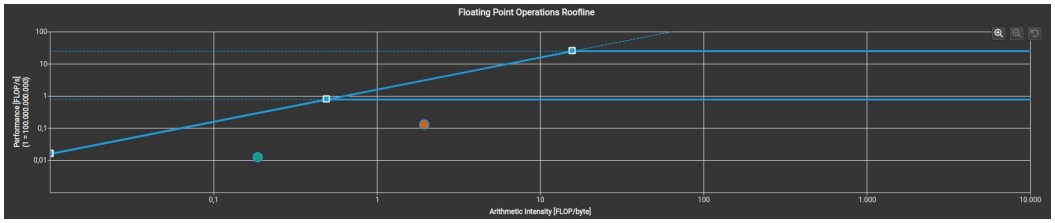


Figure 4.14.: Roofline for the double precision version of the ghost update kernel on the Quadro T2000 GPU for a grid of size 128^3 and $gh = 1$

In this version, we first shift the index, such that the first real point has index 0. Then, we apply the modulo operator to map the index to the range $[0, m-1]$, and finally shift it back into its original position. In the implementation, we need to respect that the modulo operator might return a negative result. Therefore, we use `((i - gh) % m + m) % m + gh` instead of `(i - gh) % m + gh`. Although we now need to use the modulo operator twice, this version ends up being significantly faster than the other implementation on the Quadro T2000, as we can see in table 4.22. On the A100, it does not make a noticeable difference. Just like with the fixed coefficients residual kernel, the kernel using double precision arithmetic suffers from a high tex throttle stall (16 cycles). This can also be seen in the FP64 pipe dominating the overall pipe utilization (86 %). Ultimately, this kernel is compute bound on the Quadro T2000, as shown in the roofline model in figure 4.14.

Although the `modulo` version performing pure integer instructions is a significant improvement over the version using double-precision, the modulo operator itself is a costly operation. Instead of casting to `double`, we might be able to cast to `float`, as the high precision is not needed in this place. When just replacing

```
i + floor(((double)(ghm - 1 - i)) / m + 1) * m;
```

with

```
i + floor(((float)(ghm - 1 - i)) / m + 1) * m;,
```

we might get into rounding issues if the expression

```
((float)(ghm - 1 - i)) / m
```

is close to negative zero, resulting in adding 1 not being exactly 1, therefore rounding down towards 0 instead of 1. This can be mitigated by adding a small ϵ to the increment, i.e.

```
i + floor(((float)(ghm - 1 - i)) / m) + 1.000001f * m;.
```

Furthermore, we exclude the addition from the `floor()`-function. This version yields a slight advantage over the previous version on the Quadro T2000.

	Time [µs]									
	Quadro T2000					A100				
Work-Group size	(a)	(b)	(c)	(d)	(e)	(a)	(b)	(c)	(d)	(e)
`2x2x8`	617	240	234	235	235	77	77	76	77	76
`4x4x4`	506	116	111	113	113	42	42	42	42	42
`4x4x8`	506	78	72	72	71	25	26	25	25	25
`4x4x16`	507	94	85	86	84	18	18	17	17	17
`8x8x8`	667	119	106	101	99	20	22	17	17	16
`32x1x1`	1.726	238	238	241	242	88	88	87	87	87
`64x1x1`	1.729	175	158	156	154	56	56	56	55	55
`128x1x1`	1.731	176	158	159	154	41	40	40	40	40

Table 4.22.: Kernel durations for the ghost update kernel for a grid of size 128^3 and $gh = 1$. (a) using `double`, (b) using modulo operator, (c) using `float`, (d) using `convert_int_rtn`, (e) using integer truncation.

An alternative to the `floor()`-function in combination with conversion to integer is OpenCL C's function `convert_int_rtn(float)`. We use that as a drop-in replacement for the `floor()`-function. However, this does not make any difference in kernel runtime on both tested GPUs.

A last attempt involves the avoidance of rounding the intermediate result altogether and use integer truncation instead. Since the result might be negative and integer truncation happens towards zero, we need to conditionally subtract 1 before the truncation if the result is negative, i.e.

```
float tmpi = (((float)(ghm - 1 - i)) / m) + 1;
int ireal = i + ((int)(tmpi >= 0 ? tmpi : tmpi - 1)) * m;
```

This variant does not alter the kernel duration compared to the previous attempts, too.

Now that we compared the different kernel versions, we want to advance to discussing the differences that stem from the work-group sizes. While no significant difference was noted running the residual kernel using a 1d or 3d `ndrange`, the 1d kernel is far behind for the ghost update kernel. Since a large portion of the inner grid is not touched at all by this kernel, a high fraction of warps in the 1d work-groups reach into this unused range, resulting in a high amount of idle work-items per warp. NCU shows, that on the Quadro T2000, the amount of active work-items per warp is at 17.7 % vs. 25.9 % in average for the work-groups `64x1x1` and `4x4x8`, respectively. This can also be seen in the almost double amount of integer instruction executed, the amount of eligible work-items per scheduler per cycle and the achieved occupancy. The metrics are listed in table 4.23.

Work-Group size	4x4x8	64x1x1
Integer Instructions Executed	2.2M	5.2M
Eligible Warps per Scheduler per Cycle	1.63	1.13
Avg. Active Work-Items per Warp [%]	25.88	17.74
Achieved Occupancy [%]	69.16	53.44

Table 4.23.: NCU metric selection for the modulo variant of the ghost update kernel for a grid of size 128^3 and $gh = 1$, executed on Quadro T2000.

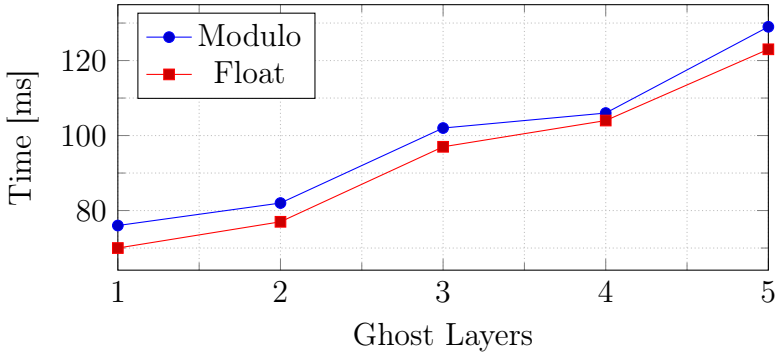


Figure 4.15.: Kernel durations for $gh \in [1, \ldots, 5]$ for the modulo and float versions. Executed on a Quadro T2000 GPU for a grid of size 128^3.

A further idea might be to change the kernel layout, such that the work-items map to ghost points only. This would lead to not having any redundant work-items and no warps finishing early. However, as those warps without any work-item mapping to a ghost point do finish early as a whole, the remaining warps can be scheduled just fine, attaining a high device saturation.

Figure 4.15 shows a plot of kernel duration for the ghost update kernel for a grid of size 128^3 and $gh \in \{1, \ldots, 5\}$ for the modulo and float versions on the Quadro T2000. We can observe a linear increase in kernel duration with the size of the ghost layer with slightly steeper slopes towards the odd numbers of layers.

4.5. Inter-Grid Transfer

The inter-grid transfer operations, i.e. prolongation and restriction, project a vector from a coarse grid to a fine grid and vice versa. Since `mgcl` implements full coarsening, and we use the full-weighted restriction and bilinear prolongation operators, one coarse point affects eight fine points in 3d.

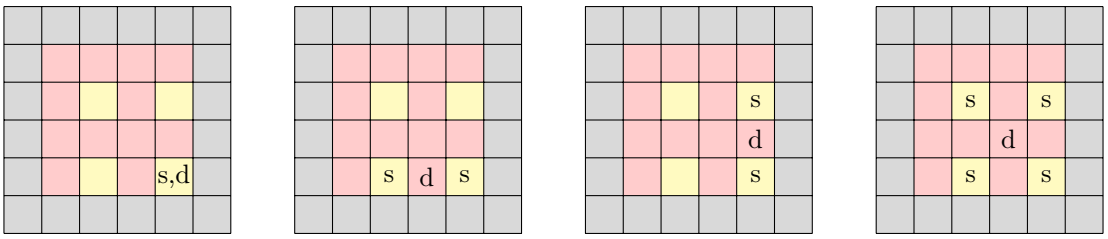


Figure 4.16.: The four cases in bilinear prolongation in 2d. "s" determines a source point, "d" a destination point. Red cells are fine grid points, yellow cells are coarse grid points.

	Time [µs]					
	Quadro T2000			A100		
Work-Group size	(a)	(b)	(c)	(a)	(b)	(c)
`4x4x4`	244	240	315	41	39	46
`8x4x4`	201	218	289	34	31	39
`8x8x4`	201	223	314	35	30	38
`8x8x8`	217	251	350	35	32	38
`32x1x1`	248	221	342	26	40	71
`64x1x1`	250	215	290	28	24	40
`128x1x1`	251	213	291	27	20	30

Table 4.24.: Durations of the prolongation kernel variants on a fine grid with size of 128^3 and $gh = 1$. (a): One work-item per coarse point, (b): 4 work-items per coarse point, (c): 8 work-items per coarse point.

4.5.1. Prolongation

For the prolongation, we can vary the amount of work-items mapping to one coarse point. We choose a one-to-one, one-to-four and one-to-eight mapping here. Mapping multiple work-items to coarse points introduces branches to determine the update rule for the fine grid points. The different cases are shown in figure 4.16 for 2d. As with the residual kernel version using multiple work-items for a single stencil application (section 4.2.3), if we organize the work-items in a way such that the different paths are taken by different warps, we can completely avoid branch divergence.

Table 4.24 shows that for a 128^3 grid the one-to-one version performs best on a Quadro T2000, followed by the four-to-one version and finally the eight-to-one version. On the A100, the four-to-one version is fastest. These trends hold for larger grid sizes, as shown in table 4.25. The figures 4.17 and 4.18, extracted from NCU, show that all three versions are memory bound on both GPUs.

Generally, using a work-group like `8x4x4` produces worse coalescing than e.g.

	Time [ms]					
	Quadro T2000			A100		
Work-Group size	(a)	(b)	(c)	(a)	(b)	(c)
4x4x4	1.85	1.84	2.46	2.91	2.16	2.56
8x4x4	1.41	1.68	2.25	2.06	1.48	2.00
8x8x4	1.61	1.73	2.44	2.02	1.47	2.00
8x8x8	1.80	1.96	2.81	2.08	1.55	2.23
32x1x1	1.60	1.73	2.69	1.18	1.95	3.89
64x1x1	1.60	1.67	2.26	1.25	1.01	1.95
128x1x1	1.61	1.67	2.23	1.24	1.02	1.61

Table 4.25.: Durations of the prolongation kernel variants on fine grids with sizes of 256^3 (Quadro T2000) and 512^3 (A100) and $gh = 1$. (a): One work-item per coarse point, (b): 4 work-items per coarse point, (c): 8 work-items per coarse point.

`64x1x1`. However, the amount of redundant work-items induced by padding is lower for the former work-group size if the real grid size extends are powers of two, hence the better performance of `8x4x4` in the case of the 128^3 and 256^3 real grids using the Quadro T2000. If we instead choose a real grid size of 126^3, such that the ghosted grid is 128^3, then there are not as many redundant work-items for a 1d work-group, resulting in 1d work-group sizes performing better than `8x4x4`. In the 128^3 case, using e.g. the work-group size `32x1x1` causes 25 % more floating point instructions being executed compared to `8x4x4`. For 126^3, the difference is only at 1.5 %. Since the Quadro T2000 has poor double precision performance compared to e.g. the A100, the worse memory access pattern of the `8x4x4` work-group cannot be compensated by the additional floating point instructions for 128^3. This is not the case for the A100, hence 1d work-groups performing better in all cases.

NCU shows that for a 128^3 grid the four-to-one version can keep the A100 busy for a higher fraction of the time than the one-to-one version (58.4 % vs. 20.8 %). This is different for the Quadro T2000: 82 % vs. 84 %. The reason is, again, that the Quadro T2000 is already close to its peak compute performance. The compute throughput is above 80 % for both kernel versions. The A100 shows a clear distinction in compute throughput: 17 % vs. 46 % for one-to-one and four-to-one versions, respectively. Although the compute throughput is higher for eight-to-one version compared to the four-to-one version, the different kernel layout with branches being spread out lead to an 51 % (A100) and 54 % (Quadro T2000) increase of sectors sent from L1.

Using local memory to share coarse grid values with other work-items might be another valid option here, but as the prolongation kernel is only called once per level in the entire v-cycle, the focus in this work is placed on other parts of the solver.

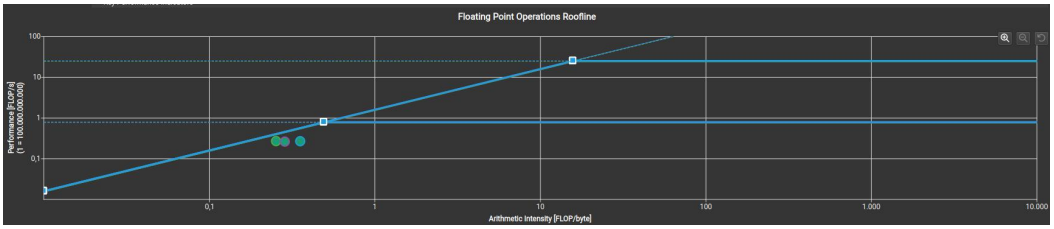


Figure 4.17.: Roofline for the prolongation kernel versions on the Quadro T2000 for a fine grid of size 128^3 with $gh = 1$.

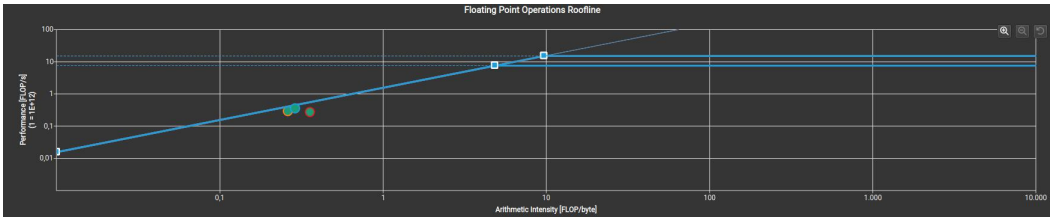


Figure 4.18.: Roofline for the prolongation kernel versions on the A100 for a fine grid of size 512^3 with $gh = 1$.

4.5.2. Restriction

With the full-weight restriction operator, we basically apply a 27-point stencil to every fine grid point that corresponds to a coarse grid point and store the result for the coarse grid point. We can again use a kernel that maps one or multiple work-items to one coarse grid point. Multiple work-items per coarse point require the usage of local memory to collect intermediate results at the end of the kernel. The concept is similar to the residual kernel version using local memory (see section 4.2.3). Table 4.26 shows kernel durations for one-to-one, two-to-one and four-to-one mappings for a 128^3 grid. There is no significant difference between the one-to-one and two-to-one version. The four-to-one version needs more time than the other two versions. For the one-to-one version both 1d and 3d work-group shapes work equally well.

4.6. Residual Sum

The residual sum is used to calculate the L2-norm of the residual, which can be used as a termination criterion. The infinity-norm is also implemented in `mgcl` and requires to find the global maximum of the residual array. Since both operations can be implemented as global reductions and therefore share the same algorithmic structure, we only elaborate on the summation of the residual.

	Time [µs]					
	Quadro T2000			A100		
Work-Group size	(a)	(b)	(c)	(a)	(b)	(c)
`4x4x4`	227	450	960	24	33	51
`8x4x4`	221	451	960	19	30	38
`8x8x4`	223	453	961	20	28	38
`8x8x8`	227	455	963	22	30	38
`32x1x1`	221	451	963	20	26	43
`64x1x1`	221	229	482	20	22	28
`128x1x1`	327	228	248	20	20	23

Table 4.26.: Durations of the restriction kernel variants on a fine grid with size of 128^3 and $gh = 1$. (a): One work-item per coarse point, (b): 2 work-items per coarse point, (c): 4 work-items per coarse point.

Sum reduction is a prime example for using local memory for communication between work-items inside a work-group, thus it is fairly well studied in the literature (e.g. chapter 6.1 of [43], chapter 5.3 of [98]). Especially, there is a presentation by Mark Harris specifically about the optimization of sum reduction using CUDA [36]. Except for the usage of template parameters, the code can be ported to OpenCL, too. Luitjens [63] follows up on this work by using warp shuffle intrinsics introduced with CC 5.0. This feature is available to OpenCL via the extension `cl_khr_subgroup_shuffle`, which includes the usage of sub-groups. Unfortunately, this is not realized in Nvidia's OpenCL implementation, therefore we resort to local memory usage only.

A first version, that launches one work-item per element in the vector to be summed, is shown in listing 4.14. Each work-group sums the elements mapped to by its work-items and stores the unary result in the global buffer `partial_sums`. The local memory buffer `buf_local` is used to store the intermediate results. The tree-based approach is illustrated in figure 4.19. Note that, instead of summing values of neighboring work-items, we fold the upper half of remaining values onto the lower half. This enables coalesced global memory access and avoids bank conflicts, similarly to the residual kernel in section 4.2.3. Since we cannot explicitly synchronize global memory, we need multiple kernel invocations to sum up the partial sums. The kernel is launched repeatedly, having `partial_sums` as both input and output starting from the second iteration. When the remaining input vector fits into a single work-group, which can easily be checked by comparing against the maximum work-group size, the kernel is invoked a final time. If the chosen work-group size is smaller than the maximum work-group size, but the vector would fit into a single work-group, we might increase the work-group size for the final kernel invocation and thus save one kernel invocation.

```
__kernel void sum_partial_global_eq_num_elements(
    __global double* restrict buf,
    __global double* restrict partial_sums,
    __local double* buf_local,
    int num_elements)
{
    int i = get_global_id(0);
    int wg_size = get_local_size(0);
    int iloc = get_local_id(0);

    buf_local[iloc] = (i < num_elements) ? buf[i] : 0;

    for (int stride = wg_size >> 1; stride > 0; stride >>= 1) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if (iloc < stride && iloc + stride < wg_size && iloc + stride < num_elements)
            buf_local[iloc] += buf_local[iloc + stride];
    }

    if (iloc == 0)
        partial_sums[get_group_id(0)] = buf_local[iloc];
}
```

Listing 4.14: Sum reduction kernel mapping one work-item to one element in the vector.

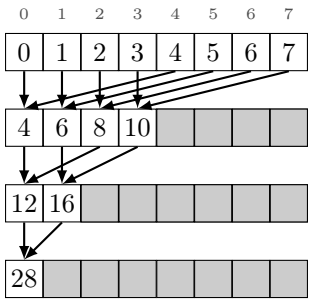


Figure 4.19.: Tree-based approach for sum reduction

For large vectors, it might be beneficial to serialize a portion of the work. Brent's theorem [10] states, that

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty,$$

where T_P is the amount of time needed by an algorithm on P processors, T_1 is the time needed by the algorithm on one processor and T_∞ is the time needed on an infinite amount of processors, i.e. the critical path. For sum reduction, $T_1 = O(N)$ and $T_\infty = O(\log_2 N)$. Asymptotically, the expression

$$T_P \leq \frac{N - \log_2 N}{P} + \log_2 N$$

can be simplified to

$$T_P \leq \frac{N}{P} + \log_2 N.$$

Therefore, the theorem suggests to use $O(N/\log_2 N)$ processors (work-items in our case) for sum reduction. This can be realized by loading a batch into local memory before starting the reduction, as shown in listing 4.15. We denote this as version (a). For this test and for the sake of simplicity, we set the batch size to 1 for subsequent kernel invocations. Of course, the batch size could also be recalculated as $\log_2 N$ in each iteration.

```
1 buf_local[iloc] = (i < num_elements) ? buf[i] : 0;
2 int idx = i + get_global_size(0);
3 for (int f = 1; f < batchSize && idx < num_elements; f++) {
4     buf_local[iloc] += buf[idx];
5     idx += get_global_size(0);
6 }
```

Listing 4.15: Loading a batch of elements into local memory

A further optimization, focusing on the implementation itself, is to unroll the loops. When we stick to work-groups with sizes of powers of two, the loops can manually be unrolled as shown in listing 4.16. On current Nvidia GPUs, the work-group size is limited to 1024 in the first dimension, therefore the set of manual unrollings is finite. If we use a preprocessor constant to define the work-group size, the conditions will be evaluated at compile time, gaining a higher chance of producing efficient code. Additionally, in this version we sum up the values of the last warp without the need of using local memory barriers. This is valid, since all work-items of the warp execute the same instruction simultaneously and we use the `volatile` keyword on the partial sums variable. An excerpt of relevant changes can be seen in listing 4.16. Using preprocessor directives to affect the path of the code also means that we need to recompile the kernels whenever we choose a different work-group size. An alternative to this would be to create entire separate kernels for all work-group sizes. We call this version (b).

```
// ...
barrier(CLK_LOCAL_MEM_FENCE);
if (SUM_WG_SIZE >= 1024) {
    if (iloc < 512) {
        buf_local[iloc] += buf_local[iloc + 512];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
if (SUM_WG_SIZE >= 512) {
    if (iloc < 256) {
        buf_local[iloc] += buf_local[iloc + 256];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
//...
if (iloc < 32) {
    if (SUM_WG_SIZE >= 64) {
        buf_local[iloc] += buf_local[iloc + 32];
    }
    if (SUM_WG_SIZE >= 32) {
        buf_local[iloc] += buf_local[iloc + 16];
    }
//...
```

Listing 4.16: Sum reduction kernel mapping multiple elements to one work-item and having unrolled loops.

Lastly, instead of finishing the reduction on the GPU, we can read the partial sums from the GPU to the host and finish it using the CPU. This is motivated by the fact, that the performance of a single CPU core is generally higher than the performance of a single GPU core, and that the size of the problem shrinks substantially with each kernel invocation. However, with this approach we need to keep in mind, that data transfer between host and device is expensive. The interruption of kernel streamlining should be negligible as in our multigrid implementation, we only need to compute the residual sum when we send it to the host afterwards anyway in order check the termination criterion. We can add a parameter to control the number of iterations executed on the GPU (and thus the size of the remaining vector to be summed up on the CPU). We denote this as version (c).

Table 4.27 shows no significant differences between the lowest durations of the different versions for both the Quadro T2000 and the A100 using double-precision data. The impact of varying the batch size can be seen in figure 4.20. The unrolled version is faster than the other two versions for low batch sizes, but the differences diminish with larger batch sizes. Note that batch size 10 is still far away from the suggested batch sizes of $\log_2(512^3) = 27$ and $\log_2(1024^3) = 30$.

4.7. Coarse Grid Operator

In the previous section we covered the parts of the solving step. Now we want to elaborate on the setup, especially the calculation of the coarse grid operator. We use the Galerkin approach to calculate the coarse grid operator $\mathbf{A}_{2h} = \mathbf{R}\mathbf{A}_h\mathbf{P}$. Since the formulation is matrix-based, but we want to retain the stencil notation,

	Time [ms]					
	Quadro T2000			A100		
N	(a)	(b)	(c)	(a)	(b)	(c)
32^3	0.038	0.039	0.034	0.049	0.050	0.043
64^3	0.052	0.050	0.047	0.053	0.047	0.052
128^3	0.152	0.149	0.147	0.051	0.056	0.057
256^3	0.909	0.908	0.906	0.137	0.146	0.146
512^3	6.945	6.951	6.942	0.792	0.798	0.797

Table 4.27.: Lowest durations for our sum reduction implementations, grouped by version and problem size. Using work-group sizes in $\{32, 64, 128, 256, 512\}$, batchSizes in range of -10 % and $+30$ % of $\log_2 N$ and a maximum of $\{1, 2, 3\}$ kernel calls on the GPU before finishing on the CPU. Versions: (a) GPU only using loops, (b) GPU only, loops unrolled, (c) start on GPU, finish on CPU.

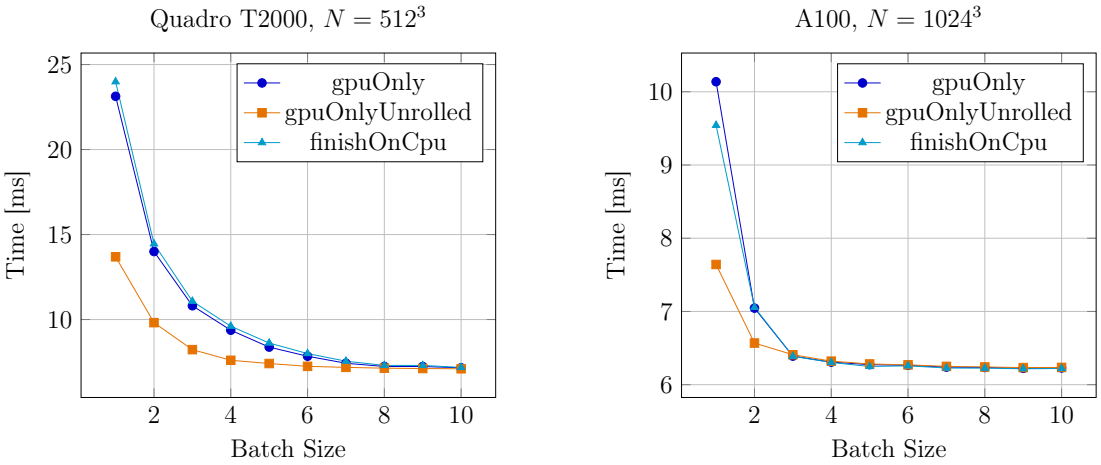


Figure 4.20.: Impact of different batch sizes for the sum reduction kernel versions

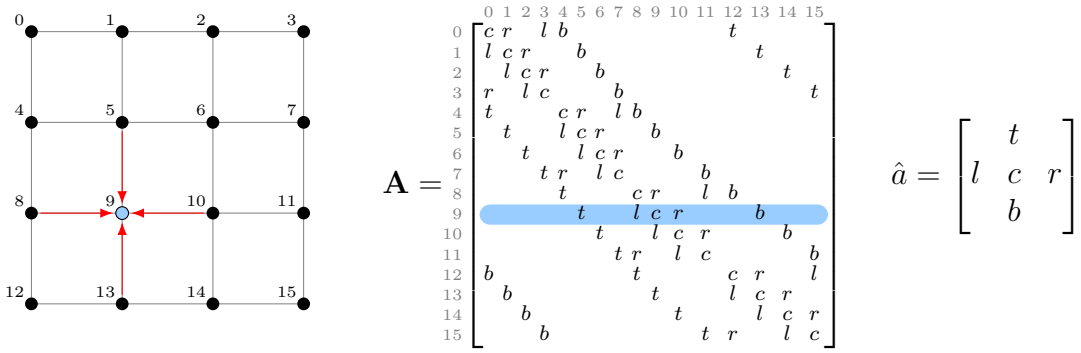


Figure 4.21.: 2d 4x4 grid with 1d grid point indices (left). A 5-point stencil is applied to grid point 9, taking values from points 5, 8, 9, 10 and 13. The affected row is highlighted in the system matrix $\mathbf{A}$ (middle). A stencil $\hat{a}$ of the corresponding set of stencils $\hat{\mathbf{A}}$ of $\mathbf{A}$ is given on the right.

we first need to discuss the relation between both representations. In the following, we stick to 2d for clarity of presentation, but the concepts can easily be extended to higher dimensions.

4.7.1. Matrix-based Approach (Stencil Arithmetic)

Consider the 4x4 grid showed in figure 4.21, the corresponding system matrix $\mathbf{A}$ and a stencil $\hat{a}$ with radius 1. We want to calculate the matrix row corresponding to grid point 9 of its corresponding set of stencils $\hat{\mathbf{A}}$. In order to apply a stencil on grid point 9, the 9th row of $\mathbf{A}$ is multiplied with the vector of grid point values $\mathbf{v}$. We observe, that row 9 of $\mathbf{A}$ only has entries in the columns with indices equal to the grid points indices involved in the stencil operation. We conclude that (a) a matrix row index corresponds to the location of the grid point a stencil is applied to and (b) a column index corresponds to the grid point that is mapped to by an entry in the stencil.

The implementation of the multiplication of two matrices $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ can be realized using three nested loops, as shown in listing 4.17. Using this structure, both matrices are iterated row-wise. Using stencil notation, iterating over rows of

```
for(int ai = 0; ai < am; ai++)
  for(int ajbi = 0; aj < an; aj++)
    for(int bj = 0; bj < bn; bj++)
      c[ai][bj] += a[ai][ajbi] * b[ajbi][bj];
```

Listing 4.17: Matrix-matrix-multiplication by row-wise iteration over both matrices.

$\mathbf{A}$ (first loop) corresponds to iterating over the grid points, iterating over columns

of **A** (second loop) corresponds to iterating over the stencil coefficients of $\hat{a}_{ij}$ and iterating over columns of **B** (third loop) corresponds to iterating over the stencil coefficients of $\hat{b}_{kl}$. $\hat{a}_{ij}$ is located at grid point (i, j) and the current coefficient maps to grid point (k, l), which is where $\hat{b}_{kl}$ is located. Thus, we can translate the matrix-based implementation to stencil notation as shown in listing 4.18.

```
for (int x = 0; x < m; x++)
for (int y = 0; y < n; y++)
  for (int ai = 0; ai < a.width; ai++)
  for (int aj = 0; aj < a.width; aj++)
    for (int bi = 0; bi < b.width; bi++)
    for (int bj = 0; bj < b.width; bj++)
    {
      int gpi = x + ai - a.radius + ghb;
      int gpj = y + aj - a.radius + ghb;

      int ci = ai + bi;
      int cj = aj + bj;

      c[x + ghc][y + ghc][ci][cj] +=
        a[x + gha][y + gha][ai][aj] *
        b[gpi][gpj][bi][bj];
    }
```

Listing 4.18: Implementation of stencil multiplication by iterating over grid points and coefficients of each input stencil.

Note that when multiplying two stencils of widths $w_{\hat{a}}$ and $w_{\hat{b}}$, the resulting stencil has a width of $w_{\hat{c}} = w_{\hat{a}} + w_{\hat{b}} - 1$, i.e. the resulting stencil is wider than the input stencils. We denote the radius of a stencil as $r_{\hat{c}} = \lfloor w_{\hat{c}}/2 \rfloor$, i.e. for $\hat{c} = \hat{a} \cdot \hat{b}$, $r_{\hat{c}} = r_{\hat{a}} + r_{\hat{b}}$. Storing stencils as 4d array with grid-first layout `sv[gpx][gpy][cx][cy]`, multiple indices must be calculated in each iteration. We take the possibility of different ghost layer sizes for each stencil into account. Furthermore, we assume stencils to be symmetric, i.e. having the same radius in each dimension. The index calculations can be derived as follows.

- The indices $\text{idx}((i, j)_{\hat{a}})$ for the location of $\hat{a}_{ij}$ and the indices $\delta(\hat{a}_{ij}(\text{xoff,yoff}))$ of the affected coefficients $\hat{a}_{ij}(\text{xoff,yoff})$ are given directly by the two outermost loops.
- The index $\delta(\hat{b}_{ij}(\text{xoff,yoff}))$ for the coefficient $\hat{b}_{ij}(\text{xoff,yoff})$ of stencil $\hat{b}$ is given by the innermost loop.
- The index $\text{idx}((i, j)_{\hat{c}})$ for the location of $\hat{c}$ is given by the outermost loop, since $(i, j)_{\hat{a}} = (i, j)_{\hat{c}}$.
- The location of $\hat{b}$ is given by adding the offsets

  $$\delta(\hat{a}_{ij}(\text{xoff,yoff})) \in [0, w_a) \times [0, w_a)$$

  of the index of current coefficient of $\hat{a}$ to the location of $\hat{a}$ minus the radius of $\hat{a}$, i.e.

  $$(k, l)_{\hat{b}} = (i + \delta(\hat{a}_i(\text{xoff})) - r_a, j + \delta(\hat{a}_j(\text{yoff})) - r_a).$$

This can be illustrated by following the path from the location of $\hat{a}$ to the location of the grid point mapped to by the current coefficient of $\hat{a}$.

- The coefficient index of $\hat{c}$ is given by adding the offsets of the current coefficient indices of $\hat{a}$ and $\hat{b}$. We can derive this by starting from the same formula that we used to calculate the location of $\hat{b}$. In 1d:

$$\begin{aligned}
\delta(\hat{c}(\text{off})) &= \text{idx}(i_{\hat{c}} + \text{off}) - \text{idx}(i_{\hat{c}}) + r_{\hat{c}} \\
&\qquad \mid \text{rowidx}(c) = \text{rowidx}(a), \text{colidx}(c) = \text{colidx}(b) \\
&= \text{idx}(i_{\hat{b}} + \text{off}) - \text{idx}(i_{\hat{a}}) + r_{\hat{c}} \\
&= \text{idx}(i_{\hat{b}}) + \delta(\hat{b}(\text{off})) - r_{\hat{b}} - (\text{idx}(i_{\hat{a}} + \text{off}) + r_{\hat{a}} - \delta(\hat{a}(\text{off}))) + r_{\hat{c}} \\
&= \text{idx}(i_{\hat{b}}) + \delta(\hat{b}(\text{off})) - r_{\hat{b}} - \text{idx}(i_{\hat{a}} + \text{off}) - r_{\hat{a}} + \delta(\hat{a}(\text{off})) + r_{\hat{c}} \\
&\qquad \mid \text{colidx(a)} = \text{rowidx(b)} \rightsquigarrow \text{idx}(i_{\hat{b}}) = \text{idx}(i_{\hat{a}} + \text{off}) \\
&= \delta(\hat{b}(\text{off})) - r_{\hat{b}} - r_{\hat{a}} + \delta(\hat{a}(\text{off})) + r_{\hat{c}} \quad \mid r_{\hat{c}} = r_{\hat{a}} + r_{\hat{b}} \\
&= \delta(\hat{a}(\text{off})) + \delta(\hat{b}(\text{off}))
\end{aligned}$$

We observe, that this implementation repeatedly accesses different locations of $\hat{c}$ in subsequent iterations. This would require an OpenCL kernel to update the value in global memory multiple times. As global memory accesses are expensive, we want to find a way to reduce the number of accesses, preferably accumulating the current value of $\hat{c}$ in private memory, i.e. a register. This can be achieved by reordering the loops as shown in listing 4.19.

```
for each grid point:
  for each coefficient of c:
    for each coefficients of a and b affecting c:
      ...
```

Listing 4.19: Reordering the loops for stencil multiplication.

The loop bounds for the outer two loops are clear. For the innermost loop, we need to find valid ranges for the coefficients of $\hat{a}$ and $\hat{b}$ that affect the current coefficient of $\hat{c}$. For each coefficient $\hat{c}(\text{off})$, $\delta(\hat{c}(\text{off})) = \delta(\hat{a}(\text{off})) + \delta(\hat{b}(\text{off}))$. For $\hat{a}$ and $\hat{b}$,

$$\begin{aligned}
0 \leq \delta(\hat{a}(\text{off})) &\leq 2r_{\hat{a}}, \\
0 \leq \delta(\hat{b}(\text{off})) = \delta(\hat{c}(\text{off})) - \delta(\hat{a}(\text{off})) &\leq 2r_{\hat{b}}.
\end{aligned} \tag{4.1}$$

For the lower bound,

$$0 \leq \delta(\hat{c}(\text{off})) - \delta(\hat{a}(\text{off})) \rightsquigarrow \delta(\hat{a}(\text{off})) \leq \delta(\hat{c}(\text{off})).$$

For the upper bound,

$$\delta(\hat{c}(\text{off})) - \delta(\hat{a}(\text{off})) \leq 2r_{\hat{b}} \rightsquigarrow \delta(\hat{a}(\text{off})) \geq \delta(\hat{c}(\text{off})) - 2r_{\hat{b}}.$$

Thus,

$$\delta(\hat{c}(\text{off})) - 2r_{\hat{b}} \leq \delta(\hat{a}(\text{off})) \leq \delta(\hat{c}(\text{off})). \tag{4.2}$$

Since both equations (4.1) and (4.2) need to be satisfied, we obtain the final result by intersection,

$$[0, 2r_{\hat{a}}] \cap [\delta(\hat{c}(\text{off})) - 2r_{\hat{b}}, \delta(\hat{c}(\text{off}))] = [\max(0, \delta(\hat{c}(\text{off})) - 2r_{\hat{b}}), \min(\delta(\hat{c}(\text{off})), 2r_{\hat{a}})]. \tag{4.3}$$

By starting at the lower bound for a and at the upper bound for b in the implementation, $\delta(\hat{c}(\text{off})) = \delta(\hat{a}(\text{off})) + \delta(\hat{b}(\text{off}))$ is hold. A 2d implementation might look like as shown in listing 4.20.

```
for (int x = 0; x < m; x++)
for (int y = 0; y < n; y++)
  for (int ci = 0; ci < c.width; ci++)
  for (int cj = 0; cj < c.width; cj++) {
      double csum = 0;
      for (int a_i = max(0, ci - 2*b.radius), b_i = min(ci, 2*b.radius);
              a_i <= min(ci, 2*a.radius) && b_i >= max(0, ci - 2*a.radius);
              a_i++, b_i--)
      for (int a_j = max(0, cj - 2*b.radius), b_j = min(cj, 2*b.radius);
              a_j <= min(cj, 2*a.radius) && b_j >= max(0, cj - 2*a.radius);
              a_j++, b_j--) {
        int gpi = i + a_i - a.radius + ghb;
        int gpj = j + a_j - a.radius + ghb;

        csum +=
            a[x + gha][y + gha][a_i][a_j] *
            b[gpi][gpj][b_i][b_j];
    }
    c[x + ghc][y + ghc][ci][cj] = csum;
  }
```

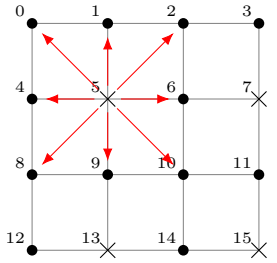
Listing 4.20: Implementation of stencil multiplication by iterating over grid points, coefficients of the output stencil and coefficients of both input stencils in a single loop.

The conversion between matrix and stencil notations is given in section A.5 of the appendix, as it is helpful when working with stencil arithmetic.

4.7.2. Galerkin via Stencil Arithmetic

In order to apply the stencil multiplication implementation to the Galerkin approach, we first need to translate the restriction matrix $\mathbf{R}$ and prolongation matrix $\mathbf{P}$ to quadratic matrices. Usually, the number of rows of $\mathbf{R}$ and number of columns of $\mathbf{P}$ are equal to the number of grid points of a coarser grid and the number of columns of $\mathbf{R}$ and number of rows of $\mathbf{P}$ are equal to the number of grid points of a finer grid, i.e. $\mathbf{R}^{N/2^d \times N}$ and $\mathbf{P}^{N \times N/2^d}$ in d dimensions for full coarsening. Due to the variational property $\mathbf{R} = c\mathbf{P}^T, c \in \mathbb{R}$, $\mathbf{R}$ and $\mathbf{P}$ inhibit the same stencil

$$\mathbf{P} = \mathbf{I}_{2h}^{h} = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_{2h}^{h}$$



$$\mathbf{R} = \mathbf{I}_{h}^{2h} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_{h}^{2h}$$

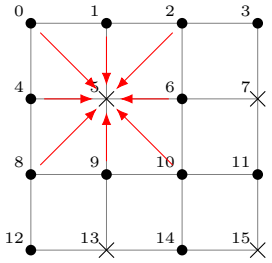


Figure 4.22.: The gathering stencil **R** collects values from surrounding fine grid points and stores the result in its coarse grid point. The distributing stencil **P** takes the value from its coarse grid point and distributes it to surrounding fine grid points.

structure, albeit **R** being a gathering stencil and **P** being a distributing stencil. This is illustrated in figure 4.22.

We can therefore expand the matrices to a square matrix **S**, pretending that we operate only on the fine grid, and cut out the correct rows and columns afterwards using a cutting matrix **K**. Thus,

$$\mathbf{RAP} = \mathbf{K} \cdot \mathbf{SAS} \cdot \mathbf{K}^T \tag{4.4}$$

with

$$\mathbf{K} = \begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{matrix} \\ \begin{bmatrix} & & & & 1 & & & & & & & & & & & \\ & & & & & 1 & & & & & & & & & & \\ & & & & & & & & & & & & & 1 & & \\ & & & & & & & & & & & & & & & 1 \end{bmatrix} \end{array} \text{ and } \mathbf{S} = \frac{1}{16}\begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{matrix} \\ \begin{bmatrix} 4 & 2 & & 2 & 2 & 1 & & 1 & & & & & 2 & 1 & & 1 \\ 2 & 4 & 2 & & 1 & 2 & 1 & & & & & & 1 & 2 & 1 & \\ & 2 & 4 & 2 & & 1 & 2 & 1 & & & & & & 1 & 2 & 1 \\ 2 & & 2 & 4 & 1 & & 1 & 2 & & & & & 1 & & 1 & 2 \\ 2 & 1 & & 1 & 4 & 2 & & 2 & 2 & 1 & & 1 & & & & \\ 1 & 2 & 1 & & 2 & 4 & 2 & & 1 & 2 & 1 & & & & & \\ & 1 & 2 & 1 & & 2 & 4 & 2 & & 1 & 2 & 1 & & & & \\ 1 & & 1 & 2 & 2 & & 2 & 4 & 1 & & 1 & 2 & & & & \\ & & & & 2 & 1 & & 1 & 4 & 2 & & 2 & 2 & 1 & & 1 \\ & & & & 1 & 2 & 1 & & 2 & 4 & 2 & & 1 & 2 & 1 & \\ & & & & & 1 & 2 & 1 & & 2 & 4 & 2 & & 1 & 2 & 1 \\ & & & & 1 & & 1 & 2 & 2 & & 2 & 4 & 1 & & 1 & 2 \\ 2 & 1 & & 1 & & & & & 2 & 1 & & 1 & 4 & 2 & & 2 \\ 1 & 2 & 1 & & & & & & 1 & 2 & 1 & & 2 & 4 & 2 & \\ & 1 & 2 & 1 & & & & & & 1 & 2 & 1 & & 2 & 4 & 2 \\ 1 & & 1 & 2 & & & & & 1 & & 1 & 2 & 2 & & 2 & 4 \end{bmatrix} \end{array}$$

and

$$\mathbf{R} = \mathbf{KS}, \quad \mathbf{P} = \mathbf{SK}^T, \quad \mathbf{S} = \mathbf{S}^T$$

for the case of bilinear prolongation and full-weighted restriction operators and $N = 16$ grid points. We implement **SAS** using two subsequent stencil multiplications. Since **S** is a fixed stencil, i.e. its coefficients do not vary over the grid, it is sufficient to only store the coefficients once. The algorithm for stencil multiplication does not change due to this, we only remove the grid point indices for the fixed stencil. For a stencil $\hat{a}$ with radius $r_{\hat{a}} = 1$, **SA** produces a stencil with radius $r_{s\hat{a}} = 2$. In order to perform the second multiplication (**SA**)**S**, we need to store two ghost layers for **SA**. The result **SAS** has radius $r_{s\hat{a}s} = 3$. Cutting out the correct rows and columns at the end can be implemented simply by iterating over the stencil set with step size 2 in each dimension, i.e. iterating only the coarse grid points.

This approach has several disadvantages. First, the resulting stencil is wider than necessary. For the previous example, we end up with a stencil of width 7, only to cut it down to a stencil of width 3 again. This increases memory consumption, limiting the whole solver to smaller grids. Second, the additional ghost layers required for intermediate results harm multi-GPU performance, as they need to be communicated between the nodes/GPUs. Third, due to the high amount of redundant intermediate computation, this approach is tremendously slow compared to the solution that is presented next.

The first version iterating over input coefficients can be implemented as OpenCL kernel by parallelizing the loops over the grid points. The second version having reordered loops can be implemented similarly or by additionally parallelizing the loops over the output coefficients . If we parallelize in 6 dimensions, the index in dimension d can be calculated from the 1d global index using the following rule.

$$\text{idx}_d = \left\lfloor \text{1d-index} \Big/ \prod_{q=0}^{q<d} M_q \right\rfloor \mod M_d,$$

where M_q is the size in dimension q. An implementation of this scheme for a grid of size $m \times n \times o$ and stencils of width 3 is shown in listing 4.21, with increment order $k \to j \to i \to ck \to cj \to ci$, i.e. k varying fastest.

```
int idx = get_global_id(0);
int k = idx % o;
idx /= o;
int j = idx % n;
idx /= n;
int i = idx % m;
idx /= m;
int ck = idx % 3;
idx /= 3;
int cj = idx % 3;
idx /= 3;
int ci = idx % 3;
```

Listing 4.21: Calculation of the 6d index from a 1d index for a coefficient `(ci,cj,ck)` of a stencil located at grid point `(i,j,k)` with grid-first indexing

	Time [ms]		
	Quadro T2000	A100	
Kernel	64^3	64^3	128^3
`mult_stencils_fix_var`	39	4	31
`mult_stencils_var_fix`	106	11	87
`mult_stencils_fix_var_reordered`	17	1	6
`mult_stencils_var_fix_reordered`	56	5	37
`mult_stencils_fix_var_reordered_parallel_c`	23	2	16
`mult_stencils_var_fix_reordered_parallel_c`	47	5	45
`cut_stencils_w7_to_w3`	1	< 1	< 1
`update_ghosts_varying_stencil`	6	< 1	1

Table 4.28.: Durations of kernels participating in the Galerkin operator computation using stencil arithmetic. The three different versions are shown in the upper three sections of the table. Each version requires the respective two kernel calls plus the two kernel calls shown in the lowest section of the table (cut stencil and update ghosts).

Table 4.28 shows the durations for the three different implementations of Galerkin operator computation via stencil arithmetic on the Quadro T2000 and the A100. On both GPUs, the reordered version is faster than the naive version. Increasing TLP by parallelizing over output coefficients results in no significant change for the Quadro T2000, while it decreases performance on the A100.

4.7.3. Operating Directly on the Coarse Grid

In the following, we want to derive an algorithm, that allows us to directly compute the coarse grid operator coefficients without calculating a larger intermediate result and cutting it down afterwards.

We stay with the `4x4` grid example and consider the calculation of $\mathbf{C}(9, 1)$ as shown in figure 4.23. We observe, that the location of $\hat{a}$ does not change, and so does the point the coefficient of $\hat{b}$ maps to. Stencil $\hat{b}$ is shifted over those coefficients of stencil $\hat{a}$, such that the final grid point, that coefficients of $\hat{b}$ and $\hat{c}$ map to, can still be reached from the location of $\hat{b}$. The relevant coefficients of $\hat{a}$ for the current coefficient of $\hat{c}$ are given by the intersection S of grid points around $\text{idx}(i_{\hat{a}})$ with radius $r_{\hat{a}}$ and grid points around $\text{idx}(i_{\hat{c}} + \text{off})$ with radius $r_{\hat{b}}$. This is illustrated in figure 4.24. Note that this results in basically the same traversal as with the reordered loops variant of the previous attempt, but coming from a different angle of view.

In the construction of the coarse grid operator, we have two subsequent stencil multiplications. We can use the above concept to connect the two multiplications

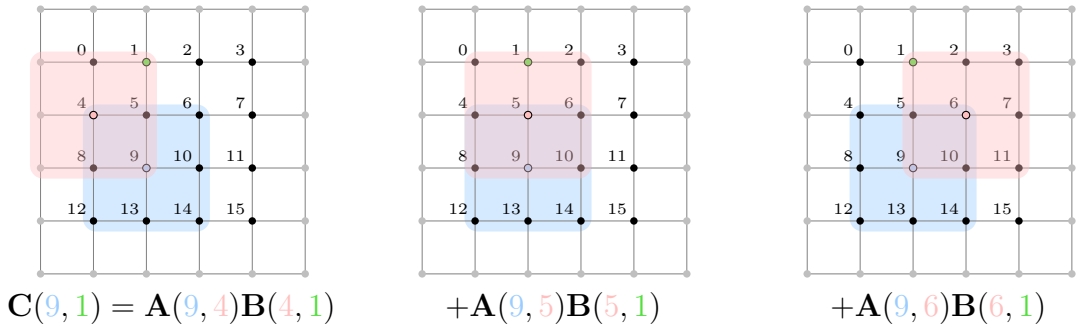
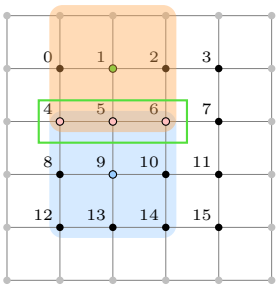
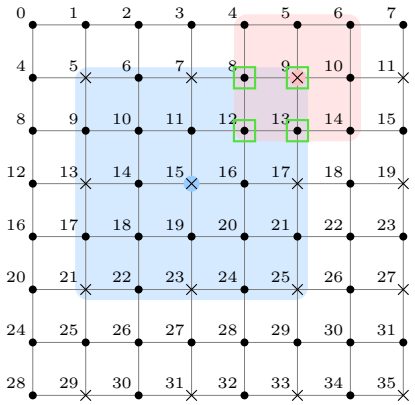


Figure 4.23.: Illustration of the computation of the entry $\mathbf{C}(9,1)$ as produced in the matrix-matrix multiplication $\mathbf{C} = \mathbf{AB}$ and the affected grid points. The blue point denotes the location of stencils $\hat{a}$ and $\hat{c}$ (i.e. rows of $\mathbf{A}$ and $\mathbf{C}$), the green point denotes the point the current coefficients of $\hat{b}$ and $\hat{c}$ map to (i.e. columns of $\mathbf{B}$ and $\mathbf{C}$) and the red point denotes the location of $\hat{b}$ (i.e row of $\mathbf{B}$). The blue region illustrates the area of influence of stencil $\hat{a}$, the red region of stencil $\hat{b}$, i.e. both stencils having a radius of 1.




```
foreach stencil C at grid point gp_c:
foreach coeff coeff_c in C mapping to grid
        point gp_c_coeff:
  find intersection S
  foreach gp_s in S:
    find coeff_a mapping to gp_s
    find coeff_b mapping to gp_c_coeff:
    C(gp_c, coeff_c) +=
        A(gp_c, coeff_a)
      * B(gp_s, coeff_b)
```

Figure 4.24.: Relevant grid points coefficients of $\hat{a}$ map to are highlighted with the green rectangle. They are given by the intersection of grid points in the blue and orange areas. The pseudo-code on the left shows the multiplication of two stencils using this scheme.




```
// c-point: coarse grid point
// f-point: fine grid point
foreach stencil RAP at c-point c_gp_RAP:
foreach coeff coeff_RAP in RAP:
  f_gp_RAP = f-point corresponding to c_gp_RAP
  f_gp_coeff_RAP = f-point mapped to by coeff_RAP
  intersection S = intersect(
    f-points around f_gp_RAP with radius=2,
    f-points around f_pg_coeff_RAP with radius=1),
  foreach f_gp_S in S
    find coeff_P mapping to f_gp_coeff_RAP
    find coeff_RA mapping to f_gp_S
    RAP(c_gp_RAP, coeff_RAP) +=
        RA(c_gp_RAP, coeff_RA)
      * P(f_gp_S, coeff_P)
```

Figure 4.25.: Calculation of (**RA**)**P** with known **RA**. The blue region denotes the area around $\hat{ra}$ with radius $r_{\hat{ra}}$, the rose region denotes the region around $\text{idx}(i_{\hat{ra}} + \text{off})$ with radius $r_{\hat{p}}$. The fine grid points in the intersection S are highlighted with green rectangles. The pseudo-code on the left shows how **RAP** is calculated with **RA** known.

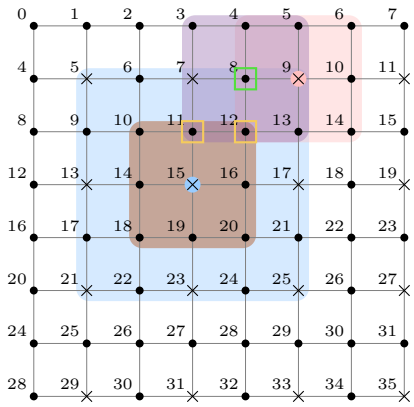
and write results onto the coarse grid immediately, instead of applying the multiplications separately as before. We begin from the end, i.e. from the calculation of (**RA**)**P**, assuming we already know **RA**. In the next step, we extend the algorithm with the calculation of **RA**.

For a fine grid operator $\mathbf{A}_h = \mathbf{A}$ consisting of stencils of width 3, the first product **RA** results in a stencil of width 5. **RA** only has stencils located at coarse grid points. For the multiplication (**RA**)**P**, we build the intersection of grid points surrounding **RA** with radius $r_{\hat{ra}}$ and grid points surrounding $\text{idx}(i_{\hat{ra}} + \text{off})$ with radius $r_{\hat{p}}$. The former is illustrated as the blue region in figure 4.25, the latter as the rose region. The affected fine grid points, which are in the intersection S, are highlighted with a green rectangle.

Now we want to incorporate the calculation of **RA**, for which we consider figure 4.26. For each grid point $\text{idx}(i_{\hat{s}}) \in S$ (green rectangle) we construct the intersection T (yellow rectangles) of grid points affected by $\hat{r}$ (brown region) and $\hat{a}$ (purple region), where $\text{idx}(i_{\hat{r}}) = \text{idx}(i_{\hat{ra}})$ and $\text{idx}(i_{\hat{a}}) = \text{idx}(i_{\hat{s}})$.

There is an optimization we can apply. The intermediate resulting set of stencils $\hat{\mathbf{RA}}$ consists of stencils located only at coarse grid points with coefficients affecting the fine grid points around it. Currently, we recalculate $\hat{ra}$ for each $\hat{s} \mid \text{idx}(i_{\hat{s}}) \in S$. However, $\hat{ra}$ does not change for one coefficient of $\hat{rap}$. Thus, we can pull the calculation of $\hat{ra}$ out of the loop over stencils located at grid points in S and thereby avoid redundant computation.

Another optimization opportunity lies in the realization, that the values of $\hat{a}$ change for different coefficients of $\hat{rap}$, but the "paths" taken through the the computation, i.e. grid points forming intersection results and coefficient indices, remain the




```
// c-point: coarse grid point
// f-point: fine grid point
foreach stencil RAP at c-point c_gp_RAP:
foreach coeff coeff_RAP in RAP:
  f_gp_RAP = f-point corresponding to c_gp_RAP
  f_gp_coeff_RAP = f-point mapped to by coeff_RAP
  intersection S = intersect(
    f-points around f_gp_RAP with radius=2 ,
    f-points around f_pg_coeff_RAP with radius=1) ,
  foreach f_gp_S in S
    find coeff_P mapping to f_gp_coeff_RAP

    // calculate RA
    intersection T = intersect(
      f-points around f_gp_RAP with radius=1 ),
      f-points around f_gp_S with radius=1 )
    RA_sum = 0
    foreach f_gp_T in T:
      coeff_R = from f_gp_RAP map to f_gp_T
      coeff_A = from f_gp_T map to f_gp_S
      RA_sum += R(f_gp_RAP, coeff_R)
              * A(f_gp_T, coeff_A)

    RAP(c_gp_RAP, coeff_RAP) +=
        RA_sum * P(f_gp_S, coeff_P)
```

Figure 4.26.: Calculation of (**RA**)**P** including the calculation of **RA**. The blue region denotes the area around $\hat{ra}$ with radius $r_{\hat{ra}}$, the rose region denotes the region around $\text{idx}(i_{\hat{ra}} + \text{off})$ with radius $r_{\hat{p}}$, the brown region denotes the area around $\hat{r}$ with radius $r_{\hat{r}}$, the purple region denotes the area around $\hat{a}$ with radius $r_{\hat{a}}$. The fine grid points in the intersection S are highlighted with green rectangles, those in the intersection T with yellow rectangles. The pseudo-code on the left shows how **RA** is calculated by iterating points of T and then used to calculate (**RA**)**P**.

same. We can therefore use the derived algorithm to generate code which directly performs the calculation for each coefficient of a stencil $\hat{rap}$. Iterating over each coarse grid point, which is parallelized in the OpenCL kernel, then completes the computation of the coarse grid operator.

Listing 4.22 shows helper functions written in OpenCL C that are used in the coarse grid calculation kernel. That is, we declare structs for defining an 1d interval and a point in 3d space. Using these, we can calculate the intersection of two 1d intervals, find the coefficient index that maps to a target grid point starting from a source grid point, find the grid point that is mapped to by adding an offset to a certain grid point index and lastly, a function to find the corresponding fine grid point for a given coarse grid point.

We implement the algorithm with and without caching of **RA** and test against the version using generated code for stencils with radius 1. The results are shown in table 4.29. Each kernel version launches one work-item per coarse grid point. In the caching variant the 125 values of **RA** are stored in private memory.

While the initial version is a significant improvement over the versions using stencil arithmetic from the previous section, caching **RA** further improves performance drastically for the versions building the paths taken through **RAP**, as, according

```
// Helper struct for galerkin. Defines an interval with integer start and end.
typedef struct Interval {
    int start;
    int end;
} Interval;

// Returns the intersection of two intervals or [-1,-1] if they don't overlap
Interval intersect(Interval a, Interval b) {
    Interval ret;
    // Check if intervals overlap
    if (a.start <= b.end && b.start <= a.end)     {
        // Calculate start and end points of intersection
        ret.start = (a.start > b.start) ? a.start : b.start;
        ret.end = (a.end < b.end) ? a.end : b.end;
        return ret;
    } else {
        // Intervals do not overlap
        ret.start = -1;
        ret.end = -1;
        return ret;
    }
};

// Helper struct for galerkin. Defines a grid point with integer 3d coordinates.
typedef struct Point {
    int x;
    int y;
    int z;
} Point;

// Returns the stencil entry indices of the stencil sitting at locationOfStencil that maps to mapsTo.
// No check is done, if the mapping is possible, i.e. the returned value might be outside of range [0,2].
// The result is just the difference of the indices plus one, since the stencil entry indices start at 0
// and not at -1.
Point stencilEntryThatMapsTo(Point locationOfStencil, Point mapsTo) {
    Point ret;
    ret.x = mapsTo.x - locationOfStencil.x + 1;
    ret.y = mapsTo.y - locationOfStencil.y + 1;
    ret.z = mapsTo.z - locationOfStencil.z + 1;
    return ret;
};

// Returns the grid point indices that is mapped to by the stencil entry of another point.
// stencilEntry must be 0-based, hence the substraction by 1.
Point pointMappedToByStencilEntry(Point locationOfStencil, Point stencilEntry) {
    Point ret;
    ret.x = locationOfStencil.x + (stencilEntry.x - 1);
    ret.y = locationOfStencil.y + (stencilEntry.y - 1);
    ret.z = locationOfStencil.z + (stencilEntry.z - 1);
    return ret;
};

// Returns the point on the fine grid that is related to the coarse grid point, respecting ghost cells.
Point coarseToFine(Point p, int ghc, int ghf) {
    Point ret;
    ret.x = (p.x - ghc) * 2 + 1 + ghf;
    ret.y = (p.y - ghc) * 2 + 1 + ghf;
    ret.z = (p.z - ghc) * 2 + 1 + ghf;
    return ret;
};
```

Listing 4.22: OpenCL helper functions for coarse grid operator calculation.

	Time [ms]							
	Quadro T2000				A100			
Grid size	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
64^3	8.16	3.65	1.58	1.55	1.00	0.43	0.16	0.14
128^3	60.08	27.38	11.34	11.18	4.89	2.35	0.88	0.79
256^3	–	–	–	–	33.75	14.52	5.95	5.34

Table 4.29.: Duration of kernel variants calculating the coarse grid operator using the derived algorithm. Each variant is called with one work-item per coarse grid point. (a) default, (b) default with cached **RA**, (c) generated code, (d) generated code with cached **RA**.

to NCU, only about 50 % of the overall instructions are executed for 64^3 (37M vs. 74M). The code generation then further saves time by reducing the overall number of operations (only 3.5M instructions executed for 64^3). Caching **RA** only yields a small improvement in performance for the generated version. A big difference between the versions using stencil arithmetic and our derived algorithm lies in the ability to consume less memory, meaning we can solve on larger grids, and the avoidance of the necessity to store multiple layers of ghosts.

4.8. Miscellaneous

4.8.1. V-cycle as a Whole

All parts that form the v-cycle can be viewed as isolated units that need to be run in a specific order. Thus, there are not many optimization opportunities affecting the code combining those parts.

One potential optimization is to define which levels of the solver to run on the GPU and which on the CPU. As suggested by Sakharnykh in [95], comutations that happen on coarse grids might benefit from running sequentially on the CPU, as GPUs excel at large workloads but the coarse grids do not expose enough parallelism to saturate the device. However, in his study the f-cycle scheme is employed, which naturally visits coarse grids more often, leveraging the benefit of optimizing coarse grid computations. Since the majority of time is spent on fine grids in the v-cycle scheme we use, and differentiation in GPU and CPU levels would require additional memory transfers between host and device, we do not expect a significant improvement by using this hybrid approach. In fact, as we see in table 4.30, we see only marginal effects of calculating coarser grids using the CPU for large problems.

	Time [ms]			
	Quadro T2000		A100	
`max_lv_ocl`	64^3	128^3	128^3	256^3
0	250	1.663	932	12.519
1	85	499	209	1.555
2	68	372	122	427
3	66	353	110	291
4	67	352	110	282
5	68	353	112	282
6	68	354	114	284
7		353	114	285
8				286

Table 4.30.: Durations for 10 v-cycle iterations, $\nu_1 = \nu_2 = 3, \omega = 0.8$, with varying maximum level `max_lv_ocl` for which the GPU is used.

Another important aspect of high performance GPU code is the choice of launch configuration and potentially tuning. While `mgcl` does not provide an automatic tuning interface, an API for configuring the work-group sizes is exposed to the user on a per-kernel basis. Together with the set of benchmarks that are used to produce the results in this thesis, users should be able to run quick tests to find sufficiently near-optimal configurations for their systems. Employing a system that allows to choose between different kernel variants, i.e. using different layouts that lead to different levels of occupancy, ILP and TLP, might be worthwhile to be considered in future work.

4.8.2. OpenCL Setup

There are multiple things to consider regarding the setup of the OpenCL environment.

OpenCL kernels can be compiled just-in-time or be shipped as pre-built binaries. The former has the advantage, that the compiler can make optimization decisions based on the hardware present in the system. On the other hand, the full kernel code must be disclosed and be parsed. Using pre-built binaries allows to hide the kernel implementation details and reduces compilation time. However, at least Nvidia provides a caching mechanism in their backend implementation that renders the usage of binaries superfluous for subsequent executions having the kernel code unchanged.

When compiling the kernel code from source code, we can read the code either from a file that is stored on the disk, or by placing all kernel code in a constant

string that is included in the program binary at compile time. The latter can easily be realized using standard CMake features and is the default in `mgcl`. However, since reading the kernel code from a file allows to adjust the code without the need of recompiling the host code itself, and to have multiple versions of kernels, `mgcl` additionally offers a function to enable loading kernel code from an external file. This is especially useful in testing and benchmark scenarios.

Chapter 5

Multi-GPU Optimizations

In the previous chapter, we went through the implementation of the multigrid method for a single GPU. In this chapter we want to focus on optimizing the performance of the implementation for multiple GPUs.

`mgcl` enables multi-GPU usage through MPI. Each MPI process uses a single GPU. In order to ensure the individual processes pick different devices present in the same system, `mgcl` scans the OpenCL platform for all available GPUs and circularly assigns the devices to the processes.

There are two types of parallel scaling: Strong scaling and weak scaling. With strong scaling, a problem with fixed global size is divided into P smaller sub-problems, where P is the number of MPI processes. As scaling progresses, the efficiency eventually reaches a natural plateau once sub-problems become too small to fully utilize a GPU. Moreover, the overhead introduced when transitioning from a single- to a multi-GPU setup is substantial, such that strong scaling beyond this point is generally not worthwhile. With weak scaling on the other hand, one process always keeps the same local grid size, but the global size scales with increasing P. This approach not only solves the scaling issues but also simplifies problem decomposition, as it naturally allows retaining a problem size that is a power of two.

Luckily, in our use case of the geometric multigrid method, we only need to consider local operators apart from the optional global sum reduction, making strong scaling obsolete. If a problem is small enough to fit on a single GPU, `mgcl` should be called with only one process. If the target global problem size exceeds what a single GPU can handle, we can easily choose an appropriate local problem size and scale it to a larger global size.

`mgcl` makes no assumptions about the system topology. Using the same MPI communicator object, a user can specify the dimensionality and periodicity. `mgcl`

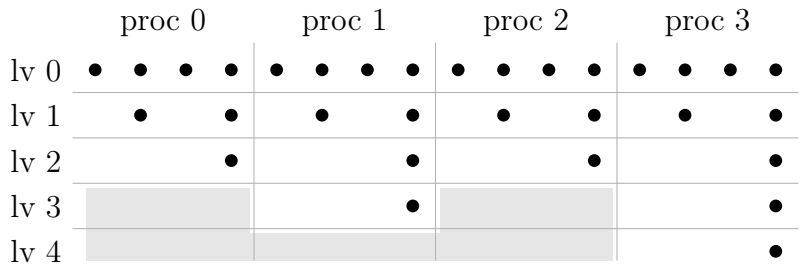


Figure 5.1.: Grid hierarchy using four processes, $N_{\text{local}}^{(0)} = 4$ and $N_{\text{global}}^{(0)} = 16$. Starting from level 3, not every process is active anymore.

then uses this information, in combination with local and global problem sizes, to determine the adjacency of the processes.

There are mainly three parts of `mgcl` that are affected by the introduction of MPI. These are

- the initialization of data structures and grid hierarchy,
- the computation of the coarse grid operator and
- the ghost update.

5.1. Grid Hierarchy

The grid hierarchy in an MPI setup is illustrated in figure 5.1. Beyond a certain threshold level, some processes are left with no grid points in their local grids. This needs to be accounted for during the data structure creation in the setup phase.

This leads to an optimization opportunity. Instead of using the same number of processes throughout the entire v-cycle, we can choose a threshold level from which the remaining problem is agglomerated onto a single process, as illustrated in figure 5.2. This requires a level object to be aware of whether it should be computed locally or distributively. Additionally, the v-cycle implementation must be extended to include the gathering of $\mathbf{f}^{(\text{threshold}+1)}$ on the way up after restricting $\mathbf{f}$ from threshold to threshold $+ 1$, and the scattering of $\mathbf{v}^{(\text{threshold}+1)}$ on the way down before prolongating $\mathbf{v}$ from threshold $+ 1$ to threshold. Lastly, care must be taken when calculating the coarse grid operator, which is described in the upcoming section. We refer to this scheme as *monadic agglomeration*.

An alternative to monadic agglomeration, which limits scalability, is to give opportunity to gather data dynamically multiple times onto multiple processes. One possibility to realize this is by introducing two parameters, $N_{\min}$ and $N_{\max}$,

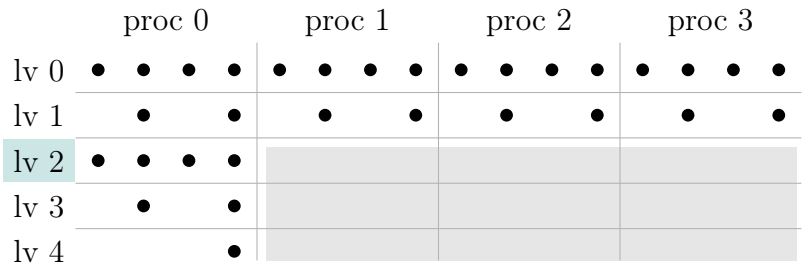


Figure 5.2.: Grid hierarchy using four processes, $N^{(0)}_{\text{local}} = 4$ and $N^{(0)}_{\text{global}} = 16$. On level 2, the grid points are agglomerated onto process 0.

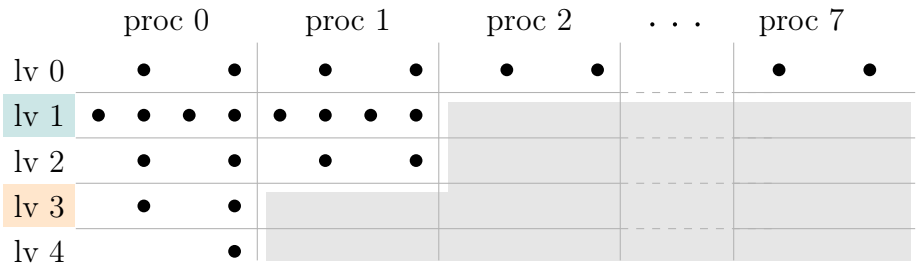


Figure 5.3.: Grid hierarchy using eight processes, $N_{\text{min}} = 2$, $N_{\text{max}} = 4$ and $N^{(0)}_{\text{global}} = 16$. On level 1, the grid points are agglomerated onto processes 0 and 1. On level 3, the remaining grid points are agglomerated onto process 0.

which define the minimum and maximum number of grid points that must be present in a local grid. Of course, N_{min} can be undershot if only one process remains. Consider the example in figure 5.3. For a global grid size of $N^{(0)}_{\text{global}} = 16$ and a local grid size of $N^{(0)}_{\text{local}} = 2$, i.e. $P = 8$, and $N_{\text{min}} = 2$, $N_{\text{max}} = 4$, there are two agglomeration points. The first occurs on level 1, since $N^{(0)}_{\text{local}} = N_{\text{min}}$ on level 0, but still $N_{\text{max}} < N^{(0)}_{\text{global}}$, hence we gather onto two processes. The second occurs on level 3, since again $N^{(2)}_{\text{local}} = N_{\text{min}}$ on level 2, but this time $N_{\text{max}} \geq N^{(2)}_{\text{global}}$, thus we can compute the remainder of the global problem on one process. While the implementation effort should be reasonable, it is beyond the scope of this thesis.

5.2. Coarse Grid Operator Computation

While the kernel of the coarse grid operator computation remains unchanged in the multi-GPU case, the control flow around the kernel call differs. More precisely, we need to take the MPI level threshold into account, i.e. the level, at which the coarse grid operator computation is agglomerated onto a single process. The control flow is shown in algorithm 1. For each level l, we calculate the coarse grid

operator $\hat{\mathbf{A}}^{(l)}$ based on the operator of the previous level $\hat{\mathbf{A}}^{(l-1)}$. If the current level is at the threshold, its size must be based on the global problem size instead of the local size. Furthermore, if $l >$ threshold, we only want to calculate the coarse grid operator on the root process (i.e. the process with rank 0). If $l =$ threshold, we gather the partial coarse grid operators from processes with rank > 0 into the globally sized coarse grid operator on the root process. Lastly, we need to differentiate between a ghost update across multiple MPI processes and a local ghost update in the case of periodic boundary conditions.

Algorithm 1: Control flow of calculating the coarse grid operator in the multi-GPU case

```
   Data: maxlv, maxdim, threshold
   Result: Â^(l) for all l > 0
 1 for l ← 1 to maxlv do
 2     for d ← 0 to maxdim−1 do
 3         if rank = 0 and l ≥ threshold then
 4             N_d(Â^(l)) ← N^(0)_global,d · 2^−l;
 5         else
 6             N_d(Â^(l)) ← N^(l)_local,d;
 7     if rank = 0 or l ≤ threshold then
 8         calculate Â^(l);
 9     if l = threshold then
10         gather onto rank 0;
11     if rank = 0 or (rank > 0 and l < threshold) then
12         if l > threshold then
13             update ghosts of Â^(l) locally;
14         else
15             update ghosts of Â^(l) across processes using MPI;
```

Since the sizes of $\hat{\mathbf{A}}^{(\text{threshold})}$ differ between process 0 and the others, we need to construct the gathering routine carefully. Using `MPI_Gatherv` we can define new MPI datatypes to represent n-dimensional arrays. Due to the mentioned difference in size, the underlying memory layout is different, which leads to the need of instantiating two different datatypes for the data to be sent and the data to be received. Furthermore, the displacements parameter must be used to prevent ghost layers of local grids from overlapping other data.

5.3. Ghost Update

The ghost update has the greatest impact on solver performance when comparing the single-GPU and multi-GPU cases. This is due to the need of communication between processes, which requires data to be sent from the device to the host, potentially over the network to another node, and back to the device again. Since we are using OpenCL, we cannot utilize CUDA-aware MPI, which enables direct GPU-to-GPU communication, assuming the correct hardware configuration.

The modification of the sequential ghost update is straightforward. For each direction, we extract slices of real grid points with the width of gh into a temporary send buffer and send it to other processes using `MPI_Sendrecv`. Data is received in another temporary buffer and pasted into the corresponding ghost layers. We restrict the number of ghost layers to $gh \leq N_{\min}$, since otherwise we would need to send data to multiple neighboring processes. This would increase the complexity of both the ghost update and the neighbor discovery without providing a reasonable benefit, as we agglomerate onto fewer processes for coarse grids.

A simple way of updating ghosts of a device buffer is to copy the entire grid from the device to the host, update the ghosts using the sequential routine and copy the result back to the device. With this approach however, the durations of memory transfers between host and device clearly dominate the performance of the ghost update. An improved approach is to extract the border planes of the real grid on the device and transfer only those between host and device. For example for a grid of size 128^3 with $gh = 1$, only $6 \cdot 130^2/130^3 \approx 4.6\,\%$ of data needs to be transferred compared to the naive variant. This method is shown e.g. in [87].

There are multiple ways to extract the border planes and choose a layout. Analogously to iterating the grid from the outermost to innermost dimension, we could keep all the ghost cells and store them in the order, the grid points are visited. While this way, only the relevant points are stored, indexing between the grid and the extraction buffer is complicated. More critically, the data that is to be sent to one neighbor is not stored contiguously, which would lead to excessive small data transfers. Another option is to take care of all directions individually, which would solve the problem of non-contiguous data and reduce mapping complexity. The drawback is that redundant data needs to be stored, and custom data copy operations need to be performed on the host after receiving data in each dimension.

Considering the example using a 128^3 grid with $gh = 1$ again, the first approach would require to extract and store $130^3 - 128^3 = 99,848$ data elements. The second approach would need to store $2 \cdot gh_m \cdot yz_{gh} + 2 \cdot gh_n \cdot xz_{gh} + 2 \cdot gh_o \cdot xy_{gh} = 6 \cdot 130^2 = 101,400$ data elements, where $yz_{gh} = n_{gh} o_{gh}$, $xz_{gh} = m_{gh} o_{gh}$ and $xy_{gh} = m_{gh} n_{gh}$. That is an increase of about $3\,\%$. Since the overhead of redundant data is still

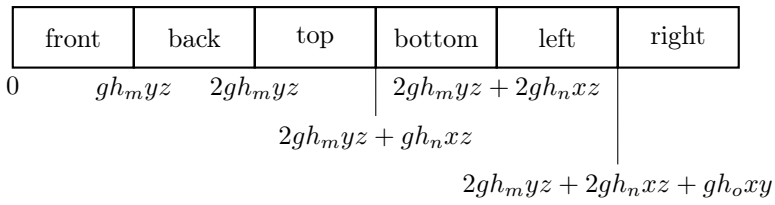


Figure 5.4.: Memory layout of extracted border planes. Grid extents are m, n, o in x,y,z-directions, respectively.

very low compared to the size of the grid and due to the advantages mentioned, we choose to implement the second approach.

In our implementation, the extracted border planes are stored in the following order: front, back, top, bottom, left, right. This is illustrated in figure 5.4. We can launch a kernel with one work-item per element that gets extracted and filter by the 1d work-item index. That is, if `idx(wi)` $< gh_myz$, we extract values from the real planes in the front, else if `idx(wi)` $< 2gh_myz$, we extract values from the real planes in the back, and so on.

Pasting the received values into ghost points works the same way, albeit the exclusion of corners when pasting in the x- and y-directions, as those values are not updated in the receive buffers. The reason of this is explained in the following.

When the border planes are extracted and sent to the host, we can send the front and back data to the corresponding neighbors. This leads to the ghost layers directly adjacent to the x-faces of the real grid to be updated. However, the diagonally adjacent points, i.e. edges and corners, do not hold the correct value yet. This is similar to the scenario in the sequential ghost update in section 4.4, after shifting data in the first dimension. We want to mimic the behavior of successively filling the diagonally adjacent ghost points by shifting data in each direction. Unlike the sequential ghost update, we have separate segments for the different directions, hence we need to copy real edges of the received yz-plane values into the corresponding edges of ghost layers, that are about to be sent to the top and the bottom processes. In the same fashion, we copy cuboid surfaces of the real grid from the xz-planes receive buffer into the send buffer for sending data to the left and the right. Using this scheme, the receive buffer for xy-planes holds the correct values for corners and edges, and only this buffer should be used to paste the values into the respective ghost points on the device.

The ghost update routine for the set of coefficients $\hat{\mathbf{A}}$ works in the same way. The difference is that the border planes are stored contiguously for each coefficient, i.e. the extracted data ranges in 1d are `sv[0..w-1][ghm..2*ghm-1]` for the left real points and `sv[0..w-1][m..m+ghm-1]` for the right real points, with `w` being the width of the stencil. This can be easily extended to higher dimensions.

	Time [ms]					
	$gh = 1$		$gh = 2$		$gh = 3$	
	naive	impr	naive	impr	naive	impr
8^3	0.05	0.06	0.06	0.08	0.08	0.10
16^3	0.11	0.08	0.14	0.11	0.19	0.19
32^3	0.51	0.14	0.61	0.27	0.73	0.40
64^3	1.63	0.40	3.42	0.74	2.07	1.14
128^3	11.67	0.63	12.27	1.29	13.37	1.87

Table 5.1.: Best durations of ghost update copying the whole grid from device to host and back (naive) and transferring border planes only (impr) on 4 MPI processes each using one A100 GPU

In contrast to single-GPU, using Dirichlet boundary conditions now requires the update of ghost cells at the locations where the global grid is divided into sub-grids.

Table 5.1 clearly shows the benefit of transferring only the border planes instead of the entire grid on the Pleiades cluster using four MPI processes on the same node, each using one A100. The optimization is more impactful for larger grids, which is expected, as the ratio of ghost cells to real cells shrinks with increasing grid size.

Table 5.2 shows how much time the individual parts of both variants take on a grid of size 128^3 with $gh = 1$ and four MPI processes on the same node. These results are reported by Nsight Systems (NSYS). We can clearly see that the memory transfer operations dominate the overall duration of the naive approach. For the improved version, the MPI transfer routines consumes most of the time, closely followed by the data transfer from device to host. The durations of the extract and paste kernels are negligible in comparison. For the naive version, there is a large discrepancy between the sum of the durations of the individual MPI routines and the total duration of inter-process communication. This is probably due to the way the sequential ghost update is implemented: Before sending data in each direction, a new send buffer is created by slicing the grid, which requires memory allocation. This is not the case for the improved version, as we extract the border planes in a format that can directly be used for the send buffer. Note that the exact durations, especially those of the memory transfer operations and the MPI transfer operations, vary from run to run. Nevertheless, it gives an impression of the relative performance of the individual parts.

The previous attempt copies the whole planes buffer from device to host, sends it to neighboring MPI processes, and finally copies the buffer back to the device. Instead of performing the data transfers in a monolithic fashion, we can interleave the transfers between device and host and the ones between processes using MPI.

	Time [µs]	
Part	naive	impr
`paste_ghosts_from_border_planes`	–	6
`extract_border_planes`	–	6
memcpy DtoH	10137	195
memcpy HtoD	2426	96
MPI_Sendrecv Total	389	384
Total inter-process transfer time	995	390

Table 5.2.: Durations of the individual parts of ghost update copying the whole grid from device to host and back (naive) and transferring border planes only (impr) on 4 MPI processes each using one A100 GPU. Grid size is 128^3 with $gh = 1$.

		Time [ms]					
		$gh = 1$		$gh = 2$		$gh = 3$	
Grid	GPUs	mon	ilv	mon	ilv	mon	ilv
32^3	2	0.09	0.14	0.13	0.17	0.17	0.26
64^3	2	0.17	0.26	0.26	0.36	0.39	0.45
128^3	2	0.43	0.50	1.45	1.38	2.14	1.96
256^3	2	2.61	2.33	5.10	4.61	4.26	4.25
32^3	4	0.19	0.21	0.33	0.30	0.49	0.49
64^3	4	0.47	0.50	0.89	0.87	1.33	1.25
128^3	4	0.73	0.70	0.98	1.00	1.48	1.35
256^3	4	1.57	1.31	2.75	2.58	3.99	3.21

Table 5.3.: Durations of the ghost update copying the planes buffer between host and device as a whole (mon) and interleaving data transfer between host and device with MPI transfers (ilv). The test used one A100 GPU per MPI process on the same node.

That is, we enqueue three read operations for the individual dimensions. When the first one is finished, we can send the data via MPI in x-direction and enqueue the data transfer of the affected portion back to the device. Repeating this for the remaining dimensions completes the ghost update. Note that there are still dependencies between the dimensions for sending the data via MPI, as edges and corners of intermediate results must be updated. Thus we can enqueue the data transfers between host and device, but the MPI transfers must still be executed sequentially. Table 5.3 shows the results for testing this scheme against the monolithic version. For larger grids, interleaving the transfers is an improvement in performance. For smaller grids, the monolithic version performs better, likely due to the overhead compared to the amount of data transferred.

5.4. Multiple Jacobi Iterations Without Intermediate Ghost Update Revisited

Since Jacobi's method is affected by the ghost update to a significant extent, we want to revisit it for the multi-GPU case. While performing multiple iterations without intermediate ghost update did not show any significant improvement in the single-GPU case, the ghost update is much more expensive in the multi-GPU case, making the ability to reduce the amounts of performed ghost updates potentially valuable.

We run the same test as in the single-GPU case, but with two MPI processes on the same node and a 64^3 grid. One entire run of Jacobi's method performing ν iterations consists of two ghost updates of v (initial and final ghost update), ν kernel calls for the residual calculation and update step and $\lceil \nu / \texttt{spi} \rceil - 1$ ghost updates between iterations, where `spi` is the number of iterations without intermediate ghost update.

According to NSYS, one run from extracting the border planes of the initial ghost update until pasting into ghost planes of the final ghost update takes 588 µs for spi = 1 and 749 µs for spi = 2. Figure 5.5 shows the procedures in NSYS. Table 5.4 shows the total durations of the individual parts summed over the entire duration of two Jacobi iterations. While the differences in kernel durations are negligible, there are substantial differences in the memory transfer operations between host and device. Notably, the CUDA API call to `cuMemcpyDtoHAsync_v2` consumes considerably more time than the referring data transfer `memcpy DtoH` itself, indicating a large overhead. Since the implementation is closed-source, we cannot ultimately determine the cause of this. One explanation could be that pageable memory must first be staged to pinned memory[1] [35]. To test this further, we can try to allocate pinned memory using the flag `CL_MEM_ALLOC_HOST_PTR` upon buffer creation and map this memory to host memory using `clEnqueueMapBuffer`, as described in section 3.1.1 in [83]. However, there is no guarantee that the allocated memory is actually pinned memory, and indeed, NSYS reveals that the target of the memory copy operations are pageable memory locations in our test run. Another attempt might be to port the code to CUDA and use `cudaMallocHost` to allocate pinned memory, but which is out of the scope of this work.

The discrepancy in the durations of the API calls between spi = 1 and spi = 2 can be explained by the difference in size that needs to be transferred: With spi = 1, $66^2 \cdot 6 \cdot 8 = 221,952$ bytes need to be transferred, while with spi = 2 it is $68^2 \cdot 12 \cdot 8 = 443,904$ bytes.

[1] `https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html#api-sync-behavior`

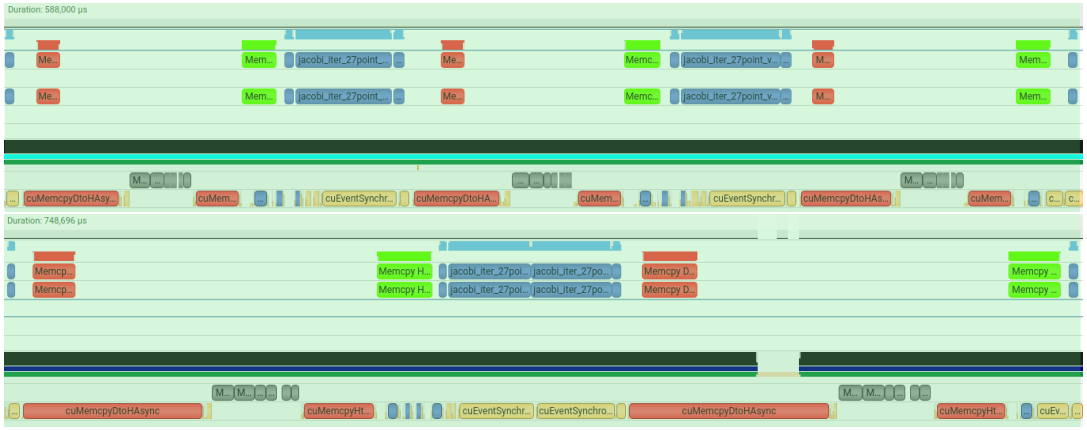


Figure 5.5.: Nsight System's view of one run of Jacobi's method with $\nu = 2$ iterations and spi $= 1$ (top) and spi $= 2$ (bottom) on a 64^3 grid.

	Time [µs]	
spi	1	2
`jacobi_iter`	106	111
`extract_border_planes`	14	11
`paste_ghosts_from_border_planes`	13	10
memcpy HtoD	56	73
memcpy DtoH	35	65
`cuMemcpyDtoHAsync_v2`	149	264
`cuMemcpyHtoDAsync_v2`	72	97
`MPI_Sendrecv`	86	109

Table 5.4.: Total durations of the individual parts of two Jacobi iterations on 2 MPI processes with spi $= 1$ and spi $= 2$. Each process uses one A100 GPU. Grid size is 64^3.

5.5. Overlapped Jacobi's Method

Another approach to optimizing the implementation of Jacobi's method, particularly in the multi-GPU scenario, is to overlap the computation of the inner region with the computation of boundary regions and the corresponding ghost update. This technique is described in multiple sources, e.g. [102, 87, 70, 58, 69], and is not exclusive to programs using GPUs. The motivation is that the inner region can be computed independently of the data from neighboring processes. Of course, for this method to be worthwhile, the overhead of splitting the calculation into two kernel executions must be significantly smaller than the actual workload.

This approach can be realized using multiple command queues in OpenCL (or streams in CUDA). One queue is responsible for the calculation of inner grid points, the other queue takes care of the boundary regions and ghost update. Both streams must be synchronized afterwards.

There are multiple fine-grained options for the implementation, as shown in figure 5.6. In version (a), we enqueue the `boundary` and `inner` kernels right after another. This leads to both kernels being executed concurrently on the device. As we can see, the `inner` kernel is already finished before the data transfer to the host starts to update the ghost points. In version (b), we delay the enqueueing of the `inner` kernel until after the kernel for extracting the border planes is enqueued. Although complying to this order, the device schedules the `inner` kernel to start earlier than the `extract` kernel, taking away resources for executing the latter. What we actually want is to overlap the `inner` kernel with memory transfer between host and device and/or between processes, as those are the expensive parts in the multi-GPU case. We can achieve this in version (c) by waiting for the `extract` kernel to finish and then enqueue the `inner` kernel. While this adds a small gap between the finish of the `extract` kernel and the start of the `inner` kernel and ghost update, the `inner` kernel now truly overlaps the memory transfers.

Figure 5.7 shows the durations for the different variants using two A100 GPUs. While the default version is fastest for small grids, there is a noticeable difference between version (c) and the other versions. We achieve the best performance when the `inner` kernel overlaps the memory transfer operations. Versions (a) and (b) perform worse than the default version. The full result set can be found in the appendix in tables A.18 and A.19.

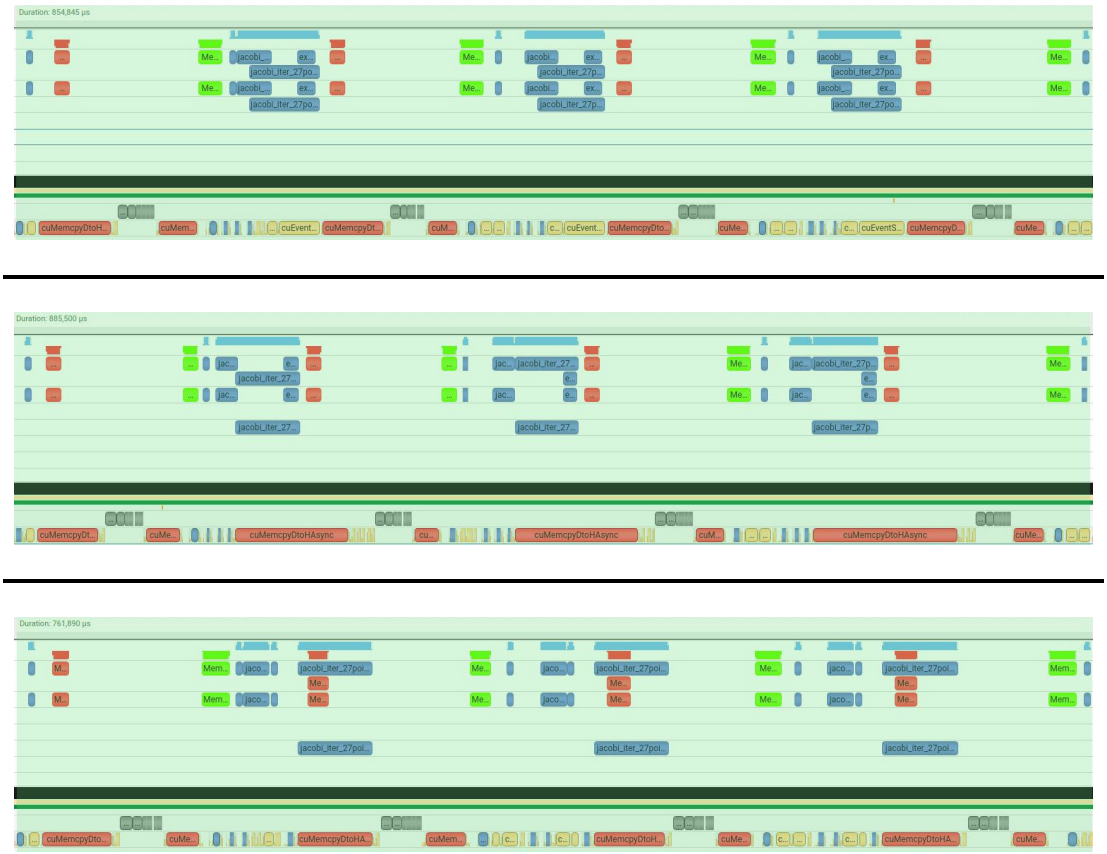


Figure 5.6.: Timelines in Nsight Systems for different implementations of overlapped Jacobi's method for 3 iterations. Executed on two A100 GPUs for a grid of size 64^3. Version (a) (top), (b) (middle) and (c) (bottom).

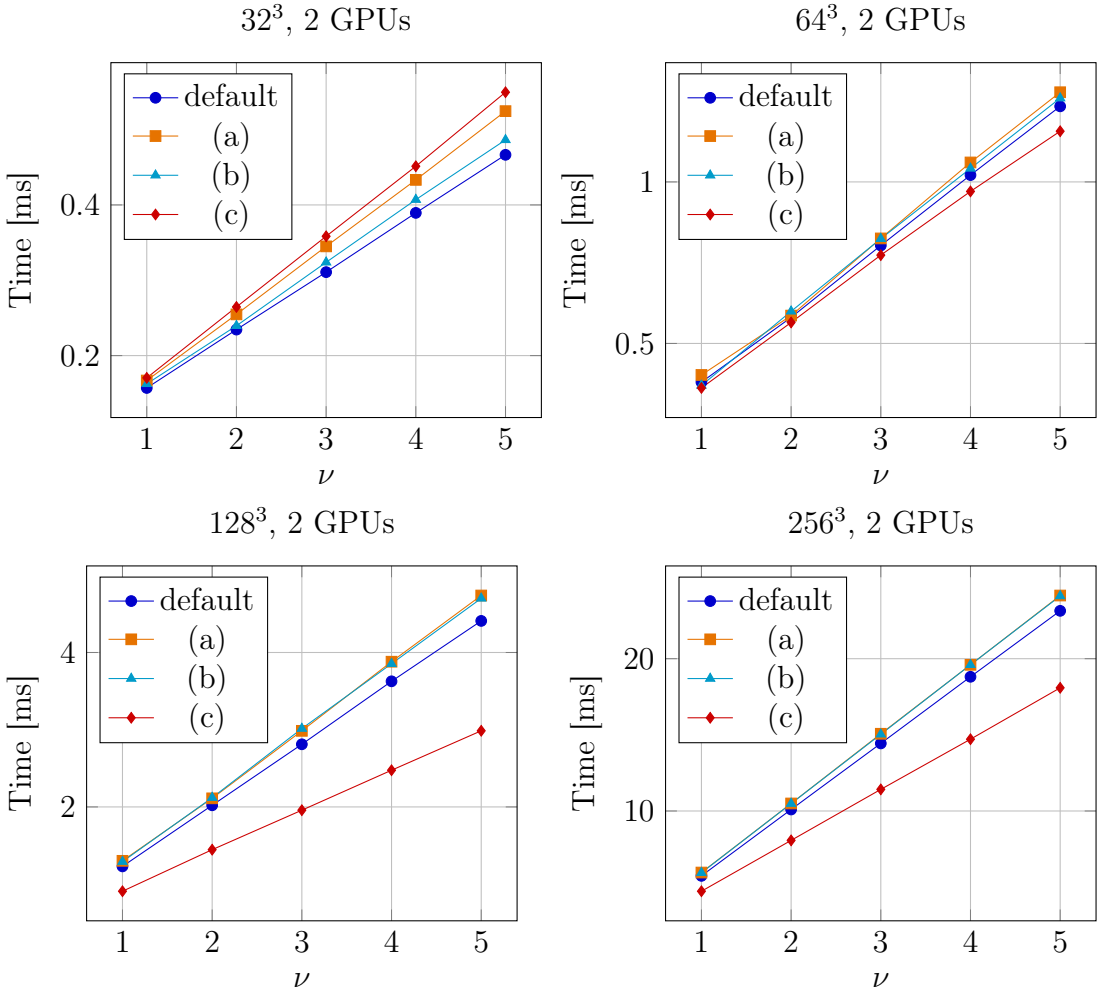


Figure 5.7.: Performance of the different implementation versions of overlapped Jacobi's method vs. non-overlapped version. Executed on two A100 GPUs on the same node.

Chapter 6

Blockstencil Implementation

In this chapter, we cover the changes in the implementation for vector-valued problems. Generally, scalar quantities like the unknowns v_i, right-hand side f_i and the residual r_i are replaced by vectors of size B. Consequently, the coefficients of the stencils $\hat{\mathbf{A}}$ are no longer scalars but matrices of size $B \times B$ instead. We refer to B as the *blocksize*. The application of a stencil onto a grid point then is a sequence of matrix-vector multiplications instead of a sequence of scalar multiplications.

Instead of storing grid quantities as 3d cuboids, we store a 4d tensor. We can choose between two viable layouts: Grid-first, i.e. `[gpx][gpy][gpz][b]`, and vector-first, i.e. `[b][gpx][gpy][gpz]`. Likewise, we can choose between two layouts for storing $\hat{\mathbf{A}}$, while keeping the grid indices in the innermost dimensions: Coefficients-first (`[cx][cy][cz][bx][by][gpx][gpy][gpz]`) and block-first (`[bx][by][cz][cx][cy][gpx][gpy][gpz]`).

6.1. Jacobi's Method and Residual Calculation Using Blockstencils

The residual calculation does not change much. We insert two loops for iterating the coefficient matrices, accumulate the result in the inner loop, and write it to r at the end of every iteration of the outer loop, as shown in listing 6.1. The kernel itself is still launched with one work-item per grid point. The `BLOCKSIZE` constant is set using a preprocessor directive when compiling the OpenCL kernels.

Table 6.1 shows the kernel durations using the different layouts for the coefficients and the grid quantities. As long as the grid point indices are the innermost dimensions, using one layout over the other for the coefficients does not make a difference. However, using the block-first layout for the grid quantities outperforms

```
// residual_27point_blockstencil_block_first_v_gp_first
int idx_block = 0;
for (int bi = 0; bi < BLOCKSIZE; bi++) {
    double stencilsum = 0;
    for (int bj = 0; bj < BLOCKSIZE; bj++) {
        stencilsum += stencilValues[index_sv_gp + (9 + 3 + 1) * svGridSize + idx_block] * v_in[index + bj]
            + stencilValues[index_sv_gp + (9 + 3) * svGridSize + idx_block] * v_in[index - koff + bj]
            + ...
        idx_block += svGridSizeCoeffs; // increase by gridsize to get to next matrix entry
    }
    r[index + bi] = f[index + bi] - stencilsum;
}
```

Listing 6.1: Residual calculation using block-first layout for the blockstencil and grid-first layout for the grid quantities.

	Time [µs]							
	Quadro T2000				A100			
	8^3	16^3	32^3	64^3	8^3	16^3	32^3	64^3
`block_first_v_block_first`	70	92	551	4.154	65	67	84	469
`block_first_v_gp_first`	39	76	563	4.546	42	45	66	503
`coeffs_first_v_block_first`	38	75	563	4.520	43	45	67	504
`coeffs_first_v_gp_first`	72	92	552	4.121	63	67	82	466

Table 6.1.: Kernel durations for the blockstencil residual layout using different layouts for the coefficients and the grid quantities, $B = 3$.

the alternative on both GPUs. For small grids, grid-first is faster. This trend can also be seen in figure 6.1, which shows the performance of the residual kernel for different blocksizes. The difference between block-first and grid-first for the grid quantities diverges as B increases.

The performance difference can be explained by the memory access pattern. In the grid-first case, the vector entries of the same grid point are stored consecutively in memory, which hinders coalescing since one work-item is mapped to one grid point. In the block-first case, neighboring work-items access neighboring memory locations because the same vector entry is stored consecutively for different grid points. This is confirmed by NCU: The ratio of sectors to requests is 8 for the block-first case and 15.4 for the grid-first case for a grid of size 64^3 with $B = 3$. Using the grid-first layout results in a strided memory access pattern with a stride of B, as shown in figure 6.2.

The update step of Jacobi's method requires the inverse of the center coefficient per grid point. In the scalar case, this is just the reciprocal of the scalar element on the main diagonal of $\mathbf{A}$. In the blockstencil case, the main diagonal consists of matrices, giving us multiple options for the update step. We could still multiply the residual with the reciprocal on the main diagonal of the center coefficient matrix (we denote this as scalar Jacobi, also known as point-wise Jacobi), or

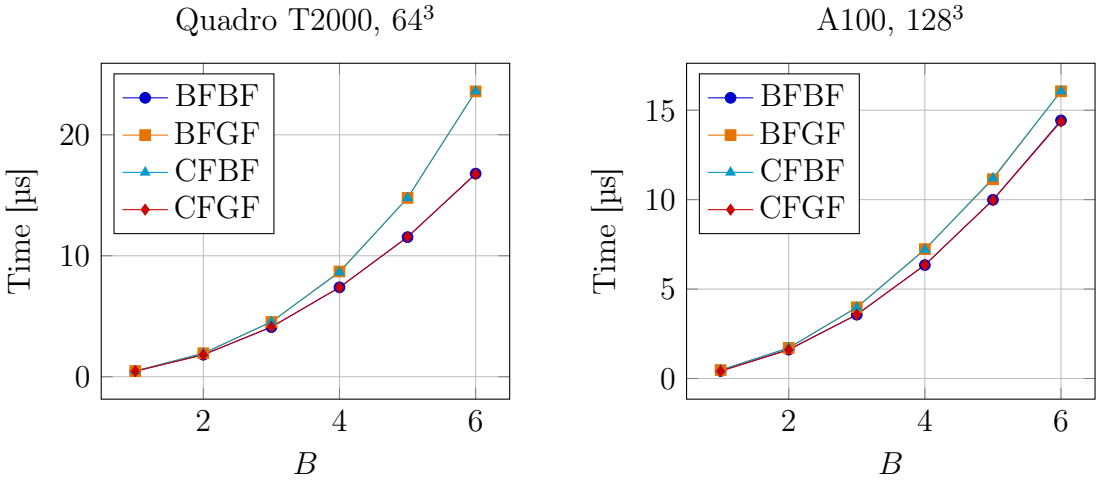


Figure 6.1.: Durations of residual kernel using different layouts for the coefficients and the grid quantities, varying the blocksize B. Results can also be found in the appendix in table A.20.
block_first_v_block_first: BFBF, block_first_v_gp_first: BFBG
coeffs_first_v_block_first: CFBF, coeffs_first_v_gp_first: CFBG



Figure 6.2.: Strided memory access for grid-first layout with stride $= B = 2$ (left), coalesced memory access for block-first layout (right). The teal cells are read in the first iteration, the orange cells are read in the second iteration. The warp size and number of grid points are illustrated as 4 for presentation purposes.

	Time [ms]	FLOPs $\times 10^3$
SP, scalar Jacobi	0.07	49
VVP, scalar Jacobi, SF 8	0.75	281
VVP, block Jacobi, SF 8	0.73	290
VVP, scalar Jacobi, SF 64	37.68	2.770
VVP, block Jacobi, SF 64	39.12	2.870

Table 6.2.: Kernel durations and FLOP count as reported by NCU for one launch of the Jacobi kernel on the Quadro T2000 and a scalar grid size of 32^3. The scaling factor is equal to the blocksize B and determines how many grid points of the scalar problems are agglomerated. SP: Scalar problem, VVP: Vector-valued problem, SF: scaling factor.

compute the inverse of the center coefficient matrix and perform the matrix-vector product $\mathbf{D}_\mathbf{A}^{-1}\mathbf{r}$ (we denote this as block Jacobi). While scalar Jacobi requires fewer instructions, block Jacobi enables the ability to respect interactions between quantities.

Table 6.2 shows the kernel durations and number of floating point operations for one launch of the Jacobi kernel on the Quadro T2000. To make the results comparable, we reduce the grid by a scaling factor for the vector-valued problem. A scaling factor of e.g. 8 means, that 8 grid points are agglomerated into one vector at one grid point. Thus, the scaling factor is equal to the blocksize. We see, that using a blockstencil is significantly more expensive than using a scalar stencil, as more instructions need to be executed. The difference between scalar Jacobi and block Jacobi on the other hand is less significant.

6.2. Remaining Multigrid Components

Mostly, only minor adjustments are needed for the remaining components of the implementation of the vector-valued multigrid method.

The paths taken in the calculation of the coarse grid operator stay the same. Only the multiplications of the coefficients of **RA** and **RAP** become matrix-matrix multiplications instead of scalar multiplications.

Just like the coefficients in the system matrix, the restriction and prolongation stencils are blockstencils in the vector-valued case, too. Thus, similar to the residual calculation, we add two loops iterating over the entries of the coefficient matrices for each grid point and accumulate the result in a temporary variable. The GPU implementation of the prolongation requires an array for storing eight temporary variables, since one work-item fills eight fine grid points corresponding to one coarse grid point.

The ghost update does not require significant changes either. Only the segments of the array storing the border planes in the multi-GPU ghost update are enlarged by a factor equal to the blocksize.

Chapter 7

Evaluation

In this chapter we evaluate the performance of our GPU implementation for different 3d problems. For `mgcl`, we can use the following variants.

- `mgcl` without OpenCL on CPU, sequential
- `mgcl` without OpenCL on CPU, parallel using MPI
- `mgcl` with OpenCL on GPU

When running on the GPU, we can additionally tweak some technical parameters, which is discussed in section 7.1.2.

We also want to compare the results against the PFMG solver of HYPRE's struct interface. PFMG is a semi-coarsening multigrid solver using point-wise Jacobi's method as the smoother. We compile HYPRE with CUDA-aware MPI and unified memory. We do not use symmetric storage for the system matrix $\mathbf{A}$.

The single-GPU tests are performed using a Quadro T2000 (in tandem with an Intel Core i7-10875H 8-core CPU) and an A100 GPU. The multi-GPU tests are performed on the Pleiades cluster featuring 5 GPU nodes with 8 A100 GPUs and two AMD EPYC 7763 32-Core CPUs each. Both `mgcl` and HYPRE are compiled using `clang` on the machine using the Quadro T2000 and GCC on the Pleiades cluster and linked using link time optimization. The OpenMPI implementation of MPI is used. The code under test is run multiple times, taking the best time as result. In multi-GPU scenarios, we prefer to use weak scaling.

This chapter is divided into the application of the implementations to scalar and vector-valued problems. For scalar problems, we first search for the optimal combination of smoothing steps ν_1 and ν_2 and v-cycle iterations `vci` to reach the tolerance. Then, we vary technical parameters specific to `mgcl`. In all tests, we use the damping parameter $\omega = 0.8$.

7.1. Scalar Problems

There are four scalar problems we want to consider.

As our first problem `p1`, we define the Poisson problem

$$-\Delta u = f, \quad (x, y, z) \in \Omega = (0, 1)^3$$

with

$$\begin{aligned}
f(x,y,z) = 12(&x^4y^4z^4(\\
&(x-1)^4(y-1)^4(z-1)^2 \\
&+ (x-1)^4(y-1)^2(z-1)^4 \\
&+ (x-1)^2(y-1)^4(z-1)^4) \\
&+ (x-1)^4(y-1)^4(z-1)^4(\\
&x^4y^4z^2 + x^4y^2z^4 + x^2y^4z^4) \\
&) \\
+ 32(& \\
&x^4y^4z^3(x-1)^4(y-1)^4(z-1)^3 \\
&+ x^4y^3z^4(x-1)^4(y-1)^3(z-1)^4 \\
&+ x^3y^4z^4(x-1)^3(y-1)^4(z-1)^4 \\
&).
\end{aligned}$$

The coefficients do not vary per grid point, but still, we set the coefficients explicitly for each grid point. It has the exact solution

$$u = x^4y^4z^4(x-1)^4(y-1)^4(z-1)^4.$$

For this problem, we use periodic boundary conditions, i.e. $u_n = u_0$, and the full 27-point Laplace stencil.

The second problem `p2` has the same definition as the first one, except for that we use Dirichlet boundary conditions instead of periodic ones,

$$u = 0 \quad (x, y, z) \in \partial\Omega.$$

The third scalar problem `p3` is an anisotropic problem with varying coefficients as shown by Trottenberg et al. in [110] on page 131,

$$\begin{aligned}
-\epsilon u_{xx} - u_{yy} - u_{zz} = f \quad (x, y, z) \in \Omega = (0, 1)^3, \\
u = 0 \quad (x, y, z) \in \partial\Omega,
\end{aligned}$$

with varying ϵ and discretizing the problem using the stencil

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ & & \end{bmatrix}\begin{bmatrix} & -1 & \\ -\epsilon & 2(2+\epsilon) & -\epsilon \\ & -1 & \end{bmatrix}\begin{bmatrix} & -1 & \\ & & \end{bmatrix}.$$

We choose ϵ as

$$\epsilon(x, y, z) = \frac{1}{2}(2 + \sin(2\pi x)\sin(2\pi y)\sin(2\pi z)).$$

We choose the same right-hand side as for problems `p1` and `p2`, modified by the multiplication of the terms induced by ϵu_{xx} with ϵ, resulting in the same analytical solution u.

A fourth problem `p4` results in using problem `p3` with a fixed ϵ. $\epsilon = 1$ creates the isotropic case, while $\epsilon \ll 1$ yields the anisotropic case. The closer ϵ is to 0, the stronger is the anisotropy of the problem.

7.1.1. Convergence Tests

For each problem, we want to check how the choice of ν_1 and ν_2 affects the number of iterations needed to reach the tolerance. We use the Jacobi weight $\omega = 0.8$ and a tolerance of $\texttt{tol} = 10^{-6}$ and test on a grid of sizes 128^3 (largest possible cube-shaped grid on the Quadro T2000). Both `mgcl` and HYPRE use $||\mathbf{r}^{(k)}||_2/||\mathbf{f}||_2 < \texttt{tol}$ as termination criterion, with $\mathbf{r}^{(k)}$ being the residual after ν_1 smoothing steps on the finest grid in iteration k. Figure 7.1 shows the result as heatmaps for the first three problems, comparing `mgcl` and HYPRE's PFMG solver. Figure 7.2 shows the heatmaps for `p4` for $\epsilon = 1$ and $\epsilon = 0.1$.

While for `p1`, the number of v-cycle iterations is quite similar for `mgcl` and HYPRE, `mgcl` needs significantly less iterations than HYPRE for `p2` and `p3`. However, for problem `p4` with $\epsilon = 0.1$ HYPRE needs less v-cycles than `mgcl`. In that case, `mgcl` does not even reach the tolerance for a low number of smoothing steps after the limit of 30 v-cycles. A convergence analysis is not part of this thesis, however we note that main differences are that `mgcl` uses full-coarsening and a fixed interpolation operator, while HYPRE's PFMG solver builds upon circular semi-coarsening in one dimension per level and operator-dependent interpolation. According to Trottenberg et al. [110] (sections 7.7.2 and 7.7.3), operator-dependent interpolation is more effective for anisotropic problems with significant jumps in the coefficients. Semi-coarsening also is known to be more effective than full-coarsening for anisotropic problems, as stated by Trottenberg et al. (section 5.1.2) and Briggs et al. [11] (chapter 7, p. 119 ff.).

N	Problem	Solver	Final rel. res.	Avg. conv. factor	$\|\|e\|\|_2$	V-cycle iters	ν_1	ν_2
128	p1	HYPRE	8.23e-08	3.93e-02	4.00e-09	5	5	5
128	p1	mgcl	8.23e-08	3.93e-02	4.00e-09	6	5	5
128	p2	HYPRE	8.50e-07	2.45e-01	3.64e-12	9	5	5
128	p2	mgcl	7.43e-07	2.94e-02	3.60e-12	5	5	5
128	p3	HYPRE	4.09e-07	2.64e-01	2.18e-12	10	5	5
128	p3	mgcl	8.38e-07	3.03e-02	2.15e-12	5	5	5
128	p4, $\epsilon = 0.1$	HYPRE	5.21e-07	2.64e-01	2.03e-12	10	5	5
128	p4, $\epsilon = 0.1$	mgcl	9.12e-07	5.27e-01	2.01e-12	20	5	5
128	p4, $\epsilon = 1$	HYPRE	4.01e-07	2.64e-01	2.18e-12	10	5	5
128	p4, $\epsilon = 1$	mgcl	8.34e-07	3.03e-02	2.15e-12	5	5	5

Table 7.1.: Best average convergence factors for mgcl and Hypre for all (solver, p)

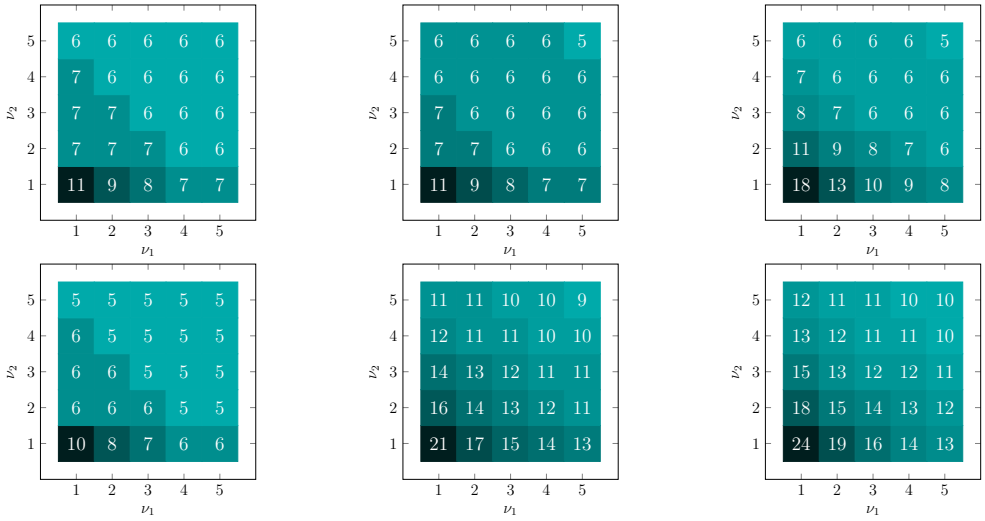
Table 7.1 shows the best average convergence factors for `mgcl` and HYPRE for all (solver, p) combinations, where

$$\text{avg. conv. factor} = \frac{1}{\text{vci}} \sum_{i=1}^{\text{vci}} \frac{\left\| r^{(i)} \right\|_2}{\left\| r^{(i-1)} \right\|_2}.$$

Since the convergence factor is determined by the relative residuals between each v-cycle, the best average convergence factors are achieved with the fewest v-cycle iterations, which usually corresponds to a high number of smoothing steps. However, the lowest average convergence factor does not necessarily correspond to the fastest run, as, on one hand, perhaps fewer smoothing steps can result in the same amount of v-cycles and on the other hand, trading more smoothing steps for fewer v-cycles might result in a lower duration overall.

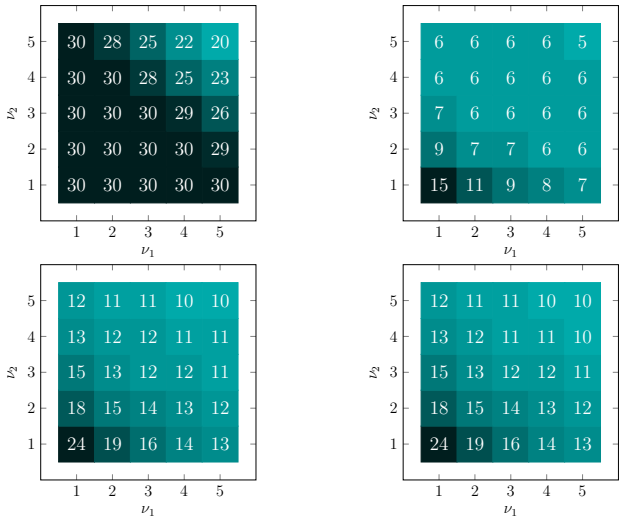
For the most part, the individual number of pre- and post-smoothing steps does not determine the amount of v-cycle iterations, but the sum of both, $\nu_1 + \nu_2$. As long as this sum stays the same, the time spent in a single v-cycle iteration does not change, i.e. pre- and post-smoothing need the same amount of time, since both just execute Jacobi's method operating on different values. Thus, to find the minimum time to reach convergence, we need to find the optimal combination of $(\nu_1 + \nu_2, \texttt{vci})$. For every $\nu_1 + \nu_2$, we use the lowest value of `vci` as reported by our previous test, freeze this value and set the tolerance to a low number that is never reached. Then we can measure the time needed by both solvers in multiple runs and take the minimum total time for comparison. We measure both, the total time from the very first API call to the corresponding solver until it finishes, and the time for the solve step only. Note that, since the total time includes time needed for memory allocations, memory transfers between host and device and initialization of the CUDA and OpenCL runtimes, the values might vary from run to run, depending on the system's status. Nevertheless, we want to include this information in our results to get an idea of the ratios between the solve step and the total time.

Table 7.2 shows the durations for the best `vci` per $(\nu_1 + \nu_2)$. The total number of Jacobi iterations is determined by $\texttt{vci} \cdot (\nu_1 + \nu_2)$. If the number of v-cycle iterations does not differ much between two cases, the total number of Jacobi



(a) p1 (b) p2 (c) p3

Figure 7.1.: Amount of v-cycle iterations until reaching $\texttt{tol} = 10^{-6}$ for pairs of pre- and post-smoothing steps (ν_1, ν_2) for mgcl (top) and HYPRE (bottom). Grid size 128^3. p1: Poisson problem with periodic bc's, p2: Poisson problem with Dirichlet bc's, p3: anisotropic problem with Dirichlet bc's.



(a) p4, $\epsilon = 0.1$ (b) p4, $\epsilon = 1$

Figure 7.2.: Amount of v-cycle iterations until reaching $\texttt{tol} = 10^{-6}$ (limit of 30 v-cycles) for pairs of pre- and post-smoothing steps (ν_1, ν_2) for mgcl (top) and HYPRE (bottom). Grid size 128^3. Anisotropic problem with Dirichlet bc's and fixed ϵ.

problem	solver	N	vci	ν_1	ν_2	vci $\cdot (\nu_1 + \nu_2)$	solve time [ms]	total time [ms]
p1	HYPRE	128	10	1	1	20	194	911
p1	HYPRE	128	6	1	2	18	150	630
p1	HYPRE	128	5	1	5	30	212	697
p1	mgcl	128	11	1	1	22	194	470
p1	mgcl	128	7	1	2	21	156	386
p1	mgcl	128	6	1	5	36	214	439
p2	HYPRE	128	21	1	1	42	378	853
p2	HYPRE	128	16	1	2	48	373	852
p2	HYPRE	128	14	1	3	56	402	880
p2	HYPRE	128	12	1	4	60	410	885
p2	HYPRE	128	11	1	5	66	436	910
p2	HYPRE	128	10	3	5	80	504	984
p2	HYPRE	128	9	5	5	90	551	1026
p2	mgcl	128	11	1	1	22	191	405
p2	mgcl	128	7	1	2	21	154	364
p2	mgcl	128	6	1	4	30	185	405
p2	mgcl	128	5	5	5	50	265	482
p3	HYPRE	128	24	1	1	48	232	398
p3	HYPRE	128	18	1	2	54	222	395
p3	HYPRE	128	15	1	3	60	225	400
p3	HYPRE	128	13	1	4	65	230	403
p3	HYPRE	128	12	1	5	72	245	425
p3	HYPRE	128	11	2	5	77	253	426
p3	HYPRE	128	10	4	5	90	283	457
p3	mgcl	128	18	1	1	36	309	528
p3	mgcl	128	11	1	2	33	239	452
p3	mgcl	128	8	1	3	32	210	428
p3	mgcl	128	7	1	4	35	215	431
p3	mgcl	128	6	1	5	36	212	435
p3	mgcl	128	5	5	5	50	265	478
p4, $\epsilon = 0.1$	HYPRE	128	24	1	1	48	263	439
p4, $\epsilon = 0.1$	HYPRE	128	18	1	2	54	257	431
p4, $\epsilon = 0.1$	HYPRE	128	15	1	3	60	263	439
p4, $\epsilon = 0.1$	HYPRE	128	13	1	4	65	271	448
p4, $\epsilon = 0.1$	HYPRE	128	12	1	5	72	290	465
p4, $\epsilon = 0.1$	HYPRE	128	11	2	5	77	302	477
p4, $\epsilon = 0.1$	HYPRE	128	10	4	5	90	341	518
p4, $\epsilon = 0.1$	mgcl	128	28	2	5	196	1087	1300
p4, $\epsilon = 0.1$	mgcl	128	25	3	5	200	1080	1293
p4, $\epsilon = 0.1$	mgcl	128	22	4	5	198	1047	1261
p4, $\epsilon = 0.1$	mgcl	128	20	5	5	200	1039	1249
p4, $\epsilon = 1$	HYPRE	128	24	1	1	48	236	423
p4, $\epsilon = 1$	HYPRE	128	18	1	2	54	223	402
p4, $\epsilon = 1$	HYPRE	128	15	1	3	60	227	405
p4, $\epsilon = 1$	HYPRE	128	13	1	4	65	230	411
p4, $\epsilon = 1$	HYPRE	128	12	1	5	72	245	422
p4, $\epsilon = 1$	HYPRE	128	11	2	5	77	254	431
p4, $\epsilon = 1$	HYPRE	128	10	4	5	90	284	462
p4, $\epsilon = 1$	mgcl	128	15	1	1	30	258	475
p4, $\epsilon = 1$	mgcl	128	9	1	2	27	197	425
p4, $\epsilon = 1$	mgcl	128	7	1	3	28	185	407
p4, $\epsilon = 1$	mgcl	128	6	1	4	30	185	407
p4, $\epsilon = 1$	mgcl	128	5	5	5	50	265	486

Table 7.2.: Durations for different problems with best `vci` per $\nu_1 + \nu_2$. Executed on a single Quadro T2000, grid size 128^3.

iterations determines the runtime. This can be clearly seen, for example, in the case p2, `mgcl`: $(\texttt{vci}, \nu_1, \nu_2) = (6, 1, 4)$ needs 185 ms, while $(5, 5, 5)$ needs 265 ms. Higher counts of `vci` result in a higher number of total smoothing iterations and thus higher time consumption, even if the number of smoothing iterations is about the same per v-cycle. This is the case, for example, for p4, $\epsilon = 1$ `mgcl`: $(15, 1, 1)$ needs 258 ms, while $(9, 1, 2)$ only needs 197 ms.

Figures 7.3, 7.4 and 7.5 (Quadro T2000) and 7.6, 7.7 and 7.8 (A100) show the durations of the solve step of `mgcl` and HYPRE for a fixed amount of smoothing steps and varying `vci` or a fixed `vci` and varying ν_1 in the single-GPU case. For `p3`, HYPRE is faster than `mgcl` in all scenarios. The reason is that HYPRE computes with only 7 coefficients on the finest grid for that problem, while `mgcl` uses all 27 coefficients, 20 of which being set to zero. Currently, there is no way to use only 7 coefficients with varying coefficients in `mgcl`. For `p1` and `p2`, the durations are quite comparable using the Quadro T2000. Using the A100, HYPRE performs better for a low total number of smoothing steps. However, `mgcl` scales better with an increasing number of smoothing steps for p1 and p2. The significant difference in the case of p2 with one v-cycle using the A100 can be explained by analysis using Nsight Systems. The OpenCL backend version used for the A100 delays device memory allocation until it is first used. In the case of Dirichlet boundary conditions, v and f are first used after the setup has been completed, thus the time needed for allocation is included in the measurement. According to Nsight Systems, each of those two memory allocation take about 10 ms, which lies in the same realm as the difference between `mgcl` and HYPRE for $\nu_1 = 1$.

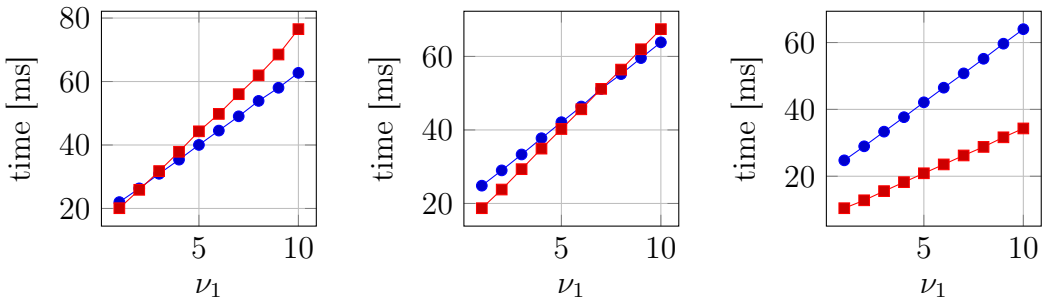
Figures 7.9, 7.10 and 7.11 (Quadro T2000) and 7.12, 7.13 and 7.14 (A100) show the total durations of the same test cases as before. For p1 and p2, our measurements suggest that `mgcl` is significantly faster than HYPRE using the Quadro T2000. For p3, HYPRE is notably faster than `mgcl`: Not only does HYPRE calculate with only 7 coefficients, but it also needs to allocate memory for only 7 coefficients per grid point instead of the 27 coefficients per grid point that `mgcl` always allocates. For both `mgcl` and HYPRE, the time needed for copying the coefficients into device memory is included in the measurement. Since we are using managed memory for HYPRE, we cannot explicitly control when the data transfer between host and device occurs. Furthermore, there is a prominent spike in the very first run of HYPRE in all cases. Analysis with linux perftools reveals, that although the CUDA memory buffer storing the coefficients is freed after each iteration (using `cudaFree`), subsequent iterations do not pay any cost for the initial memory allocations (`cudaMalloc`). While we cannot look under the hood of the Nvidia driver implementation, it leads to the assumption that the memory buffer is not released immediately upon the call of `cudaFree`, but can be reused for further memory allocations. Additionally, according to Nsight Systems, there is a noticeable overhead for the very first kernel launch: For HYPRE, the first

kernel sets up the coefficients in device memory. While the very first host-side call to this kernel needs 52 ms, a subsequent call only needs 31 µs. In the same fashion, running `mgcl` using OpenCL, Nsight Systems shows only one call to `cuCtxCreate` for creating the context, although it is not reused in the benchmark. Additionally, the very first call to `cuModuleLoad`, which loads the compiled OpenCL code into the CUDA driver, records 89 ms, while the subsequent calls only need about 10 ms.

Figures 7.15, 7.16, 7.17, 7.18, 7.19 and 7.20 show the solve and total durations of the same test cases as before using eight A100 GPUs on a single Pleiades node, operating on a global grid of size 512^3. For p1 and p2, `mgcl` records lower solve and total durations than HYPRE. For p3, the durations are comparable, with HYPRE scaling better than `mgcl`. If we compare the solve time for p1, one v-cycle, $\nu_1 = \nu_2 = 1$, we see, that `mgcl` needs about double the time in the multi-GPU case compared to the single-GPU case (25 ms vs. 44 ms). For the same case, HYPRE needs 18 ms in the single-GPU case and 74 ms in the multi-GPU case. `mgcl` keeps the ratio of 2 for high numbers of smoothing steps, e.g. for `vci` $= 10$ and $\nu_1 = 10, \nu_2 = 1$, it needs 487 ms in the single-GPU case and 1080 ms in the multi-GPU case. HYPRE's ratio remains slightly higher at approximately 2.19 (582 ms vs. 1277 ms). The large spike in the total duration for `mgcl` is due to a call to `cuCtxCreate` for each visible device, i.e. eight in our case. Although we call `clCreateContext` only once per process and limit the visibility using the environment variable `CUDA_VISIBLE_DEVICES`, the driver apparently creates a context for each device. According to Nsight Systems, the total time these calls need is approximately 1.9 s, which is about the time difference we observe between the first two results.

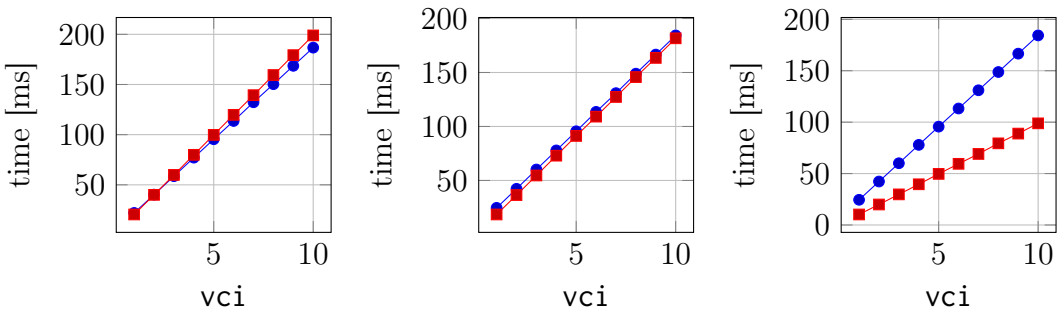
Table 7.3 compares the durations of different portions of `mgcl` and HYPRE as shown in Nsight Systems for N = 128, $\nu_1 = \nu_2 = 2$, periodic boundary conditions, on a Quadro T2000. As we can see, especially in the most expensive part, the residual calculation, mgcl outperforms HYPRE. Additionally, the whole part of coarse grid calculation is faster in `mgcl`, which is expected due to full-coarsening vs. semi-coarsening.

Table 7.4 and figure 7.21 show the durations of the different parts of the ghost update of v between two Jacobi iterations in HYPRE and `mgcl` using eight A100 GPUs on a local grid size of 256^3 and global grid size of 512^3. While the kernel durations for pre- and post-processing of the data are smaller in `mgcl`, the total duration of the ghost update is larger. As we can see in the timeline view in Nsight Systems, HYPRE achieves a high amount of host-device- and MPI-transfer overlapping due to many small host-device transfers and asynchronous MPI transfers.



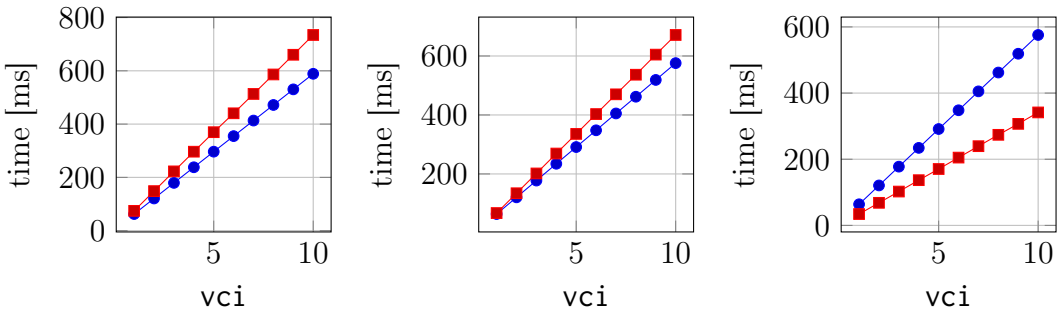
(a) p1 (b) p2 (c) p3

Figure 7.3.: Solve durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl (blue) and HYPRE (red). Grid size 128^3, executed on a single Quadro T2000.



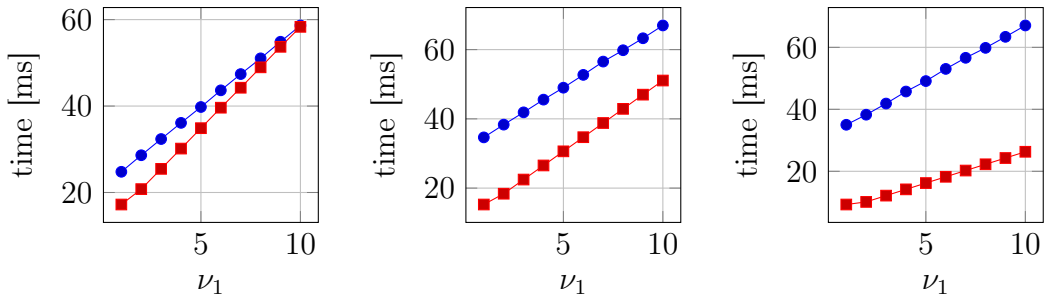
(a) p1 (b) p2 (c) p3

Figure 7.4.: Solve durations for 1-10 v-cycles, $\nu_1 = 1$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 128^3, executed on a single Quadro T2000.



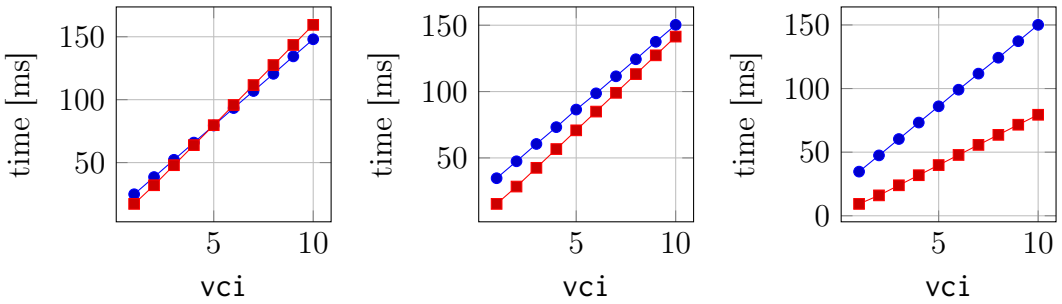
(a) p1 (b) p2 (c) p3

Figure 7.5.: Solve durations for 1-10 v-cycles, $\nu_1 = 10$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 128^3, executed on a single Quadro T2000.



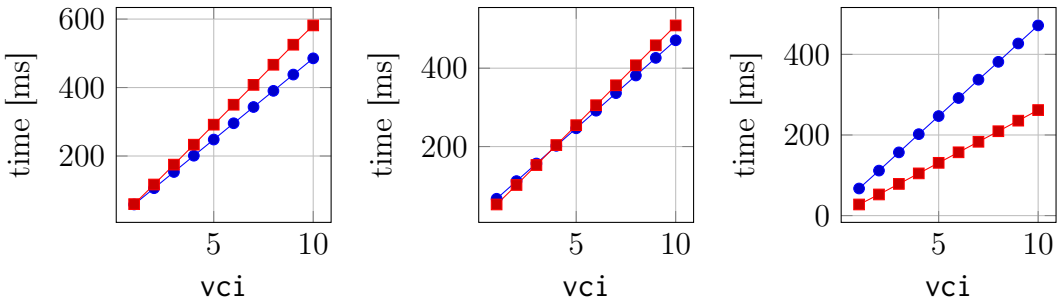
(a) p1 (b) p2 (c) p3

Figure 7.6.: Solve durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl (blue) and HYPRE (red). Grid size 256^3, executed on a single A100.



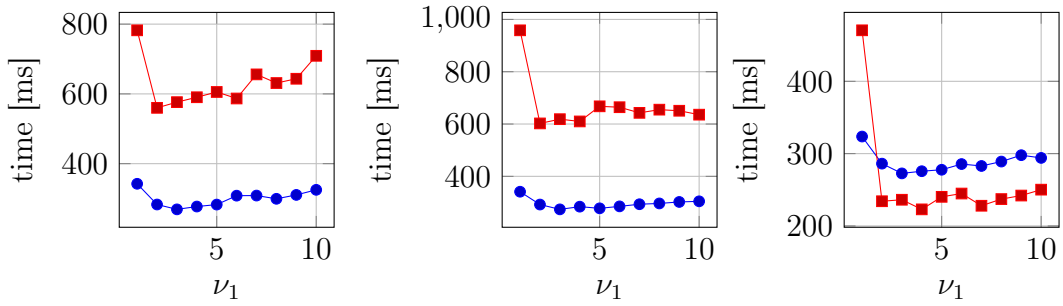
(a) p1 (b) p2 (c) p3

Figure 7.7.: Solve durations for 1-10 v-cycles, $\nu_1 = 1$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 256^3, executed on a single A100.



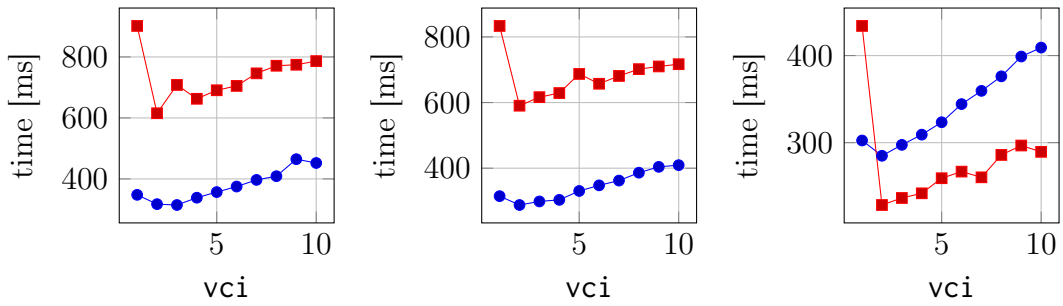
(a) p1 (b) p2 (c) p3

Figure 7.8.: Solve durations for 1-10 v-cycles, $\nu_1 = 10$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 256^3, executed on a single A100.



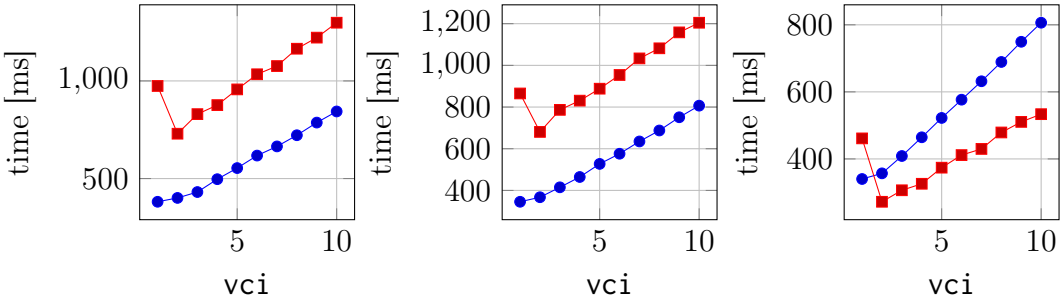
(a) p1 (b) p2 (c) p3

Figure 7.9.: Total durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl (blue) and HYPRE (red). Grid size 128^3, executed on a single Quadro T2000.



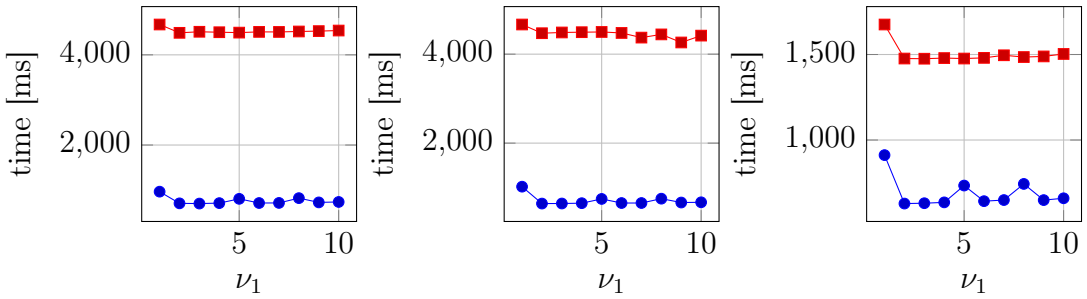
(a) p1 (b) p2 (c) p3

Figure 7.10.: Total durations for 1-10 v-cycles, $\nu_1 = 1$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 128^3, executed on a single Quadro T2000.



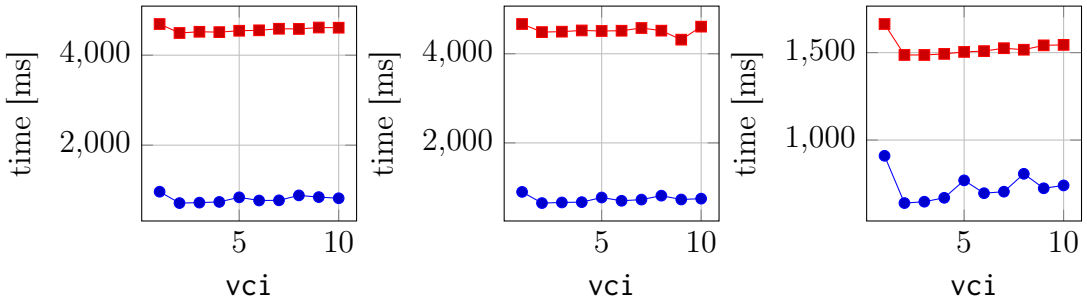
(a) p1 (b) p2 (c) p3

Figure 7.11.: Total durations for 1-10 v-cycles, $\nu_1 = 10$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 128^3, executed on a single Quadro T2000.



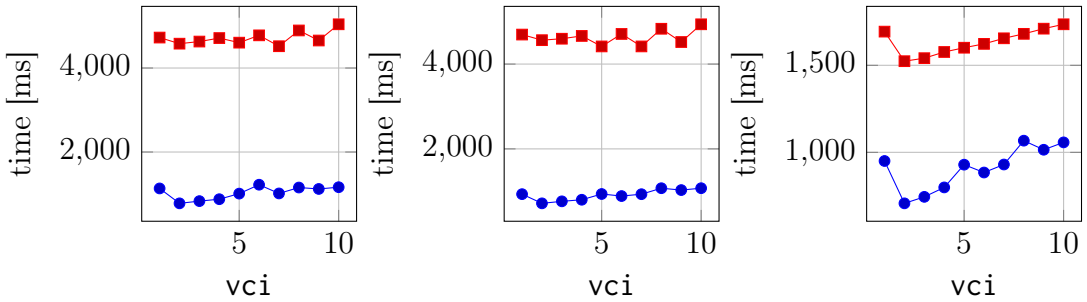
(a) p1 (b) p2 (c) p3

Figure 7.12.: Total durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl (blue) and HYPRE (red). Grid size 256^3, executed on a single A100.



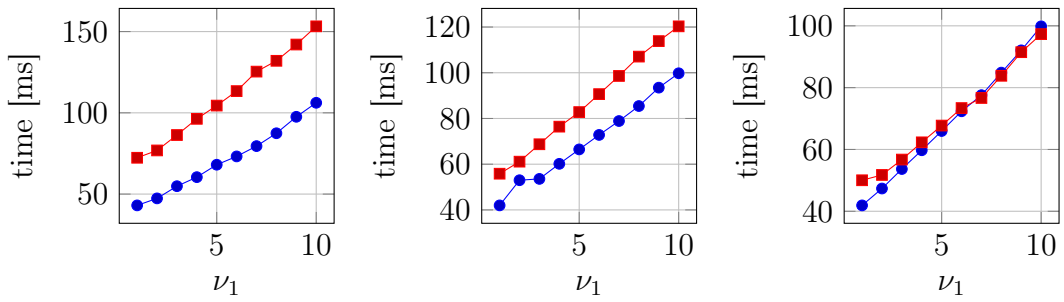
(a) p1 (b) p2 (c) p3

Figure 7.13.: Total durations for 1-10 v-cycles, $\nu_1 = 1$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 256^3, executed on a single A100.



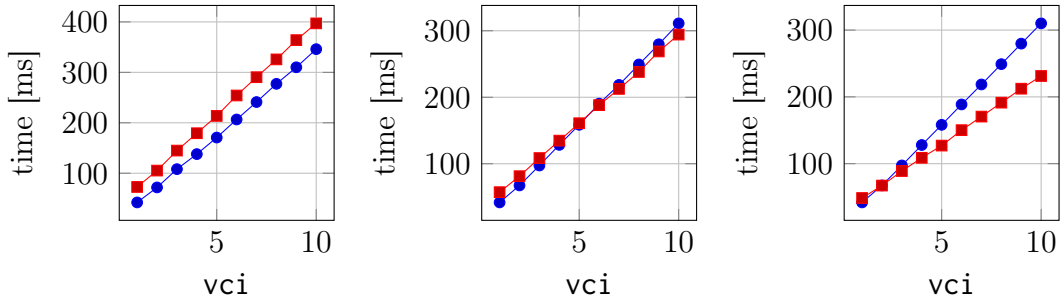
(a) p1 (b) p2 (c) p3

Figure 7.14.: Total durations for 1-10 v-cycles, $\nu_1 = 10$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). Grid size 256^3, executed on a single A100.



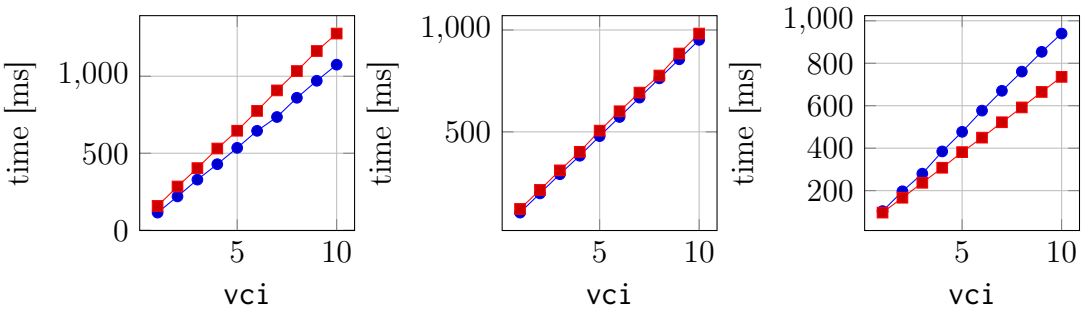
(a) p1 (b) p2 (c) p3

Figure 7.15.: Solve durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl (blue) and HYPRE (red). $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$, executed on eight A100 GPUs.



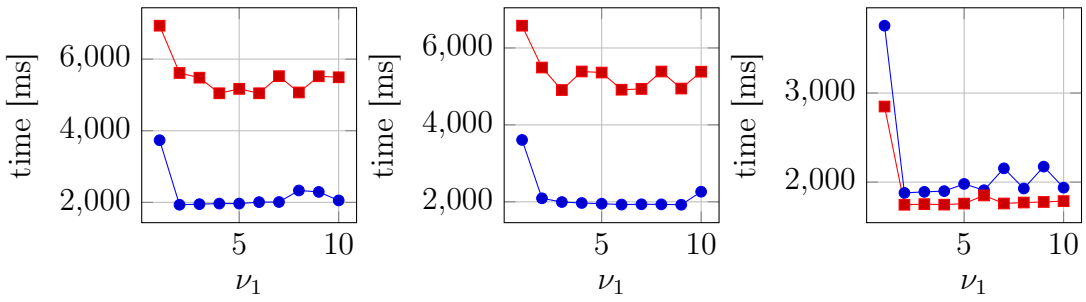
(a) p1 (b) p2 (c) p3

Figure 7.16.: Solve durations for 1-10 v-cycles, $\nu_1 = 1$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$, executed on eight A100 GPUs.



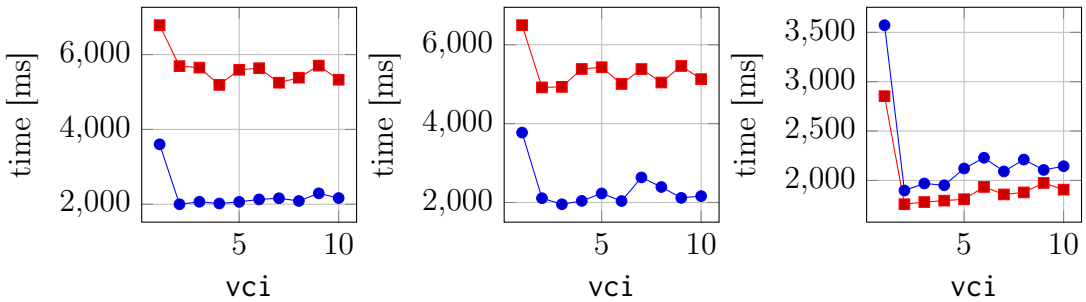
(a) p1 (b) p2 (c) p3

Figure 7.17.: Solve durations for 1-10 v-cycles, $\nu_1 = 10$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$, executed on eight A100 GPUs.



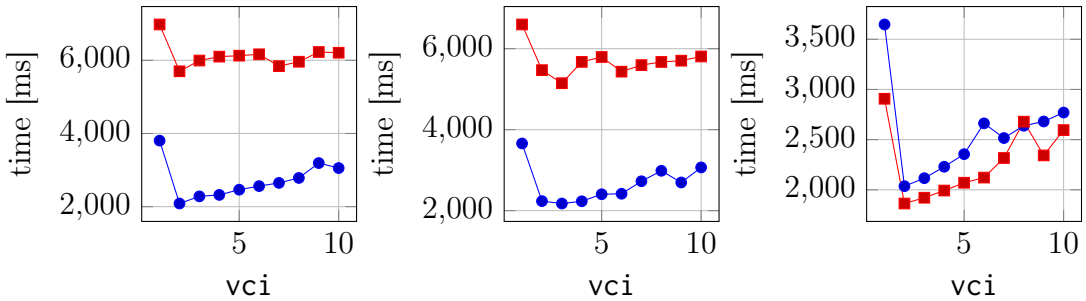
(a) p1 (b) p2 (c) p3

Figure 7.18.: Total durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl (blue) and HYPRE (red). $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$, executed on eight A100 GPUs.



(a) p1 (b) p2 (c) p3

Figure 7.19.: Total durations for 1-10 v-cycles, $\nu_1 = 1$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$, executed on eight A100 GPUs.



(a) p1 (b) p2 (c) p3

Figure 7.20.: Total durations for 1-10 v-cycles, $\nu_1 = 10$, $\nu_2 = 1$, for mgcl (blue) and HYPRE (red). $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$, executed on eight A100 GPUs.

	HYPRE	`mgcl`
one entire v-cycle ($\nu_1 = \nu_2 = 2$)	32	27
one residual calculation and Jacobi update on the finest grid	4.9	3.7
coarse grids from restrict to prolongate	9.2	4.6
ghost update on the finest grid	0.129	0.098
setup from copying coefficients until first Jacobi kernel call	317	106

Table 7.3.: Durations of different portions of `mgcl` and HYPRE as shown in Nsight Systems. Grid size 128^3, executed on a single Quadro T2000 with periodic boundary conditions. Time in ms.

Total time	Instances	Operation
	Hypre	
739.8 µs	10	memcpy DtoH
329.3 µs	10	memcpy HtoD
68.7 µs	26	hypre_StructCommunicationInitialize
64.1 µs	26	hypre_StructCommunicationFinalize
2.1 ms	1	MPI_Waitall
114.6 µs	7	MPI_Isend
2.6 µs	7	MPI_Irecv
2.5 ms	–	Duration of entire time frame selected in nsys
	mgcl	
652.4 µs	1	memcpy DtoH
305.0 µs	1	memcpy HtoD
16.0 µs	1	extract_border_planes
12.5 µs	1	paste_ghosts_from_border_planes
3.5 ms	6	MPI_Sendrecv
4.8 ms	–	Duration of entire time frame selected in nsys

Table 7.4.: Durations of different portions of the ghost update of v on the finest grid in `mgcl` and HYPRE as shown in Nsight Systems. $N^{(0)}_{\text{global}} = 512$, $N^{(0)}_{\text{local}} = 256$, executed on eight distinct A100 GPUs with periodic boundary conditions.



Figure 7.21.: Nsight System's view of one ghost update of v via MPI on a 256^3 local grid for `mgcl` (top) and HYPRE (bottom).

7.1.2. Varying Technical Parameters of `mgcl`

In this section, we want to elaborate on how varying the different technical parameters of `mgcl` affects the performance. For each problem `p1`, `p2`, and `p3`, we use the values of `vci`, ν_1 and ν_2 that leads to the minimum solve time using the Quadro T2000, as seen in the previous section. We vary the following parameters.

- Ignoring the tolerance, using a fixed value for `vci` (avoids synchronization for sending the residual sum to the host)
- Varying the number of fine grids, that are calculated on the GPU (OpenCL level threshold, remaining coarse grids are calculated sequentially on the CPU)
- Varying the amount of fine grids, that are calculated distributively (MPI level threshold, multi-GPU only)
- Varying the amount of fine grids, for which the overlapped Jacobi strategy is used (overlapped Jacobi level threshold, multi-GPU only)

Additionally, a comparison between GPU and CPU performance of `mgcl` is given.

Problem	vci	nu1	nu2	!ignoreTol	ignoreTol
p1	8	1	2	186.11	181.49
p2	8	1	2	183.95	179.39
p3	6	1	5	217.88	214.71
p4, $\epsilon = 0.1$	21	5	5	1112.6	1100.37
p4, $\epsilon = 1$	7	1	3	192.21	188.39

Table 7.5.: Solve durations for different problems with (vci, ν_1, ν_2) combination that consumes the lowest amount of solve time, ignoring or not ignoring the tolerance. Executed on a single Quadro T2000, grid size 128^3. When ignoreTol is used, the residual norm is not calculated and sent to the host after a full v-cycle, thus there is no need to synchronize with the GPU. Time in ms.

				maxLvOcl				
Problem	vci	ν_1	ν_2	0	1	2	3	4
p1	8	1	2	982	283	198	187	186
p2	8	1	2	1023	293	196	186	185
p3	6	1	5	1252	347	233	220	219
p4, $\epsilon = 0.1$	21	5	5	6373	1813	1189	1119	1113
p4, $\epsilon = 1$	7	1	3	1177	319	207	194	193

Table 7.6.: Solve durations for different problems with (vci, ν_1, ν_2) combination that consumes the lowest amount of solve time. Executed on a single Quadro T2000, grid size 128^3. Varies the maximum level, for which OpenCL is used. For maxLvOcl = 0, only the finest level is using the GPU. Time in ms.

Table 7.5 shows the effect of ignoring the tolerance, such that there is no need to synchronize with the GPU after every v-cycle. Since we use the residual calculated before the restriction to check the termination criterion and the computation of the L2-norm is relatively inexpensive, we do not see a significant impact on performance when ignoring the tolerance.

Table 7.6 shows the impact of setting a level threshold for OpenCL usage, i.e. setting a maximum level for which OpenCL is used. Clearly, it is worthwhile to compute the fine grids using OpenCL for this problem size (which is the maximum possible cube-shaped size on the Quadro T2000). However, starting from level 3 (grid size 16^3), the difference in time between using OpenCL and not using it does not vary significantly.

Table 7.7 shows the effect of varying the MPI level threshold, i.e. the first level, that is not computed distributively anymore. On this level, the data of the remaining grid points are gathered onto the root process. In mgcl, per default, there need to be at least 4 local grid points for MPI to be used, which corresponds to a MPI

				MPI LT					
Problem	vci	ν_1	ν_2	1	2	3	4	5	6
p1	8	1	2	1513	445	400	364	359	346
p2	8	1	2	1493	444	353	354	353	360
p3	6	1	5	1264	465	439	411	415	422
p4, $\epsilon = 0.1$	21	5	5	5086	2183	2158	2123	2148	2119
p4, $\epsilon = 1$	7	1	3	1366	432	394	370	372	373

Table 7.7.: Solve durations for different problems with (`vci`, ν_1, ν_2) combination that consumes the lowest amount of solve time. Executed on eight A100 GPUs, $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$. Varies the maximum level that is computed distributively, before gathering data of all remaining grid points onto the root process. MPI LT: MPI level threshold. Time in ms.

				Overlapped Jacobi LT				
Problem	vci	ν_1	ν_2	-1	0	1	2	3
p1	8	1	2	352	343	359	357	359
p2	8	1	2	351	333	349	351	357
p3	6	1	5	407	359	349	368	371
p4, $\epsilon = 0.1$	21	5	5	2078	1713	1717	1740	1744
p4, $\epsilon = 1$	7	1	3	355	337	346	351	355

Table 7.8.: Solve durations for different problems with (`vci`, ν_1, ν_2) combination that consumes the lowest amount of solve time. Executed on eight A100 GPUs, $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$. Varies the maximum level for which the overlapped Jacobi strategy is used. -1 means it is not used at all. LT: Level threshold. Time in ms.

level threshold of 6 in this test case. In all cases, it is worthwhile to calculate at least the first four levels (level 0 to 3) distributively. For p1, the duration moderately decreases up to level 6, while for p3 and p4 (both variants), gathering data on level 4 seems to be the sweet spot. Only for p2, the lowest duration is reached at level 3. The increase in time for higher level thresholds is moderate.

Table 7.8 shows the effect of varying the maximum level for which the overlapped Jacobi strategy is used, i.e. overlapping the computation of the inner domain with the computation of the halo zone plus communication via MPI. In all cases, using overlapped Jacobi on the finest level is worthwhile compared to not using it at all. As the total number of Jacobi iterations increases, the time savings become more substantial. However, the benefit stagnates or declines if overlapped Jacobi is used on coarser grids as well.

Table 7.9 compares the performance of `mgcl` using the Quadro T2000 GPU, a

problem	vci	nu1	nu2	GPU	CPU P=1	CPU P=8
p1	8	1	2	186	7695	2061
p2	8	1	2	184	8271	1744
p3	6	1	5	218	9030	2016
p4, $\epsilon = 0.1$	21	5	5	1113	40709	9914
p4, $\epsilon = 1$	7	1	3	192	7432	1963

Table 7.9.: Solve durations of `mgcl` using the Quadro T2000 GPU, the Intel Core i7-10875H 8-core CPU with one MPI process and the same CPU with 8 MPI processes. The global problem size is 128^3. Time in ms.

					CPU				
problem	vci	nu1	nu2	GPU	P=1	P=8	P=64	P=512	P=4096
p1	8	1	2	152	46778	5746	1124	196	307
p2	8	1	2	153	45710	5603	1129	181	148
p3	6	1	5	185	46811	6673	1328	196	128
p4, $\epsilon = 0.1$	21	5	5	891	230545	32877	6385	925	800
p4, $\epsilon = 1$	7	1	3	162	39970	5982	1182	176	220

Table 7.10.: Solve durations of `mgcl` using the A100 GPU and the AMD EPYC 7452 32-Core CPU with varying number of MPI processes on the Pleiades cluster. One node features two CPUs (64 cores in total) each, thus P=512 and P=4096 span over 8 and 64 nodes, respectively. The global problem size is 256^3. Time in ms.

CPU with one MPI process and a CPU with eight MPI processes. The global size of each problem is 128^3. Overall using the GPU needs an order of magnitude less time than using the CPU with a single thread. For example for `p1`, the GPU only needs 2.4 % of the time the CPU needs. Using eight processes on the CPU also reduces the time needed, but the difference to using only a single process is not as tremendous as to using the GPU, needing about one fourth of the time for `p1`. All problems under test show comparable ratios. The difference between using one GPU and a CPU with a single MPI process is even more severe on the Pleiades cluster, as shown in table 7.10. We need 512 MPI processes (16 CPUs with 32 cores each) to achieve a performance comparable with a single A100 GPU. Past this point, adding more CPUs degrades the performance as the communication overhead increases. As a rough estimate, the A100 GPU presumably only consumes a fraction of the power of the 16 CPUs: The A100 is listed with a TDP (thermal design power) of 400 W, a single EPYC 7452 CPU with 155 W, thus 2480 W in total.

N	b	final rel. res	avg. conv. factor	$\|\|e\|\|_2$	tol	vci	ν_1	ν_2
64	0	8.78e-07	1.14e-01	3.99e-09	1e-06	7	2	2
64	0	8.78e-07	1.14e-01	3.99e-09	1e-06	7	2	2
64	1	8.78e-07	1.14e-01	3.99e-09	1e-06	7	2	2
64	0	8.78e-07	1.14e-01	3.99e-09	1e-06	7	2	2
64	1	8.78e-07	1.14e-01	3.99e-09	1e-06	7	2	2
64	2	8.78e-07	1.14e-01	3.99e-09	1e-06	7	2	2

Table 7.11.: Convergence results for blocksize $B \in \{1, 2, 3\}$ on a grid of size 64^3. b is the index of the vector component.

N	B	vci	ν_1	ν_2	solve time [ms]	total time [ms]
64	1	7	2	2	27.94	230.87
64	2	7	2	2	87.28	436.95
64	3	7	2	2	184.33	792.00

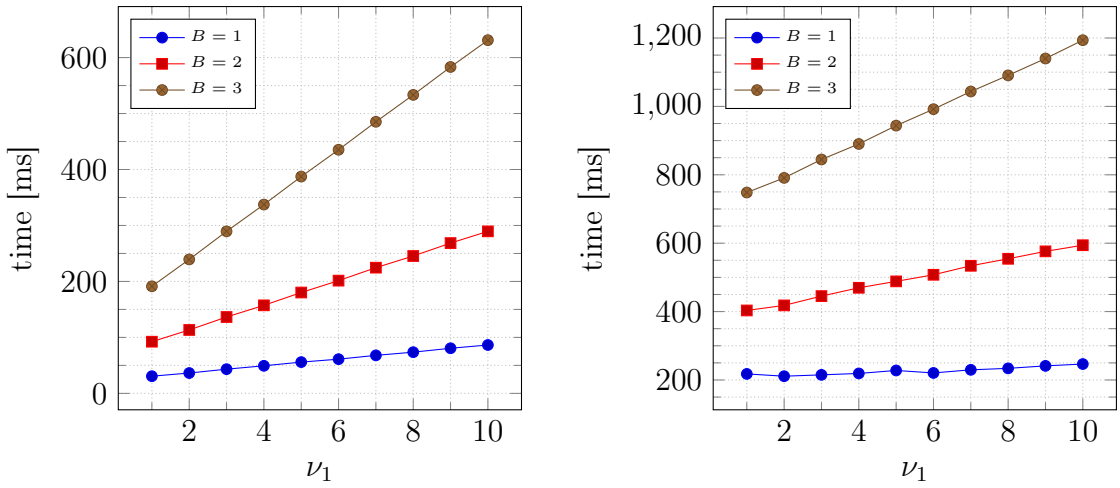
Table 7.12.: Solve and total durations for solving the vector-valued problem using blocksize $B \in \{1, 2, 3\}$ on the Quadro T2000 GPU on a grid of size 64^3.

7.2. Vector-valued Problem

In this section, we want to evaluate the performance of `mgcl` solving vector-valued problems using blockstencils. We construct a system of B independent problems, where each problem is akin to p1 from the previous section. While this does not represent a real-world scenario, it is an easy way to get an idea of the time consumption when working with vector-valued problems.

Table 7.11 shows that each problem of the system of problems converges to the same result, as expected. Thus, we can ascertain the correctness of the implementation at least for problems with independent quantities.

Table 7.12 shows the solve and total durations for solving the vector-valued problem using blocksize $B \in \{1, 2, 3\}$ on the Quadro T2000 GPU on a grid of size 64^3. We see, that adding vector components adds a significant amount of solve time, i.e. the solve time increases by more than a factor of two when incrementing B. Since applying a stencil onto a grid point mainly consists of a matrix-vector-multiplication, which has complexity $O(B^2)$, a superlinear increase in solve time is to be expected. The total time increases by a factor between 1.8 and 1.9 for increasing B. Note that a warm-up run is performed for this benchmark, such that the total durations are comparable.

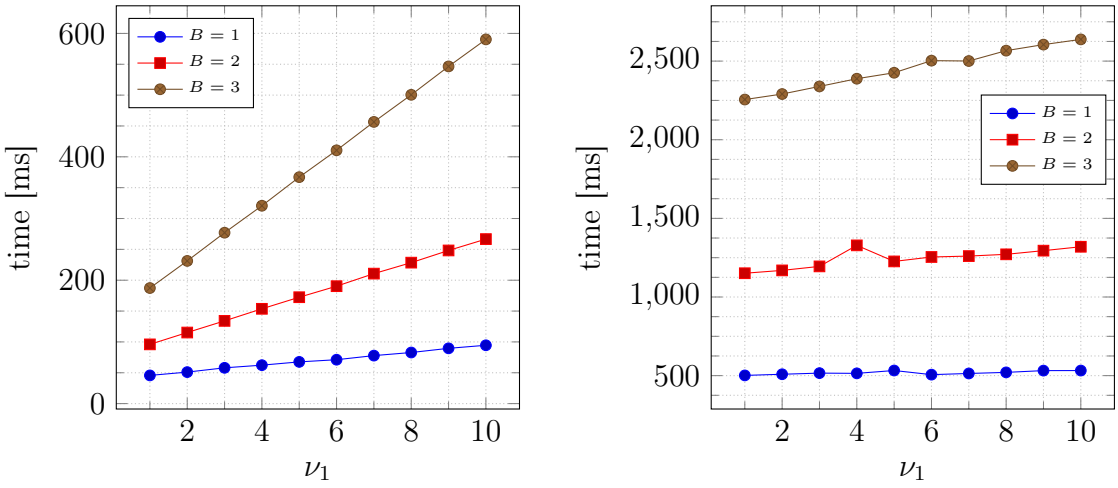


(a) solve durations

(b) total durations

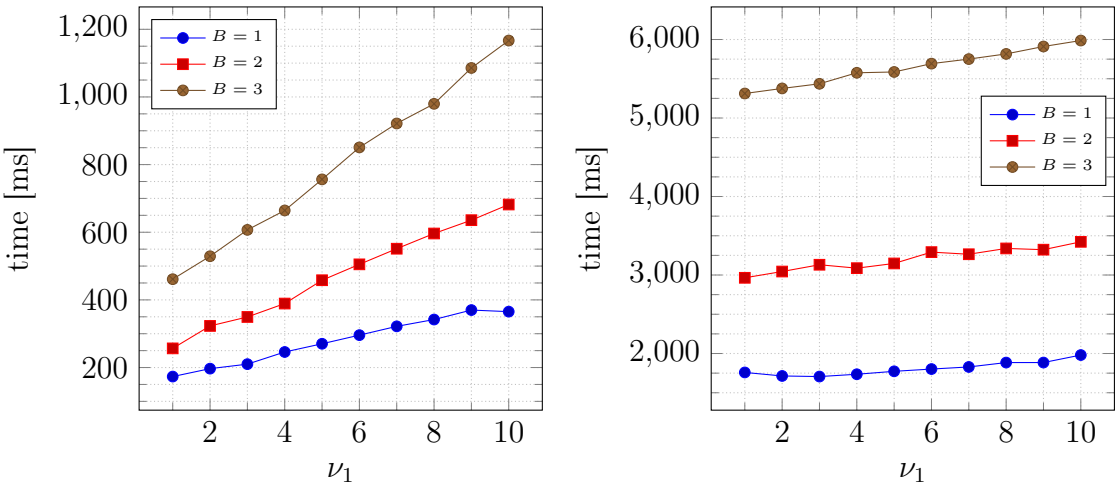
Figure 7.22.: Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. Grid size 64^3, executed on a single Quadro T2000.

Figures 7.22, 7.23 and 7.24 show the solve and total durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$, using blocksize $B \in \{1, 2, 3\}$. Results are shown for a single Quadro T2000 (64^3), a single A100 (128^3) and eight A100 GPUs ($N^{(0)}_{\text{local}} = 128^3, N^{(0)}_{\text{global}} = 256^3$). For all blocksizes, we note a linear increase in both solve duration and total duration. Larger blocksizes lead to a steeper increase in duration. Table 7.13 shows the solve and total durations for one and eight A100 GPUs in ms and the ratio of solve and total durations of eight A100 GPUs to one A100 GPU. For increasing B, the relative difference between using a single and multiple GPUs shrinks. This is expected, as the ratio of the durations of the residual calculation to the ghost update raises.



(a) solve durations (b) total durations

Figure 7.23.: Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. Grid size 128^3, executed on a single A100.



(a) solve durations (b) total durations

Figure 7.24.: Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. $N^{(0)}_{\text{local}} = 128^3, N^{(0)}_{\text{global}} = 256^3$, executed on eight A100 GPUs.

b	vci	ν_1	ν_2	solve 1 GPU	solve 8 GPUs	total 1 GPU	total 8 GPUs	solve ratio	total ratio
1	10	1	1	45.7	173.32	501.44	1758.8	3.79	3.51
1	10	2	1	51.05	196.71	508.81	1713.98	3.85	3.37
1	10	3	1	58	210.01	516.03	1706.34	3.62	3.31
1	10	4	1	62.31	245.98	514.48	1735.69	3.95	3.37
1	10	5	1	67.62	270.28	532.57	1773.11	4.00	3.33
1	10	6	1	71.14	295.79	506.28	1802	4.16	3.56
1	10	7	1	77.79	321.76	513.77	1828.71	4.14	3.56
1	10	8	1	82.81	341.9	520.71	1884.62	4.13	3.62
1	10	9	1	89.59	369.92	531.94	1884.78	4.13	3.54
1	10	10	1	94.54	365.24	532.3	1980.39	3.86	3.72
2	10	1	1	96.01	256.64	1151.07	2964.43	2.67	2.58
2	10	2	1	115.2	323.01	1169.17	3044.43	2.80	2.60
2	10	3	1	134.08	349.52	1194.63	3130.68	2.61	2.62
2	10	4	1	153.65	389.19	1328.37	3087.18	2.53	2.32
2	10	5	1	172.43	458.12	1226.28	3147.26	2.66	2.57
2	10	6	1	190.32	505.09	1254.41	3291.56	2.65	2.62
2	10	7	1	210.68	551.3	1260.17	3264.52	2.62	2.59
2	10	8	1	228.48	596.29	1271.38	3338.5	2.61	2.63
2	10	9	1	248.23	635.66	1294.71	3322.61	2.56	2.57
2	10	10	1	266.64	681.99	1319.45	3422.79	2.56	2.59
3	10	1	1	187.31	461.18	2255.49	5312.28	2.46	2.36
3	10	2	1	231.27	529.06	2290.51	5376.67	2.29	2.35
3	10	3	1	277	606.98	2338.89	5435.92	2.19	2.32
3	10	4	1	320.69	664.42	2387.97	5576.05	2.07	2.34
3	10	5	1	367.05	756.28	2425.83	5586.45	2.06	2.30
3	10	6	1	410.54	850.79	2502.9	5692.66	2.07	2.27
3	10	7	1	456.56	921.55	2500.44	5750.66	2.02	2.30
3	10	8	1	500.5	979.63	2566.37	5816.42	1.96	2.27
3	10	9	1	546.53	1085.67	2604.79	5910.9	1.99	2.27
3	10	10	1	590.31	1166.78	2637.95	5986.95	1.98	2.27

Table 7.13.: Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. Executed on one and eight A100 GPUs with local grid size of 128^3. The solve ratio is the ratio of the solve time for 8 GPUs to the solve time for 1 GPU, total ratio vice versa. Time in ms.

Chapter 8

Conclusion and Outlook

In this thesis, we elaborated on the different parts that construct a geometric multigrid implementation and their optimization opportunities using GPUs via OpenCL. We were able to distill important aspects when using GPUs in general and specific to our use-case, both in the single-GPU and multi-GPU cases. An optimized implementation for solving elliptic partial differential equations featuring spatially varying coefficients was developed and tested against the existing implementation HYPRE. `mgcl` was found to be able to compete with, and in some cases outperform, the alternative. Additionally, to our knowledge, it is the first GPU-based implementation supporting vector-valued problems.

Through rigorous testing – particularly regarding the implementation of the residual calculation and Jacobi's method using spatially varying coefficients – we contributed knowledge to an area that is primarily dominated by studies featuring problems using fixed coefficients. While most of the optimizations involving local memory usage can lead to notable performance gains for fixed coefficients (that is, if the implementation is sophisticated enough), local memory usage appears less useful in the case of varying coefficients. This is especially true since the fusion of local memory with the L1 cache on Nvidia GPUs.

An algorithm for the calculation of the Galerkin coarse grid operator based on stencil representation was developed and documented, as well as the multiplication of two stencils and the conversion between stencil and matrix representations. We were unable to find reports of this in the existing literature at the time of writing.

`mgcl` has proven to be performant in its specific area of application. Nevertheless, there are still opportunities to extend its capabilities and improve existing features.

Currently, the scaling in multi-GPU scenarios is limited due to the monadic agglomeration scheme. Implementing dynamic agglomeration on multiple levels

could alleviate this limitation. However, its benefit should be carefully gauged due to the notable performance gap between single-GPU and multi-GPU.

HYPRE has shown that a high degree of freedom in setting the stencil geometry does not necessarily negatively impact the performance. Currently, `mgcl` only supports stencils that access the 26 closest neighbors. This could be adjusted to a higher radius or non-symmetric stencils, but it would demand a significant implementation effort. The amount of applicable problems could be further extended by adding support for mixed boundary conditions – currently only either Dirichlet (with $u = 0$ on $\partial\Omega$) or periodic boundary conditions are supported. Support for user-selected restriction and prolongation operators as stencils with fixed or spatially varying coefficients could also be added. Implementing further smoothers and/or cycle schemes might also yield a higher applicability of `mgcl`.

Since peak performance is subject to the correct choice of launch parameters of GPU kernels, an automatic tuning tool could be implemented to find the best configuration for a given hardware. Currently, `mgcl` only offers an API that allows a user to set the work-group sizes, but the optimal values must be found manually, e.g. by running the benchmarks provided with the source code.

`mgcl` is optimized for GPU usage. As OpenCL supports many different device types, `mgcl` could be extended to provide optimized support for other types, too. For some kernels, it might be as simple as choosing the correct launch parameters, while for others, a completely different implementation with different underlying memory layouts might be more suitable. Additionally, it would be interesting to see how `mgcl` performs on GPUs by other vendors, e.g. AMD and Intel. Furthermore, in order to support systems without access to OpenCL, OpenMP support could be added to `mgcl`.

Appendix A

Appendix

A.1. Data Access Patterns

kernel	l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	L1 sectors / requests
coalesced, 32 els	1	8	8
coalesced, 64 els	2	16	8
coalesced, 128 els	4	32	8
accessing first vector half	1	4	4
accessing first vector quarter	1	2	2

Table A.1.: Nsight Compute metrics for the coalesced memory access kernels executed on Quadro T2000

Offset	dram__sectors_read.sum [sector]	lltex__t_sector_hit_rate.pct [%]	lltex__t_requests_pipe_lsu_mem_global_op_ld.sum	lltex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	L1 sectors / request	lts__t_requests_srcunit_tex_op_read.sum [request]	lts__t_sector_hit_rate.pct [%]	lts__t_sectors_srcunit_tex_op_read.sum [sector]
0	4198584	0	1048576	4194304	4	1048576	50.02	4194304
1	4196296	38.82	1048576	4752747	4.53	1311196	54.41	4315018
2	4197174	38.82	1048576	4752240	4.53	1310642	54.4	4314910
3	4196166	38.8	1048576	4752976	4.53	1309209	54.34	4314891
4	4196372	38.82	1048576	4753727	4.53	1311174	54.38	4314879
5	4198649	38.81	1048576	4752342	4.53	1309918	54.38	4314900
6	4196372	38.8	1048576	4754011	4.53	1310933	54.36	4314676
7	4196378	38.79	1048576	4752514	4.53	1309415	54.39	4314896
8	4196225	29.01	1048576	3659600	3.49	1379705	53.94	4194304
9	4197845	36.52	1048576	4216401	4.02	1375810	53.93	4315295
10	4196383	36.56	1048576	4216805	4.02	1374962	53.93	4315020
11	4197569	36.57	1048576	4218069	4.02	1376519	53.94	4314904
12	4197328	36.53	1048576	4215435	4.02	1375571	53.92	4315065
13	4196375	36.58	1048576	4215787	4.02	1375168	53.95	4315051
14	4196336	36.55	1048576	4216773	4.02	1376905	53.9	4314902
15	4196314	36.55	1048576	4214657	4.02	1375889	53.94	4315233
16	4196559	25.31	1048576	3126306	2.98	1380428	50.02	4194304
17	4197439	30.65	1048576	4664792	4.45	1375709	54.38	4316212
18	4196959	30.63	1048576	4664108	4.45	1377668	54.38	4316169
19	4196368	30.63	1048576	4662723	4.45	1375076	54.37	4315930
20	4196242	30.62	1048576	4666325	4.45	1376528	54.37	4315909
21	4196359	30.64	1048576	4665510	4.45	1375102	54.37	4315948
22	4198656	30.63	1048576	4666257	4.45	1375653	54.37	4316232
23	4197128	30.63	1048576	4665276	4.45	1375695	54.37	4315777
24	4198769	18.26	1048576	3659492	3.49	1378562	53.94	4194304
25	4196075	25.03	1048576	5199070	4.96	1237357	53.99	4316053
26	4198246	25.03	1048576	5199048	4.96	1237554	53.99	4316144
27	4196355	25.09	1048576	5199024	4.96	1237263	53.98	4316113
28	4196963	25.06	1048576	5199016	4.96	1236396	54	4315989
29	4196163	25.06	1048576	5199179	4.96	1237147	53.99	4316063
30	4196364	25.03	1048576	5199543	4.96	1237494	53.99	4315893
31	4196361	25.06	1048576	5198595	4.96	1238610	54.02	4316106
32	4196189	0	1048576	4194304	4.00	1048576	50.01	4194304

Table A.2.: Nsight Compute metrics for the `offsetCopy`-kernel executed on Quadro T2000 with 2^{25} elements

A.2. Residual Kernel Profiling Data

Metric	Unit	Description
Speed of Light		
gpu__time_duration.sum	ms	Duration of the entire kernel
sm__throughput.avg.pct_of_peak_sustained_elapsed	%	Compute Throughput (% of peak sustainable)
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed	%	Memory Throughput (% of peak sustainable)
l1tex__throughput.avg.pct_of_peak_sustained_active	%	L1/TEX Cache Throughput (% of peak sustainable)
lts__throughput.avg.pct_of_peak_sustained_elapsed	%	L2 Cache Throughput (% of peak sustainable)
gpu__dram_throughput.avg.pct_of_peak_sustained_elapsed	%	DRAM Throughput (% of peak sustainable)
Compute Workload Analysis		
sm__instruction_throughput.avg.pct_of_peak_sustained_active	%	SM Busy
sm__inst_executed_pipe_fp64.avg.pct_of_peak_sustained_active	%	Number of warp instruction executed by fp64 pipe (double precision unit) (% of peak sustainable)
sm__inst_executed_pipe_lsu.avg.pct_of_peak_sustained_active	%	Number of warp instruction executed by lsu pipe (load/store unit) (% of peak sustainable)
sm__inst_executed_pipe_fma.avg.pct_of_peak_sustained_active	%	Number of warp instruction executed by fma pipe (processes e.g. IMUL, IMAD, FMAD instructions) (% of peak sustainable)
sm__inst_executed_pipe_alu.avg.pct_of_peak_sustained_active	%	Number of warp instruction executed by alu pipe (processes e.g. bit manipulation and logic instructions and integer instructions exlcuding IMUL and IMAD) (% of peak sustainable)
Memory Workload Analysis		
dram__sectors_read.sum	sector	Amount of sectors sent from DRAM
l1tex__data_pipe_lsu_wavefronts_mem_shared.sum	wavefront	Total number of shared memory wavefronts
l1tex__data_pipe_lsu_wavefronts_mem_shared.sum.pct_of_peak_sustained_elapsed	%	Percentage of peak sustainable shared memory throughput
l1tex__t_sector_hit_rate.pct	%	Overall hit rate for the L1/TEX cache
lts__t_sector_hit_rate.pct	%	Overall hit rate for the L2 cache
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum	sector	Number of sectors requested for global loads
l1tex__t_sectors_pipe_lsu_mem_global_op_st.sum	sector	Number of sectors requested for global stores
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	request	Number of requests for global loads
l1tex__t_requests_pipe_lsu_mem_global_op_st.sum	request	Number of requests for global stores
sass__inst_executed_local_loads	inst	Number of local memory load instructions executed
sass__inst_executed_local_stores	inst	Number of local memory store instructions executed
Scheduler statistics		
sm__warps_active.avg.pct_of_peak_sustained_active	%	Percentage of peak sustainable active warps per scheduler
smsp__warps_active.avg.peak_sustained	warp	GPU maximum warps per scheduler Max. Warps per Scheduler
smsp__maximum_warps_avg_per_active_cycle	warp	Theoretical warps per schedulerTheoretical Warps per Scheduler
smsp__warps_active.avg.per_cycle_active	warp	Active warps per scheduler
smsp__warps_eligible.avg.per_cycle_active	warp	Eligible warps per scheduler
smsp__issue_active.avg.per_cycle_active	warp	Issued warps per scheduler

Table A.3.: Excerpt of metrics in Nsight Compute (Part 1). Local memory refers to CUDA local memory.

Metric	Unit	Description
Warp State Statistics		
smsp__average_warp_latency_per_inst_issued.ratio	cycles	Warp cycles per issued instruction
smsp__thread_inst_executed_per_inst_executed.ratio	threads	Average active threads per warp
smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio	warps	Avg. number of resident warps per issue cycle stalled for a scoreboard dependency on L1TEX
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio	warps	Avg. number of resident warps per issue cycle stalled for a free slot in the LSU instruction queue
Occupancy		
device__attribute_max_blocks_per_multiprocessor	block	GPU block limit per SM
device__attribute_max_registers_per_block	register	GPU register limit per block
device__attribute_max_registers_per_multiprocessor	register	GPU register limit per SM
device__attribute_max_shared_memory_per_block	byte	GPU shared memory default limit per block
device__attribute_max_shared_memory_per_multiprocessor	byte	GPU shared memory limit per SM
device__attribute_max_threads_per_block	thread	GPU thread limit per block
device__attribute_max_threads_per_multiprocessor	warp	GPU thread limit per SM
device__attribute_max_warps_per_scheduler	warp	GPU warp limit per scheduler
launch__occupancy_limit_blocks	block	Occupancy limit due to maximum number of blocks managable per SM
launch__occupancy_limit_registers	block	Occupancy limit due to register usage
launch__occupancy_limit_shared_mem	block	Occupancy limit due to shared memory usage
launch__occupancy_limit_warps	block	Occupancy limit due to number of warps, i.e. block size
launch__registers_per_thread	register	Number of registers per thread actually used
launch__registers_per_thread_allocated	register	Number of registers per thread allocated by the GPU
sm__maximum_warps_per_active_cycle_pct	%	Theoretical occupancy
sm__maximum_warps_avg_per_active_cycle	warps	Theoretical active warps per SM
sm__warps_active.avg.pct_of_peak_sustained_active	%	Achieved occupancy
sm__warps_active.avg.per_cycle_active	warps	Achieved active warps per SM
derived__pct_occupancy_per_register_count	%	Occupancy limit due to Register usage in percent
derived__pct_occupancy_per_block_size	%	Occupancy limit due to block size in percent
derived__pct_occupancy_per_shared_mem_size	%	Occupancy limit due to shared memory usage in percent
Source Counters		
derived__memory_l2_theoretical_sectors_global_excessive	sector	Number of excessive L2 sectors due to uncoalesced global memory access

Table A.4.: Excerpt of metrics in Nsight Compute (Part 2). Local memory refers to CUDA local memory.

metric	auto-wg (subopt idx)	4x4x4 (subopt idx)	auto-wg (opt idx)	4x4x4 (opt idx)
gpu__time_duration.sum [ms]	108.13	43.97	3.91	14.42
sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	1.73	3.70	47.72	11.25
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	41.21	39.46	91.52	47.26
derived__memory_l2_theoretical_sectors_global_excessive [Mbyte]	87,16 {59}	39,56 {59}	2,28 {59}	20,16 {59}
dram__sectors_read.sum [sector]	178585572	72396344	17338520	32965184
l1tex__t_sector_hit_rate.pct [%]	11.29	37.49	30.20	40.79
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	115052854	67619968	31266985	48582101
sm__instruction_throughput.avg.pct_of_peak_sustained_active [%]	1.73	3.68	47.72	11.23
sm__warps_active.avg.pct_of_peak_sustained_active [%]	63.57	63.66	87.41	74.20
smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	1759.15	745.82	42.38	281.80
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	200.78	168.34	62.64	84.83
smsp__issue_active.avg.per_cycle_active	0.00	0.00	0.06	0.01
smsp__thread_inst_executed_per_inst_executed.ratio	25.69	30.17	25.69	30.17

Table A.5.: Nsight Compute metrics for the varying stencil residual kernel on Quadro T2000, launched as 3d `ndrange` on a real grid of size 128^3 and $gh = 1$. Compares manually chosen work-group sizes of 4x4x4 to automatically chosen ones by OpenCL. Suboptimal indexing refers to the work-item mapping (i, j, k), optimal indexing refers to the work-item mapping (k, j, i).

variant	8x1x1	16x1x1	32x1x1	64x1x1	96x1x1	128x1x1	256x1x1
gpu__time_duration.sum [ms]	6.70	3.77	3.75	4.00	3.97	3.94	3.82
sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	94.67	88.98	49.68	46.57	46.89	47.45	48.72
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	51.04	90.61	91.52	91.04	91.35	91.47	91.70
dram__sectors_read.sum [sector]	16541902	16541124	16617810	17667078	17558896	17458080	16946068
l1tex__t_sector_hit_rate.pct [%]	32.66	33.78	27.09	27.78	29.14	30.29	32.63
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	39825327	34111577	31875372	31577984	31472573	31454886	31199044
sm__instruction_throughput.avg.pct_of_peak_sustained_active [%]	94.77	89.00	49.75	46.78	47.11	47.44	48.92
sm__warps_active.avg.pct_of_peak_sustained_active [%]	48.98	48.98	49.46	81.75	84.57	82.71	80.93
smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	0.04	0.07	6.39	37.01	38.07	28.25	15.59
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	29.71	32.28	53.14	56.69	58.65	57.56	70.80
smsp__issue_active.avg.per_cycle_active	0.12	0.11	0.06	0.06	0.06	0.07	0.07
smsp__thread_inst_executed_per_inst_executed.ratio	7.64	14.63	27.07	27.29	27.29	27.67	27.67

Table A.6.: Nsight Compute metrics for the varying stencil residual kernel on Quadro T2000, launched as 3d ndrange on a real grid of size 128^3 and $gh = 1$. Compares different 1d work-group sizes.

wg size	32x1x1	64x1x1	128x1x1	256x1x1
gpu__time_duration.sum [ms]	5.1	6.77	6.53	6.95
sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	29.86	22.46	23.3	21.98
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	92.26	91.82	91.15	90.16
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	3736497	3736498	3736500	3736504
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	70869228	70600185	70632094	69997457
L1 sectors / requests	18.97	18.89	18.90	18.73
l1tex__t_sector_hit_rate.pct [%]	41.54	37.61	38.08	36.29
lts__t_sector_hit_rate.pct [%]	71.94	63.17	64.03	63.65
smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	62.22	176.38	176.4	180.32
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	35.74	46.39	44.68	37.73

Table A.7.: Nsight Compute metrics for the varying stencil residual kernel with grid-first layout on Quadro T2000, launched as 3d `ndrange` on a real grid of size 128^3 and $gh = 1$.

wg size	16	32	64	128	256
gpu__time_duration.sum [ms]	0.46	0.61	0.67	0.67	0.67
sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	21.51	8.88	7.87	8.06	8.2
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	71.31	94.94	94.17	93.82	93.85
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	8110080	4505600	4505600	4505600	4505600
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	73567716	69895794	69576514	69572047	69474433
L1 sectors / requests	9.07	15.51	15.44	15.44	15.42
l1tex__t_sector_hit_rate.pct [%]	62.81	32.76	27.2	27.86	28.69
lts__t_sector_hit_rate.pct [%]	64.07	76	75.95	76.09	75.76
sm__instruction_throughput.avg.pct_of_peak_sustained_active [%]	11.12	4.65	4.47	4.94	4.94
sm__maximum_warps_per_active_cycle_pct [%]	50	50	100	100	100
sm__warps_active.avg.pct_of_peak_sustained_active [%]	48.61	49.21	75.88	78.83	84.57
smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	1.88	18.71	99.2	99.44	124.89
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	63.89	137.54	134.25	120.8	117.23

Table A.8.: Nsight Compute metrics for the varying stencil residual kernel with grid-first layout on A100, launched as 3d `ndrange` on a real grid of size 128^3 and $gh = 1$.

wg size	16	32	64	128	256
gpu__time_duration.sum [ms]	3.83	3.89	4.33	4.24	4.08
sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	79.19	39	35.32	35.85	37.16
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	89.41	89.47	88.58	89	89.81
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	7362969	3685005	3685006	3685008	3685008
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	31856669	30708220	30469771	30323459	30028340
L1 sectors / requests	4.33	8.33	8.27	8.23	8.15
l1tex__t_sector_hit_rate.pct [%]	33.72	25.27	23.48	24.4	32.25
lts__t_sector_hit_rate.pct [%]	25.97	28.96	24.68	24.52	18.49
smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	0.03	9.65	74.59	74.03	39.45
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	36.43	67.57	66.53	67.36	78.94
sm__instruction_throughput.avg.pct_of_peak_sustained_active [%]	79.37	39.04	35.41	35.75	37.21

Table A.9.: Nsight Compute metrics for the varying stencil residual kernel on Quadro T2000, launched as 3d `ndrange` on a real grid of size 128^3 and $gh = 1$ with 4 grid points per work-item.

wg size	16	32	64	128	256
gpu__time_duration.sum [ms]	0.45	0.4	0.39	0.39	0.39
sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	15.22	8.81	9.14	9.08	8.75
gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	73.36	83.84	84.76	84.93	85.41
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	7328640	3667840	3667840	3667840	3667840
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	32236596	30708852	29769505	29573006	29447497
L1 sectors / requests	4.40	8.37	8.12	8.06	8.03
l1tex__t_sector_hit_rate.pct [%]	31.55	27.05	29.51	32.08	39.53
lts__t_sector_hit_rate.pct [%]	41.95	40.39	39.13	38.23	33.66
smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	0.22	1.8	7.6	7.04	3.52
smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	49.97	88.56	102.18	102.12	104.79
sm__instruction_throughput.avg.pct_of_peak_sustained_active [%]	14.2	8.14	8.23	8.26	8.34
sm__maximum_warps_per_active_cycle_pct [%]	50	50	62.5	62.5	62.5
sm__warps_active.avg.pct_of_peak_sustained_active [%]	48.16	47.99	58.86	58.71	59.01

Table A.10.: Nsight Compute metrics for the varying stencil residual kernel on A100, launched as 3d `ndrange` on a real grid of size 128^3 and $gh = 1$ with 4 grid points per work-item.

A.3. Jacobi Blocking

Grid	Tile	#yz-planes	Time [ms] Quadro T2000	A100	Grid	Tile	#yz-planes	Time [ms] Quadro T2000	A100
8	4x4	8	0.27	0.15	64	8x8	64	1.93	0.77
8	4x4	4	0.18	0.11	64	8x8	32	2.71	0.46
8	4x4	2	0.14	0.10	64	8x8	16	3.16	0.30
8	4x4	1	0.12	0.08	64	8x8	8	3.93	0.41
8	4x8	8	0.27	0.15	64	8x16	64	1.73	0.87
8	4x8	4	0.18	0.11	64	8x16	32	2.05	0.47
8	4x8	2	0.14	0.09	64	8x16	16	2.32	0.32
8	4x8	1	0.12	0.08	64	8x16	8	2.65	0.39
8	8x8	8	0.20	0.12	64	16x16	64	1.95	1.02
8	8x8	4	0.14	0.09	64	16x16	32	2.40	0.55
8	8x8	2	0.11	0.08	64	16x16	16	2.89	0.32
8	8x8	1	**0.09**	**0.07**	64	16x16	8	3.29	0.38
16	4x4	16	0.49	0.24	64	4x32	64	**1.59**	0.69
16	4x4	8	0.31	0.16	64	4x32	32	1.87	0.38
16	4x4	4	0.24	0.11	64	4x32	16	2.03	0.26
16	4x4	2	0.25	0.09	64	4x32	8	2.36	0.31
16	4x8	16	0.50	0.24	64	8x32	64	1.69	0.85
16	4x8	8	0.31	0.16	64	8x32	32	1.92	0.46
16	4x8	4	0.22	0.12	64	8x32	16	2.10	0.27
16	4x8	2	0.19	0.09	64	8x32	8	2.39	0.33
16	8x8	16	0.38	0.19	64	4x64	64	1.62	0.74
16	8x8	8	0.24	0.13	64	4x64	32	1.72	0.41
16	8x8	4	0.17	0.10	64	4x64	16	1.88	0.25
16	8x8	2	0.15	0.08	64	4x64	8	2.20	0.29
16	8x16	16	0.39	0.19	64	2x32	64	1.81	0.56
16	8x16	8	0.24	0.13	64	2x32	32	2.05	0.35
16	8x16	4	0.17	0.09	64	2x32	16	2.22	**0.23**
16	8x16	2	**0.14**	**0.08**	64	2x32	8	2.52	0.30
16	16x16	16	0.57	0.29	64	2x64	64	1.66	0.60
16	16x16	8	0.35	0.18	64	2x64	32	1.70	0.34
16	16x16	4	0.23	0.12	64	2x64	16	1.84	0.23
16	16x16	2	0.18	0.09	64	2x64	8	2.15	0.27
32	4x4	32	1.01	0.41	128	4x4	128	66.30	3.43
32	4x4	16	0.90	0.25	128	4x4	64	70.67	3.06
32	4x4	8	1.06	0.16	128	4x4	32	76.52	2.87
32	4x4	4	1.40	0.12	128	4x4	16	84.11	3.45
32	4x8	32	0.94	0.42	128	4x8	128	39.54	2.12
32	4x8	16	0.62	0.25	128	4x8	64	42.01	2.46
32	4x8	8	0.59	0.16	128	4x8	32	45.39	2.07
32	4x8	4	0.82	0.12	128	4x8	16	49.70	2.13
32	8x8	32	0.72	0.34	128	8x8	128	31.87	1.71
32	8x8	16	0.51	0.20	128	8x8	64	33.26	1.90
32	8x8	8	0.47	0.13	128	8x8	32	35.38	1.61
32	8x8	4	0.64	0.11	128	8x8	16	37.88	1.78
32	8x16	32	0.79	0.38	128	8x16	128	18.68	1.78
32	8x16	16	0.50	0.22	128	8x16	64	19.40	1.80
32	8x16	8	0.41	0.14	128	8x16	32	20.17	1.52
32	8x16	4	0.47	0.10	128	8x16	16	21.63	1.61
32	16x16	32	0.79	0.57	128	16x16	128	26.60	2.01
32	16x16	16	0.49	0.33	128	16x16	64	28.57	2.16
32	16x16	8	0.35	0.20	128	16x16	32	29.43	1.73
32	16x16	4	0.48	0.14	128	16x16	16	30.78	1.69
32	4x32	32	0.53	0.33	128	4x32	128	13.83	1.66
32	4x32	16	0.34	0.20	128	4x32	64	14.34	1.69
32	4x32	8	**0.31**	0.13	128	4x32	32	15.03	1.42
32	4x32	4	0.39	0.09	128	4x32	16	16.64	1.38
32	8x32	32	0.71	0.42	128	8x32	128	15.33	1.67
32	8x32	16	0.44	0.24	128	8x32	64	15.75	1.83
32	8x32	8	0.32	0.16	128	8x32	32	16.47	1.50
32	8x32	4	0.40	0.11	128	8x32	16	17.85	1.47
32	2x32	32	0.47	0.27	128	4x64	128	13.43	1.56
32	2x32	16	0.34	0.17	128	4x64	64	13.86	1.68
32	2x32	8	0.33	0.11	128	4x64	32	14.58	1.37
32	2x32	4	0.41	**0.09**	128	4x64	16	16.08	1.30
64	4x4	64	4.41	0.92	128	2x32	128	15.21	1.46
64	4x4	32	5.95	0.57	128	2x32	64	15.60	1.57
64	4x4	16	7.35	0.69	128	2x32	32	16.46	1.32
64	4x4	8	9.43	0.63	128	2x32	16	18.23	1.27
64	4x8	64	2.63	0.99	128	2x64	128	**12.95**	1.55
64	4x8	32	3.59	0.55	128	2x64	64	13.41	1.54
64	4x8	16	4.16	0.37	128	2x64	32	14.22	1.27
64	4x8	8	5.50	0.49	128	2x64	16	15.77	**1.2**

Table A.11.: Durations of our prototype blocking implementation of Jacobi's method, performing 2 Jacobi iterations

tile	#yz-planes	gpu__time_duration.sum [µs]	sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_ld.sum	l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]
4x4	8	274.82	4.67	1.05	3240	12560	39036
4x4	4	180.58	8.97	1.89	4176	15872	48590
4x4	2	133.82	17.02	3.2	6048	22496	67693
4x4	1	109.57	32.69	5.87	9792	35744	107180
4x8	8	273.25	2.36	0.93	0	6280	24288
4x8	4	179.36	4.49	1.39	0	7936	30131
4x8	2	132.74	8.58	2.33	0	11248	41848
4x8	1	110.21	16.41	3.88	0	17872	65236
8x8	8	200.8	2.22	1.2	2430	4354	27440
8x8	4	129.98	4.31	1.91	2988	5476	33763
8x8	2	95.49	8.2	3.05	4104	7720	46355
8x8	1	78.11	15.78	4.93	6336	12208	71614

Table A.12.: NCU metrics for our prototype blocking implementation for grid size 8^3, 2 Jacobi iterations, executed on a Quadro T2000 (part 1)

tile	#yz-planes	sm__instruction_throughput.avg.pct_of_peak_sustained_active [%]	sm__maximum_warps_per_active_cycle_pct [%]	sm__warps_active.avg.pct_of_peak_sustained_active [%]	smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	smsp__inst_executed.sum [inst]	smsp__inst_executed_op_branch.sum [inst]	smsp__sass_average_branch_targets_threads_uniform.pct [%]
4x4	8	18.88	25	3.13	0	7.75	100000	6500	76.11
4x4	4	18.14	25	3.13	0	7.57	128128	8328	75.66
4x4	2	17.23	25	3.12	0	7.48	184384	11984	75.16
4x4	1	33.32	25	6.24	0	7.24	296896	19296	74.73
4x8	8	18.89	25	3.12	0	7.52	50052	3302	73.54
4x8	4	18.15	25	3.12	0	7.4	64136	4236	72.87
4x8	2	17.37	25	3.13	0	7.16	92304	6104	72.14
4x8	1	16.59	25	3.13	0	7.07	148640	9840	71.51
8x8	8	35.94	25	6.25	0	7.97	35170	2516	77.39
8x8	4	34.73	25	6.24	0	7.62	44892	3224	76.72
8x8	2	33.47	25	6.26	0	7.36	64336	4640	75.99
8x8	1	32.08	25	6.25	0	7.14	103224	7472	75.35

Table A.13.: NCU metrics for our prototype blocking implementation for grid size 8^3, 2 Jacobi iterations, executed on a Quadro T2000 (part 2)

tile	#yz-planes	gpu__time_duration.sum [ms]	sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_ld.sum	l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]
4x4	8	0.94	69.22	18.14	162432	644864	2017152
4x4	4	0.83	84.42	22.43	177408	697856	2142511
4x4	2	1.35	59.26	20.62	207360	803840	2473657
4x4	1	1.37	73.95	38.58	267264	1015808	3124584
4x8	8	0.9	36.22	14.38	0	322432	1316551
4x8	4	0.58	61.89	24.43	0	348928	1426799
4x8	2	0.54	75.3	33.57	0	401920	1627370
4x8	1	0.87	53.98	50.76	163	507904	2030931
8x8	8	0.74	30.68	16.39	128736	224800	1412673
8x8	4	0.53	46.97	29.17	137664	242752	1512913
8x8	2	0.51	56.27	56.21	155520	278656	1719108
8x8	1	0.7	51.09	63.32	191453	350464	2116506
8x16	8	0.71	25.13	12.02	123120	175984	1243257
8x16	4	0.43	45.29	26.98	131040	189664	1328890
8x16	2	0.34	65.02	40.96	146880	217024	1495606
8x16	1	0.56	48.27	47.83	178560	271744	1833745
16x16	8	1.09	14.1	9.21	360936	151576	1673615
16x16	4	0.61	27.13	18.91	383184	163120	1783719
16x16	2	0.39	48.72	38.76	427680	186208	1998344
16x16	1	0.53	44.76	57.68	516672	232384	2431384
4x32	8	0.7	25.72	11.54	0	175984	1010194
4x32	4	0.41	46.89	28.39	0	189664	1079240
4x32	2	0.34	65.39	41.83	0	217024	1215095
4x32	1	0.53	51.69	37.12	245	271744	1493481
8x32	8	0.98	15.73	7.39	120312	151576	1172325
8x32	4	0.55	30.03	21.06	127728	163120	1250017
8x32	2	0.34	53.42	39.52	142560	186208	1404131
8x32	1	0.45	50.94	38.31	172224	232384	1709838
2x32	8	0.67	34.19	14.71	117504	222432	1007106
2x32	4	0.46	53.28	27.92	124416	240064	1076158
2x32	2	0.4	69.26	44.62	138282	275328	1222055
2x32	1	0.53	66.81	45.35	166064	345856	1521321

Table A.14.: NCU metrics for our prototype blocking implementation for grid size 32^3, 2 Jacobi iterations, executed on a Quadro T2000 (part 1)

tile	#yz-planes	sm__instruction_throughput.avg.pct_of_peak_sustained_elapsed [%]	sm__maximum_warps_per_active_cycle_pct [%]	sm__warps_active.avg.pct_of_peak_sustained_active [%]	smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	smsp__inst_executed.sum [inst]	smsp__inst_executed_op_branch.sum [inst]	smsp__sass_average_branch_targets_threads_uniform.pct [%]
4x4	8	69.22	37.5	12.5	0.01	10.01	5020800	328256	76.67
4x4	4	84.42	37.5	24.93	0.09	18.74	5468416	357504	76.45
4x4	2	59.26	37.5	31.74	0.03	25.93	6363648	416000	76.11
4x4	1	73.95	37.5	35.48	0.15	28.84	8154112	532992	75.66
4x8	8	36.22	37.5	6.25	0.04	8.96	2512768	166496	74.35
4x8	4	61.89	37.5	12.45	0.19	11.08	2736896	181440	74.04
4x8	2	75.3	37.5	24.76	0.46	19.77	3185152	211328	73.54
4x8	1	53.98	37.5	32.71	1.21	27.98	4081664	271104	72.87
8x8	8	30.68	37.5	8.43	0.31	10.05	1774704	127040	78.2
8x8	4	46.97	37.5	12.88	0.44	12.23	1929440	138368	77.89
8x8	2	56.27	37.5	25.08	1.57	19.75	2238912	161024	77.39
8x8	1	51.09	37.5	31.55	2.65	23.47	2857856	206336	76.72
8x16	8	25.13	37.5	12.44	0.55	10.64	1405080	106720	82.03
8x16	4	45.29	37.5	12.5	0.63	11.85	1525040	116160	81.77
8x16	2	65.02	37.5	24.74	1.02	17.59	1764960	135040	81.34
8x16	1	48.27	37.5	35.72	1.11	24.05	2244800	172800	80.76
16x16	8	14.1	25	25	4.03	17.05	1219512	95972	85.2
16x16	4	27.13	25	24.91	4.35	17.35	1321968	104392	84.99
16x16	2	48.72	25	24.91	5.31	17.93	1526880	121232	84.65
16x16	1	44.76	25	24.54	5.54	19.16	1936704	154912	84.19
4x32	8	25.72	37.5	12.5	0.28	10.43	1403896	105536	83.81
4x32	4	46.89	37.5	12.52	0.35	11.26	1523696	114816	83.62
4x32	2	65.39	37.5	24.99	0.76	16.76	1763296	133376	83.32
4x32	1	51.69	37.5	36.28	0.96	22.21	2242496	170496	82.92
8x32	8	15.73	25	24.98	1.57	17.96	1220268	96560	84.51
8x32	4	30.03	25	25	1.64	18.57	1322840	105056	84.27
8x32	2	53.42	25	24.92	2.89	19.49	1527984	122048	83.9
8x32	1	50.94	25	24.45	2.56	19.56	1938272	156032	83.39
2x32	8	34.19	37.5	7.04	0.02	10.07	1668048	116544	84.89
2x32	4	53.28	37.5	12.59	0.08	11.9	1808288	126592	84.74
2x32	2	69.26	37.5	24.99	0.33	19.9	2088768	146688	84.49
2x32	1	66.81	37.5	32.07	0.48	24.62	2649728	186880	84.15

Table A.15.: NCU metrics for our prototype blocking implementation for grid size 32^3, 2 Jacobi iterations, executed on a Quadro T2000 (part 2)

tile	#yz-planes	gpu__time_duration.sum [ms]	sm__throughput.avg.pct_of_peak_sustained_elapsed [%]	gpu__compute_memory_throughput.avg.pct_of_peak_sustained_elapsed [%]	l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_ld.sum	l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]
4x4	8	5.83	85.21	24.68	1239552	4946944	15583973
4x4	4	6.69	78.97	39.73	1299456	5158912	16394882
4x4	2	8.39	67.08	46.87	1419264	5582848	17729420
4x4	1	10.75	60.72	49.06	1658880	6430720	20246131
4x8	8	3.1	80.55	32.67	0	2473472	10243177
4x8	4	3.74	69.54	55.46	0	2579456	10675521
4x8	2	4.59	62.57	59.67	0	2791424	11536327
4x8	1	5.93	54.31	60.42	0	3215360	13156425
8x8	8	2.45	70.57	55.46	994176	1726592	10922187
8x8	4	2.86	63.69	67.32	1029888	1798400	11325603
8x8	2	3.36	58.02	69.86	1101312	1942016	12114022
8x8	1	4.22	53.89	69.28	1244160	2229248	13714363
8x16	8	2.03	67.69	42.52	953280	1353152	9672894
8x16	4	2.28	63.61	58.7	984960	1407872	10003941
8x16	2	2.61	60	62.94	1048320	1517312	10670717
8x16	1	3.04	57.16	59.5	1175040	1736192	12006924
16x16	8	2.3	48.65	37.43	2798496	1166432	12944456
16x16	4	2.65	46.8	53.09	2887488	1212608	13392529
16x16	2	2.92	44.45	55.35	3065472	1304960	14257125
16x16	1	3.41	43.89	54.38	3421440	1489664	15985605
4x32	8	1.94	70.74	44.16	0	1353152	7807284
4x32	4	2.17	65.78	54.82	0	1407872	8086658
4x32	2	2.38	64.67	54.44	0	1517312	8636514
4x32	1	2.74	63.39	52.9	0	1736192	9737982
8x32	8	2.09	56.85	39.71	932832	1166432	8986267
8x32	4	2.3	54.14	49.2	962496	1212608	9291396
8x32	2	2.5	52.67	48.41	1021824	1304960	9900044
8x32	1	2.93	51.17	47.32	1140480	1489664	11111980
4x64	8	2.05	57.68	41.65	0	1166432	7426360
4x64	4	2.17	56.46	42.46	0	1212608	7683550
4x64	2	2.37	55.88	43.36	0	1304960	8204442
4x64	1	2.76	53.12	41.85	0	1489664	9242891
2x32	8	2.13	82.61	53.16	912517	1708928	7956081
2x32	4	2.31	78.55	65.32	941041	1779456	8255506
2x32	2	2.52	79.23	64.03	996289	1920512	8847399
2x32	1	2.95	76.33	62.35	1107004	2202624	10026821
2x64	8	1.9	72.25	53.83	912384	1339904	7389405
2x64	4	1.99	71.38	52.15	940032	1393664	7659313
2x64	2	2.19	70.75	53	995328	1501184	8197714
2x64	1	2.6	67.48	53.32	1105920	1716224	9268922

Table A.16.: NCU metrics for our prototype blocking implementation for grid size 64^3, 2 Jacobi iterations, executed on a Quadro T2000 (part 1)

tile	#yz-planes	sm__instruction_throughput.avg.pct_of_peak_sustained_elapsed [%]	sm__maximum_warps_per_active_cycle_pct [%]	sm__warps_active.avg.pct_of_peak_sustained_active [%]	smsp__average_warps_issue_stalled_lg_throttle_per_issue_active.ratio [inst]	smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.ratio [inst]	smsp__inst_executed.sum [inst]	smsp__inst_executed_op_branch.sum [inst]	smsp__sass_average_branch_targets_threads_uniform.pct [%]
4x4	8	85.21	25	24.53	0.03	18.65	38598656	2509056	76.79
4x4	4	78.97	25	24.95	0.05	20.6	40398848	2626048	76.67
4x4	2	67.08	25	24.93	0.14	23.49	43999232	2860032	76.45
4x4	1	60.72	25	24.56	0.18	25.44	51200000	3328000	76.11
4x8	8	80.55	25	25.01	0.13	20.06	19316992	1272192	74.53
4x8	4	69.54	25	24.21	0.13	23.07	20218368	1331968	74.35
4x8	2	62.57	25	24.89	0.27	25.76	22021120	1451520	74.04
4x8	1	54.31	25	24.5	0.43	29.09	25626624	1690624	73.54
8x8	8	70.57	25	25.65	0.55	21.6	13651584	971008	78.38
8x8	4	63.69	25	24.7	0.71	23.22	14273792	1016320	78.2
8x8	2	58.02	25	24.96	0.81	25.15	15518208	1106944	77.89
8x8	1	53.89	25	24.75	1.01	26.23	18007040	1288192	77.39
8x16	8	67.69	25	24.7	0.71	17.83	10814464	816000	82.19
8x16	4	63.61	25	25.1	0.73	19.03	11296768	853760	82.03
8x16	2	60	25	24.79	0.75	19.63	12261376	929280	81.77
8x16	1	57.16	25	24.92	0.69	19.29	14190592	1080320	81.34
16x16	8	48.65	25	25.37	5.56	18.38	9390320	734096	85.32
16x16	4	46.8	25	24.85	5.81	19.58	9802208	767776	85.2
16x16	2	44.45	25	24.71	6.05	20.56	10625984	835136	84.99
16x16	1	43.89	25	25.13	6.13	21.26	12273536	969856	84.65
4x32	8	70.74	25	24.91	0.39	18.12	10805632	807168	83.92
4x32	4	65.78	25	25.12	0.34	18.71	11287296	844288	83.81
4x32	2	64.67	25	25.09	0.33	18.6	12250624	918528	83.62
4x32	1	63.39	25	24.77	0.33	18.7	14177280	1067008	83.32
8x32	8	56.85	25	24.99	2.67	19.17	9395904	738496	84.65
8x32	4	54.14	25	25.17	2.44	19.95	9808256	772480	84.51
8x32	2	52.67	25	24.92	2.33	20.08	10632960	840448	84.27
8x32	1	51.17	25	25.13	2.38	20.26	12282368	976384	83.9
4x64	8	57.68	25	25.07	1.21	21.08	9387072	729664	86.54
4x64	4	56.46	25	25	1.12	20.95	9798784	763008	86.46
4x64	2	55.88	25	24.99	1.08	21.11	10622208	829696	86.31
4x64	1	53.12	25	24.87	1.14	21.41	12269056	963072	86.07
2x32	8	82.61	25	24.8	0.23	20.15	12830976	892160	84.98
2x32	4	78.55	25	24.73	0.36	20.76	13394432	932352	84.89
2x32	2	79.23	25	24.85	0.34	20.72	14521344	1012736	84.74
2x32	1	76.33	25	24.79	0.39	20.9	16775168	1173504	84.49
2x64	8	72.25	25	25	0.17	19.14	10232256	745984	89.41
2x64	4	71.38	25	24.77	0.18	18.92	10673024	779264	89.36
2x64	2	70.75	25	25.09	0.18	19.07	11554560	845824	89.26
2x64	1	67.48	25	25	0.19	19.26	13317632	978944	89.1

Table A.17.: NCU metrics for our prototype blocking implementation for grid size 64^3, 2 Jacobi iterations, executed on a Quadro T2000 (part 2)

A.4. Residual With Fixed Coefficients

```
residual_27point_fixed_stencil_
  coeffs_preproc
// ...
LDG.E.64.CONSTANT.SYS R60, [R54+-0x8]
LDG.E.64.CONSTANT.SYS R50, [R50+-0x8]
LDG.E.64.CONSTANT.SYS R6, [R6]
DMUL R4, R4, -9
DFMA R4, R48, -10, R4
DFMA R4, R46, -4, R4
DFMA R4, R44, -12, R4
DFMA R4, R42, -22, R4
DFMA R4, R40, -14, R4
DFMA R2, R38, -26, R2
DFMA R4, R36, -16, R4
DFMA R2, R34, -2, R2
DFMA R4, R32, -13, R4
DFMA R2, R30, -18, R2
DFMA R4, R28, -3, R4
DFMA R2, R26, -20, R2
DFMA R4, R24, -15, R4
DFMA R2, R22, -7, R2
DFMA R4, R20, -17, R4
DFMA R2, R18, -25, R2
DFMA R4, R16, -11, R4
DFMA R2, R14, -8, R2
DFMA R4, R12, -5, R4
DFMA R2, R10, -19, R2
DFMA R4, R8, -23, R4
DFMA R2, R58, -6, R2
DFMA R4, R60, -21, R4
DFMA R2, R50, -24, R2
DADD R4, -R6, R4
DADD R2, R4, R2
IMAD.WIDE R4, R53, R0, c[0x0][0x170]
STG.E.64.SYS [R4], R2
EXIT
```

(a) Coefficients as preprocessor constants

```
residual_27point_fixed_stencil_
  coeffs_as_global_buffer
// ...
LDG.E.64.CONSTANT.SYS R46, [R20+0x8]
LDG.E.64.CONSTANT.SYS R10, [R10]
DMUL R48, R48, R44
LDG.E.64.CONSTANT.SYS R44, [UR4+0x88]
DFMA R54, R50, R54, R48
LDG.E.64.CONSTANT.SYS R48, [UR4+0x60]
LDG.E.64.CONSTANT.SYS R50, [R6+-0x8]
DFMA R52, R56, R52, R54
LDG.E.64.CONSTANT.SYS R54, [UR4+0xb8]
LDG.E.64.CONSTANT.SYS R56, [R8+0x8]
DFMA R12, R14, R12, R52
LDG.E.64.CONSTANT.SYS R14, [R20+-0x8]
LDG.E.64.CONSTANT.SYS R52, [UR4+0x78]
DFMA R18, R18, R22, R12
LDG.E.64.CONSTANT.SYS R12, [UR4+0x20]
IADD3 R5, R5, c[0x0][0x188], RZ
LDG.E.64.CONSTANT.SYS R6, [R6+0x8]
DFMA R24, R26, R24, R18
LDG.E.64.CONSTANT.SYS R18, [UR4+0x70]
IMAD.WIDE R4, R5, R0, c[0x0][0x160]
LDG.E.64.CONSTANT.SYS R22, [UR4]
DFMA R28, R28, R30, R24
LDG.E.64.CONSTANT.SYS R20, [R16+-0x8]
IADD3 R25, R3, -c[0x0][0x188], R2
LDG.E.64.CONSTANT.SYS R26, [UR4+0x40]
IMAD.WIDE R24, R25, R0, c[0x0][0x160]
LDG.E.64.CONSTANT.SYS R30, [R4+0x8]
DFMA R34, R34, R32, R28
LDG.E.64.CONSTANT.SYS R28, [UR4+0x98]
LDG.E.64.CONSTANT.SYS R32, [R24]
DFMA R38, R36, R38, R34
LDG.E.64.CONSTANT.SYS R34, [UR4+0x30]
LDG.E.64.CONSTANT.SYS R36, [R4+-0x8]
DFMA R40, R40, R42, R38
LDG.E.64.CONSTANT.SYS R16, [R16+0x8]
LDG.E.64.CONSTANT.SYS R38, [UR4+0x10]
IMAD.WIDE R8, R59, 0x8, R8
LDG.E.64.CONSTANT.SYS R42, [R24+-0x8]
LDG.E.64.CONSTANT.SYS R4, [R4]
LDG.E.64.CONSTANT.SYS R60, [R8+-0x8]
LDG.E.64.CONSTANT.SYS R58, [UR4+0xc0]
DFMA R44, R44, R46, R40
LDG.E.64.CONSTANT.SYS R40, [UR4+0x90]
LDG.E.64.CONSTANT.SYS R46, [R8]
DFMA R50, R50, R48, R44
LDG.E.64.CONSTANT.SYS R44, [UR4+0xc8]
LDG.E.64.CONSTANT.SYS R48, [UR4+0x38]
DFMA R56, R54, R56, R50
LDG.E.64.CONSTANT.SYS R50, [UR4+0xa0]
LDG.E.64.CONSTANT.SYS R54, [R24+0x8]
DFMA R56, R52, R14, R56
LDG.E.64.CONSTANT.SYS R14, [UR4+0xd0]
LDG.E.64.CONSTANT.SYS R52, [R8+0x8]
IMAD.WIDE R24, R3, R0, c[0x0][0x168]
LDG.E.64.CONSTANT.SYS R24, [R24]
DFMA R10, R12, R10, R56
DFMA R6, R6, R18, R10
DFMA R6, R22, R20, R6
DFMA R6, R26, R30, R6
DFMA R6, R28, R32, R6
DFMA R6, R34, R36, R6
DFMA R6, R38, R16, R6
IMAD.WIDE R2, R3, R0, c[0x0][0x170]
DFMA R6, R40, R42, R6
DFMA R6, R44, R46, R6
DFMA R4, R48, R4, R6
DFMA R4, R50, R54, R4
DFMA R4, R14, R52, R4
DFMA R4, R58, R60, R4
DADD R4, R24, -R4
STG.E.64.SYS [R2], R4
EXIT
```

(b) Coefficients in global memory

Figure A.1.: Excerpts of the generated SASS code for the residual kernel with fixed coefficients on the Quadro T2000

A.5. Conversion Between Matrix and Stencil Notations

When working with matrices and their stencil notation, it is important to understand how to convert between the two representations. In order to successfully perform this conversion, information about the grid must be present. We restrict ourselves to square matrices having stencils whose location is on the same grid as the grid point the coefficients map to. A counterexample would be the restriction operator, which is located at a coarse grid point and takes values from the surrounding fine grid points.

Consider a 3d grid of size $m \times n \times o$, optionally plus ghost layers. The corresponding matrix $\mathbf{A}$ has size $mno \times mno$ and its corresponding stencil set $\hat{\mathbf{A}}$ has mno stencils $\hat{a}_{ijk}$ with width w_a in each dimension.

In order to create a matrix from a set of stencils, we iterate the grid points and the coefficients and calculate the row and column indices of the matrix that correspond to the current grid point and stencil coefficient, respectively. The row index is simply given by the flattened current grid point index, while the column index is determined by the grid point the current stencil coefficient maps to. Since a matrix does not use ghost layers, we need to respect periodic boundary conditions, if present, by wrapping around the grid point indices. Since it is possible that multiple coefficients of different stencils map to the same grid point, we need to use the `+=`-operator for a to accumulate the values. A possible implementation is shown in listing A.1.

In order to create a set of stencils from a matrix, we can first define a function `getStencilIndicesForEntry()` to query the stencil indices for a given matrix entry. This function needs the following inputs.

- Row and column index of the matrix entry,
- Size of the grid ($m \times n \times o$),
- Width of a stencil (must be a positive odd number),
- Periodic boundary conditions yes/no.

The concept is the same as for conversion in the opposite direction. We calculate the location of a stencil by transforming the 1d row index into a 3d grid point index. Similarly, we calculate the grid point the current matrix entry maps to by transforming the 1d column index into a 3d grid point index. The coefficient index is then given by the difference between the two grid point indices plus the radius of the stencil. A difference is that we need to check if the calculated coefficient indices are actually part of the stencil. If this is not the case, we return -1 for all

```
static Matrix2d fromVaryingStencil(const mgcl::VaryingStencil& s, bool periodic)
{
  int m = s.getM();
  int n = s.getN();
  int o = s.getO();
  Matrix2d c(m * n * o, m * n * o);
  int N2 = s.getWidth() >> 1;

  for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
  for (int k = 0; k < o; k++)
    for (int ii = 0; ii < s.getWidth(); ii++)
    for (int jj = 0; jj < s.getWidth(); jj++)
    for (int kk = 0; kk < s.getWidth(); kk++)
    {
      // Column index of the resulting matrix is equal to the grid point
      // the current stencil entry maps to. Therefore wrap grid point
      // indices around to account boundary conditions.
      int gpi = i + (ii - N2); // grid point index mapped to by stencil entry in x-direction
      int gpj = j + (jj - N2); // grid point index mapped to by stencil entry in y-direction
      int gpk = k + (kk - N2); // grid point index mapped to by stencil entry in z-direction

      if (periodic)
      {
        gpi = gpi % m;
        gpj = gpj % n;
        gpk = gpk % o;

        // shift mod result into positive range
        if (gpi < 0) gpi += m;
        if (gpj < 0) gpj += n;
        if (gpk < 0) gpk += o;
      }

      if (gpi >= 0 && gpi < m &&
          gpj >= 0 && gpj < n &&
          gpk >= 0 && gpk < o)
      {
        // row: current grid point
        // column: grid point the current stencil entry maps to (dependent on current grid point)
        c[i * n * o + j * o + k][gpi * n * o + gpj * o + gpk] +=
          s[ii][jj][kk][i + s.getGhostsM()][j + s.getGhostsN()][k + s.getGhostsO()];
      }
    }

  return c;
}
```

Listing A.1: Construction of a matrix from a set of varying stencils.

indices to indicate that the matrix entry is not part of any stencil. A possible implementation is shown in listing A.2. Using this function when iterating over all matrix entries allows us to construct a set of stencils from a matrix.

```
// Returns std::array<int, 6>: Indices of the stencil of the form {i, j, k, ii, jj, kk}, where i,j,k is the index of the grid point and ii,jj,kk is the index of the stencil entry.
std::array<int, 6> Matrix2d::getStencilIndicesForEntry(int row, int col,
                                                       int m, int n, int o,
                                                       int stencilWidth, bool periodic) const
{
    // Grid point mapped to by current matrix row (row number equals 1d index of grid point)
    int no_rows = n * o;
    int i = row / no_rows;
    int j = (row - i * no_rows) / o;
    int k = row % o;

    // Grid point the current stencil entry maps to (column number equals 1d index of grid point)
    int no = n * o;
    int gpi = col / no;
    int gpj = (col - gpi * no) / o;
    int gpk = col % o;

    // Stencil entry
    int ii = gpi + (stencilWidth / 2) - i;
    int jj = gpj + (stencilWidth / 2) - j;
    int kk = gpk + (stencilWidth / 2) - k;

    if (periodic)
    {
        ii = ii % m;
        jj = jj % n;
        kk = kk % o;

        // shift mod result into positive range
        if (ii < 0) ii += m;
        if (jj < 0) jj += n;
        if (kk < 0) kk += o;
    }

    // check if matrix entry is actually part of a stencil
    if (ii < 0 || ii >= stencilWidth || jj < 0 || jj >= stencilWidth || kk < 0 || kk >= stencilWidth)
        return {-1, -1, -1, -1, -1, -1};
    else
        return {i, j, k, ii, jj, kk};
}
```

Listing A.2: Querying stencil indices for a given matrix entry.

A.6. Overlapped Jacobi's method

GPUs	ν	Version	N_{local}	Time [ms]
2	1	default	32^3	0.16
2	1	(a)	32^3	0.17
2	1	(b)	32^3	0.16
2	1	(c)	32^3	0.17
2	2	default	32^3	0.23
2	2	(a)	32^3	0.25
2	2	(b)	32^3	0.24
2	2	(c)	32^3	0.26
2	3	default	32^3	0.31
2	3	(a)	32^3	0.34
2	3	(b)	32^3	0.32
2	3	(c)	32^3	0.36
2	4	default	32^3	0.39
2	4	(a)	32^3	0.43
2	4	(b)	32^3	0.41
2	4	(c)	32^3	0.45
2	5	default	32^3	0.47
2	5	(a)	32^3	0.52
2	5	(b)	32^3	0.49
2	5	(c)	32^3	0.55
2	1	default	64^3	0.38
2	1	(a)	64^3	0.40
2	1	(b)	64^3	0.37
2	1	(c)	64^3	0.36
2	2	default	64^3	0.58
2	2	(a)	64^3	0.59
2	2	(b)	64^3	0.60
2	2	(c)	64^3	0.56
2	3	default	64^3	0.80
2	3	(a)	64^3	0.83
2	3	(b)	64^3	0.82
2	3	(c)	64^3	0.77
2	4	default	64^3	1.02
2	4	(a)	64^3	1.06
2	4	(b)	64^3	1.04
2	4	(c)	64^3	0.97
2	5	default	64^3	1.23
2	5	(a)	64^3	1.28
2	5	(b)	64^3	1.26
2	5	(c)	64^3	1.16

GPUs	ν	Version	N_{local}	Time [ms]
2	1	default	128^3	1.23
2	1	(a)	128^3	1.30
2	1	(b)	128^3	1.29
2	1	(c)	128^3	0.91
2	2	default	128^3	2.02
2	2	(a)	128^3	2.11
2	2	(b)	128^3	2.12
2	2	(c)	128^3	1.45
2	3	default	128^3	2.81
2	3	(a)	128^3	2.99
2	3	(b)	128^3	3.01
2	3	(c)	128^3	1.96
2	4	default	128^3	3.63
2	4	(a)	128^3	3.88
2	4	(b)	128^3	3.85
2	4	(c)	128^3	2.48
2	5	default	128^3	4.41
2	5	(a)	128^3	4.74
2	5	(b)	128^3	4.70
2	5	(c)	128^3	2.99
2	1	default	256^3	5.74
2	1	(a)	256^3	5.96
2	1	(b)	256^3	5.94
2	1	(c)	256^3	4.73
2	2	default	256^3	10.10
2	2	(a)	256^3	10.49
2	2	(b)	256^3	10.48
2	2	(c)	256^3	8.07
2	3	default	256^3	14.44
2	3	(a)	256^3	15.07
2	3	(b)	256^3	15.04
2	3	(c)	256^3	11.42
2	4	default	256^3	18.81
2	4	(a)	256^3	19.60
2	4	(b)	256^3	19.60
2	4	(c)	256^3	14.71
2	5	default	256^3	23.14
2	5	(a)	256^3	24.15
2	5	(b)	256^3	24.12
2	5	(c)	256^3	18.09

Table A.18.: Durations of overlapped Jacobi's method for 2 A100 GPUs on the Pleiades cluster. Default does not overlap. In version (a), the `inner` kernel is enqueued right after the `boundary` kernel. In version (b), the `inner` kernel is enqueued after the `extract` kernel. In version (c), the `inner` kernel is enqueued after the `extract` kernel has finished.

GPUs	ν	Version	N_{local}	Time [ms]
4	1	default	32^3	0.18
4	1	(a)	32^3	0.20
4	1	(b)	32^3	0.19
4	1	(c)	32^3	0.19
4	2	default	32^3	0.26
4	2	(a)	32^3	0.28
4	2	(b)	32^3	0.27
4	2	(c)	32^3	0.31
4	3	default	32^3	0.37
4	3	(a)	32^3	0.39
4	3	(b)	32^3	0.38
4	3	(c)	32^3	0.40
4	4	default	32^3	0.47
4	4	(a)	32^3	0.54
4	4	(b)	32^3	0.50
4	4	(c)	32^3	0.55
4	5	default	32^3	0.60
4	5	(a)	32^3	0.63
4	5	(b)	32^3	0.61
4	5	(c)	32^3	0.66
4	1	default	64^3	0.44
4	1	(a)	64^3	0.46
4	1	(b)	64^3	0.47
4	1	(c)	64^3	0.44
4	2	default	64^3	0.72
4	2	(a)	64^3	0.73
4	2	(b)	64^3	0.73
4	2	(c)	64^3	0.68
4	3	default	64^3	0.96
4	3	(a)	64^3	1.01
4	3	(b)	64^3	1.01
4	3	(c)	64^3	0.91
4	4	default	64^3	1.24
4	4	(a)	64^3	1.28
4	4	(b)	64^3	1.27
4	4	(c)	64^3	1.19
4	5	default	64^3	1.49
4	5	(a)	64^3	1.48
4	5	(b)	64^3	1.50
4	5	(c)	64^3	1.39

GPUs	ν	Version	N_{local}	Time [ms]
4	1	default	128^3	1.47
4	1	(a)	128^3	1.47
4	1	(b)	128^3	1.45
4	1	(c)	128^3	1.15
4	2	default	128^3	2.38
4	2	(a)	128^3	2.38
4	2	(b)	128^3	2.48
4	2	(c)	128^3	1.80
4	3	default	128^3	3.34
4	3	(a)	128^3	3.44
4	3	(b)	128^3	3.48
4	3	(c)	128^3	2.46
4	4	default	128^3	4.27
4	4	(a)	128^3	4.45
4	4	(b)	128^3	4.51
4	4	(c)	128^3	3.13
4	5	default	128^3	5.22
4	5	(a)	128^3	5.48
4	5	(b)	128^3	5.43
4	5	(c)	128^3	3.80
4	1	default	256^3	6.22
4	1	(a)	256^3	6.29
4	1	(b)	256^3	6.21
4	1	(c)	256^3	4.96
4	2	default	256^3	10.99
4	2	(a)	256^3	11.06
4	2	(b)	256^3	11.01
4	2	(c)	256^3	8.22
4	3	default	256^3	15.64
4	3	(a)	256^3	15.89
4	3	(b)	256^3	15.88
4	3	(c)	256^3	11.48
4	4	default	256^3	20.13
4	4	(a)	256^3	20.76
4	4	(b)	256^3	20.65
4	4	(c)	256^3	14.85
4	5	default	256^3	24.72
4	5	(a)	256^3	25.56
4	5	(b)	256^3	25.57
4	5	(c)	256^3	18.56

Table A.19.: Durations of overlapped Jacobi's method for 4 A100 GPUs on the Pleiades cluster. Default does not overlap. In version (a), the `inner` kernel is enqueued right after the `boundary` kernel. In version (b), the `inner` kernel is enqueued after the `extract` kernel. In version (c), the `inner` kernel is enqueued after the `extract` kernel has finished.

A.7. Blockstencil Residual

blocksize	kernel	Time [ms] Quadro T2000, 64^3	A100, 128^3
1	block_first_v_block_first	0.46	0,42
1	block_first_v_gp_first	0.47	0,47
1	coeffs_first_v_block_first	0.47	0,47
1	coeffs_first_v_gp_first	0.46	0,41
2	block_first_v_block_first	1.82	1,62
2	block_first_v_gp_first	1.94	1,71
2	coeffs_first_v_block_first	1.93	1,71
2	coeffs_first_v_gp_first	1.81	1,60
3	block_first_v_block_first	4.10	3,57
3	block_first_v_gp_first	4.52	3,98
3	coeffs_first_v_block_first	4.55	3,97
3	coeffs_first_v_gp_first	4.12	3,57
4	block_first_v_block_first	7.39	6,34
4	block_first_v_gp_first	8.70	7,23
4	coeffs_first_v_block_first	8.64	7,20
4	coeffs_first_v_gp_first	7.38	6,35
5	block_first_v_block_first	11.54	9,99
5	block_first_v_gp_first	14.78	11,14
5	coeffs_first_v_block_first	14.78	11,20
5	coeffs_first_v_gp_first	11.56	9,97
6	block_first_v_block_first	16.79	14,43
6	block_first_v_gp_first	23.59	16,06
6	coeffs_first_v_block_first	23.60	16,06
6	coeffs_first_v_gp_first	16.75	14,37

Table A.20.: Kernel durations for the blockstencil residual kernel using different layouts for v

List of Figures

2.1. 1d grid on the interval $[0, 1]$ 4
2.2. 2d grid on the unit square 5
2.3. 3d compact Laplace stencils 8
2.4. Linear interpolation from coarse grid Ω^{2h} to fine grid Ω^h in 1d . 17
2.5. Interpolation of a smooth error and a highly oscillatory error . . 18
2.6. Full-weighted restriction from fine grid Ω^h to coarse grid Ω^{2h} in 1d 18
2.7. Multigrid schemes: V-cycle, W-cycle and Full Multigrid 20

3.1. OpenCL Platform Model 28
3.2. OpenCL Execution Model 29
3.3. OpenCL Memory Model 29
3.4. Full view of Nvidia's GA100 GPU architecture 33
3.5. View of one SM of Nvidia's GA100 GPU architecture 34
3.6. Turing GPU memory overview as shown in NCU 36
3.7. Workflow of the L1 cache on a global load on GA100 37
3.8. Workflow of the L2 cache on a global load on GA100 37
3.9. Overview of OpenCL runtime components 40
3.10. Aligned and misaligned memory accesses 51
3.11. Multiple work-items accessing the same memory address in a coalesced manner 53
3.12. Strided memory access pattern 54
3.13. Local memory bank conflicts 57
3.14. 64-bit elements spanning two memory banks each 57
3.15. Example of Nsight Systems GUI 61
3.16. Nsight Compute summary view 62
3.17. Nsight Compute Speed of Light bar charts 63
3.18. Regions of the roofline model 63
3.19. General roofline model in NCU 64
3.20. Double-precision roofline model in NCU 64

4.1. Memory access pattern of v in the 3d residual kernel with wg-size `4x4x4` and suboptimal work-item indexing 77
4.2. Memory access pattern of v in the 3d residual kernel with wg-size `4x4x4` and improved work-item indexing 77

4.3. Grid-first and coefficients-first data layouts 84
4.4. Mapping of one work-item to two grid points 86
4.5. Visualization of a Morton index grid and traversal 91
4.6. Durations for Jacobi's method having residual and update step kernels split vs. fused 95
4.7. Illustration of different tile shapes 99
4.8. Streaming concept for the 3.5D blocking scheme 103
4.9. Branching in `8x8` and `2x64` tiles for our blocking implementation 106
4.10. Roofline from NCU for `gmem` and `preproc` variants of the residual kernel with fixed coefficients (Quadro T2000 GPU) 107
4.11. Roofline from NCU for `gmem` and `preproc` variants of the residual kernel with fixed coefficients (A100 GPU) 108
4.12. Sequential ghost update sequence on a 2d grid 109
4.13. Calculation of the real point index based on the ghost point index for $gh \geq m$ 110
4.14. Roofline for the double precision version of the ghost update kernel, Quadro T2000, $128^3, gh = 1$ 111
4.15. Kernel durations for $gh \in [1, \ldots, 5]$ for the modulo and float versions 113
4.16. The four cases in bilinear prolongation in 2d 114
4.17. Roofline for the prolongation kernel versions on the Quadro T2000 for a fine grid of size 128^3 with $gh = 1$ 116
4.18. Roofline for the prolongation kernel versions on the A100 for a fine grid of size 512^3 with $gh = 1$ 116
4.19. Tree-based approach for sum reduction 118
4.20. Impact of the batch size for the sum reduction kernel versions . . 121
4.21. Visualization of a `4x4` grid, the corresponding system matrix and its stencil representation 122
4.22. Visualization of gathering and distributing stencils 126
4.23. Illustration of the computation of the entry $\mathbf{C}(9, 1)$ as produced in the matrix-matrix multiplication $\mathbf{C} = \mathbf{AB}$ and the affected grid points 129
4.24. Relevant grid points in the multiplication of two stencils 129
4.25. Relevant grid points for the calculation of $\mathbf{RA} \cdot \mathbf{P}$ 130
4.26. Relevant grid points for the calculation of $\mathbf{R} \cdot \mathbf{A} \cdot \mathbf{P}$ 131

5.1. Grid hierarchy without agglomeration 138
5.2. Grid hierarchy with monadic agglomeration 139
5.3. Grid hierarchy using dynamic agglomeration 139
5.4. Memory layout of extracted border planes 142
5.5. Nsight System's view of one run of Jacobi's method with $\nu = 2$ iterations and spi $= 1, 2$ on a 64^3 grid 146

5.6. Timelines in Nsight Systems for different implementations of overlapped Jacobi's method for 3 iterations 148
5.7. Performance of the different implementation versions of overlapped Jacobi's method vs. non-overlapped version 149

6.1. Durations of residual kernel using different layouts for the coefficients and the grid quantities, varying the blocksize B 153
6.2. Strided memory access for grid-first layout with stride $= B = 2$, coalesced memory access for block-first layout 153

7.1. Amount of v-cycle iterations until reaching $\texttt{tol} = 10^{-6}$ for pairs of pre- and post-smoothing steps (ν_1, ν_2) for mgcl and HYPRE . 161
7.2. Amount of v-cycle iterations until reaching $\texttt{tol} = 10^{-6}$ (limit of 30 v-cycles) for pairs of pre- and post-smoothing steps (ν_1, ν_2) for mgcl and HYPRE 161
7.3. Solve durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl and HYPRE. Grid size 128^3, Quadro T2000 165
7.4. Solve durations for 1-10 v-cycles, $\nu_1 = 1,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 128^3, Quadro T2000 165
7.5. Solve durations for 1-10 v-cycles, $\nu_1 = 10,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 128^3, Quadro T2000 165
7.6. Solve durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl and HYPRE. Grid size 256^3, A100 166
7.7. Solve durations for 1-10 v-cycles, $\nu_1 = 1,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 256^3, A100 166
7.8. Solve durations for 1-10 v-cycles, $\nu_1 = 10,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 256^3, A100 166
7.9. Total durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl and HYPRE. Grid size 128^3, Quadro T2000 167
7.10. Total durations for 1-10 v-cycles, $\nu_1 = 1,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 128^3, Quadro T2000 167
7.11. Total durations for 1-10 v-cycles, $\nu_1 = 10,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 128^3, Quadro T2000 167
7.12. Total durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl and HYPRE. Grid size 256^3, A100 168
7.13. Total durations for 1-10 v-cycles, $\nu_1 = 1,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 256^3, A100 168
7.14. Total durations for 1-10 v-cycles, $\nu_1 = 10,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 256^3, A100 168
7.15. Solve durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl and HYPRE. $N_{\text{local}}^{(0)} = 256^3, N_{\text{global}}^{(0)} = 512^3$, A100 169
7.16. Solve durations for 1-10 v-cycles, $\nu_1 = 1,\ \nu_2 = 1$, for mgcl and HYPRE. $N_{\text{local}}^{(0)} = 256^3, N_{\text{global}}^{(0)} = 512^3$, A100 169

7.17. Solve durations for 1-10 v-cycles, $\nu_1 = 10,\ \nu_2 = 1$, for mgcl and HYPRE. $N_{\text{local}}^{(0)} = 256^3, N_{\text{global}}^{(0)} = 512^3$, A100 169
7.18. Total durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for mgcl and HYPRE. Grid size 256^3, A100 170
7.19. Total durations for 1-10 v-cycles, $\nu_1 = 1,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 256^3, A100 170
7.20. Total durations for 1-10 v-cycles, $\nu_1 = 10,\ \nu_2 = 1$, for mgcl and HYPRE. Grid size 256^3, A100 170
7.21. Nsight System's view of one ghost update of v via MPI on a 256^3 local grid for `mgcl` and HYPRE 172
7.22. Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. Grid size 64^3, Quadro T2000 177
7.23. Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. Grid size 128^3, A100 178
7.24. Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. $N_{\text{local}}^{(0)} = 128^3, N_{\text{global}}^{(0)} = 256^3$, A100 . . . 178

A.1. Excerpts of the generated SASS code for the residual kernel with fixed coefficients on the Quadro T2000 203

List of Tables

3.1. Comparison of OpenCL and Nvidia CUDA terminology 31
3.2. Excerpt of device attributes for A100 and Quadro T2000 GPUs . 32
3.3. Excerpt of work-item functions in OpenCL C 42

4.1. Performance of different 3d residual kernel variants on a single Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ . . . 76
4.2. Performance of different 3d residual kernel variants on a single Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ . . . 78
4.3. Sectors per request for different 3d residual kernel variants, Quadro T2000, real grid 128^3, $gh = 1$ 80
4.4. Occupancies of different 3d residual kernel variants on a single Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ . . . 80
4.5. Warp stalls for the residual kernel with varying coefficients on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ . . . 82
4.6. Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using one work-item for one grid point 82
4.7. Metrics for the varying stencil residual kernel on a A100 GPU for a real grid size of 128^3 and $gh = 1$ using one work-item for one grid point 83
4.8. Metrics for the varying stencil residual kernel on a A100 GPU for a real grid size of `256x256x512` and $gh = 1$ using one work-item for one grid point 84
4.9. Metrics for the varying stencil residual kernel, Quadro T2000, real grid size 128^3, $gh = 1$, grid-first layout 85
4.10. Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using one work-item for four grid points 86
4.11. Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using four consecutive work-items for one grid point 89

4.12. Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using four spread out work-items for one grid point 89
4.13. Metrics for the varying stencil residual kernel on a Quadro T2000 GPU for a real grid size of 128^3 and $gh = 1$ using two spread out work-items for one grid point 90
4.14. Durations for the varying stencil residual kernel on a A100 GPU for a real grid size of 128^3 and $gh = 1$ using multiple work-items for one grid point 90
4.15. Durations for Jacobi's method having residual and update step kernels split vs. fused 95
4.16. Durations for Jacobi's method for various iteration counts without intermediate ghost update for a grid of size 128^3 and $gh = 1$ 97
4.17. Kernel durations for Jacobi's method for 5 iterations with varying amount of ghost updates for a grid of size 128^3 and $gh = 1$. 97
4.18. Durations for Jacobi's method (our blocking implementation) . . 103
4.19. Amount of work-items choosing different branches in our prototype blocking implementation for a grid of size 64^3 105
4.20. Durations for residual calculation with fixed coefficients for a grid of size 128^3 108
4.21. NCU metrics for residual calculation with fixed coefficients for a grid of size 128^3 on a Quadro T2000 108
4.22. Kernel durations for the ghost update kernel for a grid of size 128^3 and $gh = 1$ 112
4.23. NCU metric selection for the modulo variant of the ghost update kernel for a grid of size 128^3 and $gh = 1$ 113
4.24. Durations of the prolongation kernel variants on a fine grid with size of 128^3 and $gh = 1$ 114
4.25. Durations of the prolongation kernel variants on fine grids with sizes of 256^3 (Quadro T2000) and 512^3 (A100) and $gh = 1$ 115
4.26. Durations of the restriction kernel variants on a fine grid with size of 128^3 and $gh = 1$ 117
4.27. Lowest durations for our sum reduction implementations, grouped by version and problem size 121
4.28. Durations of kernels participating in the Galerkin operator computation using stencil arithmetic 128
4.29. Duration of kernel variants calculating the coarse grid operator using the derived algorithm 133
4.30. Durations for 10 v-cycle iterations, $\nu_1 = \nu_2 = 3, \omega = 0.8$, with varying maximum level `max_lv_ocl` for which the GPU is used . 134

5.1. Best durations of ghost update copying the whole grid from device to host and back (naive approach) and transferring border planes only (improved approach) on 4 MPI processes each using one A100 GPU 143
5.2. Durations of the individual parts of ghost update copying the whole grid from device to host and back (naive approach) and transferring border planes only (improved approach) on 4 MPI processes each using one A100 GPU 144
5.3. Durations of the ghost update copying the planes buffer between host and device as a whole and interleaving data transfer between host and device with MPI transfers 144
5.4. Total durations of the individual parts of two Jacobi iterations on 2 MPI processes with spi = 1 and spi = 2 146

6.1. Kernel durations for the blockstencil residual layout using different layouts for the coefficients and the grid quantities, $B = 3$. 152
6.2. Kernel durations and FLOP count as reported by NCU for one launch of the Jacobi kernel on the Quadro T2000 and a scalar grid size of 32^3 for a vector-valued problem 154

7.1. Best average convergence factors for mgcl and Hypre for all (solver, p) 160
7.2. Durations for different problems with best `vci` per $\nu_1 + \nu_2$, Quadro T2000, grid size 128^3 162
7.3. Durations of different portions of `mgcl` and HYPRE as shown in Nsight Systems 171
7.4. Durations of different portions of the ghost update of v on the finest grid in `mgcl` and HYPRE as shown in Nsight Systems . . . 171
7.5. Solve durations for different problems with (`vci`, ν_1, ν_2) combination that consumes the lowest amount of solve time, ignoring or not ignoring the tolerance 173
7.6. Solve durations for different problems with (`vci`, ν_1, ν_2) combination that consumes the lowest amount of solve time, Quadro T2000, grid size 128^3. Varies the maximum level, for which OpenCL is used. For maxLvOcl = 0, only the finest level is using the GPU. Time in ms. 173
7.7. Solve durations for different problems with (`vci`, ν_1, ν_2) combination that consumes the lowest amount of solve time, A100, $N^{(0)}_{\text{local}} = 256^3, N^{(0)}_{\text{global}} = 512^3$. Varies the maximum level that is computed distributively, before gathering data of all remaining grid points onto the root process. 174

7.8. Solve durations for different problems with (`vci`, ν_1, ν_2) combination that consumes the lowest amount of solve time, A100, $N_{\text{local}}^{(0)} = 256^3$, $N_{\text{global}}^{(0)} = 512^3$. Varies the maximum level for which the overlapped Jacobi strategy is used. 174
7.9. durations of `mgcl` using the Quadro T2000 GPU, the Intel Core i7-10875H 8-core CPU with one MPI process and the same CPU with 8 MPI processes 175
7.10. Solve durations of `mgcl` using the A100 GPU and the AMD EPYC 7452 32-Core CPU with varying number of MPI processes on the Pleiades cluster 175
7.11. Convergence results for blocksize $B \in \{1, 2, 3\}$ 176
7.12. Solve and total durations for solving the vector-valued problem using blocksize $B \in \{1, 2, 3\}$ on the Quadro T2000 GPU on a grid of size 64^3 176
7.13. Durations for one v-cycle, $\nu_1 \in \{1, \ldots, 10\}, \nu_2 = 1$ for `mgcl`, varying the block size B. Executed on one and eight A100 GPUs with local grid size of 128^3. 179

A.1. Nsight Compute metrics for the coalesced memory access kernels executed on Quadro T2000 183
A.2. Nsight Compute metrics for the `offsetCopy`-kernel 184
A.3. Excerpt of metrics in Nsight Compute (Part 1) 186
A.4. Excerpt of metrics in Nsight Compute (Part 2) 187
A.5. Nsight Compute metrics for the varying stencil residual kernel on Quadro T2000, launched as 3d `ndrange` 188
A.6. Nsight Compute metrics for the varying stencil residual kernel on Quadro T2000, launched as 3d `ndrange` 189
A.7. Nsight Compute metrics for the varying stencil residual kernel with grid-first layout on Quadro T2000, launched as 3d `ndrange` 190
A.8. Nsight Compute metrics for the varying stencil residual kernel with grid-first layout on A100, launched as 3d `ndrange` 191
A.9. Nsight Compute metrics for the varying stencil residual kernel (4 grid points per work-item) 192
A.10. Compute metrics for the varying stencil residual kernel on A100 (4 grid points per work-item) 193
A.11. Durations of our prototype blocking implementation of Jacobi's method, performing 2 Jacobi iterations 195
A.12. NCU metrics for our prototype blocking implementation for grid size 8^3, 2 Jacobi iterations, Quadro T2000 (part 1) 196
A.13. NCU metrics for our prototype blocking implementation for grid size 8^3, 2 Jacobi iterations, Quadro T2000 (part 2) 197
A.14. NCU metrics for our prototype blocking implementation for grid size 32^3, 2 Jacobi iterations, Quadro T2000 (part 1) 198

A.15. NCU metrics for our prototype blocking implementation for grid size 32^3, 2 Jacobi iterations, Quadro T2000 (part 2) 199
A.16. NCU metrics for our prototype blocking implementation for grid size 64^3, 2 Jacobi iterations, Quadro T2000 (part 1) 200
A.17. NCU metrics for our prototype blocking implementation for grid size 64^3, 2 Jacobi iterations, Quadro T2000 (part 2) 201
A.18. Durations of overlapped Jacobi's method for 2 A100 GPUs on the Pleiades cluster 207
A.19. Durations of overlapped Jacobi's method for 4 A100 GPUs on the Pleiades cluster 208
A.20. Kernel durations for the blockstencil residual kernel using different layouts for v 209

Bibliography

[1] Aggarwal, A. and Vitter S., J. The Input/Output Complexity of Sorting and Related Problems. In: *Communications of the ACM* 31.9 (Sept. 1988), pp. 1116–1127. DOI: 10.1145/48529.48535.

[2] Ahmed, N., Mateev, N., and Pingali, K. "Tiling Imperfectly-nested Loop Nests". In: *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. SC '00: The 2000 ACM/IEEE Conference on Supercomputing. Nov. 2000, pp. 31–31. DOI: 10.1109/SC.2000.10018.

[3] AMD. *AMD ROCm*. URL: https://www.amd.com/en/products/software/rocm.html (visited on 10/02/2026).

[4] Anderson, S. E. *Bit Twiddling Hacks*. 2005. URL: https://graphics.stanford.edu/~seander/bithacks.html#InterleaveBMN (visited on 10/02/2026).

[5] Ashby, S. F. and Falgout, R. D. A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations. In: *Nuclear Science and Engineering* 124.1 (Sept. 1996), pp. 145–159. DOI: 10.13182/nse96-a24230.

[6] Beckingsale, D. A., Scogland, T. R., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., and Ryujin, B. S. "RAJA: Portable Performance for Large-Scale Scientific Applications". In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). Denver, CO, USA: IEEE, Nov. 2019, pp. 71–81. DOI: 10.1109/P3HPC49587.2019.00012.

[7] Bell, N. and Hoberock, J. "Thrust". In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 359–371. DOI: 10.1016/B978-0-12-385963-1.00026-5.

[8] Bergische Universität Wuppertal. *Wissenschaftliches Rechenzentrum PLEIADES*. URL: http://pleiades.uni-wuppertal.de/de/ (visited on 10/02/2026).

[9] Bondhugula, U., Bandishti, V., and Pananilath, I. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations. In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (May 2017), pp. 1285–1298. DOI: 10.1109/TPDS.2016.2615094.

[10] Brent, R. P. The Parallel Evaluation of General Arithmetic Expressions. In: *Journal of the ACM* 21.2 (Apr. 1974), pp. 201–206. DOI: 10.1145/321812.321815.

[11] Briggs, W. L., Henson, V. E., and McCormick, S. F. *A Multigrid Tutorial, Second Edition*. Second. Society for Industrial and Applied Mathematics, Jan. 2000. DOI: 10.1137/1.9780898719505.

[12] Buelow, M. von, Guthe, S., and Fellner, D. W. Fine-Grained Memory Profiling of GPGPU Kernels. In: *Computer Graphics Forum* 41.7 (2022), pp. 227–235. DOI: 10.1111/cgf.14671.

[13] Carter Edwards, H., Trott, C. R., and Sunderland, D. Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. In: *Journal of Parallel and Distributed Computing*. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing 74.12 (Dec. 1, 2014), pp. 3202–3216. DOI: 10.1016/j.jpdc.2014.07.003.

[14] Crovella, R. "GPU PERFORMANCE ANALYSIS". Aug. 18, 2020.

[15] Crovella, R. *Requesting clarification for Shared Memory Bank Conflicts and Shared memory access?* NVIDIA Developer Forums. Oct. 6, 2023. URL: https://forums.developer.nvidia.com/t/requesting-clarification-for-shared-memory-bank-conflicts-and-shared-memory-access/268574/3 (visited on 10/02/2026).

[16] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. "Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures". In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08: The 2008 ACM/IEEE Conference on Supercomputing. Nov. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5222004.

[17] De La Cruz, R. and Araya-Polo, M. Algorithm 942: Semi-Stencil. In: *ACM Transactions on Mathematical Software* 40.3 (Apr. 2014), pp. 1–39. DOI: 10.1145/2591006.

[18] Demouth, J. *CUDA Pro Tip: Minimize the Tail Effect*. NVIDIA Technical Blog. June 4, 2014. URL: https://developer.nvidia.com/blog/cuda-pro-tip-minimize-the-tail-effect/ (visited on 10/02/2026).

[19] Donea, J. and Huerta, A. *Finite Element Methods for Flow Problems*. 1st ed. Wiley, Apr. 11, 2003. DOI: 10.1002/0470013826.

[20] Einkemmer, L. and Ostermann, A. "Exponential Integrators on Graphic Processing Units". In: *2013 International Conference on High Performance Computing & Simulation (HPCS)*. 2013 International Conference on High Performance Computing & Simulation (HPCS). Helsinki, Finland: IEEE, July 2013, pp. 490–496. DOI: 10.1109/HPCSim.2013.6641458.

[21] Elman, H., Silvester, D., and Wathen, A. *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*. Oxford University Press, June 1, 2014. DOI: 10.1093/acprof:oso/9780199678792.001.0001.

[22] Falgout, R. D. and Jones, J. E. "Multigrid on Massively Parallel Architectures". In: *Multigrid Methods VI*. Ed. by Dick, E., Riemslagh, K., and Vierendeels, J.

Red. by Griebel, M., Keyes, D. E., Nieminen, R. M., Roose, D., and Schlick, T. Vol. 14. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 101–107. DOI: 10.1007/978-3-642-58312-4_13.

[23] Falgout, R. D. and Yang, U. M. "Hypre: A Library of High Performance Preconditioners". In: *Computational Science — ICCS 2002*. Ed. by Sloot, P. M. A., Hoekstra, A. G., Tan, C. J. K., and Dongarra, J. J. Red. by Goos, G., Hartmanis, J., and Van Leeuwen, J. Vol. 2331. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 632–641. DOI: 10.1007/3-540-47789-6_66.

[24] Fang, J., Fu, H., Yang, G., Wu, W., and Dai, N. "GPU-based Explicit Time Evolution Method". In: *SEG Technical Program Expanded Abstracts 2015*. SEG Technical Program Expanded Abstracts 2015. New Orleans, Louisiana: Society of Exploration Geophysicists, Aug. 19, 2015, pp. 3549–3553. DOI: 10.1190/segam2015-5856116.1.

[25] Fang, J., Fu, H., Zhang, H., Wu, W., Dai, N., Gan, L., and Yang, G. "Optimizing Complex Spatially-Variant Coefficient Stencils for Seismic Modeling on GPU". In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS). Melbourne, VIC: IEEE, Dec. 2015, pp. 641–648. DOI: 10.1109/ICPADS.2015.86.

[26] Flynn, M. Very High-Speed Computing Systems. In: *Proceedings of the IEEE* 54.12 (Dec. 1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[27] Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M. W., and Keutzer, K. *AI and Memory Wall*. Mar. 21, 2024. DOI: 10.48550/arXiv.2403.14123. arXiv: 2403.14123 [cs]. URL: http://arxiv.org/abs/2403.14123 (visited on 10/02/2026). Pre-published.

[28] Golub, G. H. and Van Loan, C. F. *Matrix Computations*. Fourth edition. Johns Hopkins Studies in the Mathematical Sciences. Baltimore: The Johns Hopkins University Press, 2013. 756 pp.

[29] Grebhahn, A., Engwer, C., Bolten, M., and Apel, S. Variability of Stencil Computations for Porous Media. In: *Concurrency and Computation: Practice and Experience* 29.17 (Sept. 10, 2017), e4119. DOI: 10.1002/cpe.4119.

[30] Gregg, B. "Visualizing Performance with Flame Graphs". Talk. USENIX ATC 2017 (Santa Clara, CA). July 13, 2017.

[31] Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., and Verdoolaege, S. "Hybrid Hexagonal/Classical Tiling for GPUs". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '14: 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization. Orlando FL USA: ACM, Feb. 15, 2014, pp. 66–75. DOI: 10.1145/2544137.2544160.

[32] Grosser, T., Cohen, A., Kelly, P. H. J., Ramanujam, J., Sadayappan, P., and Verdoolaege, S. "Split Tiling for GPUs: Automatic Parallelization Using Trape-

zoidal Tiles". In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. GPGPU-6: Sixth Workshop on General Purpose Processing Using GPUs. Houston Texas USA: ACM, Mar. 16, 2013, pp. 24–31. DOI: 10.1145/2458523.2458526.

[33] Hackbusch, W. *Iterative Solution of Large Sparse Systems of Equations*. Vol. 95. Applied Mathematical Sciences. Cham: Springer International Publishing, 2016. DOI: 10.1007/978-3-319-28483-5.

[34] Harris, M. *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. NVIDIA Technical Blog. Apr. 23, 2013. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (visited on 10/02/2026).

[35] Harris, M. *How to Optimize Data Transfers in CUDA C/C++*. NVIDIA Technical Blog. Dec. 5, 2012. URL: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/ (visited on 10/02/2026).

[36] Harris, M. "Optimizing Parallel Reduction in CUDA". 2007.

[37] Harris, M. *Unified Memory for CUDA Beginners*. NVIDIA Technical Blog. June 20, 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (visited on 10/02/2026).

[38] *High Performance Geometric Multigrid*. URL: https://hpgmg.org/ (visited on 10/02/2026).

[39] Hijma, P., Heldens, S., Sclocco, A., Van Werkhoven, B., and Bal, H. E. Optimization Techniques for GPU Programming. In: *ACM Computing Surveys* 55.11 (Nov. 30, 2023), pp. 1–81. DOI: 10.1145/3570638.

[40] Hoffmann, S. *Mgcl*. Version v1.0.0. Zenodo, Feb. 14, 2026. DOI: 10.5281/ZENODO.18642994.

[41] Holewinski, J., Pouchet, L.-N., and Sadayappan, P. "High-Performance Code Generation for Stencil Computations on GPU Architectures". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS'12: International Conference on Supercomputing. San Servolo Island, Venice Italy: ACM, June 25, 2012, pp. 311–320. DOI: 10.1145/2304576.2304619.

[42] Hux, A. "OpenCL 2.0 Overview". SIGGRAPH Asia 2013 (Hong Kong). 2013.

[43] Hwu, W.-M. W., Kirk, D., and El Hajj, I. *Programming Massively Parallel Processors: A Hands-on Approach*. Fourth edition. Cambridge, MA: Morgan Kaufmann, Elsevier, 2023. 1 p. DOI: 10.1016/C2020-0-02969-5.

[44] Intel. *Intel oneAPI DPC++/C++ Compiler*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html (visited on 10/02/2026).

[45] Intel. *Intel oneAPI Level Zero*. URL: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-0/intel-oneapi-level-zero.html (visited on 10/02/2026).

[46] Intel. *Intel oneAPI Math Kernel Library*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html (visited on 10/02/2026).

[47] Intel. *oneAPI*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html (visited on 10/02/2026).

[48] Jääskeläinen, P., De La Lama, C. S., Schnetter, E., Raiskila, K., Takala, J., and Berg, H. Pocl: A Performance-Portable OpenCL Implementation. In: *International Journal of Parallel Programming* 43.5 (Oct. 2015), pp. 752–785. DOI: 10.1007/s10766-014-0320-y.

[49] Jia, Z., Maggioni, M., Smith, J., and Scarpazza, D. P. *Dissecting the NVidia Turing T4 GPU via Microbenchmarking*. Mar. 18, 2019. DOI: 10.48550/arXiv.1903.07486. arXiv: 1903.07486 [cs]. URL: http://arxiv.org/abs/1903.07486 (visited on 10/02/2026). Pre-published.

[50] Jia-Wei, H. and Kung, H. T. "I/O Complexity: The Red-Blue Pebble Game". In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing - STOC '81*. The Thirteenth Annual ACM Symposium. Milwaukee, Wisconsin, United States: ACM Press, 1981, pp. 326–333. DOI: 10.1145/800076.802486.

[51] John, V. *Finite Element Methods for Incompressible Flow Problems*. Vol. 51. Springer Series in Computational Mathematics. Cham: Springer International Publishing, 2016. DOI: 10.1007/978-3-319-45750-5.

[52] Kaeli, D. R. *Heterogeneous Computing with OpenCL 2.0*. 3rd edition. Elsevier, 2015.

[53] Khan, A., Al-Mouhamed, M., Fatayar, A., Almousa, A., Baqais, A., and Assayony, M. "Padding Free Bank Conflict Resolution for CUDA-based Matrix Transpose Algorithm". In: *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). June 2014, pp. 1–6. DOI: 10.1109/SNPD.2014.6888709.

[54] Khronos OpenCL Working Group. *The OpenCL™ C Specification*. July 10, 2025. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html (visited on 10/02/2026).

[55] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M. S., and Nagel, W. E. "The Vampir Performance Analysis Tool-Set". In: *Tools for High Performance Computing*. Ed. by Resch, M., Keller, R., Himmler, V., Krammer, B., and Schulz, A. Berlin, Heidelberg: Springer, 2008, pp. 139–155. DOI: 10.1007/978-3-540-68564-7_9.

[56] Knüpfer, A. et al. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Ed. by Brunst, H., Müller, M. S., Nagel, W. E., and Resch, M. M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. DOI: 10.1007/978-3-642-31476-6_7.

[57] Koestler, H., Ritter, D., and Feichtinger, C. "A Geometric Multigrid Solver on GPU Clusters". In: *GPU Solutions to Multi-scale Problems in Science and Engineering*. Ed. by Yuen, D. A., Wang, L., Chi, X., Johnsson, L., Ge, W., and Shi, Y. Berlin, Heidelberg: Springer, 2013, pp. 407–422. DOI: 10.1007/978-3-642-16405-7_26.

[58] Komatitsch, D., Erlebacher, G., Göddeke, D., and Michéa, D. High-Order Finite-Element Seismic Wave Propagation Modeling with MPI on a Large GPU Cluster. In: *Journal of Computational Physics* 229.20 (Oct. 2010), pp. 7692–7714. DOI: 10.1016/j.jcp.2010.06.024.

[59] Kraus, J. *An Introduction to CUDA-Aware MPI*. NVIDIA Technical Blog. Mar. 14, 2013. URL: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/ (visited on 07/22/2025).

[60] Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. Effective Automatic Parallelization of Stencil Computations. In: *ACM SIGPLAN Notices* 42.6 (June 10, 2007), pp. 235–244. DOI: 10.1145/1273442.1250761.

[61] Lake, A. *Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics*. 2014. URL: https://www.intel.com/content/www/us/en/developer/articles/training/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics.html (visited on 10/02/2026).

[62] Liu, S., Zhang, Z., and Wu, W. DHTS: A Dynamic Hybrid Tiling Strategy for Optimizing Stencil Computation on GPUs. In: *IEEE Transactions on Computers* 72.10 (Oct. 2023), pp. 2795–2807. DOI: 10.1109/TC.2023.3271060.

[63] Luitjens, J. *Faster Parallel Reductions on Kepler*. NVIDIA Technical Blog. Feb. 14, 2014. URL: https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/ (visited on 10/02/2026).

[64] Luitjens, J. and Devaramani, R. *CUDA Pro Tip: Increase Performance with Vectorized Memory Access*. NVIDIA Technical Blog. Aug. 4, 2025. URL: https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/ (visited on 10/02/2026).

[65] Matsumura, K., Zohouri, H. R., Wahib, M., Endo, T., and Matsuoka, S. "AN5D: Automated Stencil Framework for High-Degree Temporal Blocking on GPUs". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. Feb. 22, 2020, pp. 199–211. DOI: 10.1145/3368826.3377904. arXiv: 2001.01473 [cs].

[66] McIntosh-Smith, S. and Deakin, T. *Hands On OpenCL*. 2014. URL: https://handsonopencl.github.io/ (visited on 10/02/2026).

[67] McKee, S. A. "Reflections on the Memory Wall". In: *Proceedings of the 1st Conference on Computing Frontiers*. CF04: Computing Frontiers Conference. Ischia Italy: ACM, Apr. 14, 2004, p. 162. DOI: 10.1145/977091.977115.

[68] Mei, X. and Chu, X. *Dissecting GPU Memory Hierarchy through Microbenchmarking*. Mar. 14, 2016. DOI: 10.48550/arXiv.1509.02308. arXiv: 1509.02308 [cs]. URL: http://arxiv.org/abs/1509.02308 (visited on 10/02/2026). Pre-published.

[69] Michéa, D. and Komatitsch, D. Accelerating a Three-Dimensional Finite-Difference Wave Propagation Code Using GPU Graphics Cards: Accelerating a Wave Propagation Code Using GPUs. In: *Geophysical Journal International* (May 2010), no–no. DOI: 10.1111/j.1365-246X.2010.04616.x.

[70] Micikevicius, P. "3D Finite Difference Computation on GPUs Using CUDA". In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU '09: Second Workshop on General-Purpose Computation on Graphics Processing Units. Washington D.C. USA: ACM, Mar. 8, 2009, pp. 79–84. DOI: 10.1145/1513895.1513905.

[71] Morton, G. M. *A Computer Oriented Geodetic Data Base, and a New Technique in File Sequencing*. IBM Canada Ltd., 1966.

[72] Mukund, R. *NVIDIA Is Now OpenCL 3.0 Conformant*. NVIDIA Technical Blog. Apr. 12, 2021. URL: https://developer.nvidia.com/blog/nvidia-is-now-opencl-3-0-conformant/ (visited on 10/02/2026).

[73] Munshi, A., Gaster, B. R., Mattson, T. G., Fung, J., and Ginsburg, D. *OpenCL Programming Guide*. Upper Saddle River, NJ: Addison-Wesley, 2012. 603 pp.

[74] Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs". In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010 SC - International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans, LA, USA: IEEE, Nov. 2010, pp. 1–13. DOI: 10.1109/SC.2010.2.

[75] Nugteren, C. and Codreanu, V. "CLTune: A Generic Auto-Tuner for OpenCL Kernels". In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). Turin, Italy: IEEE, Sept. 2015, pp. 195–202. DOI: 10.1109/MCSoC.2015.10.

[76] NVIDIA Corporation. *CUDA C++ Best Practices Guide*. Oct. 2, 2025.

[77] NVIDIA Corporation. *CUDA C++ Programming Guide*. Aug. 20, 2025.

[78] NVIDIA Corporation. *CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-toolkit (visited on 10/02/2026).

[79] NVIDIA Corporation. *Data Types Used by CUDA Runtime*. CUDA Runtime API. Oct. 9, 2025. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (visited on 10/02/2026).

[80] NVIDIA Corporation. *Nsight Compute Documentation*. Aug. 14, 2025. URL: https://docs.nvidia.com/nsight-compute/index.html (visited on 10/02/2026).

[81] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. Whitepaper. 2020.

[82] NVIDIA Corporation. *NVIDIA GameWorks Documentation Rev. 1.0.150630 - Memory Statistics - Global*. 2015.

[83] NVIDIA Corporation. *NVIDIA OpenCL Best Practices Guide*. Aug. 10, 2009.

[84] NVIDIA Corporation. *NVIDIA TESLA V100 GPU ARCHITECTURE*. Whitepaper. 2017.

[85] NVIDIA Corporation. *NVIDIA TURING GPU ARCHITECTURE*. Whitepaper. 2018.

[86] O'Reilly, R. C. and Beck, J. M. *A Family of Large-Stencil Discrete Laplacian Approximations in Three Dimensions*. 2006. URL: https://ccnlab.org/papers/OReillyBeck06.pdf (visited on 06/02/2026).

[87] Pekkilä, J., Väisälä, M. S., Käpylä, M. J., Rheinhardt, M., and Lappi, O. Scalable Communication for High-Order Stencil Computations Using CUDA-aware MPI. In: *Parallel Computing* 111 (July 2022), p. 102904. DOI: 10.1016/j.parco.2022.102904.

[88] Perdacher, M., Plant, C., and Böhm, C. "Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition". In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020 IEEE International Conference on Big Data (Big Data). Dec. 2020, pp. 351–360. DOI: 10.1109/BigData50022.2020.9378385.

[89] Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., and Panda, D. K. "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs". In: *2013 42nd International Conference on Parallel Processing*. 2013 42nd International Conference on Parallel Processing. Oct. 2013, pp. 80–89. DOI: 10.1109/ICPP.2013.17.

[90] *Questions about Cacheline & Sector - CUDA / CUDA Programming and Performance*. NVIDIA Developer Forums. Mar. 2, 2025. URL: https://forums.developer.nvidia.com/t/questions-about-cacheline-sector/325647 (visited on 10/02/2026).

[91] Rawat, P. S. "Optimization of Stencil Computations on GPUs". The Ohio State University, 2018.

[92] Rawat, P. S., Vaidya, M., Sukumaran-Rajam, A., Rountev, A., Pouchet, L.-N., and Sadayappan, P. "On Optimizing Complex Stencils on GPUs". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Rio de Janeiro, Brazil: IEEE, May 2019, pp. 641–652. DOI: 10.1109/IPDPS.2019.00073.

[93] Saad, Y. *Iterative Methods for Sparse Linear Systems*. Second edition. Society for Industrial and Applied Mathematics, Jan. 2003. DOI: 10.1137/1.9780898718003.

[94] Sai, R., Mellor-Crummey, J., Meng, X., Araya-Polo, M., and Meng, J. "Using the Semi-Stencil Algorithm to Accelerate High-Order Stencils on GPUs". In: *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). Nov. 2021, pp. 63–68. DOI: 10.1109/PMBS54543.2021.00012.

[95] Sakharnykh, N. *High-Performance Geometric Multi-Grid with GPU Acceleration*. NVIDIA Technical Blog. Feb. 23, 2016. URL: https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/ (visited on 10/02/2026).

[96] Sakharnykh, N. *Hpgmg-Cuda*. Bitbucket. URL: https://bitbucket.org/nsakharnykh/hpgmg-cuda/src/master/ (visited on 07/23/2025).

[97] Sakharnykh, N. *Maximizing Unified Memory Performance in CUDA*. NVIDIA Technical Blog. Nov. 20, 2017. URL: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/ (visited on 10/02/2026).

[98] Sanders, J. and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Upper Saddle River, NJ: Addison-Wesley, 2011. 290 pp.

[99] Saviankou, P., Knobloch, M., Visser, A., and Mohr, B. Cube v4: From Performance Report Explorer to Performance Analysis Tool. In: *Procedia Computer Science* 51 (2015), pp. 1343–1352. DOI: 10.1016/j.procs.2015.05.320.

[100] Shan, B. and Araya-Polo, M. *Evaluation of Programming Models and Performance for Stencil Computation on Current GPU Architectures*. Aug. 12, 2024. arXiv: 2404.04441 [cs]. URL: http://arxiv.org/abs/2404.04441 (visited on 10/02/2026). Pre-published.

[101] Shende, S. S. and Malony, A. D. The Tau Parallel Performance System. In: *The International Journal of High Performance Computing Applications* 20.2 (May 2006), pp. 287–311. DOI: 10.1177/1094342006064482.

[102] Sourouri, M., Gillberg, T., Baden, S. B., and Cai, X. "Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads". In: *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS). Hsinchu, Taiwan: IEEE, Dec. 2014, pp. 981–986. DOI: 10.1109/PADSW.2014.7097919.

[103] Spotz, W. F. and Carey, G. F. A High-Order Compact Formulation for the 3D Poisson Equation. In: *Numerical Methods for Partial Differential Equations* 12.2 (Mar. 1996), pp. 235–243. DOI: 10.1002/(SICI)1098-2426(199603)12:2<235::AID-NUM6>3.0.CO;2-R.

[104] Stewart, G. W. *Introduction to Matrix Computations*. Computer Science and Applied Mathematics. Olrando: Academic Press, 1973. 441 pp.

[105] *The Granularity of L1 and L2 Caches - CUDA / CUDA Programming and Performance*. NVIDIA Developer Forums. Apr. 18, 2024. URL: https://forums.developer.nvidia.com/t/the-granularity-of-l1-and-l2-caches/290065 (visited on 10/02/2026).

[106] The Khronos Group. *OpenCL*. URL: https://www.khronos.org/opencl/ (visited on 10/02/2026).

[107] The Khronos Group. *OpenCL-Guide*. OpenCL Guide. July 26, 2025. URL: https://github.com/KhronosGroup/OpenCL-Guide (visited on 10/02/2026).

[108] The Khronos Group. *SYCL*. URL: https://www.khronos.org/sycl/ (visited on 10/02/2026).

[109] Thiyagalingam, J., Beckmann, O., and Kelly, P. H. J. "An Exhaustive Evaluation of Row-Major, Column-Major and Morton Layouts for Large Two-Dimensional Arrays". In: *Proceedings of the UK Performance Engineering Workshop (UKPEW 2003)*. UK Performance Engineering Workshop (UKPEW 2003). Warwick, UK: University of Warwick, July 2003.

[110] Trottenberg, U., Oosterlee, C. W., Schüller, A., Brandt, A., Oswald, P., and Stüben, K. *Multigrid*. Transferred to digital print. Amsterdam Heidelberg: Elsevier Academic Press, 2007. 631 pp.

[111] Ujeniya, A. *CUDA-Aware-MPI: Part 3: Solving 2D Poisson Equation with 2D Domain Decomposition*. Aditya Ujeniya's blog. Feb. 27, 2025. URL: https://blogs.fau.de/adityauj/2025/02/27/cuda-aware-mpi-part-3-solving-2d-poisson-equation-with-2d-domain-decomposition/ (visited on 10/02/2026).

[112] Varga, R. S. *Matrix Iterative Analysis*. Vol. 27. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. DOI: 10.1007/978-3-642-05156-2.

[113] Volkov, V. Better Performance at Lower Occupancy. In: *Proceedings of the GPU Technology Conference, GTC* 10 (Jan. 2015).

[114] Volkov, V. "Understanding Latency Hiding on GPUs". PhD thesis. Berkeley: University of California, Aug. 16, 2016.

[115] Wang, P. "Fundamental Optimizations in CUDA". Nvidia GTC.

[116] *What Is the Expected L1/L2 Hit Rate for Fully Coalesced Accesses? - CUDA / CUDA Programming and Performance*. NVIDIA Developer Forums. Jan. 7, 2025. URL: https://forums.developer.nvidia.com/t/what-is-the-expected-l1-l2-hit-rate-for-fully-coalesced-accesses/316346/5 (visited on 10/02/2026).

[117] Wijngaart, R. V. der and Oh, F. *Boosting Application Performance with GPU Memory Prefetching*. NVIDIA. Mar. 23, 2022. URL: https://resources.nvidia.com/en-us-nsight-developer-tools/boosting-application (visited on 10/02/2026).

[118] Williams, S., Waterman, A., and Patterson, D. *Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*. 1407078. Sept. 1, 2009, p. 1407078. DOI: 10.2172/1407078.

[119] Wolf, M. E. and Lam, M. S. A Data Locality Optimizing Algorithm. In: *ACM SIGPLAN Notices* 26.6 (June 1991), pp. 30–44. DOI: 10.1145/113446.113449.

[120] Wolfe, M. "More Iteration Space Tiling". In: *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. Supercomputing '89. New York, NY, USA: Association for Computing Machinery, Aug. 1, 1989, pp. 655–664. DOI: 10.1145/76263.76337.

[121] Wulf, Wm. A. and McKee, S. A. Hitting the Memory Wall: Implications of the Obvious. In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. DOI: 10.1145/216585.216588.

[122] Yang, C. *Hierarchical Roofline Analysis: How to Collect Data Using Performance Tools on Intel CPUs and NVIDIA GPUs*. Oct. 4, 2020. DOI: 10.48550/arXiv.2009.02449. arXiv: 2009.02449 [cs]. URL: http://arxiv.org/abs/2009.02449 (visited on 10/02/2026). Pre-published.

[123] You, X., Yang, H., Jiang, Z., Luan, Z., and Qian, D. "DRStencil: Exploiting Data Reuse within Low-order Stencil on GPU". In: *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. 2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys). Haikou, Hainan, China: IEEE, Dec. 2021, pp. 63–70. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00036.

[124] Young, D. M. and Rheinboldt, W. *Iterative Solution of Large Linear Systems*. Burlington: Elsevier Science, 2014. 599 pp.

[125] Zhang, L., Wahib, M., Chen, P., Meng, J., Wang, X., Endo, T., and Matsuoka, S. "Revisiting Temporal Blocking Stencil Optimizations". In: *Proceedings of the 37th International Conference on Supercomputing*. ICS '23: 37th International Conference on Supercomputing. Orlando FL USA: ACM, June 21, 2023, pp. 251–263. DOI: 10.1145/3577193.3593716.