Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DIPLOMARBEIT

Algebraic Multigrid Methods on Parallel Architectures

Ausgeführt am

Institut für Mikroelektronik
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Erasmus Langer
Dipl.-Ing. Karl Rupp Msc. BSc.
Dipl.-Ing. Josef Weinbub BSc.

durch

Mag. Markus Wagner BSc.

Fleschgasse 15-17/7/2
1130 Wien

Matrikelnummer 9672208
Studienkennzahl 438

Datum

Unterschrift

Acknowledgement

First, I want to thank Prof. Erasmus Langer for supervising this thesis and for granting me a scholarship from the Institute for Microelectronics. I also owe a big thank you to my co-supervisors Karl Rupp and Josef Weinbub for providing a great working environment and for several discussions about the scope and direction of this work, including ideas for the implementation part. I further thank Karl Rupp for his support and advice on the written part. Certain parts of this work were developed during the Google Summer of Code 2011 program and I thank Google for this opportunity.

I want to thank my parents Josef and Herta Wagner for their consistent financial support as well as their encouragement and guidance during my studies. The same way I want to thank my grandparents Georg and Ella Reitmair for supporting me financially and for showing interest in my work. I further thank my brother Andreas Wagner for proof-reading certain parts and for some general discussions about this work.

Last but not least, I want to thank my girlfriend Sigrid Fasching for her encouragement and consistent support during my studies and the work on this thesis.

Abstract

Algebraic multigrid methods (AMG) are multigrid methods which construct grid hierarchies using algebraic information contained in the system matrix of the linear equation to be solved. This makes AMG useful for more general applications, including problems for which no grid interpretation is available. AMG offers optimal linear complexity and is therefore especially well-suited for large problem-sizes. An efficient stand-alone solver for linear equations, AMG is even more often used as a preconditioner for Krylov methods to improve convergence. Many variations of the AMG algorithm have been developed and the approaches usually differ in terms of convergence, setup time and complexity. One important aspect, driven by recent hardware development trends, is parallelism: Although parts of the basic AMG approaches are sequential, this can be overcome by certain modifications to the algorithm. For this thesis, implementations of several AMG preconditioners were developed for ViennaCL, a mathematical C++ library supporting parallel computing via OpenCL. These implementations can take advantage of parallel execution by using OpenMP threads for the setup phase (CPU) and OpenCL for the solver phase (GPU). Benchmark results show that GPU computing can increase solver performance for large matrices by a factor close to the theoretical maximum, but the total computation time is bounded by the setup phase computed on the CPU. For various problems, different AMG methods lead to optimal results, justifying the multitude of AMG variations available. An interesting consequence of these benchmarks is the relative independence of computation time from convergence factor and this is even stronger if the GPU is used. Methods with lower convergence usually offer faster setup but also a lower number of coarse levels and operator matrix density, reducing solver time even though more iterations are required. Complexity, therefore, seems to be a better indicator for AMG performance than convergence.

Contents

1 Introduction **1**

2 Components and Variants **9**

2.1 AMG Concepts 10

2.1.1 Setup Phase 10

2.1.2 Precondition Phase 11

2.1.3 Metrics 12

2.2 Smoother 13

2.3 Coarsening 15

2.3.1 RS Coarsening 15

2.3.2 CLJP Coarsening 18

2.3.3 Other RS-related Strategies 18

2.3.4 Aggregation 19

2.4 Interpolation 20

2.4.1 Direct Interpolation 21

2.4.2 Classical Interpolation 22

2.4.3 Other Interpolation Procedures 23

2.4.4 Interpolation for Aggregation-based AMG 24

2.5 Summary 25

3 Tools and Technologies 27

3.1 OpenCL 27

3.2 ViennaCL 29

3.3 OpenMP 31

3.4 uBlas 32

3.5 Renumbering Tool 32

3.6 Hardware 34

4 Implementation 35

4.1 Code and File Organization 36

4.2 Data Structures 38

4.2.1 User Options 38

4.2.2 Sparse Matrix 38

4.2.3 Sparse Vector 43

4.2.4 Point Management 43

4.2.5 AMG Classes 45

4.2.6 Parallel Coarsening 46

4.3 Setup Phase 47

4.3.1 Coarsening 48

4.3.2 Interpolation 52

4.3.3 Matrix Product 54

4.4 Precondition Phase 57

4.4.1 Smoother 57

4.4.2 Direct Solver 60

4.5 Summary 60

5 Benchmarks **63**

5.1 Systems 64

5.2 AMG Variants 65

5.3 Numerical Results 66

5.3.1 Scaling Analysis 66

5.3.2 Aggressive and Parallel Coarsening 70

5.3.3 Interpolation Approaches 72

5.3.4 Limitations to AMG 74

5.3.5 Summary 74

6 Summary, Conclusions and Possible Extensions **77**

Bibliography **79**

Chapter 1

Introduction

Partial differential equations (PDE) play a central role in many models of physical nature and are therefore an important aspect of applied mathematics [1, p.47]. Examples are the Poisson equation (see below), the wave equation or the heat equation [2, p.2]. To solve PDEs, a typical approach is to discretize the problem on a grid to transform it into a set of linear or nonlinear algebraic equations. Discretization means that the differential equation at hand is approximated on a certain geometry by solving for chosen grid points only [2, p.3-7]. Then, the unknowns in the original PDE correspond to grid point values of the discretized equation.

This system of linear equation, formally written as $Ax = f$ with a system matrix A, a vector of unknowns x and a right-hand-side vector f, is of size N, the number of points used for discretization on the respective grid. There are many different approaches to choosing grids which is discussed in [2, p.3ff]. A simple example for such a discretization can be obtained using the 1D Poisson equation with Dirichlet boundary conditions [1, p.53f]:

$$-u_{xx} = f(x) \tag{1.1}$$

$$u(0) = u(1) = 0 \tag{1.2}$$

Discretizing this equation on a regular grid using Taylor approximations for the

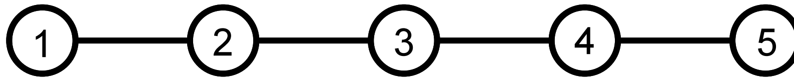

Figure 1.1: Graph respresentation of the discretization of equation 1.1. Numbered circles denote points, lines symmetric connections between points.

differential operators with 5 points located in $(0, 1)$ leads to $Ax = f$ with $h = \frac{1}{5}$ and

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \quad f = h^2 \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ f(x_5) \end{pmatrix} \tag{1.3}$$

The system can also be described using a graph notation. In that case, grid points i and j are connected to each other if the respective coefficients a_{ij} and a_{ji} for the matrix A are non-zero. The graph for the example system from equation 1.3 is depicted in figure 1.1. More examples of discretizations of PDEs can be found in [2, p.389-412] and [3, p.78-112] for different PDEs and grids.

Usually, system matrices constructed from a grid discretization have the property that they are relatively sparse, meaning that only a few coefficients per row are non-zero [1, p.47]. The number of unknowns, or equivalently, the number of rows in the system matrix, however, can be quite large, as the grid approximation improves by using finer grids with more grid points. Those linear equations certainly can be solved using general approaches like a direct Gauss solver or an iterative solver, but this can be problematic.

An important feature of solvers is its complexity, which is defined as the relationship between the number of operations and the number of unknowns N. A comparison between the complexities of different solvers for the Poisson problem is given in [2, p.14]: For the direct banded Gaussian solver, complexity squares with the number of unknowns $O(N^2)$ which makes it inefficient for fine grids with many grid points and a large resulting linear equation. For basic iterative procedures like Jacobi and Gauß-Seidel [1, p.105ff], the complexity is also quadratic such that the direct banded solver is usually preferred, especially since convergence and stability

can be an issue for the latter. More involved iterative solvers like Krylov subspace methods show better complexity: For example, the complexity for CG [1, p.157ff] is $O(N^{\frac{3}{2}} \log \epsilon)$[1]. Still, for large systems, performance of the solver can become a problem. Therefore, for certain problems, specialized solvers have been developed, for example the Fast Poisson Solver (FPS) for the discretization of the Poisson equation on a rectangular grid, leading to a complexity of $O(N \log N)$ [1, p.58-62].

The best complexity, however, can be obtained using multigrid methods: These offer linear complexity $O(N)$ and therefore are an attractive choice for large equations. Multigrid methods are known since the 1960s and have become very popular since the 1980s when numerous scientific contributions were made [2, p.23-24]. Multigrid methods can be distinguished into geometric multigrid methods and algebraic multigrid methods: For the geometric approach, the grid and grid points are constructed using discretizations as mentioned above and this is usually tailored to the geometry of the problem at hand.

Algebraic multigrid methods (AMG), on the other hand, do not use a grid in the geometric sense. Instead, grid points are constructed using algebraic information only, that is, information contained in the system matrix A. Still, the formal properties of AMG are the same as for geometric multigrid and AMG can also be used for geometric problems with an underlying grid [3, p.8-12]. AMG, however, is more general as no geometric grid is necessary and the approach can also be used for any linear equation as long as certain requirements on the system matrix are met. Although a general requirement is still lacking in the literature, AMG works best if the system matrix is a symmetric M-matrix, that is, a symmetric matrix with positive diagonal coefficients, negative off-diagonal coefficients and non-negative row sums such that at least one is strictly positive. Experience, however, shows that this is not a necessary requirement as AMG still works as long as the properties of the respective system matrix is not too far off from an M-matrix [3] [4]. As mentioned in [3], the terms “grids” and “points” are still used for AMG even though “sets of variables” and “variables” would be more reasonable. On the other hand, using the grid terminology makes it easier to relate to an underlying geometric problem and also gives more intuitive insight into the algorithm.

[1] The term $\log \epsilon$ reflects the stopping criterion for the iterative procedure.

Both multigrid approaches are based on two major principles - smoothing and coarse grid correction [2, p.15f]:

1. Many relaxation procedures (like Gauß-Seidel or Jacobi) have the property that the error $e^{(i)} = x^{(i)} - x$ from an iteration of the equation $Ax = f$ becomes smoother the more iteration steps i are applied, especially for problems of certain discretized partial differential equations. This means that oscillatory error parts are damped more efficiently than the smooth parts such that the error averages across the grid points (see figure 1.2). This principle is called the smoothing principle. If the error is already smooth, relaxation procedures are relatively inefficient (see above).

2. If the overall error is smooth, the error value at one point is close to the error value of neighboring points. This means that for an approximation of the error values at grid points, just a subset of the total grid points are needed, while errors at the other points can be approximated from neighboring points. This principle is called the coarse grid principle. It means that a coarser grid with just a subset of grid points can be built from the original grid and an error correction can be done by computing only the error on the coarser grid. If the error across the grid is sufficiently smooth, the error at the all grid points can be well approximated from errors computed for points on the coarser grid only. This approach is called "coarse grid correction".

A combination of both principles can be used to solve linear equations by applying smoothing and coarse grid correction: First, a few relaxation steps are applied to make the error smooth. Then, the error is computed on a coarser grid consisting of only a fraction of the total points and approximated for all points on the original grid. The computation of the error is achieved by taking into account that after a relaxation step i the equation $Ae^{(i)} = r^{(i)}$ has to hold, where $e^{(i)}$ is the error to be computed and $r^{(i)}$ is the residual after relaxation as $r = f - Ax^{(i)} = Ax - Ax^{(i)} = Ae^{(i)}$ ($x^{(i)}$ is the solution vector after relaxation). Therefore, the equation can be solved for the error vector using for example a direct Gauss solver on the coarser grid, approximated for the points not used on that grid and used to correct the result from the relaxation. This is efficient as long as the number of relaxation steps needed for

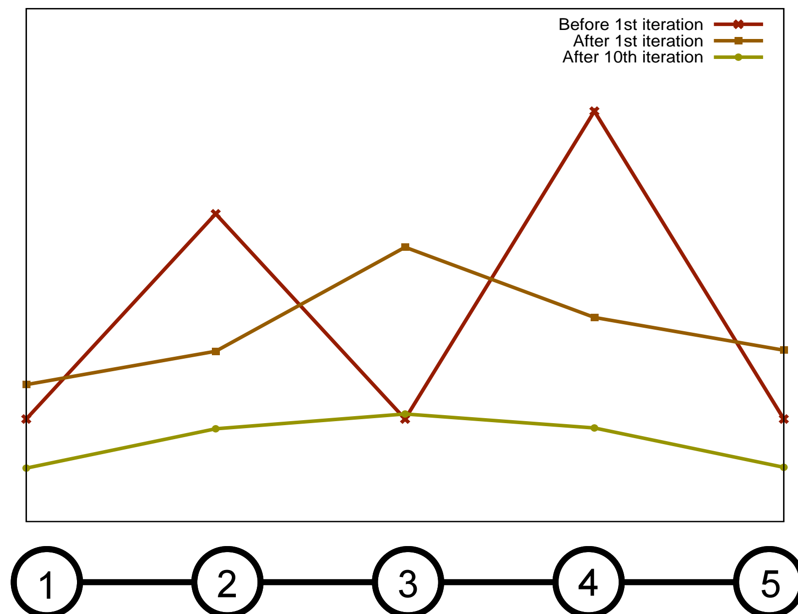

Figure 1.2: Error smoothing: Weighted Jacobi relaxation with weight factor 0.67 of equation 1.3 with a random right-hand-side vector. Shown are the errors at all 5 points after 0, 1 and 10 iterations.

smoothing and the number of coarse points needed for approximation are relatively small.

Multigrid methods generalize these two principles by using a hierarchy of grids for smoothing and error correction (see figure 1.3). Then, the computation of the error on the coarse level is done by building an even coarser grid and again using smoothing and coarse grid correction on that level and so on. Only on the coarsest grid the actual error is computed by using for example the Gauß method [2, p.39]. As shown in many examples, for example in [2, p.53-56] or [3], such an approach can be very efficient.

How coarse points are chosen from the set of total points and how the approximation is done from errors at coarse points to errors at fine points (called interpolation) is certainly the important aspect of how a multigrid method is constructed. Geometric multigrid methods usually use relatively simple approaches for choosing coarse points, for example the use of every second point in any direction on structured grids. Interpolation is then accomplished by computing the algebraic average of the errors at the neighboring coarse points. This approach, however, needs more complex

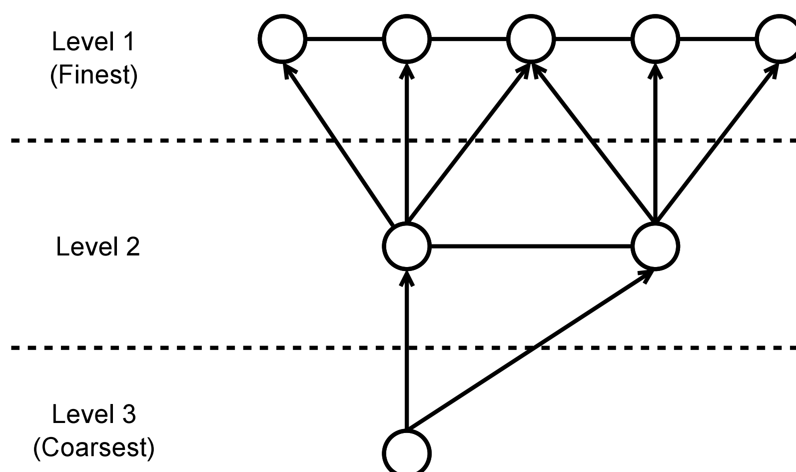

Figure 1.3: Multigrid approach using three levels. Arrows denote possible error approximations.

smoothers, especially for anisotropic problems, for good convergence [3, p.9]. Algebraic multigrid on the other hand can use simple smoothers, but the construction of the coarse grids and the interpolation is in general more complex: Coarse grids are constructed using the so-called Galerkin operator by employing only information contained in the system matrix A and the interpolation is constructed using matrix information only. Then, coarse-grid correction even works if the error is not smooth in a geometric sense but only smooth in an algebraic sense. An error is algebraically smooth when the residuals are small: $Ae \approx 0$ [4, p.218]. Another interpretation of an algebraically smooth error is that "relaxation is slow to converge" [3, p.29]. In this case, coarse grid correction can improve overall convergence. Therefore, while the geometric approach uses simple coarsening and interpolation but complex smoothers, the algebraic approach uses more complex matrix-dependent coarsening and interpolation but simple smoothers. This makes AMG very useful for problems on unstructured grids [3, p.8].

A two-level AMG system can be shown to uniformly converge for certain classes of positive definite matrices. This, however, does not generally hold for multilevel approaches or more general classes of matrices [3, p.60-63]. Still, even if these assumptions do not hold, AMG can usually be applied even though performance might be affected. Benchmarks have shown that an AMG standalone solver is an efficient process to solve linear equations for many classes of problems [3, p.78-112].

However, AMG can also be used very efficiently as a preconditioner for Krylov methods like CG [1, p.171ff] or GMRES [1, p.196ff]. In that case, if used as a preconditioner, AMG does not solve the actual system but is used to improve the convergence of those iterative solvers by correcting the iteration result. However, the computation of this correction vector is still done by solving a system of linear equations and therefore there is no algorithmic difference in the approach to an AMG solver. To avoid confusion, the term precondition phase will be used for the solve phase if AMG is used as a preconditioner. Using a combination of Krylov solver and AMG preconditioner is often advised in the literature due to its good convergence properties [3] [4].

One important aspect of the development of an AMG preconditioner is to use parallel architectures, as this is a recent trend in hardware development: This is described in [4] for highly parallel computer clusters, consisting of high numbers of servers and processors that have to work together. But even on single workstations and personal computers, parallelism is becoming more important as processors provide a number of parallel cores and threads. Furthermore, GPU (graphical processing unit) computing has become a trend in recent years due to the fact that GPUs consist of a huge number of processors, leading to a theoretical computing power that is much higher than that of a CPU (see chapter 3.6). Technologies like OpenCL allow developers to efficiently make use of this computing power in mixed CPU-GPU software (section 3.1).

The main research goal of this thesis is to develop an AMG preconditioner which uses parallel computation to improve performance and in effect, it has to be scrutinized how this can be done such that different architectures are used in an efficient way. This includes the use of multiple CPU cores as well as GPU computing units. Furthermore, the goal is to implement and compare different approaches and variations to AMG in terms of certain performance measures on parallel systems.

In the next chapter I will go into more details on AMG, namely, the different components and certain variations of the algorithm. Then, in chapter 3, I describe certain tools and technologies used for the implementation and the benchmarks. In the following chapter, an explanation of the implementation of the AMG preconditioner, focusing on features, interfaces and challenges, is given. In chapter 5, I will

present benchmark results for the different AMG implementations for different linear systems for both CPU and GPU execution. In the last chapter I will revisit the research goals, summarize the results and discuss possible extensions of my work.

Chapter 2

Components and Variants

In this chapter I will go into more details about the different components and variants of AMG. First, I will discuss the general concepts of AMG, including the core components and the two phases, namely, the setup and precondition phase. Then, I will go into more details about variants of these components, namely the smoother, the coarsening and the interpolation. In the literature, there are many different approaches and I will discuss the most important ones, including the Gauß-Seidel and Jacobi smoother, RS serial and parallel coarsening, aggregation-based coarsening and different interpolation strategies like direct, classical or smoothed interpolation.

Some parts of the AMG algorithm do not require a more detailed theoretical analysis. This includes the coarse-grid correction and also the computation of the Galerkin operator, because these parts are quite straightforward and parallel as these require only matrix-matrix multiplications, matrix-vector multiplications and vector additions and substractions. In that case, the row of a matrix or the entry of a vector can be computed independently of the other rows or entries and therefore parallelism is not an issue. Still, certain aspects have to be taken into account to efficiently exploit parallelism in the implementation, which will be discussed in chapter 4.

2.1 AMG Concepts

Formalizing the AMG approach described in chapter 1, the components of AMG are the following [4, p.212]:

1. The problem is a system of linear equations, formally: $Au = f$ with A the $n \times n$ system matrix, u the $n \times 1$ vector of unknowns and f the $n \times 1$ right-hand-side vector. Using geometric terminology n can be related to the number of grid points on the finest grid while u_i is the value of u at point i.
2. Grids $1, ..., M$ are defined such that on level k there is a number of C^k coarse and F^k fine points. Coarse points make up the grid on the next level $k+1$ until the coarsest level is reached.
3. For every level k, an operator matrix A^k, an interpolation or prolongation matrix P^k, a restriction matrix $R^k = (P^k)^T$ and a smoother S^k are defined. The operator on the finest level is the system matrix: $A^1 = A$.
4. A setup phase is used to build the grids and operators on all levels starting from the finest level (see algorithm 1).
5. In the solve phase a cyclic algorithm is used to perform smoothing and coarse grid correction on the defined levels (see algorithm 2).

Certainly, one drawback of using an AMG preconditioner is the need for a setup phase, causing additional overhead. Furthermore, the data structures built and saved during that phase require additional memory which might be a problem if memory is scarce. Also, in the precondition phase, smoothing and coarse-grid correction operations have to be done on maybe a high number of levels which leads to worse performance per iteration cycle. Usually, using the AMG preconditioner therefore only makes sense for systems with a very large number of points.

2.1.1 Setup Phase

The setup phase is done before the actual computation and is used to build the grids and the operators on all levels. It can be performed in a step-by-step manner,

starting with the finest level until certain criteria are met for the coarsest level. Usually, the criterion is an upper bound on the number of points on the coarsest level to ensure that the direct solver in the solve phase can be run efficiently:

Algorithm 1 (AMG Setup Phase)**:**

1. *Define smoother*[1] S.
2. *Start at the finest level* $k = 1$.
3. *Partition all points into coarse points* C^k *and fine points* F^k. *Coarse points are taken to the next (coarser) level and make up the points on that level.*
4. *Define an interpolation operator* P^k *and a restriction operator* $R^k = (P^k)^T$.
5. *Calculate Galerkin operator matrix* A^{k+1} *on next level:* $A^{k+1} = R^k A^k P^k$.
6. *If coarsest level is reached, set* $M = k$ *and stop. If not, set* $k \leftarrow k + 1$ *and go back to 3.*

Variations of the AMG approach involve the smoother, the coarsening algorithm and the interpolation operator.

2.1.2 Precondition Phase

The precondition phase does the actual computation to improve convergence of the underlying iterative method. It is a recursive procedure that starts on the finest level with a pre-smooth operation and a coarse-grid correction. Except for the coarsest level, the coarse grid correction again uses pre-smoothing and a coarse-grid correction to solve the residual equation and so on. If the coarsest level is reached, then the residual equation is solved using a direct solver and the error is prolonged to the finer level to correct the error. The presented algorithm is called a V-cycle due to the fact that levels are traversed from finest to coarsest and back. A schematic overview is shown in figure 2.1. In the literature sometimes also more complex cycles like W-cycles are presented [2, p.46].

[1] The smoother could be different for pre- and postsmoothing and also on different levels but I assume just one smoother for all levels and purposes.

Algorithm 2 (AMG Precondition Phase (V-cycle))**:**

- *If* $k = M$*, use a direct solver to solve* $A^M u^M = f^M$*.*
- *Else, perform V-cycle:*
  1. *Do* μ_1 *pre-smooth iterations of* S *for* $A^k u^k = f^k$*.*
  2. *Do coarse-grid correction:*
     (a) *Calculate residual from smoother:* $r^k = f^k - A^k u^k$*.*
     (b) *Restrict residual to next level:* $r^{k+1} = R^k r^k$*.*
     (c) *Set* $f^{k+1} = r^{k+1}$ *and repeat algorithm for* $k = k + 1$*.*
     (d) *Prolongate error from coarser level:* $e^k = P^k u^{k+1}$*.*
     (e) *Correct error/solution:* $u^k \leftarrow u^k + e^k$*.*
  3. *Do* μ_2 *post-smooth iterations of* S *for* $A^k u^k = f^k$*.*

2.1.3 Metrics

To discuss aspects of different AMG algorithms and approaches, several metrics have to be used, both in a theoretical discussion and for benchmarks. Regular metrics are convergence and solver time, but for AMG some more have to be used: First, the setup time plays a role, especially for systems for which the solver time is rather small. Also, one needs a metric for memory usage of the internal data structures built during setup. In the literature this is called complexity and two different metrics are usually defined [4, p.213]:

- Operator complexity is the sum of the number of non-zero coefficients on all levels A^k divided by the sum of the number of non-zero coefficients for the system matrix A^1.
- Average stencil size is the average number of non-zero coefficients per row in the operator matrix on a certain level. This shows average memory usage on a certain level.

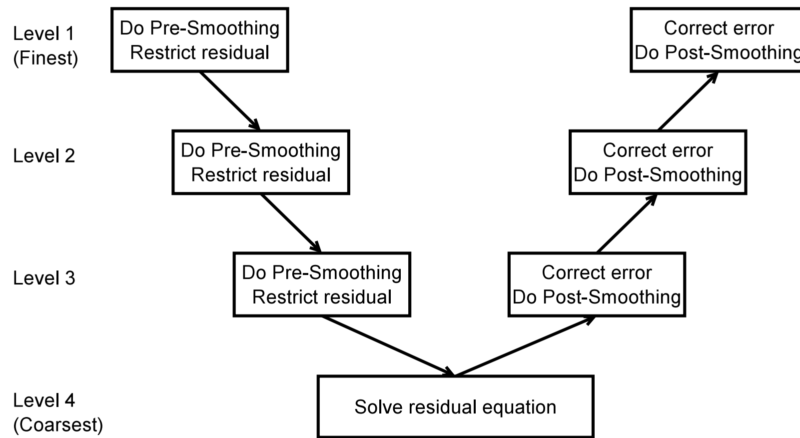

Figure 2.1: Schematic overview of a V-cycle with four levels: Coarse grid correction is done on the fine levels while the residual equation is solved on the coarsest level.

Note that both metrics also give some hint on setup and solve time as high numbers of operator complexity and average stencil size mean that the number of certain operations, for example a matrix-vector product, in both the setup and precondition phase is high. Another metric that has to be discussed from a theoretical standpoint is parallelism: As will be shown in the next sections, some variations are sequential in nature while others allow parallel execution.

2.2 Smoother

An important part of the AMG precondition phase is the smoother S. Smoothing is usually both done before and after a coarse-grid correction. In the first case, this is useful as then interpolation is able to approximate the overall error well. In the second case it is useful as non-smooth error components from the correction can be damped efficiently. Traditionally, AMG uses Gauß-Seidel relaxation which has been shown to be a good smoother for many problems [4, p.226]. The component form of the Gauß-Seidel iteration is defined as[2] [1, p.107]

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(-\sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k + b_i \right), \qquad i = 1, ..., n$$

[2] x_i^k denotes component i of vector x at iteration step k, a_{ij} is the component of matrix A at row i and column j, b_i is component i of vector b.

Clearly, Gauß-Seidel is a sequential iteration as for the computation of vector component i, components $0, ..., i-1$ have to be computed first. A less effective [4, p.227] but fully parallel approach is the Jacobi relaxation which only uses components of the last iteration step [1, p.106]:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(- \sum_{j=1, j \neq i}^{n} a_{ij} x_j^k + b_i \right), \qquad i = 1, ..., n$$

Here, in principle, vector components can be computed all at the same time.

One extension to these basic relaxation procedures is to weight the respective result of the iteration with a certain weight parameter ω [1, p.431]. Using this approach for the Jacobi relaxation leads to

$$x_i^{k+1} = \omega \frac{1}{a_{ii}} \left(- \sum_{j=1, j \neq i}^{n} a_{ij} x_j^k + b_i \right) + (1-\omega) x_i^k, \qquad i = 1, ..., n$$

Here, the new vector is built by using a fraction ω of the Jacobi relaxation and a fraction $(1-\omega)$ of the old vector from the previous iteration step. This approach is still parallel but has improved smoothing properties similar to the ones by Gauß-Seidel [3, p.30]. The parameter ω is usually chosen between 0 and 2 and a typical choice is $\omega = \frac{2}{3}$, although the optimal parameter usually depends on the actual problem [1, p.431]. Note that for $\omega = 1$ the weighted Jacobi relaxation is equivalent to the basic Jacobi relaxation.

Another extension that is suggested for the Gauß-Seidel smoother is to first smooth the coarse points and then the fine points for pre-smoothing and in reverse order for post-smoothing [3]. This approach is called C-F (or F-C) smoothing. Of course, this can also be done for the Jacobi relaxation with the advantage that it can be implemented in parallel [4, p.227]. Many more ideas can be found in [4, p.227-229], like hybrid smoothers, multicoloring approaches, polynomial smoothers and also other preconditioners could be used as smoothers, for example the approximate inverse approach or incomplete LU factorization. It is, however, not clear how effective those smoothers are: They usually seem to need good matrix- or problem-dependent parameter choices or involve a costly setup, for example the computation of eigenvalues, to work well.

2.3 Coarsening

Coarsening is the part of the setup phase where points are chosen to become either coarse (C) points taken to the coarser grid, or fine (F) points for which the error is interpolated from the coarse point errors. Choosing coarse points is crucial to the effectiveness of the AMG algorithm and different approaches consequently show very different properties in terms of complexity and convergence (see chapter 5).

2.3.1 RS Coarsening

The traditional coarsening algorithm described in [3, p.63-76] and in [4, p.216f] is usually called RS (Ruge-Stüben) approach. The basic idea of this approach is that points (or variables in the algebraic interpretation) are connected to each other via couplings that depend on the matrix coefficients between those points. A point i is then strongly coupled to (or dependent on or strongly influenced by) a point j if

$$-a_{ij} \geq \theta \max_{k \neq i}(-a_{ik}).$$

The assumption is that the diagonal element of a certain row i is positive while the non-diagonal elements are negative. If the diagonal is negative and the non-diagonals are positive, the same criterion holds except that the signs have to be changed. θ is the strength of threshold parameter and is set to $0 < \theta < 1$. Note that whether a point is strongly influenced by another point is also dependent on the highest negative coefficient in the respective row of matrix A and therefore dependent on the couplings to all the other points.

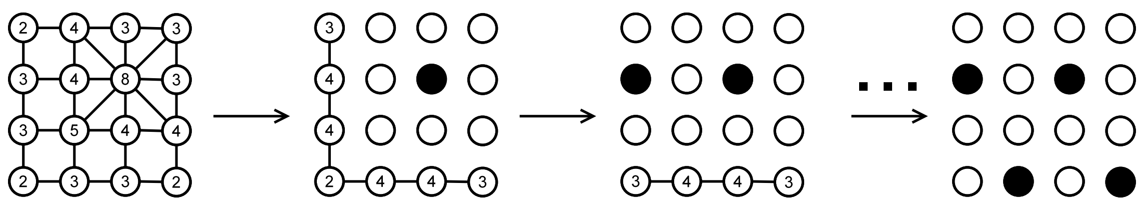

Figure 2.2: First pass [4, p.215]. Numbers denote the current state of λ_i for point i. Black points are C points, white points without number F points.

The coarsening algorithm then proceeds with two passes: The first pass described in algorithm 3 and shown graphically in figure 2.2 generates C and F points such that there are enough C points for interpolation but as few as possible to minimize complexity.

Algorithm 3 (RS coarsening, 1st pass)**:**

1. *Every point i is associated with a measure λ_i that stores the number of points that are strongly influenced by i.*
2. *Pick a point with the maximum value λ_i to be C point*[3]*.*
3. *All points that are strongly influenced by this new C point become F points.*
4. *For every point that has become F point in the last step: Increment λ_i if point i strongly influences an F point.*
5. *If there are points left that are neither C nor F point, go back to 2.*

The second pass checks whether all strong influences between pairs of F points are connected to a common C point. This is described in algorithm 4 and shown in figure 2.3. If the check fails for a F-F connection, then more C points are created to improve interpolation and convergence at the expense of a higher complexity[4].

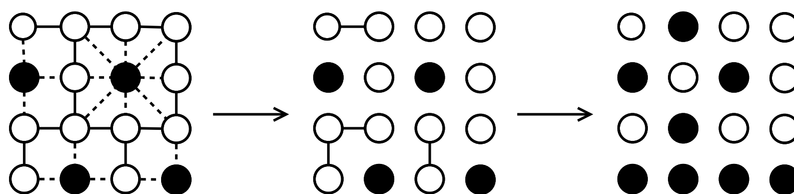

Figure 2.3: Second pass [4, p.215]. Left figure: Strong lines denote F-F connections, dotted lines C-F connections. Central figure: Strong lines denote F-F connections without common C point connection. Right figure: C points are added.

[3]There might be more points with maximum λ_i. If that is the case then a point can be picked in an arbitrary way.

[4]Note that usually there are many options of how C points can be created to fulfill that requirement.

Algorithm 4 (RS coarsening, 2nd pass)**:**

1. *Pick one of the F points.*
2. *If F point is strongly influenced by another F point, check whether both are influenced by a common C point.*
3. *If not, switch one of the mentioned F points to C point.*
4. *If not all F points are checked, pick an unchecked F point and go back to 2.*

The first pass makes those points C points that have a high influence on other points such that a minimal number of C points are selected. The second pass ensures that if one F point is strongly influenced by another F point, there is also one C point both F points are strongly coupled to. This improves interpolation although it increases the number of C points and therefore a higher complexity on the coarser levels is to be expected.

Even though RS-coarsening works very well for many applications, the main drawback of the algorithm is that it is highly sequential, only the computation of the dependence and the first step of the first pass can be done in parallel. Other related coarsening procedures are one-pass-RS-coarsening, where only the first pass of the algorithm is carried out, and aggressive coarsening [4, p.216]. For the latter, strong relations are defined via paths of points. In this terminology, RS is a special case with paths of unit length. The algorithm then proceeds with the first pass of RS coarsening (algorithm 3) and runs another RS first pass on the C points from the first run. Altogether, only those points become C points that are flagged as C points in both runs.

Still, all of these approaches are single-threaded and therefore ideas have been developed to parallelize the coarsening procedure. One idea is to partition all the points into subsets (partition the matrix into many smaller matrices) and then coarsen each set of points independently [4, p.217] [5, p.162-164]. This is called RS0 coarsening and is efficient in the sense that coarsening can run in parallel on (theoretically) as many processors as are available. To make this work, certain couplings between points have to be neglected during coarsening as influencing points

may reside on other processors. Therefore, in general, worse convergence is to be expected. RS3 coarsening improves convergence by, after running RS0 coarsening, introducing a third pass which is similar to the second pass of RS coarsening but is only run on the processor boundary points - points which are influenced by points on a different processor. This third pass therefore adds more C points and is again sequential in nature.

2.3.2 CLJP Coarsening

A problem of both RS0 and RS3 coarsening is that at least one C point has to exist on the coarsest level on each processor, which is a problem if the number of processors becomes very large. In that case the direct solver on the coarsest level might become quite slow as the overall number of points stays relatively large. A different parallel approach that does not have this disadvantage is CLJP coarsening [4, p.218-220] [5, p.160-162]. Here, points are picked as C points if their influence measure λ_i is greater than the same measure for all points that either influence this C point or are strongly influenced. This can be done in parallel. To be able to find unique maximum points, random values between zero and one are added to λ_i for all points. Now, instead of making all points strongly influenced by those C points to F points, the value λ_i is decremented by the number of strongly influencing C points. This is also done for all influencing point pairs that have a common C point. In the end, only points for which $\lambda_i < 1$ become F points. The process is then repeated until all points are either C or F points. Note that for a parallel execution, information between processors has to be exchanged about updated values λ_i for strong influences between processor boundaries.

2.3.3 Other RS-related Strategies

Although CLJP in general works well, complexities are rather high as can be seen in [4, p.227ff]. A similar approach, which however leads to less complexity, is PMIS coarsening [4, p.220]. The difference to CLJP is the fact that for PMIS all points immediately become F points when points are strongly influenced by C points. Subdomain blocking is another parallel approach that coarsens from the boundary

points to the inside [4, p.220]. The Falgout approach is a combined approach that uses RS0 coarsening on finer levels while proceeding with CLJP for coarser levels [4, p.221] [5, p.164-165]. The advantage is that this leads to lower complexities than CLJP and avoids the RS0 problem that at least one C point has to exist per processor. HMIS coarsening combines RS0 and PMIS, using the former for finer and the latter for coarser levels [4, p.221].

2.3.4 Aggregation

A completely different approach to coarsening is called coarsening by aggregation, which uses a different strength of dependence measure [3, p.112-117] [4, p.216] [6]. A point i is strongly connected to a point j if

$$|a_{ij}| \geq \theta\sqrt{|a_{ii}a_{jj}|}.$$

The idea of coarsening by aggregation is to form aggregates of points that are represented as one point on the next coarser level. This concept is developed for symmetric matrices, although it can be in principle also used for non-symmetric ones.

Algorithm 5 (Coarsening by Aggregation)**:**

1. *Build neighborhoods N_i, which are sets of points that include point i and all points connected to i.*
2. *Pick one neighborhood N_i by selecting a root point i.*
3. *Make this neighborhood an aggregate. The root point i of the aggregate can be interpreted as C point while the others are F points.*
4. *If there are points included in a neighborhood but not in an aggregate, pick another N_i that is not included in an aggregate and go back to 3.*
5. *If there are points left that are not in an aggregate, either join them with existing aggregates or build new ones.*

Coarsening by aggregation usually leads to lower complexities than RS coarsening, but this depends on the interpolation used (see section 2.4.4). The original aggregation-based AMG uses a very simple interpolation procedure where all points in an aggregate just use the same value for all points in the aggregate. Smoothed aggregation on the other hand also interpolates from other aggregates, which also leads to higher complexities from the Galerkin operator. Note also that coarsening by aggregation is a sequential algorithm with the exception of the first step. However, decoupled coarsening can be used similar to RS0 coarsening by splitting the points into disjoint sets that are coarsened on different processors.

2.4 Interpolation

Interpolation is used to prolongate the error from a coarser level to the respective fine level. The usual approach for C points is to use the same value as the respective value on the coarser level, while F point values are linear combinations of C point values. Which C points to interpolate from and which interpolation weights to use is determined by the respective interpolation algorithm.

Formally, interpolation defines weights w_{ij} such that for all F points i and coarse interpolatory points $\tilde{C}_i$[5]:

$$e_i = \sum_{j \in \tilde{C}_i} w_{ij} e_j$$

The basis for interpolation is a smooth error, defined in the algebraic sense as $Ae \approx 0$ such that the error for point i can be approximated by the equation

$$a_{ii} e_i + \sum_{j \in N_i} a_{ij} e_j = 0,$$

where N_i is the set of all neighboring points.

[5]Usually, the set of coarse interpolatory points is the same as the set of strongly influencing C point neighbors. For aggregation-based interpolation it is usually the C point that is the root of the respective aggregate.

2.4.1 Direct Interpolation

Equation weights are then set such that the approximation is as good as possible. The problem is that for the interpolation from a coarser level only the points C_i (the points that appear on that coarser level) can be used and not all points N_i. However, interpolation is still possible in a variety of ways. One approach used for RS and related coarsening procedures is direct interpolation [3, p.39f] [4, p.223]. Here, influences from F points are neglected with the underlying assumption that the approximation can be done using only direct connections to C points. Then, weights can be defined as

$$w_{ij} = -\left(\frac{\sum_{k \in N_i} a_{ik}}{\sum_{l \in C_i} a_{il}}\right) \frac{a_{ij}}{a_{ii}}. \tag{2.1}$$

Certainly, this approach only makes sense when a high portion of influence of an F point can be captured in the neighboring C points. This has to be accomplished by the coarsening procedure used, otherwise interpolation and in turn convergence will suffer. This is one reason why the distinction of strong and weak dependence is used and the coarsening algorithms are tailored such that points with many strong connections become C points. Still, most applications of the algorithm leave strong F connections and therefore convergence for direct interpolation can be worse than for other interpolation approaches. Still, the formula is rather simple and can be computed in parallel such that good setup times are to be expected.

To reduce complexity, interpolation truncation can be used to restrict the number of interpolating C points [3, p.74]. Then, interpolation is only done from points for which the interpolation weight is larger than a certain threshold ϵ times the maximum weight per point[6]. All the weights that are smaller than the threshold are set to zero while the others are scaled such that the total sum of interpolation weights per point stays the same. Then, complexity is smaller as both the interpolation and the Galerkin operator become more sparse and, if the weight is chosen correctly, interpolation does not suffer as only small weights are neglected.

[6] Choosing $\epsilon = 0.2$ is suggested in [3].

2.4.2 Classical Interpolation

A more sophisticated interpolation formula is classical interpolation [4, p.222f]. Here, the distinction is made not only between strongly influencing C points C_i and other points, but also between strongly influencing F points F_i^s and weakly influencing points F_i^w. Now, connections are treated differently for weak and strong F connections if the pair of strongly connected F points has a common C point:

$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in F_i^w} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik} a_{kj}}{\sum_{l \in C_i} a_{kl}} \right) \tag{2.2}$$

This is always the case for RS coarsening when the full algorithm is used, as the second pass makes sure that any strong F connection has a common C point. This is also the case for RS3 coarsening, but cannot be guaranteed by one-pass or RS0 coarsening. In that case, the interpolation formula breaks down as the right denominator becomes zero. The computation can again be done in parallel but is more complex such that setup times are expected to be worse than for direct interpolation. However, as more information is used for the interpolation (C point influence over strong F connection), convergence should improve.

Experience suggests that classical interpolation works very well if the system matrix is close to an M matrix, where coefficients in a row have the opposite sign of the diagonal coefficient in that row. However, if the matrix for the system to be solved has many non-diagonal coefficients with the same sign as the diagonal, then interpolation weights might become very large as coefficients in the denominator of the right sum in equation 2.2 might cancel out [5, p.166]. This can lead to bad interpolation performance and even divergence in some cases.

A proposed modification for this problem is to define weights

$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in F_i^w} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik} \hat{a}_{kj}}{\sum_{l \in C_i} \hat{a}_{kl}} \right) \tag{2.3}$$

such that

$$\hat{a}_{ij} = \begin{cases} 0, & \text{if } sign(a_{ij}) = sign(a_{ii}), \\ a_{ij}, & \text{otherwise.} \end{cases}$$

In this case, only coefficients are used which have a different sign as the diagonal. Depending on the actual matrix, this might also lead to worse convergence as certain strong F connections are neglected. On the other hand, this ensures that coefficients in the denominator in the right sum have the same sign, eliminating the divergence problem.

2.4.3 Other Interpolation Procedures

Even more information is used for standard interpolation [3, p.70f] [4, p.215]. It works like direct interpolation but eliminates all strongly influencing F points F_i^s from the equation by approximating these first:

$$e_j = -\sum_{k \in N_j} \frac{a_{jk} e_k}{a_{jj}}$$

Then, the error equation resembles a larger neighborhood, namely all C points that strongly influence i or strongly influence an F point that strongly influences i:

$$\hat{a}_{ii} e_i + \sum_{j \in \hat{N}_i} \hat{a}_{ij} e_j = 0$$

The coefficients in the equation are the combined coefficients from the elimination of the points in F_i^s together with the original ones and $\hat{N}_i$ is the extended neighborhood. For this equation, direct interpolation can be used for the interpolation weights. Standard interpolation offers very good convergence but the computation of the extended neighborhood leads to higher setup times.

Multipass interpolation is used for coarsening procedures for which not every F point has a C point connection, for example with aggressive coarsening [4, p.223]. Here, direct interpolation is used for all F points with a C point neighbor and standard interpolation for the others.

2.4.4 Interpolation for Aggregation-based AMG

The basic idea of aggregation-based AMG is to identify all values in an aggregate with the aggregate value on the coarse level, resulting in interpolation weights of 1 if the point is part of the respective aggregate, and 0 if not [3, p.112]. This is equivalent to interpolating an error at an F point exactly by the error at the C point that corresponds to the respective aggregate for which the C point is the root point. As this interpolation can be implemented in parallel and furthermore the interpolation is very simple, one can expect fast setup times and low complexity. However, as only one C point is always used for interpolation, the approximation is not very good compared to other approaches. Therefore, convergence of the basic aggregation-based method is relatively bad.

Smoothed aggregation uses the same approach, however, the interpolation is improved by doing one step of a weighted Jacobi smoother such that a broader interpolation base is used, including values from other aggregates [6]. In matrix form, if $\hat{P}$ is the basic prolongation matrix at a certain level, the final interpolation is determined by computing

$$P = (I - \omega D^{-1} A^F)\hat{P}$$

I is the identity matrix, ω the weight of the Jacobi method[7], D the diagonal of the operator matrix A and A^F the filtered matrix. The latter is constructed from A by transferring all off-neighborhood coefficients into the diagonal to decrease complexity (see [6] for more details).

Nevertheless, complexity is usually increased compared to the basic aggregate interpolation, but convergence is improved due to the broader interpolation base. Furthermore, the additional matrix product leads to higher computation times for the smoothed aggregation approach and therefore also to higher setup times.

[7] $\omega = \frac{2}{3}$ is often advised in the literature.

2.5 Summary

This chapter shows that AMG is in fact not a single method or algorithm but a very broad concept that can be tweaked in a variety of ways. This includes certain parameter choices, for example for the strength of dependence, but even more so the respective coarsening and interpolation schemes used. Usually, there is a trade-off involved between setup time and operator complexity on one hand and convergence on the other.

It is important to note that not all combinations of coarsening and interpolation work well together: The aggregation-based coarsening methods have to be combined with aggregation-based interpolation, aggressive coarsening needs multipass interpolation and one-pass coarsening cannot be used together with classical interpolation. However, in the end it is up to the user which approach to apply to a certain problem. This of course includes the trade-off mentioned above, but also the underlying physical problem that might be more suitable for one approach or the other. This is certainly also the case for the partitioning approaches like RS0 or RS3 which only make sense if the off-processor influences are small.

Chapter 3

Tools and Technologies

In this chapter I want to discuss and summarize certain tools and technologies used for the work on the practical part of this thesis. This includes certain parallel programming tools (OpenCL, OpenMP) and libraries (ViennaCL, uBlas), a tool to renumber matrix indices and a description of the used hardware.

3.1 OpenCL

OpenCL (Open Computing Language) is an open computing standard for parallel architectures and supports many different heterogenous platforms [7]. OpenCL drivers are available for many different vendors including NVIDIA (GPU), AMD (GPU and CPU) and Intel (CPU). OpenCL includes a runtime, a compiler and a C dialect programming language also usually called OpenCL. It therefore offers a convenient way to develop applications which efficiently use the parallel execution units available on a given device. OpenCL applications are usually called kernels.

OpenCL uses a host/device-based platform model: An application runs on a host which controls and issues streams of instructions (kernels) to the devices which are executed on the processing elements of these devices. A typical example of such a model is a CPU-host that runs the program but delegates certain tasks to GPU-devices. To distinguish between the different execution units, an index space

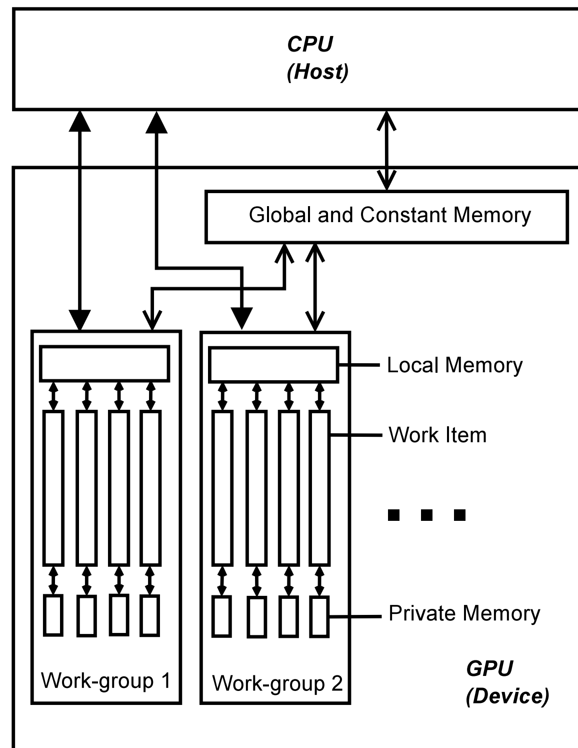

Figure 3.1: Simplistic overview of the OpenCL architecture including memory for a CPU host and a GPU device with 4 work-items per work-group.

is used which separates the parallel ressources on the device in work-items such that every work-item is associated with a unique index. Each work-item executes the same code in parallel, but this code can be index-dependent such that different calculations are possible (see below).

Work-items are further grouped into work-groups which allows for a better composition of work-items for execution. Furthermore, synchronization barriers can be used inside work-groups to control the execution on work-items. OpenCL uses four different types of memory: Global memory that is shared between all work-items, constant memory which holds constants for all work-items, local memory which is shared among a work-group and private memory only visible to a certain work-item. To copy memory between host and an OpenCL device the global memory is used which is controlled by the host but can be accessed from the work-items on a device. A schematic overview of the architecture for a typical CPU-GPU system is shown in figure 3.1.

The OpenCL C language is used to develop kernels which execute on the work-items. The most important part is the handling of the parallel execution using the work-item id. A typical code to distinguish the execution between different work-items is the following:

```
for (i = get_global_id(0); i < max_i; i += get_global_size(0))
{
  // Computation for work-item i
}
```

Here, `get_global_id(0)` returns the id of the work-item, while `get_global_size(0)` returns the total number of work-items available. Work-item 0 in this case computes values for `i = 0, get_global_size(0), 2 x get_global_size(0)` and so on. This can be used for example for vector manipulations where vector-entries with index `i` are computed in parallel.

ViennaCL offers an API for C++ and Fortran to write code for an OpenCL host (see next section). This includes memory handling, device control and a compiler for OpenCL code that is run on the device. From a performance point of view, this introduces some overhead compared to a host-only application mostly due to data transfer from host to device but also due to kernel calls and for the device handling. A performance gain from using OpenCL is therefore only to be expected if the difference in computation time between a host-only application and a combined application is at least as large as this overhead.

3.2 ViennaCL

ViennaCL (Vienna Computing Library) is a computing library written in C++ [8]. The current version 1.1.2 offers a variety of linear algebra functions (BLAS level 1, 2 and 3), a direct solver (LU decomposition), three iterative solvers (CG, BiCGStab, GMRES) and three preconditioners (LU with threshold, Jacobi, row normalization).

ViennaCL further offers a variety of data types for mathematical computation like vector, dense and sparse matrix types. There are also many functions for data

manipulation like matrix-vector multiplication or matrix-matrix multiplication. The library is distributed as C++ header files and can therefore easily be used in any C++ project. ViennaCL further has a similar interface to Boost uBlas (see section 3.4) and offers wrappers to uBlas, Eigen [9] and MTL4 [10].

The main feature of ViennaCL is its ability to optionally use GPUs or multi-core CPUs to speed up computation. For this purpose, ViennaCL uses OpenCL and offers a convenient way to access OpenCL functionality. This include handling of OpenCL data objects and memory and certain convenience functions for OpenCL source code:

- OpenCL kernels usually have to be programmed as C++ strings which is not very convenient. ViennaCL offers a tool such that OpenCL source code can be written into source files which are then translated into C++ strings.
- ViennaCL furthermore checks whether an OpenCL device and driver provide double precision support. This is currently not included in the standard but is nevertheless implemented in some devices as an extension. If double precision is supported, then the OpenCL code is adjusted for this purpose.
- ViennaCL also eases the OpenCL code compilation: Kernels are associated with certain data types (e.g. compressed_matrix) and are automatically pre-compiled when an object of this type is created. This can even improve performance because compilation is only done once instead of every time a kernel is to be run.

ViennaCL is therefore a library to ease the implementation of parallel applications. A programmer can either use the built-in functionality or use the framework to implement his or her own parallel computation code. Of course, ViennaCL functionality can also be used for plain single-threaded computation just like Boost uBlas, for example when the underlying algorithm cannot be parallelized in a reasonable way.

Benchmark results show that ViennaCL indeed offers a performance advantage over single-threaded uBlas computations whenever massive parallel execution of a certain large computation is possible. Then, the superior parallel computing power

of a GPU outweighs the OpenCL overhead which includes memory transfers, kernel calls and device handling[1]. For more details, see the benchmark results in the ViennaCL manual and those in chapter 5.

3.3 OpenMP

OpenMP (Open Multi-Processing) is an API for multithreaded programming in C, C++ and Fortran in a shared memory environment like multi-core CPUs and supports many platforms and compilers including GCC [11]. The difference to OpenCL is the way parallelism is used: In case of OpenCL by several workers on an OpenCL device, for example a GPU or multicore CPU, in case of OpenMP by threads on the CPU. A single-threaded C++ program can therefore be altered to a parallel program if the execution or algorithm allows it. The main advantage of OpenMP is therefore that single-threaded code can easily be altered to parallel code using simple code constructs to for example run a for-loop in parallel:

```
#pragma omp parallel for private (var_list1) shared (var_list2)
for (unsigned int i = 0; i < 10; ++i)
{ ... }
```

In this code example the compiler runs the loop by starting threads with the interior code for i=0,1,2,...,10 in parallel. Of coarse, the calculations in those iterations have to be independent of the other iterations. OpenMP further offers ways to deal with variables by using the `private` and `shared` list. Private variables can only be accessed by the respective thread while shared variables can be accessed by all threads in parallel. Variables that are created inside the loop-block are always treated as private variables.

Parallel write-access to shared variables, however, is problematic and can result in race conditions and even memory errors ("Segmentation Fault"). Therefore, the directive `#pragma omp critical` makes sure that the block below the directive

[1] A typical example for the differences in raw computing power between CPU and GPU is given in section 3.6.

is only executed by one thread at a time. There are many more ways to include or control OpenMP parallel execution in C++ code, for example by using parallel sections, tasks, singular regions, atomic operations and barriers. However, many problems can already be programmed using `parallel for` and `critical`. OpenMP further defines global functions to control parallel execution and the system environment. The number of available processors/threads for example can be determined by calling `omp_get_num_procs()`.

OpenMP is therefore a very useful tool when parallel execution on a CPU is required. For this case, OpenMP offers a lightweight approach using shared memory and threads with less overhead than OpenCL.

3.4 uBlas

uBlas is part of the Boost libraries and offers mathematical functionality for the C++ language by providing certain mathematical data types (e.g. vector or matrix in different formats) and operations [12]. uBlas also provides a lot of linear algebra functionality (Blas level 1, 2 and 3). ViennaCL is interface-compatible to uBlas and offers some of the same features, but the main difference is that uBlas is a single-threaded CPU-only library[2].

In this work uBlas is used as both a basis for the development of the preconditioner (see chapter 4) but also as a way to compare CPU versus GPU performance (see chapter 5).

3.5 Renumbering Tool

For certain coarsening strategies (RS0, RS3) the space of points has to be grouped into different sub-spaces such that parallel execution of the coarsening algorithm is possible. For this to make sense, the grouping has to be such that as many strong connections as possible are retained, or to put it in another way: As few as possible

[2]There are of course many more differences as for example uBlas does not offer iterative solvers.

strong connections should be cut. This is tough to do if neighboring points have very different indices. Therefore, a renumbering of the points eases the development of such a coarsening algorithm as then neighbor points also have close indices.

A simple example can be derived from the matrix defined in equation 1.3. By switching indices for points 2 and 4 as well as 3 and 5, the matrix

$$\tilde{A} = \begin{pmatrix} 2 & 0 & 0 & -1 & 0 \\ 0 & 2 & -1 & 0 & -1 \\ 0 & -1 & 2 & -1 & 0 \\ -1 & 0 & 0 & 2 & -1 \\ 0 & -1 & 0 & -1 & 2 \end{pmatrix}$$

is obtained, which clearly has a higher bandwidth than the original matrix. Although the mathematical interpretation of this matrix is equivalent and there are no negative effects on performance if for example non-zero iterators are used, the situation is different if the matrix is split into sub-matrices. For example, a splitting into two matrices of size 2 and 3 where A^1 and A^2 are the parts of the original matrix and $\tilde{A}^1$ and $\tilde{A}^2$ the parts of the renumbered matrix leads to

$$\tilde{A}^1 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \qquad A^1 = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}$$

$$\tilde{A}^2 = \begin{pmatrix} 2 & -1 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} \qquad A^2 = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$$

Clearly, the matrices are different and the splitting of the lower-bandwidth-matrix retains more connections between points than the one with the higher-bandwidth-matrix. Therefore, in practice, the quality of the coarsening of RS0 and RS3 depends very much on the numbering of the points in a matrix with the effect that an optimal result can be obtained for a numbering for which the matrix has the lowest bandwidth possible.

However, matrices in practice are not necessarily numbered in an optimal way. For RS0 and RS3 it is therefore useful to first employ a renumbering tool that works on the system matrix and renumbers the points such that the matrix is diagonally

dominant. There are different algorithmic approaches to doing such a renumbering and I used an implementation of the Reverse Cuthill-McKee algorithm described in [13][3]. The algorithm uses two sets: R to order points and Q to save neighboring points. First, the point with the fewest neighbors is chosen and put into the first free place in R, all the neighbors in order of their number of neighbors into Q. Then, the algorithm prioritizes points in Q and does the same operation again: Transfer the point with the fewest neighbors to R and it's neighbors to Q. This is done until all points are in R. In the end, the ordering of R is reversed.

The tool is able to read and write matrices in the matrix-market format which is also used in ViennaCL. After computation, the bandwidth of the matrix is minimized, meaning that the indices of neighboring points are a lot closer to each other. The mentioned tool might be included in ViennaCL in later versions, until then however, a user of the coarsening algorithms RS0 and RS3 is advised to use a similar renumbering tool before doing the coarsening.

3.6 Hardware

The software was developed and benchmarked on a personal computer equipped with an Intel Core i7-960 (3,2 GHz), 12 GB RAM memory and a NVIDIA GeForce GTX 470 (OpenCL driver version: 270.41.19) with a maximum of 14 compute units and 1024 OpenCL work-items on Funtoo Linux (64 Bit) and kernel version 2.6.38. The processor has 4 physical cores and can run a total of 8 threads via OpenCL or OpenMP. The theoretical total computing power of the CPU is about 50 GFlops [14] while for the GPU it is about 550 GFlops [15]. Although these theoretical values have to be taken with a a grain of salt, it shows nicely how the GPU can improve performance if the software is able to exploit its parallel architecture for problems for which the ViennaCL/OpenCL overhead is small enough.

[3]The tool was developed by Philipp Grabenweger at the Institute for Microelectronics, Vienna University of Technology in 2011 for his Bachelor thesis.

Chapter 4

Implementation

During the work on this thesis a C++ implementation of an AMG preconditioner has been developed for ViennaCL. It features support for both CPU and GPU computing devices and can be used with the already existing iterative solvers in ViennaCL. The AMG preconditioner can be customized by the user as several coarsening and interpolation schemes can be chosen and combined. Other options include the number of coarse levels set manually or automatically by the algorithm during setup, the number of pre- and postsmooth cycles and the strength of dependence threshold parameter for the setup phase (see table 4.1 for a full list).

The implementation combines several approaches: While the precondition phase can be run both on the CPU and GPU by just using CPU or GPU vectors and matrices as input, the setup phase runs on the CPU only. The reason for this design decision was that the precondition steps are rather straightforward, involving a smoother, matrix-vector multiplications, vector-vector additions and a direct solver. This can be developed easily by using the existing ViennaCL framework and some OpenCL code[1]. The setup phase on the other hand is rather complex, involving the handling and build-up of many different data structures, a task for which GPUs are not very efficient. Furthermore, the setup phase involves many data transfers which would also slow down a GPU implementation. Still, even though the setup phase is CPU-only, the implementation can be speed up by using OpenMP on a

[1] The direct solver also runs on the CPU right now but will run on the GPU as soon as ViennaCL supports pivoting for the LU factorization.

multicore CPU. For the GPU implementation, the data structures are copied to the GPU after the setup phase has been completed on the CPU.

The software therefore takes advantage of the strength of the respective architecture: While the general purpose CPU is efficient for more complex tasks including sorting, dynamic memory handling and conditional branching, the GPU is efficient for standard and repetitive parallel tasks like vector computations or iterations. While important parts of the setup phase like the construction of coarsening and interpolation consist of many tasks of the former, the precondition algorithm is solely based on tasks of the latter. On both architectures the implementation makes full use of the available computing units by taking advantage of parallelism as much as possible.

4.1 Code and File Organization

The code of the AMG preconditioner is distributed among several files and can be found in *viennacl/linalg/*. All the classes and global functions are also built into the namespace *viennacl::linalg*.

- *amg.hpp*: The main file contains both classes for the AMG preconditioner (CPU and GPU version), the precondition methods including the smoothers and certain global functions for the setup phase. These functions build the internal data structures, run the setup cycle and copy data to the GPU (if necessary).
- *amg_base.hpp*: This file is included in all the other files and contains the internal data structures and several functions, including the computation of the Galerkin operator.
- *amg_coarse.hpp*: In this file all the different coarsening algorithms are included.
- *amg_interpol.hpp*: This file contains the different interpolation algorithms.
- *amg_debug.hpp*: To ease code debugging, functions were written to print certain matrices or vectors or write them to a file. These are included here.

For debugging and benchmarking I tweaked the built-in ViennaCL solver benchmark to support the AMG preconditioner. This source file can be found in the ViennaCL benchmark directory *examples/benchmarks* and is named *amgbench.cpp*.

ViennaCL enforces certain coding standards for the preconditioners such that they can work with the existing iterative solvers. First, the class and template structure has to match both for the CPU and GPU type. The default implementation which uses the CPU for all computations uses the structure

```
template <typename MatrixType>
class amg_precond
```

while the specialization for GPU computation via ViennaCL and OpenCL uses

```
template <typename ScalarType, unsigned int MAT_ALIGNMENT>
class amg_precond< compressed_matrix<ScalarType, MAT_ALIGNMENT> >
```

The templates for the class `amg_precond` therefore enable the choice between CPU or GPU implementation. In the latter case the class is templated by the type `viennacl::compressed_matrix` using a `ScalarType` (`float` or `double`) and a memory alignment `MAT_ALIGNMENT`. For all other classes the CPU implementation is used and `MatrixType` shows the type of the system matrix, for example `ublas::compressed_matrix`. If common functionality is needed for both classes which is the case for most of the setup phase, then global functions are used.

The precondition operation is done by using a member function of these classes, in both cases using the following interface:

```
template <typename VectorType>
void apply(VectorType & vec) const
```

Note that the method is `const` and therefore data structures inside the class do not change. `vec` is the vector of type `VectorType` (e.g. `viennacl::vector` or `ublas::vector`) to which the preconditioner is applied to. From the point of view

of the AMG preconditioner this is the right-hand-side of the precondition equation to be solved which is equivalent to the residual of the system equation (see chapter 1).

4.2 Data Structures

An important aspect of the implementation are the data structures. Those are not only important to exchange information between the user and the implementation but also to efficiently transmit the relevant information between different parts of the preconditioner which often factors into the overall performance quite substantially. This includes information about the system matrix and its coefficients, information about the influence between different points and about the division of information between different threads and so on. In the following section I will explain how those data structures are set up and used and the reasons behind the most important design decision.

4.2.1 User Options

To setup the properties of the AMG preconditioner a tag-class `amg_tag` is used, which is distributed to the AMG classes and saves the user options for the algorithm. Options include the coarsening and interpolation strategy used, parameters for the smoother, the strength of dependence calculations and many more. The options can be set via the constructor and getter/setter functions for every parameter. Table 4.1 shows the implemented options with the restrictions on the values and the default value.

4.2.2 Sparse Matrix

Most of the data structures used for AMG are in a matrix format which includes the operator matrix, the restriction and the prolongation matrix. To use memory efficiently it makes sense to use a sparse matrix format that only saves non-zero

Option Name	Description	Value Restrictions	Default
coarse	Coarsening algorithm. (see sections 2.3 and 4.3.1 for descriptions)	VIENNACL_AMG_COARSE_RS (=1) VIENNACL_AMG_COARSE_ONEPASS (=2) VIENNACL_AMG_COARSE_RS0 (=3) VIENNACL_AMG_COARSE_RS3 (=4) VIENNACL_AMG_COARSE_AG (=5) VIENNACL_AMG_COARSE_SA (=6)	1
interpol	Interpolation algorithm. (see section 2.4 and 4.3.2 for descriptions)	VIENNACL_AMG_INTERPOL_DIRECT (=1) VIENNACL_AMG_INTERPOL_CLASSIC (=2) VIENNACL_AMG_INTERPOL_AG (=3) VIENNACL_AMG_INTERPOL_SA (=4)	1
threshold	Strength of dependence threshold.	$0 < threshold \leq 1$	0.25
interpolweight	Weight parameter for the SA interpolation. Truncation parameter for RS interpolations.	$0 < interpolweight \leq 2$	1
jacobiweight	Weight parameter for the Jacobi smoother. $jacobiweight = 1$ leads to classic Jacobi relaxation.	$0 < jacobiweight \leq 2$	1
presmooth	Number of presmooth iterations.	$presmooth \geq 0$	1
postsmooth	Number of postsmooth iterations.	$postsmooth \geq 0$	1
coarselevels	Number of coarse levels to construct. If zero, setup runs until a maximum of COARSE_LIMIT (=50) points are found. For other values *coarselevels* are constructed if possible. If only $<$ *coarselevels* levels can be constructed, coarsening is stopped there.	$coarselevels \geq 0$	0

Table 4.1: Options for the ViennaCL AMG preconditioner via class `viennacl::linalg::amg_tag`.
.

entries. In a first development the `ublas::compressed_matrix` type was used for this purpose but its performance traversing through the matrix is rather weak and in some cases not much better than the regular `ublas::matrix`.

Therefore, the decision was to implement a custom sparse matrix format that is compatible to `ublas::compressed_matrix` but also includes special functionality needed for AMG (see below). The class is defined as

```
template <typename ScalarType>
class amg_sparsematrix
```

where `ScalarType` is either `float` or `double` and denotes the type of the matrix entries. Those entries are saved in a structure `std::vector<std::map<unsigned int,ScalarType>>` which means that rows are saved in `std::map` structures while the row indices are used to access elements in `std::vector`. This approach is very flexible as a map can be dynamically extended and it is also very efficient as it uses an index-value mapping for the actual entries in a certain row. Therefore, memory is only allocated for entries that are explicitly written. The underlying assumption to this approach is that all entries that are not present in the respective map are zero values. A sparse matrix typically consists of just a few non-zero entries per row and therefore such a structure is efficient both in terms of memory requirements and iteration performance.

The interface of this class is similar to `ublas::compressed_matrix` as single entries can be accessed using parenthesis operators like `mat(i,j)` and the matrix can be traversed using row and column iterators. This is usually more efficient than using indices as iterators only run through non-zero entries. The functionality of these iterators is used from the built-in `viennacl::tools::sparse_matrix_adapter` class.

One feature of `amg_internalmatrix` is that it works very efficiently with zero values. To check whether an entry at a certain index is non-zero, one can use `bool isnonzero(unsigned int i, unsigned int j) const` which uses the fast `find()` operation from the `std::map` type and is therefore very efficient. Furthermore, the implementation takes care that no zero-entry is written to the matrix. For this purpose an additional class is used:

```
template <typename InternalType,
          typename IteratorType,
          typename ScalarType>
class amg_nonzero_scalar
```

If the parenthesis operator for `amg_sparsematrix` is used to access a certain matrix entry for write access, an object of this type is created and returned which saves both indices of the matrix entry, an iterator position and a pointer to the `amg_internalmatrix` it belongs to. If a non-zero value is written to this `amg_internal_nonzero_scalar` object, then the method `addscalar(IteratorType & iter, unsigned int i, unsigned int j, ScalarType s)` is called, saving `s` to the iterator position denoted by `iter` or to position `(i,j)` if a new entry is created. In this way no zero-entry can be written to the matrix and therefore memory consumption and computational overhead are minimized. Using the iterator is not necessary but it leads to an improved performance if an already existing value is overwritten as only one search operation in the `std::map` is necessary.

Another feature is that the data type not only saves the matrix itself but it can also easily deal with the transposed of the saved matrix which is needed for example for building the restriction matrix ($R = P^T$). For this purpose another structure of type `std::vector<std::map<unsigned int,ScalarType>>` is used, which saves the transposed of the matrix when needed. Using the transposed is necessary because iterating over rows is a lot faster than iterating over columns due to the fact that in the first case only one `std::map` has to be traversed while in the second every single map in the outer vector (see figure 4.1). An object of type `amg_sparsematrix` can therefore be switched to a transposed mode, either by using the method `void set_trans(bool mode)` or by using the iterators in transposed mode, for example by calling `iterator1 begin1(bool trans = false)` with parameter `true`. To save memory usage and computation time the transposed is only built when requested and saved for later usage. An internal state variable tracks changes to the matrix such that the transposed is rebuilt if necessary.

A last important implementation aspect I want to discuss is the efficient parallel access of the `amg_internalmatrix` type with OpenMP. Usually, a parallel computation with a matrix is designed such that rows are traversed in parallel. However,

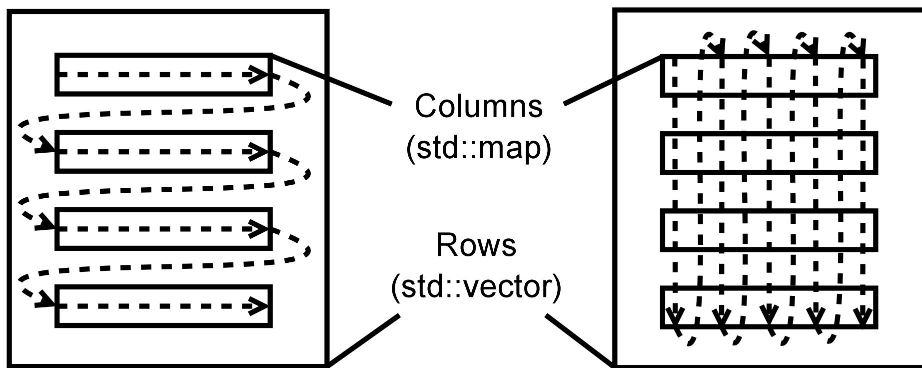

Figure 4.1: Matrix iterations. Left: row by row. Right: column by column. The right iteration is a lot slower because a vector-entry-switch and a map-entry-find has to be done in every iteration step.

the iterators defined for the `amg_internalmatrix` type cannot be used to parallelize the execution via OpenMP as the latter can only work with basic types. For this purpose, the "+=" operator can be used to combine the advantages of sparse-matrix iterators with the advantage of parallel execution:

```
#pragma omp parallel for
for (unsigned int i=0; i<A.size1(); ++i)
{
  // Use row i in matrix A
  RowIterator row_iter = A.begin1();
  row_iter += i;

  // Use column iterator in row i
  for (ColIterator col_iter = row_iter.begin();
        col_iter != row_iter.end(); ++col_iter)
  {  /* Computations in row i */ }
}
```

In this code example, rows are traversed in parallel while the column iterator only iterates over non-zero entries. With this combination very efficient matrix computations can be realized.

4.2.3 Sparse Vector

A similar approach is used for a sparse vector format in this implementation. Again, only coefficients that are non-zero are saved which improves iteration performance and memory usage:

```
template <typename ScalarType>
class amg_sparsevector
```

Internally, the structure uses a `std::map <unsigned int,ScalarType >` to save the vector coefficients and therefore is similar to one row in the sparse matrix structure. Furthermore, the interface is very similar to `ublas::vector` with the extension that a function `bool isnonzero()` is used to efficiently determine whether a certain entry is non-zero. The `amg_nonzero_scalar` described in section 4.2.2 is used to only write non-zero entries and the function `unsigned int internal_size()` returns the number of non-zero entries in the vector. Care has to be taken when using the sparse vector in a parallel environment. Although read-access is possible in parallel, entries can only be written one after another due to the underlying map structure: In a std::map type, entries are sorted and re-ordered for every write access. Certainly, writing entries in parallel can disturb this operation and is therefore impossible.

4.2.4 Point Management

An important aspect of the AMG implementation is the handling of points. This is important for coarsening when points become either coarse (C) or fine (F) points and for interpolation when points from the coarse level have to prolongate their values to points on the fine level. The management of the points is done using the classes `amg_point` which saves information for each point and `amg_pointvector` which holds the points and stores global information about them. The following information is stored in class `amg_point`:

1. The point index and the current influence measure used and updated during coarsening (see section 2.3).

2. The state of the point: C point, F point or undecided point.

3. The index on the coarse level if the point is a C point. This information is constructed before the interpolation procedure by calling `void build_index()` in `amg_pointvector`. The method uses a simple counter to index points on the coarse level, for example: If on the fine level points 1, 3 and 7 are chosen C points, then those points get indices of 0, 1 and 2 on the coarse level.

4. Each point furthermore holds a list of points influencing this point and a list of points that are influenced by this point, using pointers to the respective `amg_point` objects. This information is necessary for RS coarsening as points that are influenced by C points become F points and the influence measure has to be changed for points that influence an F point (see section 2.3.1 and 4.3.1). For AG coarsening the list of influencing points holds the neighborhood of the point. The two lists both use the `amg_sparsevector` structure as for a sparse operator matrix only a few points have to be stored.

5. For parallel coarsening an offset is saved to translate between the global point index and the local index inside the thread.

6. For AG coarsening an aggregate identifier is saved to hold the information to which aggregate a point belongs to.

The class `amg_pointvector` stores pointers to objects of type `amg_point`, one for each point. This structure can be accessed like a traditional pointer using the bracket operator to access a single point via its index: `amg_point* point = pointvector[index]`. However, internally the points are additionally saved using a list of type `std::set` which also orders points much like a `std::map`[2]. This is done because the first pass of RS coarsening traverses points by their influence measure (see section 2.3.1). Although this could be done by traversing all undecided points before each iteration, sorting points in a set is a lot more efficient. The sorting is

[2]The difference is that a map saves key-value pairs while in a set the value and the key are the same.

done automatically for each write operation to the set. How to sort the entries is determined by a structure passed to the set via the constructor:

```
struct classcomp
{
  bool operator() (amg_point* l, amg_point* r) const
  {
    return ( l->get_influence() < r->get_influence() ||
            (l->get_influence() == r->get_influence()
            && l->get_index() > r->get_index()) );
  }
};
```

This structure ensures that points are sorted by influence measure using the index as a tie-breaker to enforce a deterministic behavior. Calling `get_nextpoint()` returns a pointer to the undecided point with the highest influence measure.

4.2.5 AMG Classes

As already described, the data structures for the AMG preconditioner are saved in `amg_precond` while the preconditioning operation is done via the member function `void apply(VectorType & vec) const`. Currently, two implementations are available, one for the ViennaCL GPU operation and one for all other types for CPU operation. Both classes also include methods for the weighted Jacobi smoother, where one is an implementation via an OpenCL kernel call (GPU) and the other an OpenMP CPU implementation (see section 4.4.1 for more details).

The data structures saved are in the form vector-of-matrix or vector-of-vector and hold the respective structures for the different levels such that index 0 refers to the finest level and `amg_tag.coarselevels` the coarsest (see table 4.1). Note that on the highest level only the operator matrix has to be stored. while on the other levels also the restriction and prolongation matrices are needed. Furthermore, information is saved for the setup process, including the points of structure `amg_point` saved in the `amg_pointvector` for every level.

For the setup phase, the matrices are stored in the `amg_sparsematrix` format which includes the operator matrix, the prolongation matrix and the restriction matrix, implicitely included in the transposed part of the prolongation matrix. This improves performance for the setup phase and eases the development due to the extended features of the `amg_internalmatrix` type. For the precondition phase, however, those structures are transformed into the respective matrix format requested by the user, for example `ublas::compressed_matrix` for CPU operation or `viennacl::compressed_matrix` for GPU operation.

Furthermore, some data structures used in the precondition phase are also stored in the AMG classes: This includes the vectors used during precondition operation like residuals, results and errors on all levels and also the pivoting information from the factorization of the operator matrix on the coarsest level used for the direct solver (see section 4.4.2).

4.2.6 Parallel Coarsening

For RS0 coarsening and RS3 coarsening the grid is partitioned into several slices and RS coarsening is run on each of the slices separately. For this purpose the class `amg_slicing` is used, which saves the data structures for the threads separately and offers functionality to partition and join the data structures. In this class essentially the same coarsening-related information is held as in `amg_precond`, which includes the operator matrix and the point management classes, but the difference is that the information is held on a per-thread level. The data structures are built when `void init(unsigned int levels, unsigned int threads = 0)` is called, allowing the user to specify the number of threads for RS0 or RS3. If the default value (0) is chosen, then the system uses as many threads as processors are available. By calling `void slice (unsigned int level, InternalType1 const & A, InternalType2 const & Pointvector)`, the operator matrix and the point information are partitioned into as many parts as threads are available. This is equivalent to slicing the matrix into quadratic parts around the diagonal, effectively setting coefficients too far off the diagonal to zero.

The matrix parts are constructed such that they are of roughly the same size: The number of points per matrix part is the size of the operator matrix divided by the number of threads. If a remainder is present in this integer division, then the remaining points are added to the last matrix part in the split. An example how a splitting of a matrix with size five is done for two threads can be found in section 3.5. Furthermore, it is assumed that neighboring points also have a close index such that non-zero coefficients are always close to the diagonal (see discussion about renumbering in the same section).

Note that an initial slicing is built on the finest level and the slicing on the coarser levels is done using the same boarders as on the finest level. This ensures that points are grouped in the same way on all levels, which is important as strong connections in RS0 or RS3 coarsening are usually in the same partition. To use the partitioned information for the computation of the interpolation and Galerkin operator, the structures from the different threads have to be joint into a single data structure which is done by calling `void join (unsigned int level, InternalType2 & Pointvector) const`. Afterwards, the same routines for interpolation and the Galerkin operator can be used as if a single-threaded coarsening was called. Note that in the case of parallel coarsening, points essentially have two indices: One global index derived from the system matrix and one local index inside the thread derived from the slicing procedure.

4.3 Setup Phase

When an object of the `amg_precond` class is built using the system matrix and a proper `amg_tag`, `void amg_init(...)` is called directly from the constructor to transform the system matrix into a `amg_sparsematrix` used for the setup phase. The setup routine can be started right afterwards by calling `void setup()` on this object and it proceeds by calling the chosen coarsening and interpolation algorithm and computing the Galerkin operator for all levels. The implementations of these steps will be discussed in more detail below. After setup has finished, the prolongation, restriction and operator matrices are transformed back into the respective matrix type such that they can be used in the precondition phase.

4.3.1 Coarsening

The implementation of the AMG preconditioner supports five different coarsening methods: Classical RS coarsening, RS one-pass coarsening, RS0, RS3 and standard aggregation-based (AG) coarsening. Which coarsening procedure to apply is chosen by the user via the tag-class (see table 4.1). For every coarsening procedure one function is defined, which is called by `void amg_coarse(...)` from the setup routine `void amg_setup(...)`. The implementation of the coarsening approaches can basically be grouped into two approaches: RS one-pass, RS0 and RS3 are all variants of RS, while AG is a different approach to coarsening (see chapter 2.3).

The implementation of the first group is therefore developed to maximize code-reuse and functions are called in a nested way:

- RS uses RS one-pass and then runs the second pass.
- RS0 runs RS in parallel on sets of subpoints and combines the results afterwards.
- RS3 runs RS0 first and then runs the third pass.

A common task for all strategies is to define criteria when coarsening has to terminate. In this implementation this is done when

1. the number of coarse levels reaches the value set by the user in the `amg_tag` class.
2. the user chose the automatic creation of coarse levels and the number of points on the coarsest level reaches the value of the preprocessor constant `COARSE_LIMIT` defined in amg.hpp. The default value is 50.
3. the maximum possible coarse level is found. This is the case when coarsening cannot construct any C or F points on that level.

The second criteria is designed to take into account that the direct solver is inefficient for large matrix sizes and therefore automatic coarsening is done until

a reasonably small size is reached, where an efficient execution can be expected. Experiments, however, showed that doing the coarsening until the maximum possible coarse level is reached is basically equivalent to stopping at `COARSE_LIMIT` as the overhead for the construction of those last coarse levels is negligible and the convergence of the resulting preconditioners shows no difference.

Influence

An important step before the actual coarsening is done is to figure out which points strongly influence other points. Although the approach is different for RS and AG coarsenings, in both cases the information has to be present beforehand. For RS it is also necessary in a second step to find out how many points depend on a certain point since coarsening starts with the point that has the highest influence measure. The information about the respective influences (RS) or neighborhoods (AG) is stored in lists in the `amg_point` class (see section 4.2.4).

Both approaches are efficiently implemented in parallel using OpenMP by searching for strong influences on a per-point, or equivalently, on a per-matrix-row basis. The information about the transposed influence is determined in a second step by transposing the information stored in the `amg_point` objects. This has to be done serially as the list cannot be written in parallel due to the fact that an `amg_sparsevector` type is used (see section 4.2.3 and 4.2.4). Of course, one could also use a regular `std::vector` which allows for parallel write access, but for sparse operator matrices, iteration can be done a lot faster with `amg_sparsevector` outweighing the performance loss. In a third step the number of influenced points are counted for each point which again can be done in parallel.

RS One-Pass

RS one-pass coarsening uses only the first pass of the classical RS algorithm for coarsening. This has the disadvantage that strong F-F connections without common C point might remain after the process, leading to worse interpolation and convergence. However, the advantage is that some setup time is saved and the com-

plexity is reduced as there are less C points and therefore the matrix on the coarser level is smaller (see section 2.3).

The implementation uses the `amg_pointvector` structure to traverse through points ordered by their influence measure (see section 4.2.4). The point with the highest order becomes C point while all points influenced by this point become F point. As those points are stored in the influenced list for each point, this can be done very efficiently. Then, for all newly created F points, all influencing points increment their influence measure. These points are stored in the influencing list which again allows for an efficient operation and shows why these lists are set up before coarsening.

The procedure stops whenever there are no undecided points left or only points with influence measure equal to zero. These points do not influence other points and are further not influenced by any C points constructed during coarsening. In such a situation it makes sense to leave them undecided as coarse-grid correction cannot improve the error for these points anyway. The interpolation row for this point in effect remains an all-zero row as no interpolation to this point can be done.

Classical RS

Classical RS first runs RS one-pass coarsening as the first pass and this also implicitely builds the influence information needed. The goal of the second pass is to prevent strong influences from one F point to another if both do not have a common neighboring C point. In that case interpolation from the C points does not lead to a very good approximation in all likelihood. If such a situation is found then one of the two F points becomes C point, improving interpolation but also worsening complexity as more points are drawn to the coarser level. Which of the two F points becomes C point is not important for the algorithm, however, it can lead to very different outcomes, especially when it comes to complexity. The decision was to always make the point with the higher index to a C point. This counteracts the fact that points with lower index are more likely to become C points due to the fact that a lower index is a tie-breaker if points have the same influence measure. Using the higher indices for C point creation in the second phase therefore leads to

a better distribution of C points on average. Again, this has to be done sequentially as F to C point transformations have to be done step by step. The implementation, however, is straightforward such that a more detailed discussion is not necessary at this point.

RS0

RS0 coarsening runs RS coarsening in parallel after points are partitioned into groups (see chapter 2.3 and 4.2.6 for more details). The number of different threads and matrix parts in use is decided by the user via the `amg_tag` class or can be set automatically to the number of threads available for the processor. The implementation starts by building the parallel data structures and slicing the system matrix into as many parts as threads are requested. Then, RS coarsening is run in parallel on those parts using OpenMP. Note that influence measures are calculated in the sub-matrices and not on the full operator matrix. This improves the coarsening for certain cases, for example when a high coefficient value lies on the outside of the respective part, practically approximating it to zero. After the coarsening is finished, data structures for the coarsening information are joined to single data structures.

RS3

RS3 coarsening runs RS0 coarsening and then adds a third pass similar to the second pass of RS coarsening. The only difference is that this third pass only operates on F-F connections crossing thread boundaries of the RS0 coarsening, such that in most cases only a few points have to be visited. There are many ways this can be implemented: The easiest is to just run the third pass on one processor after all RS0 threads have finished, while the alternative is to compute the additional C points on the different processors separately. Even though the first choice might lead to worse setup times, this approach was taken as the overhead is negligible. Furthermore, doing the pass on all processors in parallel might lead to more C points and therefore higher complexity or certain C points have to be removed afterwards to reduce complexity at the expense of some computational overhead.

AG

AG coarsening uses the aggregation approach and therefore runs a different approach for computing dependencies (see section 2.3.4). In the implementation, neighborhoods are built first, which includes all influencing points and then neighborhoods are made into disjoint aggregates. C points are the root points of those aggregates while all points in the neighborhood of this root point become F points. The algorithm is sequential in nature, although compared to RS coarsening it is less expensive: For AG coarsening, neighborhoods can be built in a sequential way while for RS coarsening, influence measures have to be computed and points have to traversed in the order of their influence measures such that sorting is necessary.

4.3.2 Interpolation

For this thesis four different interpolation procedures were implemented, two for the different RS related approaches and two for aggregation-based AMG. Direct interpolation and classical interpolation belong to the first group while the basic aggregation-based interpolation and smoothed aggregation interpolation belong to the second. The interpolation is chosen by the user in the `amg_tag` class and the respective interpolation routine is called from the setup routine after coarsening has finished.

A common task for all interpolation procedures is to change the C point indices from the fine to the coarse level. This is necessary as the operator matrix on the coarse level only contains the C points from the fine level such that the prolongation matrix is usually rectangular with as many rows as on the fine level and as many columns as on the coarse level. The respective coarse level index is saved in the `amg_point` class and constructed before interpolation by calling `void build_index()` (see section 4.2.4 for more details).

Direct Interpolation

Direct interpolation uses only direct C-F point connections for interpolation. This can be implemented using either only strongly connected C points or all C points with at least a weak connection to the respective F point. The decision was to implement the first approach because this usually leads to sparser prolongation matrices and also to sparser operator matrices via the Galerkin operation, improving complexity and performance. The disadvantage is that interpolation might be worse, although experiments have shown that the effect is negligible. The interpolation is otherwise quite straightforward. To improve performance, the summations of the C point coefficients and coefficient row sum in equation 2.1 are calculated on a per-row level. Furthermore, the rows of the prolongation matrix are computed in parallel. If chosen by the user ($interpolweight > 0$), each row is truncated to further reduce the number of non-zero points per row (see section 2.4.1). The truncation operation is quite straightforward and therefore not described.

Classical Interpolation

Classical interpolation also takes strong F-F connections via a common C point into account, which leads to better convergence but the computation of the interpolation is more complex because neighbors of neighbors have to be visited. The implementation, however, is very similar to direct interpolation: Interpolation is done from strongly coupled coarse points only, rows are computed in parallel and the denominator and sum of C point coefficients in equation 2.2 is determined before the actual computation of the interpolation weights. Interpolation truncation is done for each row if chosen by the user.

Basic Aggregation-based Interpolation

Basic aggregation-based interpolation only uses the root point of each aggregate as basis for interpolation. Therefore, interpolation is done from only one point with unit weight. This is easy to implement and compute and it further keeps complexity low at the expense of the interpolation quality. The implementation is again parallel.

Smoothed Aggregation

Smoothed aggregation interpolation improves the basic aggregation by employing a smoother, for example one iteration of a weighted Jacobi. In that case, interpolation is done also from neighboring aggregates which improves convergence. However, the computation of the prolongation matrix is a lot more expensive as it involves a matrix-matrix product (see section 2.4.4). The implementation is done in parallel using OpenMP and is rather straightforward: It involves building the Jacobi matrix from the filtered operator matrix and the computation of the matrix-matrix product. The former is constructed without explicitely building the filtered matrix which improves performance substantially. The latter is done via the function `void amg_mat_prod (SparseMatrixType & A, SparseMatrixType & B, SparseMatrixType & RES)` and will be explained next.

4.3.3 Matrix Product

Matrix product are used in two scenarios: To compute the smoothed aggregation (see last section) and to compute the Galerkin operator (see section 2.1.1). This is done after coarsening is defined and interpolation is computed and finishes up the setup phase on a certain level. That computation is a triple-matrix product involving the restriction, prolongation and operator matrix on the respective level (see section 2.1.1). For this purpose the function `void amg_galerkin_prod(SparseMatrixType & A, SparseMatrixType & P, SparseMatrixType & RES)` was developed to compute the Galerkin product $A^{k+1} = R^k A^k P^k$.

The implementation of the matrix product is quite straightforward and can be parallelized conveniently using OpenMP by computing the rows of the result matrix in parallel. However, certain aspects have to be taken into account to make the implementation efficient for sparse matrices. While a typical implementation of a dense matrix product would directly compute all the row-column multiplications on after another, this is not efficient if matrices are relatively sparse. Then, this type of multiplication can involve row-column computations with non-overlapping coefficients, leading to unnecessary iterations.

Take the five-point example matrix from chapter 1 together with a very simple interpolation scheme such that[3]

$$R^kA^kP^k = \begin{pmatrix} 1 & 1 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}$$

Here, the computation of row 1 of R^k with column 5 in A^k leads to $1 \times 0 + 1 \times 0 + 0.5 \times 0 + 0 \times (-1) + 0 \times 2 = 0$. Depending on the actual implementation a high number of unnecessary iterations are carried out in this case even though the result is zero anyways. Furthermore, as column iterations are slow (see section 4.2.2), the transposed matrix of A^k would have to be built first.

A more efficient approach for sparse-matrix multiplication is not to do row-column multiplications directly, but to iterate over non-zero coefficients only, which can be done very efficiently using the `amg_sparsematrix` structure (see same section). Then, iterations are minimized as only those rows/columns are iterated that lead to overlapping coefficients. Algorithm 6 shows how these iterations are done: If a non-zero entry in the left matrix is found, then all computations with that entry are done by traversing through the respective row in the right matrix and adding up the resulting coefficient multiplications.

Algorithm 6 (Sparse Matrix Product: $AB = C$)**:**

1. *Choose row i in A that has not been chosen yet.*
2. *For all non-zero entries a_{ij} in row i of matrix A do:*
3. *For all non-zero entries b_{jk} in row j of matrix B compute $c_{ik} \leftarrow c_{ik} + a_{ij}b_{jk}$ for result matrix C.*
4. *If not all rows in A have been visited, go back to 1.*

[3] In this example points 2 and 4 are coarse points while 1, 3 and 5 are fine points. The interpolation is defined by simply using a weighted sum via strongly coupled C points.

Clearly, this algorithm can proceed in parallel as the computations for rows $i = 0, 1, ...$ are independent. Furthermore, no matrix transposition is necessary.

For a computation of $R^k A^k$ from above, the steps proceed in the following way:

1. Choosing the first row in R^k.
2. $r_{00} = 1$. Switching to row 0 in A.
3. $a_{00} = 2$. $c_{00} \leftarrow 0 + 1 \times 2 = 2$.
4. $a_{01} = -1$. $c_{01} \leftarrow 0 + 1 \times -1 = -1$.
5. $r_{01} = 1$. Switching to row 1 in A.
6. $a_{10} = -1$. Computing $c_{00} \leftarrow 2 + 1 \times (-1) = 1$.

The triple-matrix product used for the Galerkin operator could be computed by doing the matrix product twice via a temporary result matrix. However, only a minor modification to the algorithm is necessary such that the Galerkin product can be computed at once, which has the advantage that more work can be done in one OpenMP thread, improving the utilization of the CPU cores. This algorithm is described in algorithm 7.

Algorithm 7 (Galerkin Operator: $R^k A^k P^k = A^{k+1}$)**:**

1. *Choose row i in R^k that has not been chosen yet.*
2. *For all non-zero entries r_{ij} in row i of matrix R^k do:*
3. *For all non-zero entries a_{jk} in row j of matrix A^k compute $t_k \leftarrow t_k + a_{ij} b_{jk}$ for temporary row t.*
4. *For all non-zero entries t_k in t do:*
5. *For all non-zero entries p_{kl} in row k in P^k compute $a_{il}^{k+1} \leftarrow a_{il}^{k+1} + t_k p_{kl}$ for result matrix A^{k+1}.*
6. *If not all rows in A have been visited, go back to 1.*

For the temporary row t the `amg_sparsevector` type can be used to minimize iterations. Experiments during the development showed that those algorithms run considerably faster for all matrices used for the benchmarks in chapter 5 than any implementation using conventional dense matrix algorithms. Furthermore, if the number of non-zero entries per row in A stays constant, the computation time grows linearly with the number of rows N. In the grid terminology this means that the computation time grows linearly with the number of points if the number of connections per point stays constant.

4.4 Precondition Phase

The precondition phase is started from the iterative solver calling `void apply (VectorType & vec) const` in `amg_precond`. If the method is called for the first time, some preparatory work is done including building the necessary data structures (see section 4.2) and running the LU factorization on the coarsest level (see section 4.4.2). This speeds up the precondition phase due to the fact that these steps only have to be taken once instead of for every iteration cycle.

Then, the first part of the V cycle with pre-smoothing and restriction operations is started until the coarsest level is reached. There, the residual equation is solved using a direct solver and then the second part of the V cycle is started with prolongation operations and post-smoothing. In the following, I will discuss the smoother and the implementation of the direct solver in more detail. The coarse-grid correction only consists of basic matrix-vector operations from the computation of the residuals/errors and the respective restrictions/prolongations and therefore there is no need to go into additional details.

4.4.1 Smoother

At present, a weighted Jacobi smoother is implemented for both classes, using the same interface: `void smooth_jacobi (int level, unsigned int iterations, VectorType & x, VectorType const & rhs) const`. Although in the literature

Gauß-Seidel is the standard smoother for AMG, Jacobi has the advantage that it can be computed in parallel on a per vector-entry level and the smoothing quality is at a similar level (see chapter 2.1.1). The weight of the linear combination of former and newly computed Jacobi vector can be set using the `amg_tag` class (see table 4.1).

GPU Implementation

One iteration of the GPU implementation is written in an OpenCL kernel called from `void smooth_jacobi(..) const` for every iteration. This makes sense as then memory can be accessed efficiently. If the smoother was implemented using ViennaCL objects in C++, then one would either use matrix operations or have one memory transfer per computed vector-entry which is both a lot slower than an OpenCL implementation.

An important aspect of the implementation is memory access. ViennaCL already provides tools to ease the usage of GPU memory such that whenever an OpenCL kernel is called, a `viennacl::vector` can be used directly as a parameter and can be accessed in OpenCL like a pointer/array. A `viennacl::compressed_matrix` can be used indirectly using the methods `handle1()`, `handle2()` and `handle()`. Those translate the matrix into three different pointers/arrays using the CSR (compressed sparse row) format [1, p.92ff] which can be accessed from an OpenCL kernel:

- `handle()` returns all matrix entries in a one-dimensional memory array.
- `handle2()` returns an array of indices that can be used to access that memory.
- `handle1()` returns an array that holds the ranges for the respective row in the index array from `handle2()`.

If `row_indices` holds the array from `handle2()` and `elements` the array from `handle()`, then an iteration over the elements of a certain row `i` can be done by

```
for (unsigned int j = row_indices[i]; j < row_indices[i+1]; j++)
{
  int col = column_indices[j];
  // Element access via elements[j]
}
```

The second important aspect is the parallel computation of the resulting vector. This is done on a per-element basis using built-in functionality in OpenCL (see also section 3.1 for more details). The complete kernel code for the Jacobi smoother looks like this (variable initialization not shown):

```
for (unsigned int i = get_global_id(0); i < size;
      i += get_global_size(0))
{
  sum = 0;
  for (unsigned int j = row_indices[i]; j < row_indices[i+1]; j++)
  {
    col = column_indices[j];
    if (i == col)
      diag = elements[j];
    else
      sum += elements[j] * old_result[col];
  }
  new_result[i] = weight * (rhs[i]-sum) / diag
                + (1-weight) * old_result[i];
}
```

CPU Implementation

The CPU implementation is straightforward and uses OpenMP to parallelize computation, again, on a per-element basis and the operator matrix is accessed using iterators. The code is very similar to the one in OpenCL with the only difference that matrix access is simpler because iterators can be used.

4.4.2 Direct Solver

The direct solver uses LU factorization to solve the residual equation on the coarsest level. For both CPU and GPU implementation the LU substitution is done on the CPU as the direct solver for ViennaCL does not support pivoting yet. This is certainly bad for the overall GPU precondition performance because two memory transfers have to be done on the coarsest level.

As already mentioned, the two parts of the direct solver are split such that the factorization of the operator matrix on the coarsest level is only done once, independent of the number of iterations of the underlying solver method and precondition operations. The implementation of the direct solver is done using built-in functionality in ublas: `ublas::lu_factorize(op,Permutation)` factorizes the matrix `op` and saves pivoting information in `Permutation` while `ublas::lu_substitute(op, Permutation, result)` does the substitution. The variable `result` holds the right-hand-side vector of the residual equation before the function call and the result of the substitution afterwards.

4.5 Summary

The implementation of the AMG preconditioner makes use of the different processing units available to the system. While the GPU is not a good option for the setup phase due to its complex nature, it can considerably improve the performance of the precondition phase, which uses standard linear operations. A user can therefore choose to use the available GPU devices, combining it with the already existing GPU computing capabilities in ViennaCL. The setup phase on the other hand makes use of parallel threads on the CPU: This is relatively simple for the setup of the interpolation operator and the computation of the Galerkin operator as rows can be computed independently. However, the standard coarsening approaches are serial in nature and parallel variations affect the overall precondition performance as discussed in chapter 2. The most important aspect for the efficiency of the implementation, however, is the handling of the data structures: For this purpose a sparse matrix and a sparse vector data type were implemented to improve perfor-

mance, especially for iterations. Furthermore, the point management is done to quickly access necessary information when needed, including the sorting of points by their influence measure before and during coarsening and the lists of influencing and influenced points saved for each point. Furthermore, an efficient sparse matrix multiplication algorithm was developed to take advantage of the underlying sparse matrix structure.

Chapter 5

Benchmarks

In this chapter numerical results obtained from solving certain systems with different combinations of AMG methods and parameters are presented and discussed. For this purpose, discretizations of physical problems are used which lead to linear equations, formally $Ax = b$, as discussed in chapter 1. The result vector x is constructed arbitrarily for each matrix A such that entries x_i are in the range $[1, 2]$. The right-hand-side vector b is computed from A and x, using the matrix-vector product from the Boost uBlas library (see section 3.4) before the benchmark is run. Then, benchmarks use A and b and solve the equation for x. For the benchmarks, different AMG preconditioners are used for either a CG or BiCGStab iterative solver in ViennaCL. The initial guess for all benchmarks is the all-zero-vector.

In the following section the different systems are described with emphasis on the structure of the matrix. This is followed by a presentation of the different coarsening and interpolation combinations as well as parameter values used for these benchmarks. In the last section, numerical results of the different AMG approaches are compared and discussed.

5.1 Systems

The benchmarks use a total of nine different system matrices from five discretized physical problems. These system matrices have a different size and different properties in terms of the number of coefficients per row and the coefficient structure:

- FEM2D: Four matrices are built by a piecewise linear finite element method for the discretization of the Poisson equation on the unit square. The problem and structure is very similar to the 1D case described in chapter 1. The difference is that in the 2D case one point has up to four neighbors and that the matrix has a slightly higher bandwidth after renumbering. Martrices with different numbers of points are constructed: 3969 for fem2d_3969, 16129 for fem2d_16129, 65025 for fem2d_65025 and 261121 for fem2d_261121. Although the scaling leads to a higher number of rows, the number of non-zero coefficients remains constant with an average stencil size of close to 5. The structure is a symmetric M-matrix for which good AMG performance is to be expected as discussed in chapters 1 and 2.

- FEM3D: Another matrix for the Poisson equation is built by a finite element discretization on the unit cube using 31713 points (fem3d_31713). There are more non-zero coefficients per row with an average stencil size of 11.6 and the bandwidth is higher compared to the 2D case. The matrix is symmetric and positive definite but no M-matrix as a few off-diagonal coefficients have the same sign as the diagonal. Still, the structure is not too far off the optimal case such that good AMG performance is still to be expected.

- KRATOS: This system matrix is a 3D finite elements discretization of the non-stationary Navier-Stokes equation (fluid dynamics) with 24202 points (kratos_24202). The overall structure is relatively similar to FEM3D as it is close to an M-matrix with a few off-diagonal coefficients having the "wrong" sign. However, the matrix is denser with an average stencil size of 14.3 as the number of non-zero coefficients per row is higher.

- LAME: This matrix is constructed using a 3D discretization of the stationary linear elasticity equation under the influence of gravity (mechanics) with 13005

points (lame_13005). The discretization is carried out on the unit cube using a tetrahedron grid. The number of non-zero coefficients per row is the highest with an average stencil size of 36.9. The matrix, however, is not even close to an M-matrix as the number of coefficients with the "wrong" sign is close to the number of coefficients with the "right" one.

- SHE1: The last matrix is constructed using a first order spherical harmonics expansion of the Boltzmann equation (semiconductors) with 30126 points (she1_30126). The matrix is relatively sparse with an average stencil size of 5.9. It is furthermore an M-matrix but non-symmetric.

5.2 AMG Variants

To test how certain AMG variants work with different matrices, a number of combinations of components and parameters were chosen for these benchmarks. Of course, the implementation described in chapter 4 allows for numerous combinations to construct AMG methods by choosing the coarsening and interpolation method, smoothing parameter, threshold and all other parameters shown in table 4.1. However, to preserve a reasonable scope of the benchmark, the choices described below are using combinations, which are recommended in the literature and make sense from a theoretical standpoint. The following combinations of coarsening and interpolation methods are used:

- RS coarsening with direct interpolation (rs_direct)
- RS coarsening with classical interpolation (rs_classic)
- RS one-pass coarsening with direct interpolation (rsop_direct)
- RS0 coarsening with direct interpolation (rs0_direct)
- RS3 coarsening with direct interpolation (rs3_direct)
- Aggregation-based coarsening with basic interpolation (ag)
- Aggregation-based coarsening with smoothed aggregation interpolation (sa)

All benchmarks use the automatic construction of coarse levels until there are a maximum of 50 points on the coarsest level. All methods use three pre- and postsmooth iterations for the precondition phase and the weight for the Jacobi relaxation is chosen to $\omega = 0.67$. The threshold parameters are $\theta = 0.25$ for RS-based coarsening and $\theta = 0.08 \times (0.5)^{l-1}$ for AG-coarsening[1]. Direct and classical interpolation use interpolation truncation with a weight of $\epsilon = 0.2$. The weight for the smoothed aggregation interpolation is chosen to $\omega = 0.67$.

5.3 Numerical Results

In this section, I discuss certain benchmark results, comparing different AMG approaches. Before running the benchmarks, the system matrices are renumbered using the tool discussed in section 3.5. The benchmarks are run on the hardware described in section 3.6. AMG is used as a preconditioner for the BiCGStab solver if the matrix is non-symmetric (she1_30126) and for the CG solver if the matrix is symmetric (all other matrices). Iteration is performed until the quadratic norm of the estimated relative residual[2] from the CG iteration is smaller than 10^{-9}. Note that the CG and BiCGStab solvers in ViennaCL do not compute the actual residuals during iteration but use estimates which is more efficient. Benchmarks which determine computation times are run 50 times and the average of these runs is taken as the result. Benchmarks that determine convergence, complexity measures and residuals are extracted from a single run only.

5.3.1 Scaling Analysis

One important property of AMG and multigrid methods in general is the linear complexity as described in chapter 1. To check for this property, the four FEM2D matrices are used where the number of points scale by a factor of 4. The scaling benchmark uses the RS coarsening approach with classical interpolation as well as

[1] l is the level index where $l = 0$ denotes the finest level.

[2] The quadratic norm of the estimated relative residual is computed by dividing the quadratic norm of the estimated residual by the quadratic norm of the right-hand-side vector.

	fem2d_3969	fem2d_16129	fem2d_65025	fem2d_261121
rs_classic (CPU)	0.052 (0.031;6)	**0.233** (0.141;6)	1.011 (0.604;6)	4.069 (2.219;6)
rs_classic (GPU)	**0.060** (0.037;6)	0.204 (0.171;6)	0.785 (0.716;6)	2.890 (2.704;6)
rs3_direct (CPU)	0.061 (0.031;8)	0.234 (0.124;7)	**0.969** (0.459;8)	**3.705** (1.756;8)
rs3_direct (GPU)	0.070 (0.032;8)	0.180 (0.130;7)	0.600 (0.494;8)	2.119 (1.870;8)
sa (CPU)	0.155 (0.092;8)	0.761 (0.355;9)	3.675 (1.432;12)	18.975 (5.677;16)
sa (GPU)	0.138 (0.084;8)	0.531 (0.419;9)	2.192 (1.795;12)	8.388 (6.481;16)
cg (CPU)	**0.037** (-;192)	0.286 (-;355)	2.265 (-;672)	17.564 (-;1262)
cg (GPU)	0.063 (-;192)	**0.131** (-;355)	**0.380** (-;672)	**1.695** (-;1262)

Table 5.1: Scaling benchmark: Shown are the total computation time in seconds with the setup time and the number of iterations in parenthesis. Best results are shown in bold.

RS3 with standard interpolation and aggregation-based coarsening with smoothed interpolation such that a maximum number of overlapping routines is used: RS uses RS one-pass coarsening, while RS3 uses RS0, and smoothed interpolation the basic aggregation-based interpolation.

Table 5.1 shows the results for the different combinations, comparing them with the unpreconditioned CG solver. The benchmark times are determined by computing the average of 50 runs of each solver. Figure 5.1 further shows diagrams of the total computation times for the different problem sizes on CPU and GPU. Due to the matrix scaling, it is to be expected that for the AMG preconditioned solvers, total computation and setup times scale with a factor of about 4 and the number of iterations should stay constant. The unpreconditioned solver should, however, scale with a factor of about 8 due to its complexity of $O(N^{\frac{3}{2}})$ (see chapter 1).

Comparing expectations with the numerical results shows a good match for CPU computation. The only exception is the smoothed aggregation preconditioner for which the number of iterations does not stay constant, leading to a stronger increase in solver time. The situation for the GPU computation is different: Here, the unpreconditioned CG performs fairly well and shows much better scaling even up to the largest matrix size. The reason for this result is that the OpenCL overhead is relatively large compared to the computation time of one CG iteration for the matrix sizes used in this benchmark. The same behavior can be observed for the smoothed aggregation preconditioner.

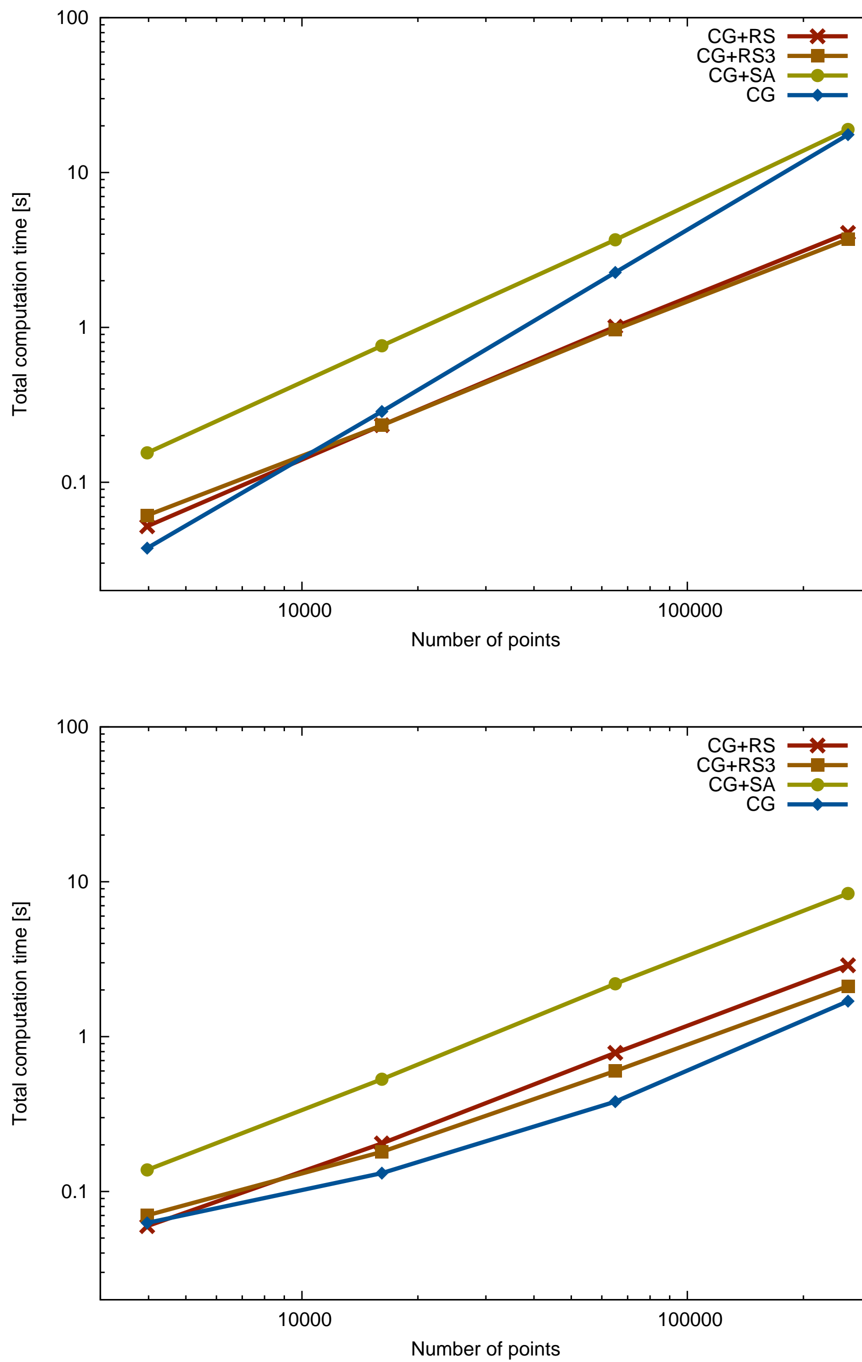

Figure 5.1: Scaling benchmark: Total computation times for different AMG preconditioners to CG on the CPU (top) and the GPU (bottom) from table 5.1. The unpreconditioned CG is shown for comparison.

Therefore, for the systems at hand and if the GPU can be used, it is not meaningful to use an AMG preconditioner as the overhead of the setup phase is larger than the savings in solver time. The only exception is the smallest matrix for which the OpenCL overhead is large compared to the setup time. But then, it certainly does not pay off to use the GPU anyways. Using an AMG preconditioner on the GPU would only be efficient for a faster CPU such that setup time goes down or a larger system such that CG is not as efficient. For the GPU benchmark results presented, almost all of the computation time is taken by the setup phase.

However, if no GPU was available such that the CPU also had to be used for the solver and precondition phase, the RS and RS3 preconditioners are both useful for smaller matrix sizes as the break-even point with the unpreconditioned CG is at around 10,000 points. The benchmark results for both methods are quite similar, although RS3 offers a little better performance for larger matrices. Smoothed aggregation on the other hand does not seem useful at all for the FEM2D problems.

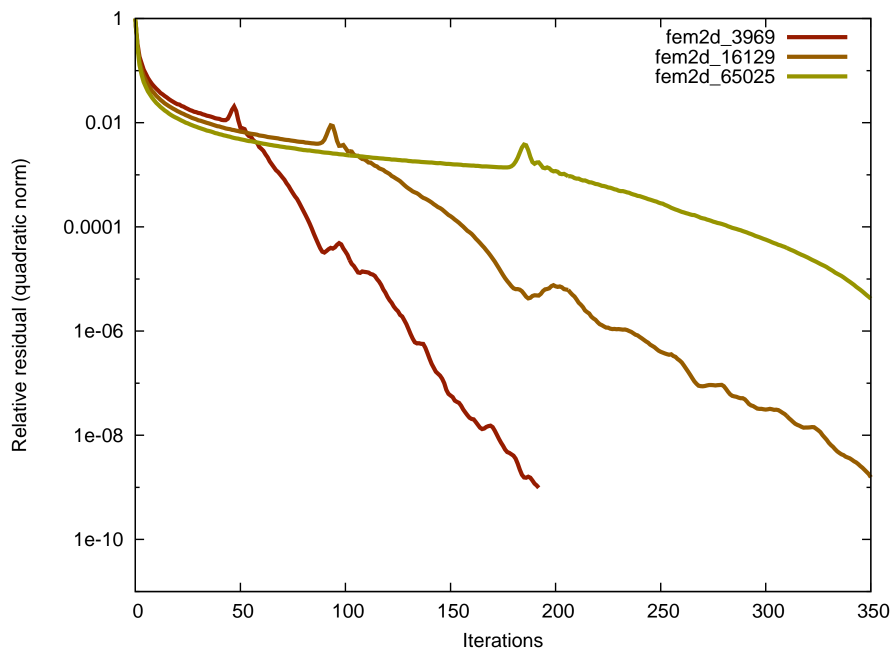

Figure 5.2: Squared relative residuals for CG and different FEM2D matrices.

A comparison between CPU and GPU solver time shows that only for the largest matrix the theoretical factor 10 in computation power between GPU and CPU can

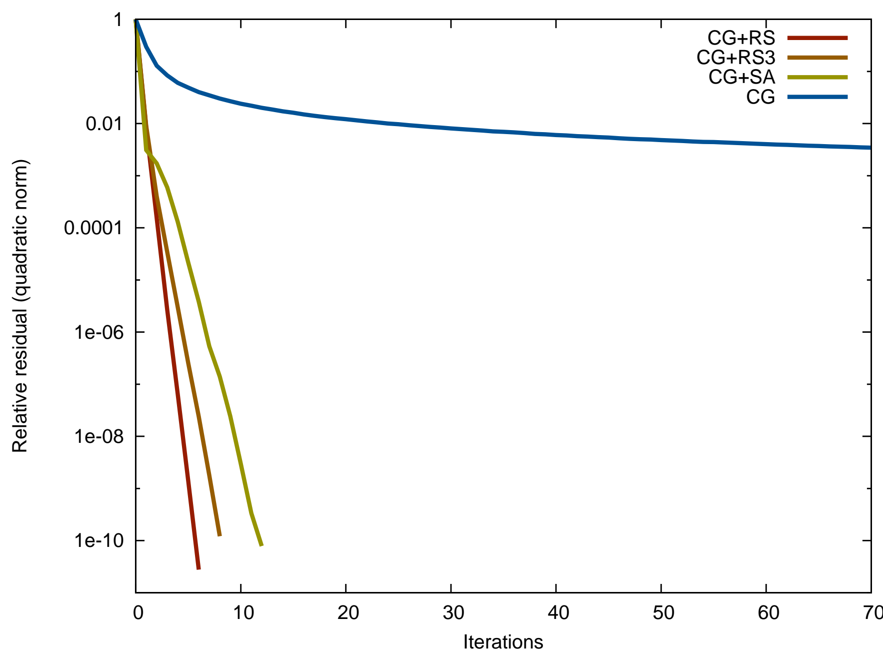

Figure 5.3: Squared relative residuals for different solvers and matrix fem2d_65025.

be observed. For example, the solver time for the RS method on the CPU is 1.850 while for the GPU it is 0.186. For the smaller matrices the factor is a lot smaller and for fem2d_3969 there is no difference in solver time at all. The reason for this result is that only for large matrices the overhead for OpenCL is negligible and the parallel processors on the GPU can be utilized efficiently.

Figure 5.2 shows the scaling of the unpreconditioned CG method in terms of relative residuals, while figure 5.3 shows the progress of the relative residuals during iteration for preconditioned and unpreconditioned CG. It can be seen how CG convergence depends on the matrix size and how AMG preconditioning drastically improves overall convergence.

5.3.2 Aggressive and Parallel Coarsening

Using aggressive or parallel coarsening is often advised in the literature to improve the setup time at the expense of interpolation quality. Table 5.2 shows results, com-

	fem2d_65025	fem3d_31713	kratos_24202	she1_30126
rs_direct (CPU)	1.064 (0.609;6)	5.161 (1.777;10)	5.476 (1.501;13)	9.447 (0.292;90)
rs_direct (CPU)	0.794 (0.724;6)	1.940 (2.587;10)	1.640 (2.393;13)	2.229 (0.324;90)
rsop_direct (CPU)	1.123 (0.632;6)	**1.947** (0.806;12)	**1.631** (0.480;21)	-
rsop_direct (GPU)	0.772 (0.702;6)	**1.090** (0.854;12)	**0.789** (0.508;21)	-
rs0_direct (CPU)	1.192 (0.434;13)	4.460 (1.540;10)	4.777 (1.154;17)	-
rs0_direct (GPU)	**0.586** (0.452;13)	2.354 (1.730;10)	2.166 (1.319;17)	-
rs3_direct (CPU)	**0.988** (0.470;8)	6.215 (2.132;10)	6.395 (1.555;16)	-
rs3_direct (GPU)	0.591 (0.486;8)	3.298 (2.427;10)	2.922 (1.801;16)	-

Table 5.2: Coarsening benchmark: Shown are the total computation time in seconds with the setup time and the number of iterations in parenthesis. Best results are shown in bold.

	fem2d_65025	fem3d_31713	kratos_24202
rs_direct	(6; 2.20; 9.68)	(8; 7.24; 327.01)	(8; 7.22; 297.36)
rsop_direct	(6; 2.20; 8.94)	(5; 2.42; 64.17)	(4; 1.72; 46.75)
rs0_direct	(6; 2.20; 13.79)	(8; 7.34; 300.07)	(8; 6.31; 259.71)
rs3_direct	(7; 2.33; 24.61)	(9; 10.20; 377.43)	(9; 8.68; 326.07)

Table 5.3: Coarsening benchmark, additional information: (Coarse levels; Operator complexity; Maximal stencil size)

paring one-pass, RS0 and RS3 coarsening to the standard RS coarsening approach for different matrices.

The results show that one-pass coarsening is a very good approach for the FEM3D and KRATOS matrices, both 3D grids. Although convergence is worst among the approaches, both setup and solver times are best for both systems. The reasons behind these results can be seen in table 5.3 and figure 5.4: Due to the aggressive coarsening, only roughly half as many coarse levels have to be constructed and furthermore, the number of non-zero coefficients is a lot smaller. RS, RS0 and RS3, on the other hand, construct larger numbers of coarse points, leading to dense matrices on the coarser levels which also slows down the precondition phase.

The results, however, are different for FEM2D: In that case, the second phase does not add many C points such that the results for RS and one-pass coarsening are very similar. Although RS0 and RS3 worsen convergence, they also improve the setup time using parallel coarsening. The total computation time is best for RS3 on the CPU and RS0 on the GPU. The reason for the difference is that on the GPU the additional iterations required by RS0 are less expensive than on the CPU.

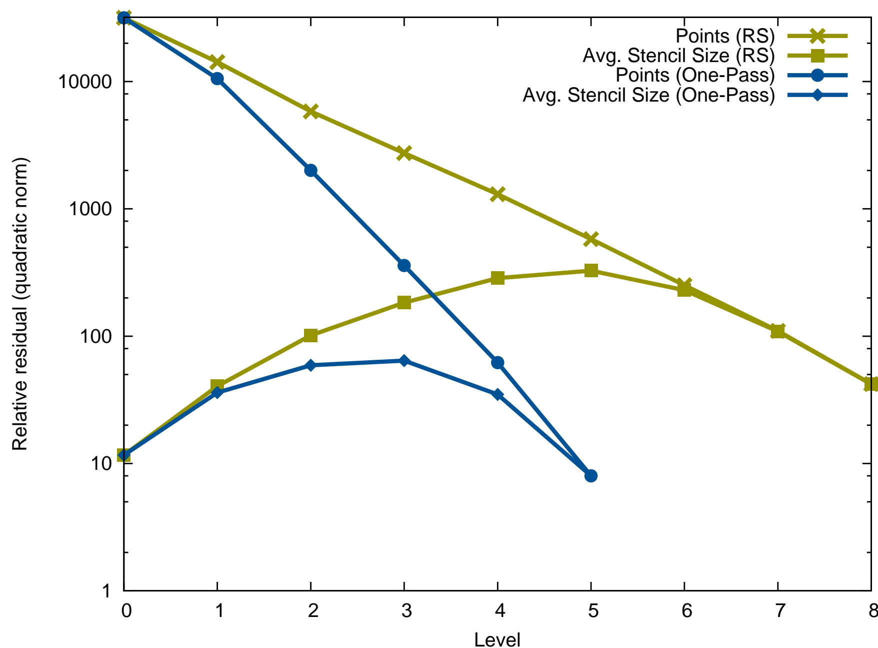

Figure 5.4: Comparison of RS and one-pass coarsening: Shown are the number of points and the average stencil sizes on all levels. RS constructs more C points such that more levels are constructed and the operator matrices become denser.

The benchmark for the SHE1 matrix did not work for the methods except RS because the computation of the estimated residuals for the BiCGStab solver was far off from the real residual value. Therefore, the iteration stopped too early and the results cannot be compared. The result for RS, however, shows that for SHE1 most of the computation time is used on the solver.

Comparing overall results for CPU and GPU computing shows that for the matrices at hand, GPU computing improves solver times by a factor of about 4-5 and total execution times by a factor of about 2.

5.3.3 Interpolation Approaches

The implementation of the AMG preconditioner offers two interpolation schemes for each coarsening approach: Direct and classical interpolation for RS coarsening and basic and smoothed interpolation for aggregation-based coarsening. Tables 5.4

	fem2d_65025	fem3d_31713	kratos_24202	she1_30126
rs_direct (CPU)	**1.034** (0.596;6)	6.206 (1.966;10)	5.976 (1.568;13)	10.543 (0.304;90)
rs_direct (GPU)	0.702 (0.772;6)	2.818 (2.170;10)	2.437 (1.685;13)	2.240 (0.335;90)
rs_classic (CPU)	1.134 (0.675;6)	4.790 (1.819;8)	6.332 (1.543;15)	**10.145** (0.310;87)
rs_classic (GPU)	0.839 (0.769;6)	2.459 (2.001;8)	2.491 (1.687;15)	**1.899** (0.334;74)
ag (CPU)	3.177 (0.292;48)	**1.465** (0.241;25)	**1.500** (0.176;41)	-
ag (GPU)	**0.706** (0.333.48)	**0.534** (0.271;25)	**0.594** (0.191;41)	-
sa (CPU)	4.206 (1.582;12)	3.701 (1.348;11)	2.853 (0.756;19)	-
sa (GPU)	2.281 (1.872;12)	1.882 (1.490;11)	1.304 (0.870;19)	-

Table 5.4: Interpolation benchmark: Shown are the total computation time in seconds with the setup time and the number of iterations in parenthesis. Best results are shown in bold.

	fem2d_65025	fem3d_31713	kratos_24202
rs_direct	(6; 2.20; 9.68)	(8; 7.24; 327.01)	(8; 7.22; 297.36)
rs_classic	(6; 2.20; 9.68)	(8; 6.34; 283.68)	(8; 6.77; 265.38)
ag	(6; 1.93; 6.96)	(5; 1.41; 13.74)	(5; 1.25; 14.67)
sa	(6; 5.71; 107.32)	(5; 4.51; 204.50)	(5; 2.84; 161.76)

Table 5.5: Interpolation benchmark, additional information: (Coarse levels; Operator complexity; Maximal stencil size)

and 5.5 show the results for the different approaches on different system matrices.

The results from this benchmark show a similar result as in section 5.3.2: For 3D problems the method with the worst convergence is the one with the best overall computation time. In this benchmark, this is the aggregation-based coarsening with basic interpolation. The reason for this result is again that fewer coarse points and coarse levels are constructed and the operator matrix stays relatively sparse even on coarser levels. Although smoothed aggregation improves convergence, total and setup times are larger as operator matrices become more dense. However, the results are still better than for RS coarsening.

For the 2D problem, the situation however is different: Here, the convergence of RS coarsening for both interpolation approaches is a lot better and so is the solver time. Aggregation-based coarsening with basic interpolation still has the best setup time, though, such that for GPU computing the total computation time is still best among the approaches. The difference between direct and classical coarsening is only marginal and there is no general result about convergence, setup time or complexity. The total computation time is best for direct interpolation for FEM2D and KRATOS, while it is best for classical interpolation for FEM3D and SHE1.

The benchmark for SHE1 is problematic for the aggregation-based coarsening approaches as the iteration stops too early due errors in the computation of the residuals estimates (see section 5.3.2).

5.3.4 Limitations to AMG

As described in chapters 1 and 2, AMG works very well for symmetric M-matrices and matrices that are not too far off this optimal case. LAME, however, is a matrix which is not even close to being an M-matrix as described in section 5.1. This shows in the benchmark results as the preconditioner actually worsens convergence: While an unpreconditioned CG solver needs 286 iterations to converge, the best preconditioned CG (RS3, direct interpolation) needs 614. Of course, the difference in computation times is even worse.

5.3.5 Summary

The benchmark results show that the implemented AMG preconditioner indeed offers linear complexity for both the setup and the solver phase for most approaches. This makes AMG useful for large matrices when the performance for the unpreconditioned CG and BiCGStab methods goes down. A comparison of the different coarsening and interpolation procedures shows that the performance varies very much between different problems and system matrices. One interesting result, however, is that convergence does not play an important role in determining the overall computation time: Methods offering worse convergence not only lead to smaller setup times, but often reduce the solver time also. The reason behind this result is that these methods, most notably RS one-pass coarsening or aggregation-based coarsening with basic interpolation, lead to fewer coarse points, which in turn leads to fewer coarse levels and sparser operator matrices. In that case, the solver time is reduced even though more iterations are required. It therefore seems that complexity measures might be a better indicator for AMG performance than convergence factors. This result is even stronger if the GPU is used to compute the precondition phase: In that case, additional iterations can be computed very efficiently such that convergence rates factor even less into the total computation time. In such

a situation, the setup phase computed on the CPU dominates total computation time. The speedup from using the GPU scales with the size of the system matrix as the OpenCL overhead becomes relatively small for large matrices. For very small matrices, however, the overhead is relatively large such that it is more efficient to run the solver on the CPU.

Chapter 6

Summary, Conclusions and Possible Extensions

Algebraic multigrid methods can be used to efficiently solve linear equations, either as a stand-alone solver or as a preconditioner for iterative solvers to improve convergence. This is especially useful for large systems as AMG offers linear complexity and therefore scales very well. Although the plethora of variations of AMG components, including smoother, coarsening and interpolation routines, may seem odd, results show that this is justified as different approaches lead to different results in terms of convergence, setup time and complexity. Furthermore, the performance of an AMG method in terms of these properties depends very much on the problem at hand such that an AMG method can be chosen given certain requirements. Benchmark results, however, suggest that the most important factor for AMG performance is complexity, that is, the number of coarse levels as well as the number of non-zero coefficients on each level. This is important in many ways as it determines the memory requirements as well as the number of computations done in the setup and precondition phase. Convergence, on the other hand, is not as important as the gain in performance due to a smaller number of iterations is often counteracted by a much higher number of computations per iteration cycle as shown in chapter 5.

Parallelism is used in a number of ways in the implementation presented: The setup phase uses OpenMP threads on the CPU, while the precondition phase uses

OpenCL to take advantage of the parallel cores on the GPU. Although the latter introduces a certain overhead in terms of data transfers and device control, the results show that this does not play a role if the system matrix is sufficiently large. In that case, GPU computing offers a significant speedup in solver time that comes close to the theoretical maximum for very large matrices. However, the total computation time is bounded by the setup time, which for large matrices dominates the overall computation time if the solver is run on the GPU. Experience gained during the implementation furthermore shows that using efficient data structures and efficient algorithms is at least as important as an efficient use of parallelism. This includes using sparse types for matrices and vectors, but also a useful management of point information and an efficient sparse matrix product algorithm.

Possible extensions to the existing implementation are the implementation of even more AMG variations, especially ones that offer low complexity like the aggressive coarsening approaches. Furthermore, using the block parallelism approach for aggregation-based coarsening much like RS0 would certainly be useful given the good results shown in chapter 5. Although never mentioned in the literature, parallel one-pass coarsening could lead to good results, too. Other parallel coarsening approaches like CLJP or HMIS could also be implemented although relatively high complexity is to be expected.

Some further improvement in performance could be obtained if certain parts of the setup phase could be computed on the GPU. Although this is certainly not possible for the coarsening procedure due to its complex and sequential nature, this could be done for the sparse matrix products, especially the Galerkin operator. However, as OpenCL currently does not support dynamic memory allocation, certain extensions to the algorithm would have to be made: The number of non-zero coefficients would have to computed first, then memory would have to be allocated and then the actual computation could start. This is certainly not very efficient although there could be a net gain due to the high performance of the GPU. Furthermore, the direct solver is currently run on the CPU as the ViennaCL direct solver does not support pivoting. Doing at least the LU substitution on the GPU would therefore improve performance of the GPU preconditioner even more, especially since two data CPU-GPU transfers could be saved in each iteration.

Bibliography

[1] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2nd edition, 2003.
www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.

[2] Ulrich Trottenberg, Cornelis Oosterlee, and Anton Schüller. *Multigrid.* Academic Press, 2001.

[3] Klaus Stüben. Algebraic Multigrid (AMG): An Introduction with Applications. Technical Report November, GMD-Forschungszentrum Informationstechnik, 1999.

[4] Ulrike Meier Yang. Parallel Algebraic Multigrid Methods - High Performance Preconditioners. In Are Magnus Bruaset and Aslak Tveito, editors, *Numerical Solutions of Partial Differential Equations on Parallel Computers (Lecture Notes in Computational Science and Engineering)*, pages 209–236. Springer Verlag, 2006.

[5] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, April 2002.

[6] Petr Vanek, Jan Mandel, and Marian Brezina. Algebraic multigrid on unstructured meshes. Technical report, 1994.

[7] Khronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems.*
http://www.khronos.org/opencl. 2011/08/05.

[8] ViennaCL.
http://viennacl.sourceforge.net. 2011/08/05.

[9] Eigen. *Eigen Library.*
http://eigen.tuxfamily.org. 2011/09/07.

[10] MTL4. *Matrix Template Library 4.*
http://www.mtl4.org. 2011/09/07.

[11] OpenMP. *The OpenMP API specification for parallel programming.*
http://openmp.org. 2011/08/05.

[12] Boost C++ Library. *Basic Linear Algebra Library.*
http://www.boost.org/doc/libs/1_47_0/libs/numeric/ublas/doc.
2011/08/13.

[13] Philipp Grabenweger. *Vergleich von Algorithmen zur Bandbreitenreduktion von bei Finite-Elemente-Methoden auftretenden Matrizen*, 2011.

[14] Intel. *Intel® microprocessor export compliance metrics.*
http://www.intel.com/support/processors/sb/CS-023143.htm.
2011/08/29.

[15] NVIDIA.
http://www.nvidia.com.