NTNU – Trondheim
Norwegian University of
Science and Technology

Krylov Subspace Accelerated Algebraic Multigrid for Mimetic Finite Differences on GPUs

Simen Andreas Andreassen Lønsethagen

Master of Science in Physics and Mathematics
Submission date: June 2012
Supervisor: Helge Holden, MATH
Co-supervisor: Knut-Andreas Lie, SINTEF
Bård Skaflestad, SINTEF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

The topic of this thesis is GPU accelerated sparse linear algebra for subsurface reservoir modeling. Numerical techniques for reservoir simulations are described and we present the open source reservoir simulation software toolbox MRST. We discuss some of the challenges related to linear algebra and reservoir simulation. Furthermore, we discuss the possibility GPU-acceleraing the linear algebra for reservoir simulation, and implement a GPU based CG solver preconditioned with AMG for MRST, using the open source linear algebra library CUSP.

Sammendrag

Temaet for denne oppgaven er GPU-akselerert glissen lineær algebra for undergrunnen reservoarmodellering. Numeriske teknikker for reservoarsimulering beskrives og vi presenterer MRST, programvare for reservoarsimulering med åpen kildekode. Vi diskuterer noen av utfordringene knyttet til lineær algebra og reservoarsimulering. Videre diskuterer vi GPU-akselert lineær algebra for reservoarsimulering, og implementerer en GPU-basert CG-løser prekondisjonert med AMG for MRST, ved å bruke CUSP, et åpen kildekode-bibliotek for glissen lineær algebra på GPUer.

Contents

Abstract i

Sammendrag ii

Preface xi

1 Introduction 1
- 1.1 Research questions 2
- 1.2 Organization of thesis 2

2 Reservoir Simulation 5
- 2.1 Reservoir modeling 5
- 2.2 Discretization methods 7
  - 2.2.1 Two-point flux approximation 8
  - 2.2.2 Mimetic finite difference methods 11
- 2.3 Matlab Reservoir Simulation Toolbox 14

3 Linear Solvers 17
- 3.1 Conjugate gradients 17
  - 3.1.1 Algorithm 18
  - 3.1.2 Convergence 18
  - 3.1.3 Preconditioning 19
- 3.2 Algebraic multigrid 19
  - 3.2.1 Galerkin formulation 20
  - 3.2.2 Algorithm 21
  - 3.2.3 Smoothed aggregation 22
  - 3.2.4 Choosing aggregates for anisotropic problems 22
  - 3.2.5 AMG as preconditioner 25

4 Heterogeneous Computing 27
- 4.1 GPU architecture 27
- 4.2 CUDA 28

4.2.1 Programming model 29
4.3 Linear algebra on GPUs 31
4.3.1 Sparse matrix-vector products 33
4.3.2 CUSP 35

5 Implementation **37**
5.1 External interfaces in Matlab 37
5.2 General setup 38
5.3 Linear solver using CUSP 39

6 Numerical Results **43**
6.1 Test setup 43
6.1.1 Performance 43
6.1.2 Correctness 44
6.1.3 Reference solvers 44
6.1.4 Test platform 45
6.2 Test cases 45
6.2.1 Box 45
6.2.2 Realistic scenarios 48
6.3 Results 49
6.3.1 A note on memory consumption and problem size 49
6.3.2 Performance 50
6.3.3 Correctness 59

7 Conclusions and Further Work **61**
7.1 Conclusion 61
7.2 Further work 62

List of Figures

2.1 Two adjacent cells in TPFA 8
2.2 A minimal MRST program 16

3.1 $N_i(\theta)$ for the five-point discretization of (3.15) 23

4.1 Schematic illustration of a CPU and GPU. 27
4.2 Heterogenous computing with CUDA. 29
4.3 Thread block hierarchy of CUDA. 30
4.4 CUDA memory hierarchy. 32
4.5 Structure of OPL. 33

5.1 Vector addition MEX-file 39

6.1 Visualization of test case 1. 46
6.2 Visualization of well cells in test case 6. 48
6.3 Visualization of grids in realistic models. 49
6.4 Run times for box test cases, $50 \times 50 \times 50$ cells. 50
6.5 Run times for box test cases, GPU solver with CSR and AGMG. . . 51
6.6 Run times for GPU solver with CSR and HYB matrix formats on realistic scenarios. 52
6.7 Effect of θ on run-time on test case 3 54
6.8 Operator complexity as function of θ, test case 3 55
6.9 Test case 3, filtered matrix in prolongator smoother 56
6.10 Running times for GPU solver with and without filtered matrix, and AGMG, all box test cases, $n = 50$. 57
6.11 Run time for realistic scenarios. 58

List of Tables

6.1 Overview of box test cases. 47
6.2 Preconditioner statistics for test case 3, $n = 25$. 56
6.3 Preconditioner properties all test cases, $n = 50$. 58
6.4 Preconditioner properties, realistic models. 59
6.5 2-norm of relative error for box test cases, $n = 50$ 59
6.6 2-norm of relative error for realistic scenarios 59
6.7 Maximal relative divergence for box test cases 60

Preface

This thesis describes the work I have done for my master's studies in industrial mathematics at the Department of Mathematics at the Norwegian University for Science and Technology. The work has been performed at SINTEF ICT's Department of Applied Mathematics in Oslo.

I would like to thank my advisors, Knut-Andreas Lie and Bård Skaflestad at SINTEF ICT, and Helge Holden at the Department of Mathematics at NTNU, for their help and support during my work, and providing office space in Oslo. I would also like to thank André Brodtkorp and Jon Hjelmervik at SINTEF ICT for helping me with some of my frustrations with GPUs and CUDA, and Lars Jahr Røine for listening to my bad jokes. The picture on the cover of this thesis is created with MRST in GNU Octave, using visualization functionality implemented by Lars as part of his master thesis.

Chapter 1

Introduction

This thesis deals with the topic of numerical linear algebra for subsurface reservoir simulation. Reservoir simulation is an active field of research, which includes modeling and numerical discretization amongst its topics. When developing numerical software for reservoir simulation, one faces the challenge of solving large, sparse linear systems. Finding a solution to these may prove to be a bottleneck computation, where exploiting information about the problem at hand may be crucial for performance. In this thesis, we will combine algorithms suited to make use of this information with the processing power of modern graphics processing units (GPUs) in an attempt to implement an efficient linear solver for reservoir simulation.

Our thesis will make use of the Matlab Reservoir Simulation Toolbox (MRST), which is an open source toolbox developed by SINTEF Applied Mathematics [24]. It offers functionality for reservoir simulation and visualization. Intended for prototyping and development of new ideas, MRST provides a framework for reservoir simulation in a high-level language. Our aim will be to implement a black-box solver for the MRST. This will serve us a dual purpose; using MRST, we can readily generate scenarios to test our linear solver, and if our linear solver proves to be efficient and robust, it can offer a way to speed up computations in MRST without any effort required from the end-user.

We have chosen to restrict ourselves to only consider the conjugate gradients method and algebraic multigrid for the algorithms we will base our solvers on. These are well-known algorithms, that have been proved to be efficient for solving the large, sparse linear systems of the symmetric, positive definite type, typically arising in reservoir simulation [25].

Initially developed for 3D graphics rendering, GPUs have become widely used also for scientific computing over the last decade. With their massive parallelism, GPUs offer attractive computational capabilities, and have been used to speed up computations in a wide field of applications [22]. This speed-up comes, however

with the cost of extra programmer effort to map applications onto the parallel architecture of GPUs. On the other side, the rise of general purpose GPU computing has led to the development of several programming frameworks aimed at making the computing power of GPUs more available.

In our work, we have focused on GPUs produced by Nvidia. Nvidia has developed CUDA, which provides a programming model for GPU computing, and an interface based on C/C++. There are several frameworks for sparse linear algebra on Nvidia GPUs. Two of the alternatives are CUSPARSE, which is featured in the official CUDA Toolkit, and CULA, a closed source, proprietary library. We have chosen to use CUSP [5], which is an open source library which implements both conjugate gradients and an algebraic multigrid preconditioner.

1.1 Research questions

To guide the our thesis, we will pose the following questions:

1. Can we use CUSP to create a GPU-based linear solver for MRST implementing conjugate gradients with algebraic multigrid as preconditioner?

2. Are we able to use GPUs to speed up the solution of linear systems in MRST compared to existing CPU-based alternatives?

3. Can we propose and implement improvements to CUSP's linear solver, making it better suited for our application in reservoir simulation?

1.2 Organization of thesis

The thesis is organized as follows:

- Chapter 2 gives a brief introduction to reservoir modeling and numerical methods for discretization of the equations governing the flow in subsurface reservoirs. We also discuss MRST and some of its functionality.

- Chapter 3 presents the conjugate gradients method and algebraic multigrid.

- In Chapter 4, we discuss GPU computing with CUDA and introduces CUSP.

- In Chapter 5, we introduce MEX-interface, which we will need to use CUSP with Matlab. Furthermore, we discuss the implementation details of our linear solver.

- Chapter 6 presents the results of numerical experiments.

- In Chapter 7, we draw conclusions based on the numerical results and suggests further work.

Chapter 2

Reservoir Simulation

Simulation is an important part of oil reservoir management. It is used to model the pressure and flow scenario of a reservoir, and can give valuable information in the planning of the production process.

Realistic reservoirs have complicated geometry, and some of the quantities influencing the flow are hard to measure. Furthermore, as the rock properties are determined by small scale factors such as the size of pores in the rock, exploiting this information to compute quantities on the scale of a realistic reservoir would require tremendous computational capabilities.

In this chapter, we will give a brief introduction to reservoir modeling, and discuss numerical methods used to approximate the solutions of the equations for flow and pressure. We will also introduce MRST [24], a MATLAB toolkit for reservoir simulation developed by SINTEF Applied Mathematics. For a more detailed overview of reservoir modeling, we refer to [1].

2.1 Reservoir modeling

The rock in subsurface reservoirs is a porous medium, which means that even though the rock is solid, a fraction of the volume occupied by the rock will be void, allowing fluids to flow in the rock. This is measured using the rock porosity, ϕ, defined as the void volume fraction of the rock (implying $0 \leq \phi \leq 1$). Typically, ϕ will be a function of spatial coordinates and the pressure p. However, in simple models, it might be sufficient to only consider spatial dependencies for ϕ. Otherwise, it is common to consider a first order linearization in p.

Another important quantity in reservoir modeling is the permeability of the rock, $\mathbf{K}$, which measures the rock's ability to transmit a single fluid at certain conditions. Although it obviously is correlated with the porosity, $\mathbf{K}$ is not necessarily proportional to ϕ. For instance, $\mathbf{K}$ also depends on the spatial orientation

of the rock pores, as the rock will be more permeable if pores are interconnected. This also shows that $\mathbf{K}$ is a tensor, since a specific orientation of the rock pores can cause the rock to be more permeable in one spatial direction. Although $\mathbf{K}$ generally will be dense, it can often be diagonalized through a change of basis, which can simplify computations. If $\mathbf{K}$ can be represented by a scalar function, the problem is called isotropic, otherwise anisotropic. As real world reservoirs consist of different rock types with different permeability, $\mathbf{K}$ can be discontinuous and vary with several orders of magnitude.

Whenever more than one phase is present in the reservoir, it is also necessary to consider the saturation s_i of each phase, which is defined as the fraction of void volume occupied by phase i. Because all of the void volume is assumed to be occupied,

$$\sum_{all\ phases} s_i = 1. \tag{2.1}$$

In reservoir management, it is often sufficient to consider at most three phases, oleic (o), gaseous (g), and aqueous (w). Whenever more than one phase is present, the permeability for one phase will be different from the absolute permeability, $\mathbf{K}$. To adjust for this, relative permeability k_{ri} is defined. As it adjusts for the presence of other phases, k_{ri} is a function of the saturation of the other phases. The effective permeability experienced by a phase α is

$$\mathbf{K}_\alpha = k_{r\alpha}\mathbf{K}. \tag{2.2}$$

Each phase can consists of several components corresponding to different chemical compounds. As there might be many different compounds present in the reservoir, several compounds with fairly equal fluid properties are typically grouped together into pseudocomponents and treated as one component. The mass fraction of a component i in phase j is denoted by c_{ij}, meaning that $\sum_i c_{ij} = 1$ for all phases. As the phase of hydrocarbons can change depending on the pressure and temperature conditions, a component can exist in several phases.

To describe the fluid properties of the phases, we consider the density ρ_α and the kinematic viscosity μ_α of each phase, as these will occur in the equations governing the flow and pressure in the reservoir. These are in general functions of the phase pressure p_j, and the mass fractions of the different components present in the phase. Following [1], we will ignore thermodynamic effects on the reservoir.

The flow and pressure will be governed by the principle of mass conservation and the empirical Darcy's law. If only one phase is considered, mass conservation can be stated as

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla \cdot (\rho \boldsymbol{v}) = q. \tag{2.3}$$

Here, q is a source term modeling sources and sinks, while $\boldsymbol{v}$ is the fluid velocity. Equation (2.3) can be simplified if the porosity is constant in time and the fluid is

incompressible, as the first term vanishes, giving

$$\nabla \cdot \boldsymbol{v} = \frac{q}{\rho}. \tag{2.4}$$

Darcy's law states that

$$\boldsymbol{v} = -\frac{\mathbf{K}}{\mu}(\nabla p - \rho g \nabla z), \tag{2.5}$$

where g the gravitational constant. It is often convenient to consider a pressure potential on the form $u = p - \rho g z$, meaning that Darcy's law can be stated as

$$\boldsymbol{v} = -\frac{\mathbf{K}}{\mu}\nabla u. \tag{2.6}$$

In the case of more than one phase, the multiphase equivalents of the equations above must be considered. Conservation must be stated for each component l, giving the saturation equations,

$$\frac{\partial}{\partial t}(\phi \sum_{\alpha} c_{l\alpha}\rho_\alpha s_\alpha) + \nabla \cdot (\sum_{\alpha} c_{l\alpha}\rho_\alpha \boldsymbol{v}_\alpha) = \sum_{\alpha} c_{l\alpha} q_\alpha, \quad \alpha = i, o, w. \tag{2.7}$$

where $s_\alpha \in [0, 1]$ is the saturation of each phase. It is assumed that the source term can be specified for each phase. For multiphase flow, Darcy's law is stated for each phase, giving

$$\boldsymbol{v}_\alpha = -\mathbf{K}\frac{k_{r\alpha}}{\mu_\alpha}(\nabla p_\alpha - \rho_\alpha g \nabla z). \tag{2.8}$$

To make the models well posed, it is also necessary two specify boundary conditions for (2.7) and (2.8). Although these in principle are problem dependent, it should be pointed out that no flow common boundary conditions are common. Physically, this means that no fluid can leave or enter the reservoir along the boundaries. Mathematically, this means $\boldsymbol{v}_\alpha \cdot \boldsymbol{n} = 0$ on $\delta\Omega$, where $\delta\Omega$ is the boundary of the domain and $\boldsymbol{n}$ is a unit normal vector on the boundary.

Equations (2.7) and (2.8) are the general statements of the equations governing flow and pressure. However, these can often be simplified by assumptions on the porosity and the phases and components present. The simplest example of this is the case of one phase incompressible flow with time constant porosity, which leads to the system of equations given by (2.4) and (2.6).

2.2 Discretization methods

In general, Eqations (2.7) and (2.8) lead to a nonlinearly coupled system of equations. There are several possible discretization strategies for solving the full system

numerically. One possibility, which is common in the industry, is to make a full implicit discretization of both equations, and solving a discretized nonlinear system of equations. However, it is also possible to use a sequential splitting strategy. This is done by at each time step solve a discretized system for the pressure, using the saturations from last time step. After computing the corresponding phase velocities, the saturation equations are then solved keeping the velocities constant. Solving the pressure equations, which becomes elliptic PDEs, are typically the biggest computational challenge. In the following, we will outline some of the techniques that are frequently used for solving these. This is the approach used in MRST [15].

For simplicity, we will derive our numerical schemes for single-phase, incompressible flow. Hence, the equations we seek to discretize are Equations (2.6) and (2.4). To simplify notation, we will drop the constants ρ and μ, and hence we will state the equations as

$$\boldsymbol{v} = -\mathbf{K}\nabla p \tag{2.9}$$

$$\nabla \cdot \boldsymbol{v} = q. \tag{2.10}$$

The first step of developing numerical schemes is to discretize the domain. A natural way of doing this is to create a mesh of polyhedral grid cells. We will assume that the permeability tensor $\mathbf{K}$ can be represented as a cell-wise constant symmetric, positive definite matrix. For technical reasons, we also require eigenvalues of $\mathbf{K}$ to be uniformly bounded from below and above.

2.2.1 Two-point flux approximation

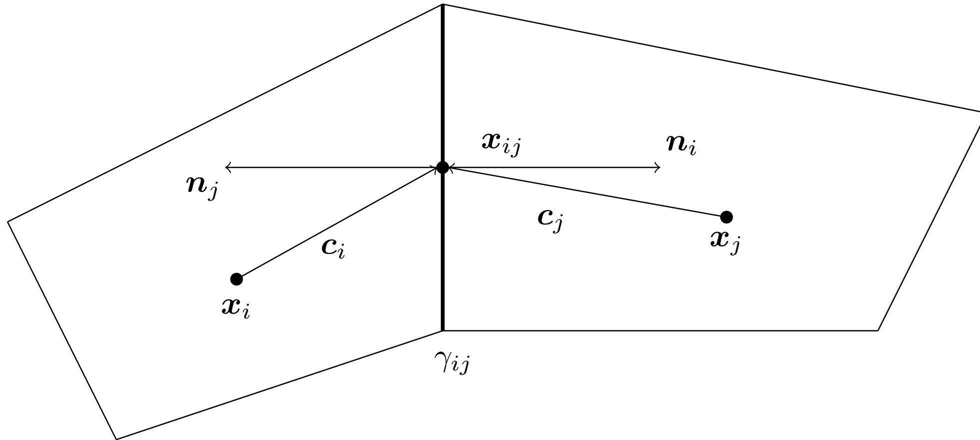


Figure 2.1: Two adjacent cells in TPFA

The most widely used numerical scheme in industry is the two-point flux approximation (TPFA). This is despite the fact that TPFA puts special requirements on the grid, which in practical cases often are not met. Given a partitioning of the domain into polyhedral grid cells, say Ω_i, we integrate (2.10) over one grid cell,

$$\int_{\Omega_i} \nabla \cdot \boldsymbol{v} dV = \int_{\Omega_i} q dV. \tag{2.11}$$

Assuming that $\boldsymbol{v}$ is sufficiently smooth, we use the divergence theorem to obtain

$$\int_{\delta\Omega_i} \boldsymbol{v} \cdot \boldsymbol{n} d\nu = \int_{\Omega_i} q dV. \tag{2.12}$$

The right-hand side of the equation is assumed to be known. Hence, a discretization scheme should find an approximation for the flux over the faces of each grid cell.

Following [2], we will derive an approximation for the flux over cell faces using a cell centered volume method, where the flux over the face between two adjacent cells is approximated by a relation to the potentials in the cell centroids of the respective cells. Figure 2.1 shows this situation for two two-dimensional cells. There, $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are the respective cell centroids, γ_{ij} is the face between the cells, and $\boldsymbol{x}_{ij}$ is the midpoint of γ_{ij}. The vectors pointing from the cell centroids to $\boldsymbol{x}_{ij}$ are denoted $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$, and the unit normals pointing out of the cells $\boldsymbol{n}_i$ and $\boldsymbol{n}_j$, respectively. It should be noted that the derivation will hold in both two and three dimensions, as we makes no assumption on the grid being two-dimensional.

The flux over γ_{ij}, say v_{ij}, is given by integrating $\boldsymbol{v}$ over γ_{ij}, and substituting for (2.9), giving

$$v_{ij} = \int_{\gamma_{ij}} \boldsymbol{v} \cdot \boldsymbol{n} d\nu = -\int_{\gamma_{ij}} \boldsymbol{n}^T \mathbf{K} \nabla u d\nu. \tag{2.13}$$

Introducing $\boldsymbol{w} = \mathbf{K}\boldsymbol{n}$, this can be written as

$$v_{ij} = -\int_{\gamma_{ij}} \boldsymbol{w} \cdot \nabla u d\nu, \tag{2.14}$$

which shows that v_{ij} is the integral over γ_{ij} of the directional derivative of ∇u along $\boldsymbol{w}$. In TPFA, we seek to express a discretization of this expression using the potentials u_i and u_j in $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, respectively. To this end, we introduce the fictious point $\boldsymbol{x}_{ij}$, associated with a potential u_{ij}. Since the integrand is the directional derivative of ∇u along $\boldsymbol{w}$, it can be discretized consistently using u_i and u_{ij} if and only if $\boldsymbol{c}_i$ and $\boldsymbol{w}_i$ are parallel [2]. In this case, we obtain the approximation

$$\boldsymbol{w} \cdot \nabla u \approx (u_{ij} - u_i)\frac{\|\boldsymbol{w}_i\|}{\|\boldsymbol{c}_i\|}, \tag{2.15}$$

and an analogous expression using u_j. If we approximate the integrand to be constant over γ_{ij} and denote the area of the face as $|\gamma_{ij}|$, this leads to the following approximations for the flux for the two cells,

$$v_{ij} = |\gamma_{ij}|(u_{ij} - u_i)\frac{\|\boldsymbol{w}_i\|}{\|\boldsymbol{c}_i\|} \tag{2.16}$$

$$v_{ji} = |\gamma_{ij}|(u_{ij} - u_j)\frac{\|\boldsymbol{w}_j\|}{\|\boldsymbol{c}_j\|}. \tag{2.17}$$

The second expression follows by symmetry. Since the flux must be continuous, it follows that $v_{ji} = -v_{ij}$. Hence, by summing and eliminating u_{ij}, we get the following formula relating the cell centroid potentials to the flux,

$$v_{ij} = |\gamma_{ij}|\left(\frac{\|\boldsymbol{c}_i\|}{\|\mathbf{K}_i\boldsymbol{n}_i\|} + \frac{\|\boldsymbol{c}_j\|}{\|\mathbf{K}_j\boldsymbol{n}_j\|}\right)^{-1}(u_j - u_i). \tag{2.18}$$

Writing

$$t_{ij} = |\gamma_{ij}|\left(\frac{\|\boldsymbol{c}_i\|}{\|\boldsymbol{K}_i\boldsymbol{n}_i\|} + \frac{\|\boldsymbol{c}_j\|}{\|\mathbf{K}_j\boldsymbol{n}_j\|}\right)^{-1}, \tag{2.19}$$

the flux over γ_{ij} is

$$v_{ij} = t_{ij}(u_j - u_i). \tag{2.20}$$

Summing over the edges j of Ω_i and inserting the corresponding flux approximations from (2.20) in (2.12), we get

$$\sum_j t_{ij}(u_j - u_i) = \int_{\Omega_i} q dV \quad \forall \Omega_i \subset \Omega, \tag{2.21}$$

which can be written as a symmetric, positive definite linear system for the cell potentials [1].

As mentioned above, TPFA requires that for each grid cell, $\boldsymbol{c}_i$ is parallel with $\mathbf{K}_i\boldsymbol{n}_i$ for all faces in the cell. A grid that meets this condition is said to be **K**-orthogonal. In practical situation, there is no guarantee that the grid can be constructed to meet the requirement of **K**-orthogonality. If the grid is not **K**-orthogonal, TPFA can not be expected to converge to the correct solution, and this is the big drawback of this method.

2.2.2 Mimetic finite difference methods

The mimetic finite difference (MFD) method represents a different approach to solving (2.9) and (2.10). In contrast to TPFA, mimetic methods do not require a **K**-orthogonal grid. In [7], Brezzi et al proves convergence properties for unstructured polyhedral meshes. The conditions required on the mesh for the convergence analysis are very loose, and will almost always be met in practice [7]. Mimetic methods can be developed by defining inner products on the function spaces of pressures and velocities, for which the gradient and divergence operator are adjoint. A mimetic method can then be derived by defining discrete analogues to the inner products, function spaces and differential operators such that the discrete versions have similar properties to the continuous. This is where the name of the methods come from. However, we will discuss MFD using the approach of [15], which uses a matrix approach to develop the formulation of the method.

Given a polyhedral grid cell Ω_i, we define N_e as the number of faces, $|E|$ as the cell volume and p_i as our approximation of the pressure in the cell centroid. Furthermore, we let $\boldsymbol{u}$ and $\boldsymbol{\pi}$ denote vectors containing the fluxes over the cell faces and the pressures in the cell centroids, respectively. Hence, we have $\boldsymbol{u},\ \boldsymbol{\pi} \in \mathbb{R}^{N_e}$. For each face j, we recall that $\boldsymbol{c}_j$ is the vector pointing from the cell centroid to the face centroid. Moreover, $\boldsymbol{n}_j$ will now denote the area-weighted normal vector pointing out of the cell; that is, a normal vector of length equal to the area of the face. Using these, we define $\boldsymbol{N}$ and $\boldsymbol{C}$ as the matrices where $\boldsymbol{n}_j$ and $\boldsymbol{c}_j$ are the jth rows. We will use that $\boldsymbol{N},\ \boldsymbol{C} \in \mathbb{R}^{N_e \times 3}$ have full rank and that $\boldsymbol{C}^T\boldsymbol{N} = |E|\boldsymbol{I}$, where $\boldsymbol{I}$ is the $N_e \times N_e$ identity matrix [15].

The mimetic method seeks to find an expression relating flux to the face centroid pressures on the form

$$\boldsymbol{M}\boldsymbol{u} = p_i\boldsymbol{e} - \boldsymbol{\pi}, \tag{2.22}$$

where $\boldsymbol{e} = (1, \ldots, 1)^T$ and $\boldsymbol{M} \in \mathbb{R}^{N_e \times N_e}$ is a symmetric, positive definite matrix [15]. This can also be expressed using the transmissibility matrix $\boldsymbol{T} = \boldsymbol{M}^{-1}$ as

$$\boldsymbol{u} = \boldsymbol{T}(p_i\boldsymbol{e} - \boldsymbol{\pi}). \tag{2.23}$$

As $\boldsymbol{M}$ is symmetric, positive definite, it is also referred to as an inner product.

MFD methods are required to give exact approximations for linear pressure fields, i.e

$$p(\boldsymbol{x}) = \boldsymbol{a} \cdot \boldsymbol{x} + b, \tag{2.24}$$

where $\boldsymbol{a} \in \mathbb{R}^3$ and $b \in \mathbb{R}$ are constant. We note that (2.24) implies that the difference between the pressure p_i in the cell centroid and the pressure π_j at the jth face centroid, is simply

$$p_i - \pi_j = \boldsymbol{c}_j^T\boldsymbol{a}. \tag{2.25}$$

Combining (2.24) and (2.9) gives a constant fluid velocity,

$$\boldsymbol{v} = -\mathbf{K}\boldsymbol{a}, \tag{2.26}$$

which together with (2.13) means that the flux u_j over face j is

$$u_j = -\boldsymbol{n}_j^T\mathbf{K}\boldsymbol{a}. \tag{2.27}$$

Inserting (2.27) and (2.25) into (2.22) we get

$$\boldsymbol{M}\boldsymbol{N}\mathbf{K}\boldsymbol{a} = \boldsymbol{C}\boldsymbol{a}. \tag{2.28}$$

Since the mimetic method is required to be exact for all linear pressure distributions, this must hold for any $\boldsymbol{a}$, and we arrive at our consistency requirement for $\boldsymbol{M}$,

$$\boldsymbol{M}\boldsymbol{N}\mathbf{K} = \boldsymbol{C} \tag{2.29}$$

Using the fact that $\boldsymbol{C}^T\boldsymbol{N} = |E|\boldsymbol{I}$, we post-multiply Equation (2.29) with $\mathbf{K}^{-1}\boldsymbol{C}^T\boldsymbol{N}$ to derive

$$\boldsymbol{M}\boldsymbol{N} = \frac{1}{|E|}\boldsymbol{C}\mathbf{K}^{-1}\boldsymbol{C}^T\boldsymbol{N}. \tag{2.30}$$

As $\boldsymbol{N}$ has full rank, this implies that a valid choice for $\boldsymbol{M}$ must be on the form

$$\boldsymbol{M} = \frac{1}{|E|}\boldsymbol{C}\mathbf{K}^{-1}\boldsymbol{C}^T + \boldsymbol{M}_2, \tag{2.31}$$

where $\boldsymbol{M}_2$ is any matrix such that $\boldsymbol{M}_2\boldsymbol{N} = \mathbf{0}$ and $\boldsymbol{M}$ is symmetric positive definite. Since $\mathbf{K}$ is symmetric, positive definite, $\frac{1}{|E|}\boldsymbol{C}\mathbf{K}^{-1}\boldsymbol{C}^T$ is symmetric, positive semi-definite, meaning it is clearly necessary that $\boldsymbol{M}_2$ is a symmetric positive semi-definite matrix which can be expressed as

$$\boldsymbol{M}_2 = \boldsymbol{Q}_N^{\perp}\boldsymbol{S}_M{\boldsymbol{Q}_N^{\perp}}^T, \tag{2.32}$$

where $\boldsymbol{Q}_N^{\perp}$ is an orthonormal basis for the null space of $\boldsymbol{N}$ and $\boldsymbol{S}$ is a symmetric positive definite matrix. As this implies that $\boldsymbol{M}$ is symmetric positive semi-definite, we only need to show that it in fact implies positive definiteness for sufficiency. To show this, we can use the fact that any vector $\boldsymbol{z} \in \mathbb{R}^n$ can be split uniquely as $\boldsymbol{z} = \boldsymbol{z}_1 + \boldsymbol{z}_2,\ \boldsymbol{z}_1 \in \mathrm{Ran}(N^T),\ \boldsymbol{z}_2 \in \mathrm{Null}(N^T)$ [21], or equivalently, $\boldsymbol{z} = \boldsymbol{N}\tilde{\boldsymbol{z}}_{\mathbf{1}} + \boldsymbol{z}_2,\ \tilde{\boldsymbol{z}}_{\mathbf{1}} \in \mathbb{R}^n$. Using (2.31), (2.32) and $\boldsymbol{C}^T\boldsymbol{N} = |E|\boldsymbol{I}$, we get

$$\boldsymbol{z}^T\boldsymbol{M}\boldsymbol{z} = |E|\tilde{\boldsymbol{z}}_1^T\mathbf{K}^{-1}\tilde{\boldsymbol{z}}_1 + \boldsymbol{z}_2^T\frac{1}{|E|}\boldsymbol{C}\mathbf{K}^{-1}\boldsymbol{C}^T\boldsymbol{z}_2 + \boldsymbol{z}_2\boldsymbol{Q}_N^{\perp}\boldsymbol{S}_M{\boldsymbol{Q}_N^{\perp}}^T\boldsymbol{z}_2 > 0 \tag{2.33}$$

as at least either the first or last term are positive since $\mathbf{K}$ and $\boldsymbol{S}_M$ are positive definite and the middle term is nonnegative as the matrix is positive semi-definite. To sum up, any inner product $\boldsymbol{M}$ can be written as

$$\boldsymbol{M} = \frac{1}{|E|}\boldsymbol{C}\mathbf{K}^{-1}\boldsymbol{C}^T + \boldsymbol{Q}_N^{\perp}\boldsymbol{S}_M{\boldsymbol{Q}_N^{\perp}}^T. \tag{2.34}$$

A similar calculation for $\boldsymbol{T} = \boldsymbol{M}^{-1}$ yields

$$\boldsymbol{T} = \frac{1}{|E|}\boldsymbol{N}\mathbf{K}\boldsymbol{N}^T + \boldsymbol{Q}_C^{\perp}\boldsymbol{S}_T{\boldsymbol{Q}_C^{\perp}}^T, \tag{2.35}$$

with $\boldsymbol{S}_T$ and $\boldsymbol{Q}_C^{\perp}$ analogous to $\boldsymbol{S}_M$ and $\boldsymbol{Q}_N^{\perp}$. Hence, MFD gives a class of discretization schemes, where a grid and a choice of a symmetric positive definite matrix gives a specific discretization scheme. One example is $\boldsymbol{S}_T = \frac{2}{|E|}\boldsymbol{P}_C^{\perp}\mathrm{diag}(\boldsymbol{N}\mathbf{K}\boldsymbol{N}^T){\boldsymbol{P}_C^{\perp}}^T$, where $\boldsymbol{P}_C^{\perp}$ is the projector onto $\mathrm{Null}(\boldsymbol{C}^T)$, which will give a method that coincides with TPFA for $\mathbf{K}$-orthogonal grids [15].

Another inner product is `IP_SIMPLE`, which is the default inner product in MRST. It is given by

$$\begin{aligned}\boldsymbol{Q} &= \mathrm{orth}(\boldsymbol{A}^{-1}\boldsymbol{N}) \\ \boldsymbol{M} &= \frac{1}{|E|}\boldsymbol{C}\mathbf{K}^{-1}\boldsymbol{C}^T + \frac{d|E|}{6\mathrm{tr}(\mathbf{K})}\boldsymbol{A}^{-1}(\boldsymbol{I} - \boldsymbol{Q}\boldsymbol{Q}^{\boldsymbol{T}})\boldsymbol{A}^{-1},\end{aligned} \tag{2.36}$$

where $\boldsymbol{A}$ is a diagonal matrix with the face areas on the diagonal and $d = 2, 3$ is the dimension of $\mathbb{R}^d$. For $\boldsymbol{T}$, the approximate inverse is typically used,

$$\begin{aligned}\boldsymbol{Q} &= \mathrm{orth}(\boldsymbol{A}\boldsymbol{C}) \\ \boldsymbol{T} &= \frac{1}{|E|}\left[\boldsymbol{N}\mathbf{K}\boldsymbol{N}^T + \frac{6}{d}\mathrm{tr}(\mathbf{K})\boldsymbol{A}(\boldsymbol{I} - \boldsymbol{Q}\boldsymbol{Q}^{\boldsymbol{T}})\boldsymbol{A}\right].\end{aligned} \tag{2.37}$$

In addition to (2.22), where $\boldsymbol{M}$ must satisfy (2.34), the flux for each cell must satisfy (2.12), and for two adjacent cells, the flux over the adjacent face must be equal, but with opposite sign. Given a global ordering of the cells and faces, this can be formulated as a linear system on hybrid form,

$$\begin{bmatrix}\boldsymbol{B} & \boldsymbol{C} & \boldsymbol{D} \\ \boldsymbol{C}^T & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}^T & \boldsymbol{0} & \boldsymbol{0}\end{bmatrix}\begin{bmatrix}\boldsymbol{u} \\ -\boldsymbol{p} \\ \boldsymbol{\pi}\end{bmatrix} = \begin{bmatrix}\boldsymbol{0} \\ \boldsymbol{q} \\ \boldsymbol{0}\end{bmatrix}. \tag{2.38}$$

Here, $\boldsymbol{B}$ and $\boldsymbol{C}$ are block diagonal matrices where the ith block contains the inner product $\boldsymbol{M}_i$ and the vector $\boldsymbol{e}_i$ for cell i, respectively. The columns of $\boldsymbol{D}$ correspond

to unique faces and has ones in the rows corresponding to the face in the global ordering (hence a column corresponding to a face has two nonzero entries for interior faces and one for boundary faces). The vectors $\boldsymbol{u}$, $\boldsymbol{p}$ and $\boldsymbol{\pi}$ contain all face fluxes, centroid pressures and face pressures respectively, consistently with the global cell and face ordering. The ith entry of the vector $\boldsymbol{q}$ is the integral $\int_{\Omega_i} q dV$.

The hybrid system (2.38) can be rewritten to Schur form using a block wise Gaussian elimination, which results in the symmetric positive definite system [15]

$$(\boldsymbol{D}^T\boldsymbol{B}^{-1}\boldsymbol{D} - \boldsymbol{F}^T\boldsymbol{L}^{-1}\boldsymbol{F})\boldsymbol{\pi} = \boldsymbol{F}^T\boldsymbol{L}^{-1}\boldsymbol{q}, \tag{2.39}$$

where $\boldsymbol{F} = \boldsymbol{C}^T\boldsymbol{B}^{-1}\boldsymbol{D}$ and $\boldsymbol{L} = \boldsymbol{C}^T\boldsymbol{B}^{-1}\boldsymbol{C}$. Once (2.39) is solved, the fluxes and centroid pressures can be computed by

$$\boldsymbol{L}\boldsymbol{p} = \boldsymbol{q} + \boldsymbol{F}\boldsymbol{\pi},\ \boldsymbol{u} = \boldsymbol{B}^{-1}(\boldsymbol{C}\boldsymbol{p} - \boldsymbol{D}\boldsymbol{\pi}). \tag{2.40}$$

This suggests that when computing the inner products, instead of computing $\boldsymbol{B}$ and then inverting it, it is more efficient to compute $\boldsymbol{B}^{-1}$ directly, using that it is block diagonal with the transmissibility matrices $\boldsymbol{T}_i$ as blocks. As $\boldsymbol{L}$ is diagonal, his shows that computing $\boldsymbol{p}$ and $\boldsymbol{u}$ is computationally inexpensive once (2.39) is solved. This also means that solving (2.39) will dominate the computational cost of finding a numerical approximation to the flux and pressure.

2.3 Matlab Reservoir Simulation Toolbox

Matlab Reservoir Simulation Toolbox (MRST) [24] is an open source toolbox for reservoir modeling and simulation in Matlab developed by SINTEF Applied Mathematics. MRST contains routines for grid processing, modeling physical quantities such as rock and fluid properties, numerical discretization and visualization. It is intended to provide tools for prototyping and testing methods and concepts on complex grids, and supports one and two phase, incompressible flows.

The grid structure in MRST can handle grids consisting of irregular, polyhedral cells, by using a general storage format. To this end, MRST contains several functions for creating and manipulating grids, and functions for reading input from files. In addition to the grid, setting the fluid and rock properties are necessary to set up a simulation scenario. These can be generated by MRST, set explicitly by the user, or read from file. MRST assumes no-flow boundary conditions by default, but features various routines for adding different boundary conditions, and it uses a well structure for handling sources and sinks.

As discussed in Section 2.2, a mimetic discretization scheme can be created using the geometry of the grid and the permeability of the rock. The function `computeMimeticIP` creates a matrix using MFD. By default, MRST will solve the

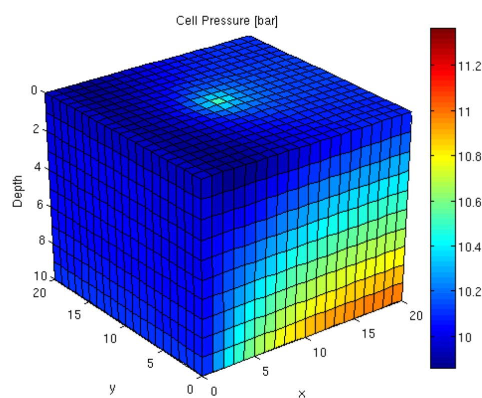
discretized pressure equations using hybrid form, given by (2.38). As (2.38) can be solved on Schur form (2.39), `computeMimeticIP` will output $\boldsymbol{B}^{-1}$ in (2.39). Although MRST supports other forms than (2.38), this is the form we will use in our work. The type of inner product can be specified with the option `'InnerProduct'`. If no option is given, the default in MRST is `ip_simple`, using (2.37).

To solve the discretized pressure equations on hybrid form, the function `solveIncompFlow` creates the full system (2.38), reduces it to Schur form, and calls a linear solver to solve (2.39). By default, the linear solver is `mldivide`, Matlab's standard linear solver. However, it is possible to pass another solver as a function handle using the option `'LinSolve'`.

Figure 2.2 shows a minimal MRST program and the plot it produces based on one of the examples from [15]. The program creates a $20\times20\times10$-cell Cartesian grid where the permeability is constant. In addition to gravitational forces, constant pressure is applied on the left hand side on the grid, and in the middle, a column of sources is inserted.

```
1  % Create a 20-by-20-by-10-cell Cartesian grid
2  nx = 20; ny = 20; nz = 10;
3  G = cartGrid([nx, ny, nz]);
4  G = computeGeometry(G);
5
6  % Set rock and fluid properties
7  rock.perm = repmat(100 * milli*darcy, [G.cells.num, 1]);
8  fluid = initSingleFluid('mu', 1*centi*poise, 'rho', ...
       1014*kilogram/meter^3);
9  gravity reset on
10
11 % Set sources and pressure boundary conditions
12 c = (nx/2*ny+nx/2 : nx*ny : nx*ny*nz) .';
13 src = addSource([], c, ones(size(c)) ./ day());
14 bc = pside([], G, 'LEFT', 10*barsa());
15
16 % Create and solve a
17 S = computeMimeticIP(G, rock, 'Type', 'hybrid', 'InnerProduct', ...
       'ip_simple');
18 rSol = initResSol(G, 0);
19 rSol = solveIncompFlow(rSol, G, S, fluid,'src', src, 'bc' , bc, ...
       'LinSolve', @mldivide);
20
21 % Plot cell pressure
22 clf
23 plotCellData(G, convertTo(rSol.pressure(1:G.cells.num), barsa()), ...
24              'EdgeColor', 'k');
25 title('Cell Pressure [bar]')
26 xlabel('x'), ylabel('y'), zlabel('Depth');
27 view(3); shading faceted; camproj perspective; axis tight;
28 colorbar
```

(a) Code



(b) Plot produced

Figure 2.2: A minimal MRST program

Chapter 3

Linear Solvers

As seen in chapter 2, when doing numerical reservoir simulation, it is necessary to solve the symmetric, positive definite system (2.39). In this chapter, we will describe conjugate gradients and algebraic multigrid, two algorithms for solving systems of this kind.

Notation

It is convenient to introduce some notation for the sections below. Following standard notation of linear algebra, uppercase latin letters will denote real-valued $n \times n$ matrices, where $n \in \mathbb{N}$. In particular, A will always be used to denote a symmetric, positive definite matrix. Lowercase latin letters will denote real valued vectors in $\mathbb{R}^n$, except for i, j, k, l and n, which will denote natural numbers. Lowercase greek letters will denote real valued scalars.

3.1 Conjugate gradients

The conjugate gradients algorithm (CG) is a well known iterative method for solving sparse, symmetric, positive definite linear systems. It was first introduced by Hestenes and Stiefel in 1952 as a direct method [12], but has since become one of the most popular iterative methods for sparse SPD problems [9]. CG is a Krylov subspace method, that is, it searches for a solution of the linear system

$$Ax = b \tag{3.1}$$

in the k-dimensional Krylov subspace $\mathcal{K}_k(A, b)$, defined as

$$\mathcal{K}_k(A, b) = \text{span}\{b, Ab, \ldots, A^{k-1}b\}. \tag{3.2}$$

CG is characterized by the fact that the solution at step k, say x_k minimizes the A-norm of the error over all vectors in $\mathcal{K}_k$ [21], that is

$$x_k = \arg\min_{y \in \mathcal{K}_k} \|x^* - y\|_A, \tag{3.3}$$

where $x^* = A^{-1}b$ is the exact solution of the linear system, and $\|\cdot\|_A$ is the norm defined by $\|x\|_A^2 = x^T A x$.

In the following, we will describe some of the key features of CG. For a more complete overview, we refer to [9, 21, 23].

3.1.1 Algorithm

Algorithm 1 shows CG as presented in [9]. For simplicity, x_0 is set to the zero vector. CG searches for a solution along an A-orthogonal basis $\{p_0, \ldots, p_k\}$ for $\mathcal{K}_k$. At each iteration step k, it computes the next update x_{k+1} such that the next residual is orthogonal to $\mathcal{K}_k$. This makes the error A-orthogonal to $\mathcal{K}_k$ (and hence minimal), and it also ensures that r_{k+1} is A-orthogonal to all the p_i's except p_k, which in turn means that only p_k needs to be stored to compute p_{k+1} [23].

Algorithm 1 Conjugate gradients

1: $x_0 = 0, r_0 = p_0 = b$
2: **for** $k = 0, 1, \ldots$ until convergence **do**
3: $\quad y_k = Ap_k$
4: $\quad \alpha_k = (r_k^T r_k)/(y_k^T p_k)$
5: $\quad x_{k+1} = x_k + \alpha_k p_k$
6: $\quad r_{k+1} = r_k - \alpha_k y_k$
7: $\quad \beta_k = (r_{k+1}^T r_{k+1})/(r_k^T r_k)$
8: $\quad p_{k+1} = r_{k+1} + \beta_k p_k$
9: **end for**

3.1.2 Convergence

In exact arithmetic, CG gives the exact solution $x^* = A^{-1}b$ in at most n steps. In practice, this may not happen because round off errors are magnified [9]. As the algorithm still may give good approximations after much fewer than n iterations, it is the convergence analysis of the iterative version that is of interest.

The convergence analysis of CG is closely related to Chebychev polynomials. As the the error is related to the residual by $r_k = -Ae_k$, the definition of $\mathcal{K}_k$ implies that the error at the kth step can be expressed as

$$e_k = p_k(A)e_0, \tag{3.4}$$

where p_k is a polynomial of degree k with $p_k(0) = 1$ and e_0 is the initial error. As CG minimizes $\|e_k\|_A$, CG must pick p_k to minimize this error over all such polynomials. Using the optimality properties of the Chebyshev polynomials, it can be shown that [21]

$$\|e_k\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|e_0\|_A, \tag{3.5}$$

where κ is the condition number of A.

3.1.3 Preconditioning

The error bound (3.5) in the end of section 3.1.2 illustrates the idea behind preconditioning, as convergence is limited by the condition number of A. In order to improve the convergence rate, one may attempt to lower the condition number of A by multiplying with a symmetric, positive definite matrix M^{-1} that in some sense approximates A^{-1}. By choosing M^{-1} such that $\kappa(M^{-1}A) \ll \kappa(A)$ the number of iterations needed to reach convergence may be significantly reduced. However one cannot apply CG directly to the system $M^{-1}Ax = M^{-1}b$, as the resulting matrix generally is not symmetric [9]. This problem can be overcome, for instance by formally introducing the linear system

$$(M^{-1/2}AM^{-1/2})(M^{1/2}x) = M^{-1/2}b, \tag{3.6}$$

and performing algorithm 1 on it. As M^{-1} (and hence M) is assumed to be symmetric positive definite, $M^{-1/2}$ is well defined [9]. It should also be noted that since $M^{-1/2}AM^{-1/2}$ and $M^{-1}A$ are similar matrices, they have the same condition number. Performing CG on (3.6) will produce an estimate for $M^{1/2}x$, which is not really of interest. However, through a change of variables, it can be shown [9] that performing CG on (3.6) is mathematically equivalent to algorithm 2, which produces an estimate for the solution of the original system.

From line 7 of Algorithm 2, we see that an operation on the form $M^{-1}v$ must be performed for each iteration. As this may mean applying some linear operator on v or solving the system $Mu = v$, it is clear that there is a trade off when choosing the preconditioner between choosing M^{-1} to be "close" to A^{-1} and the cost of applying M^{-1}.

3.2 Algebraic multigrid

Algebraic multigrid (AMG) represents an approach to solving sparse linear systems that is fundamentally different from the Krylov subspace approach of CG. AMG

Algorithm 2 Preconditioned conjugate gradients

1: $x_0 = 0, r_0 = b, p_0 = z_0 = M^{-1}r_0$
2: **for** $k = 0, 1, ...$ until convergence **do**
3: $\quad y_k = Ap_k$
4: $\quad \alpha_k = (r_k^T z_k)/(y_k^T p_k)$
5: $\quad x_{k+1} = x_k + \alpha_k p_k$
6: $\quad r_{k+1} = r_k - \alpha_k y_k$
7: $\quad z_{k+1} = M^{-1} r_{k+1}$
8: $\quad \beta_k = (r_{k+1}^T z_{k+1})/(r_k^T z_k)$
9: $\quad p_{k+1} = z_{k+1} + \beta_k p_k$
10: **end for**

was developed as an extension of the ideas of geometric multigrid [21] to handle problems were the geometric information required by multigrid is unavailable or the geometry is so complex that multigrid implementations are either very complicated or not feasible at all. A complete introduction to AMG can be found in [25].

3.2.1 Galerkin formulation

The key idea of AMG is to extend the idea of solving a linear system stemming from some discretization of a partial differential equations on successively coarser grids to situations where grids are unavailable, using only the information contained in the matrix. This is usually done in the formal framework of the Galerkin formulation, which makes it possible to define analogues to the fine and coarse grids of geometric multigrid, and analyze the transfer between them.

Following [25], two-level AMG is described using the index sets Ω^h and Ω^H to denote the sets of degrees of freedom on the fine and coarse levels, respectively. Given $\Omega^h = \{1, \ldots, n\}$, the fine problem is defined as

$$A_h x^h = b^h \iff \sum_{j \in \Omega^h} a_{ij}^h x_j^h = b_j^h \quad (i \in \Omega^h). \tag{3.7}$$

To transfer the fine problem to the coarse one, Ω^h is split into two disjoint sets C and F such that $\Omega^h = C \cup F$, where C represents the coarse level, such that $C = \Omega^H$. Corresponding to this splitting, a restriction operator, P_h^H, and interpolation operator, P_H^h, mapping between the coarse and fine problem must be defined. This allows the degrees of freedom in F to be interpolated onto C and

vice versa. Assuming the operators to be given, the coarse problem is defined as

$$A_H x^H = b^H, \tag{3.8}$$

$$A_H = P_h^H A_h P_H^h, \tag{3.9}$$

$$x^H = P_h^H x^h, \quad b^H = P_h^H b^h. \tag{3.10}$$

The matrix A_H is referred to as a Galerkin product. Both the interpolation and restriction operators must be of full rank. For symmetric positive definite systems, we also require

$$P_H^h = (P_h^H)^T. \tag{3.11}$$

3.2.2 Algorithm

The last missing piece to describe a two-level AMG algorithm is a smoothing operator on the fine level, say S_h. S_h is typically a simple relaxation scheme, such as Jacobi or Gauss-Seidel. As for geometric multigrid, the smoothing operator should reduce "oscillatory" components of the error, while "smooth" components must be dealt with on the coarser level. However, as the notion of smoothness cannot be related to the geometry of the grid in AMG, a smooth error is defined as an error components that cannot be resolved efficiently by the smoother, i.e. $S_h e \approx e$, and hence must be treated on the coarser level [25].

With this in place, the two-level AMG algorithm is presented in Algorithm 3 as in [21]. To extend the two-level cycle to the full AMG algorithm, the two-level is initially called with A from (3.1) as the matrix on the finest level. Then, assuming that operators for all the coarser levels are defined, the solve in line 4 of Algortihm 3 is replaced with a recursive call to the two-level cycle itself. When the recursive call reaches the coarsest level, the coarse system is typically solved using a direct solver such as LU. When the recursive call is done once at each level, one full cycle of the resulting algorithm is called a V-cycle, refering to how the algorithm traverses the recursive call stack. There are other possible options, two recursive calls at each level is for example called a W-cycle. In our numerical work, we will use the V-cycle algorithm.

Algorithm 3 Two-level AMG cycle

1: $x^h = S^h(A_h, x_0^h, b^h)$ Pre-smooth
2: $r^h = b^h - A_h x^h$ Get residual
3: $r^H = P_h^H r^h$ Coarsen
4: Solve $A_H \delta^H = r^H$
5: $x^h = x^h + P_H^h \delta^H$ Correct
6: $x^h = S^h(A_h, x^h, b^h)$ Post-smooth

3.2.3 Smoothed aggregation

In contrast to geometric multigrid, the interpolation and restriction operators on each level cannot be predefined for AMG. Instead, the algorithm has an initial setup phase, where the operators are constructed together with the matrices for the coarse level problems. If the levels are numbered from 1 to L, where L is the coarsest level, the operators on each level l depends on A_{l-1}, the the Galerkin product on level $l-1$.

As the smoothing operation by definition operates inefficiently on smooth error components, the restriction (and hence the interpolation operator) must be constructed in a way that allows the smooth error components to be resolved on the coarser level. One way to construct an interpolation operator is smoothed aggregation (SA). SA was introduced by Vaněk [27] and can be considered as modification on the idea of restriction by aggregation. Following [16], an aggregation method consists of decomposing $\Omega^h = \{1, \ldots, n\}$ into disjoint sets $\mathcal{A}_i$, based on the matrix entries of A_h. A tentative restriction operator $\tilde{P}_h^H$ is then constructed by

$$(\tilde{P}_h^H)_{ij} = \begin{cases} 1 & \text{if } \omega_j \in \mathcal{A}_i \\ 0 & \text{if } \omega_j \notin \mathcal{A}_i. \end{cases} \tag{3.12}$$

In SA, the tentative restriction operator is then multiplied with a prolongator smoother M_H, typically weighted Jacobi,

$$M_H = I - \omega D_h^{-1} A_h, \tag{3.13}$$

where D_h is the diagonal of A_h, and ω a weight parameter. This defines the operator P_h^H from Algorithm 3 as

$$P_h^H = M_H \tilde{P}_h^H. \tag{3.14}$$

3.2.4 Choosing aggregates for anisotropic problems

As mentioned above, the construction of the restriction operator must ensure that smooth error components are resolved on the coarser level. For anisotropic problems, this typically requires coarsening in the direction of strong connections of the underlying PDE [25]. This can be illustrated by considering the simple anisotropic equation

$$u_{xx} + \varepsilon u_{yy} = 0, \tag{3.15}$$

discretized with a standard five-point stencil with step size h, illustrated in Figure 3.1. If $0 < \varepsilon \ll 1$, the values of u are very weakly connected in the y-direction. Heuristically, coarsening in this direction and applying a simple relaxation scheme

can be viewed as averaging values that are nearly uncorrelated, and hence, good convergence cannot be expected.

It should be noted that this argument can be equally applied to the case where the step size h is different in the x- and y-directions. To see this, it suffices to note that one only need to reinterpret ε in Figure 3.1 as the square of the aspect ratio, i.e.

$$\varepsilon = \left(\frac{h}{h_y}\right)^2, \tag{3.16}$$

where h and h_y are the step sizes in the x- and y-directions, respectively.

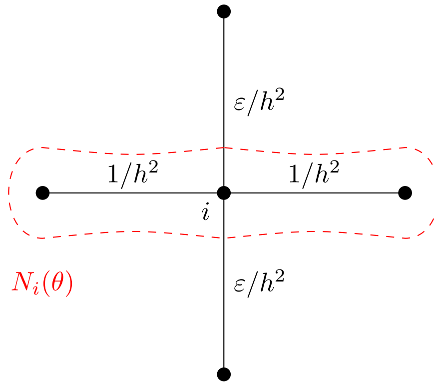


Figure 3.1: $N_i(\theta)$ for the five-point discretization of (3.15)

The standard way of creating aggregates for anisotropic problems was introduced by Vaněk et. al. [26]. Motivated by the requirement to coarsen only in the direction of strong connection, a strongly-coupled neighborhood of $i \in \Omega^h$ is defined as

$$N_i(\theta) = \left\{ j \in \Omega^h : |a_{ij}^h| \geq \theta \sqrt{|a_{ii}^h a_{jj}^h|} \right\}, \tag{3.17}$$

where $0 < \theta < 1$ is a user-specified tolerance [26]. This also suggests defining to degrees of freedom, say $i,\ j \in \Omega^h$ to be strongly coupled if

$$|a_{ij}^h| \geq \theta \sqrt{|a_{ii}^h a_{jj}^h|}. \tag{3.18}$$

To illustrate the idea of a strongly-coupled neighborhood, Figure 3.1 shows the five-point stencil for the discretization of (3.15) with step-size h at the ith degree of freedom. The edge weights correspond to off-diagonal matrix entries. The red dashed line shows $N_i(\theta)$ for θ sufficiently large, in particular

$$\frac{\varepsilon}{2(1+\varepsilon)} < \theta \leq \frac{1}{2(1+\varepsilon)}. \tag{3.19}$$

Ideally, the aggregates $\mathcal{A}_i$ should consist of a disjoint covering of Ω^h. However, this is in practice not always achievable. Instead, Algorithm 4 [26] presents a greedy algorithm consisting of three phases. The first phase greedily creates aggregates from strongly-coupled neighborhoods that are disjoint from the ones already added. The second adds the remaing degrees of freedom to aggregates if they are strongly coupled to any of the degrees of freedom in the aggregates created in the first phase. If any degrees of freedom are left after the second phase, they are added to a new aggregate together with any unanggregated degrees of freedom to which they are strongly coupled.

Algorithm 4 Construction of aggregates

1: Let $R = \Omega^h$ **and** $j = 0$
2: **for** $i \in R$ **do**
3:    **if** $N_i(\theta) \subset R$ **then**
4:       $\mathcal{A}_i \leftarrow N_i(\theta)$
5:       $R \leftarrow R \setminus \mathcal{A}_i$
6:       $j \leftarrow j + 1$
7:    **end if**
8: **end for**
9:
10: **for** $k = 0$ **to** j **do**
11:    $\tilde{\mathcal{A}}_k \leftarrow \mathcal{A}_k$ Copy
12:    **for all** i such that $N_i(\theta) \bigcap \tilde{\mathcal{A}}_k) \neq \emptyset$ **do**
13:       $\mathcal{A}_i \leftarrow \mathcal{A}_i \bigcup \{i\}$
14:       $R \leftarrow R \setminus \{i\}$
15:    **end for**
16: **end for**
17:
18: **while** $R \neq \emptyset$ **do**
19:    Pick $i \in R$
20:    $\mathcal{A}_{j+1} \leftarrow R \bigcap N_i(\theta)$
21:    $R \leftarrow R \setminus \mathcal{A}_{j+1}$
22:    $j \leftarrow j + 1$
23: **end while**

Using anisotropic aggregation according to Algorithm 4 might, however, not be sufficient for achieving an efficient V-cycle. This can be explained [10] by considering the way the coarse level Galerkin products are computed when coarsening is performed by smoothed aggregation. By substituting Equation (3.14) and Equa-

tion (3.13) into Equation (3.9), we get,

$$A_H = \tilde{P}^T(I - D_h^{-1}A_h)A_h(I - D_h^{-1}A_h)\tilde{P}. \tag{3.20}$$

If one ignores the scaling by D_h^{-1}, which does not alter the sparsity pattern of A_H, this can be written as

$$A_H = \tilde{P}^T q_3(A_h)\tilde{P}, \tag{3.21}$$

where $q_3(A_h)$ is a third degree polynomial in A_h. Hence, even with anisotropic aggregation, any two aggregates i and j, for which there exists a path of length three or less in the matrix graph of A_h will result in a non zero entry a_{ij} in A_H, independent of whether the couplings on this path are weak or strong with regard to θ. Consequently, weak couplings will lead to non zeros in A_H even though they are disregarded in the creation of the tentative prolongator. This may lead to severe fill-in in the coarse level matrices of the V-cycle.

One way to resolve this problem is proposed by Vaněk et. al. [26]. They suggest modifying the prolongator smoother defined in Equation (3.13) into

$$M_H = I - \omega D_h^{-1}A_h^F, \tag{3.22}$$

where A_h^F is defined as

$$a_{ij}^F = \left\{ \begin{array}{ll} a_{ij} & \text{if } j \in N_i(\theta) \\ 0 & \text{if } j \notin N_i(\theta) \end{array} \right\} \text{if } i \neq j, \;\; a_{ii}^F = a_{ii} - \sum_{j=1, j\neq i}^{n_l} a_{ij} - a_{ij}^F. \tag{3.23}$$

A_h^F is refered to as the filtered matrix. The filtered matrix can be viewed of the matrix consisting only of strong couplings, but with the values of weak couplings added to their corresponding diagonals. This is done to preserve the row sum from the unfiltered matrix, and is required to guarantee convergence [10].

3.2.5 AMG as preconditioner

Although AMG was originally introduced as a standalone solver, it can also be used as a preconditioner for CG or other Krylov subspace methods [25]. When used as a preconditioner, the coarsening may be done more aggressively than when used as a standalone server, making a cycle cheaper to run.

Chapter 4

Heterogeneous Computing

GPUs - graphics processing units - were developed as devices specialized for rendering graphics. To meet the specific needs of graphics rendering, GPUs have an architecture that differs from conventional CPUs, focusing on massive on-chip parallelism and high memory bandwidth. Although GPUs were originally developed for graphics processing, their potential in high-performance and scientific computing has long been acknowledged, and they are frequently used to speed up computations in other fields than graphics [11]. Amongst these are numerical linear algebra, and porting linear algebra applications to GPUs is an active field of research. In the following chapter, we will give a brief description of GPU computing and present the background for CUDA and CUSP, which will enable us to use GPUs for performing the numerical methods described in Chapter 3.

4.1 GPU architecture

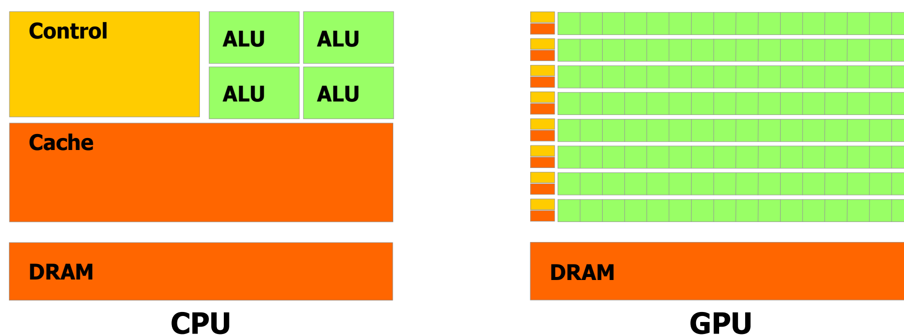


Figure 4.1: Schematic illustration of a CPU and GPU. Picture taken from [20].

Figure 4.1 shows a simplified model that demonstrates the most important

architectural differences between CPUs and GPUs. Although the details vary between different chips, the figure shows how a CPU is built for high single-core performance with a few floating-point units, big caches, and a large portion dedicated to flow control. GPUs on the other side, dedicate more transistors to floating-point units, while deemphasizing flow control and sophisticated caches. As a result, a GPU can perform thousands of structured computations in parallel.

GPUs are built around streaming multiprocessors (SMs). A SM schedules and executes computations in parallel along several lanes of execution, where each lane does the same computation. This type of parallelism is typically known as SIMD (Single Instruction Multiple Data). Although it is possible to perform SIMD operations on CPUs as well, the SIMD width of the chip, the number of possible concurrent threads of computation, is typically much lower, meaning that the GPU is able to perform a much higher number of computations in parallel. Furthermore, exploiting SIMD on CPUs can be cumbersome as it typically involves writing code on assembly level using libraries such as SSE, and because the programmer has to structure the computations after the SIMD width of the chip [8]. This is not the case for GPUs as the CUDA programming model offers a scalable thread abstraction to the SIMD lanes.

An important aspect related to the architecture of GPUs is that because the GPU is a separate chip with its own memory space, any computation done by the GPU must be preceded with an initial data transfer from the CPU's memory. This memory transfer is relatively slow, and should be minimized. As a result, there is a trade-off between the possible speed-up of doing a computation on the GPU and the time spent communicating between the CPU and GPU.

4.2 CUDA

CUDA [20] was released in 2006 by Nvidia. CUDA is a parallel computing architecture designed to enable writing general-purpose applications to Nvidia GPUs. Because of the architectural differences between GPUs and CPUs, CUDA provides a programming model designed to take advantage of the strengths of GPUs, centered around a thread abstraction to the SIMD lanes of the GPU architecture that makes CUDA programs scalable.

CUDA programs can be implemented in two ways. The first is a low-level driver API that makes it possible to write programs on assembly level. On a higher level, it is possible to implement CUDA programs with CUDA C, which consists of a small set of extensions to C. We will use CUDA C, because it is easier to program, and because CUSP, which we will use for our linear solver, is implemented with it.

4.2.1 Programming model

The CUDA programming model assumes that computations are done in a heterogenous setting. Specifically, it assumes a host, a CPU, running a sequential C program. The GPU (or GPUs) is available to the host as a device, on which computations can be done in parallel by invoking calls to parallel functions, or kernels. The host and device (or devices) are assumed to have separate memory, meaning that data must be transferred physically between them. This is illustrated in Figure 4.2.

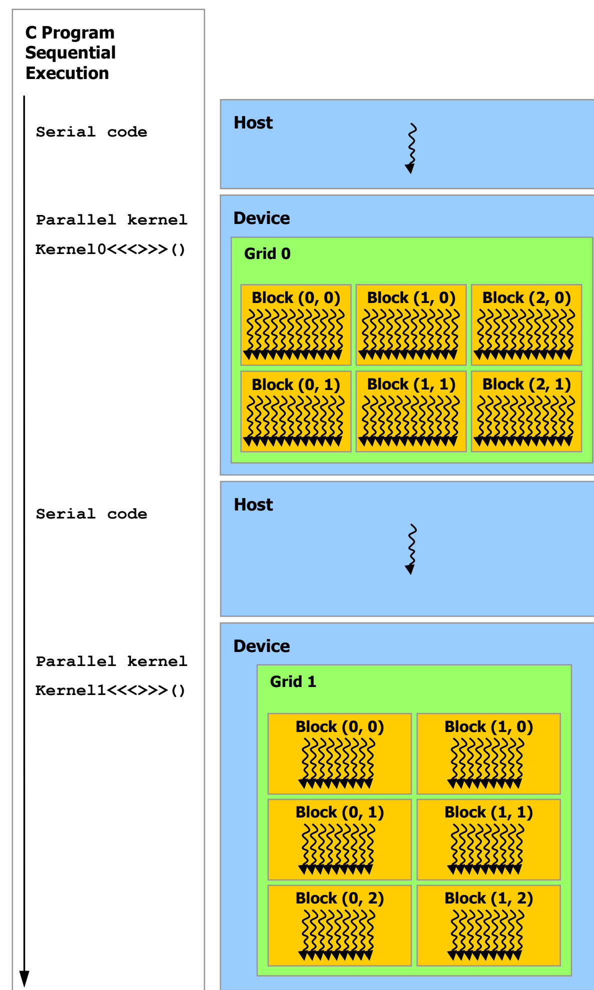


Figure 4.2: Heterogenous computing with CUDA. Picture taken from [20].

The CUDA programming model is based on the concept of computational kernels being executed by a hierarchy of threads. A kernel is basically a function that is executed in parallel, while a thread is the unit of parallelism. A kernel

is conceptually launched on a grid, divided into thread blocks, where the threads are laid out. This layout can be done in one or two dimensions. A 2-dimensional layout is demonstrated in Figure 4.3. The threads then execute the kernel in parallel, independently from the other threads. Synchronization can be achieved in form of barriers, either between the threads in a thread block, or through a global barrier for all threads in a kernel invocation. There is an implicit barrier between kernels.

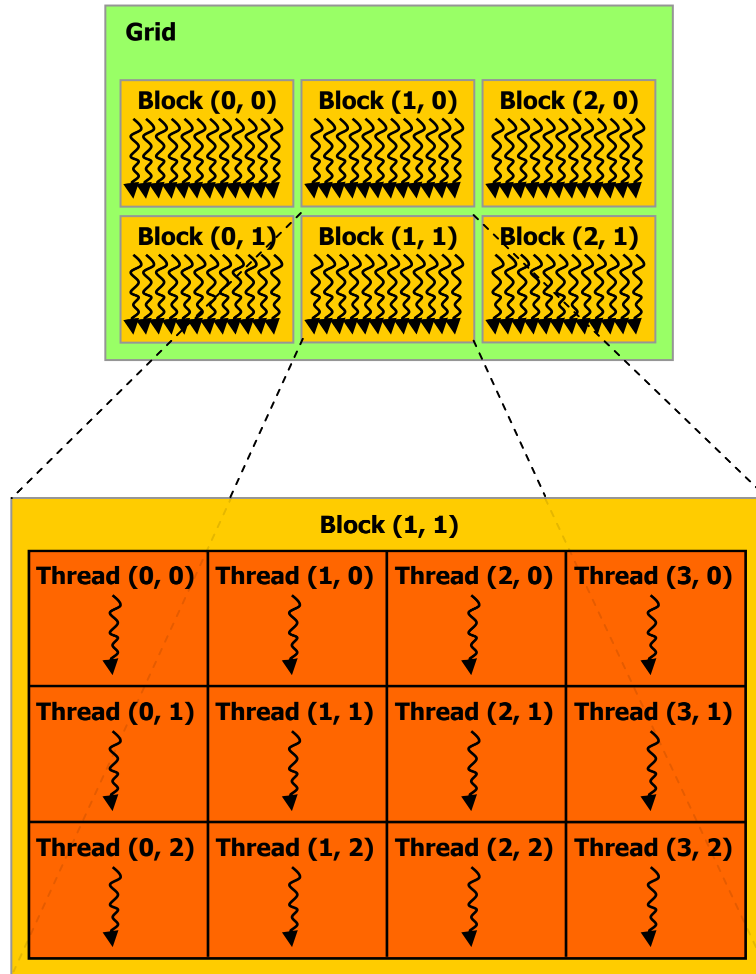


Figure 4.3: Thread block hierarchy of CUDA. Picture taken from [20].

As mentioned above, the device is assumed to have separate memory from the host. The device memory model is shown schematically in Figure 4.4 from [20]. Every thread has access to a shared global memory space, which is persistent between kernel calls. In addition, a thread block has a shared memory space to which all threads in the blocks has access, and each thread has its own memory

space. This makes it possible for threads within a thread block to cooperate by sharing data. The shared memory of a thread block is also assumed to be faster than the global device memory, which gives an example of the fact that to write efficient code, it is important to keep details in the memory hierarchy in mind.

CUDA is designed to be functionally forgiving in the sense that one does not have to think about how the programming model is implemented on the hardware in order to get running code. However, the details of the hardware implementation is important for the code performance. When a kernel is called, the programming model is mapped to the physical GPU by assigning each thread block to one streaming multiprocessor. Each multiprocessor will then schedule and execute threads in groups, so called warps, that will run until all threads of the warp complete the kernel. The GPU achieves high performance when the latency of memory transactions can hidden by always letting a warp do computation. To achieve this, it important to fit several thread blocks onto the same SM, so that whenever a warp is waiting for a memory transaction, another can start executing. This means that the programmer has to think about issues like register count and shared memory usage, as this is limited for each SM. Another detail is that because each warp executes sequentially, a kernel with a divergent execution path within a warp will be slower than a kernel where branch divergence happens between warps. More details can be found in [19, 20].

4.3 Linear algebra on GPUs

In [14], Keutzer and Mattson argue that most problems in computing will follow certain patterns. These patterns are classified into a pattern language (OPL), which is intended as a tool for writing efficient parallel software. According to Keutzer and Mattson, design problems will on the highest level exhibit certain structural and computational patterns, which will influence design choices such as strategies for algorithms, implementation and parallel execution. Sparse linear algebra is one of the 13 computational patterns recognized. The concepts of Keutzer and Mattson can help motivate why GPUs are appropriate when doing sparse linear algebra, and in particular Krylov subspace methods such as CG. As a lot of the computation typically consists of performing the same type of operation on independent sets of the data, such as dot products and SpMVs, a data-parallel strategy is often in place. Data-parallel algorithms can often be efficiently handled using the SIMD execution pattern [14].

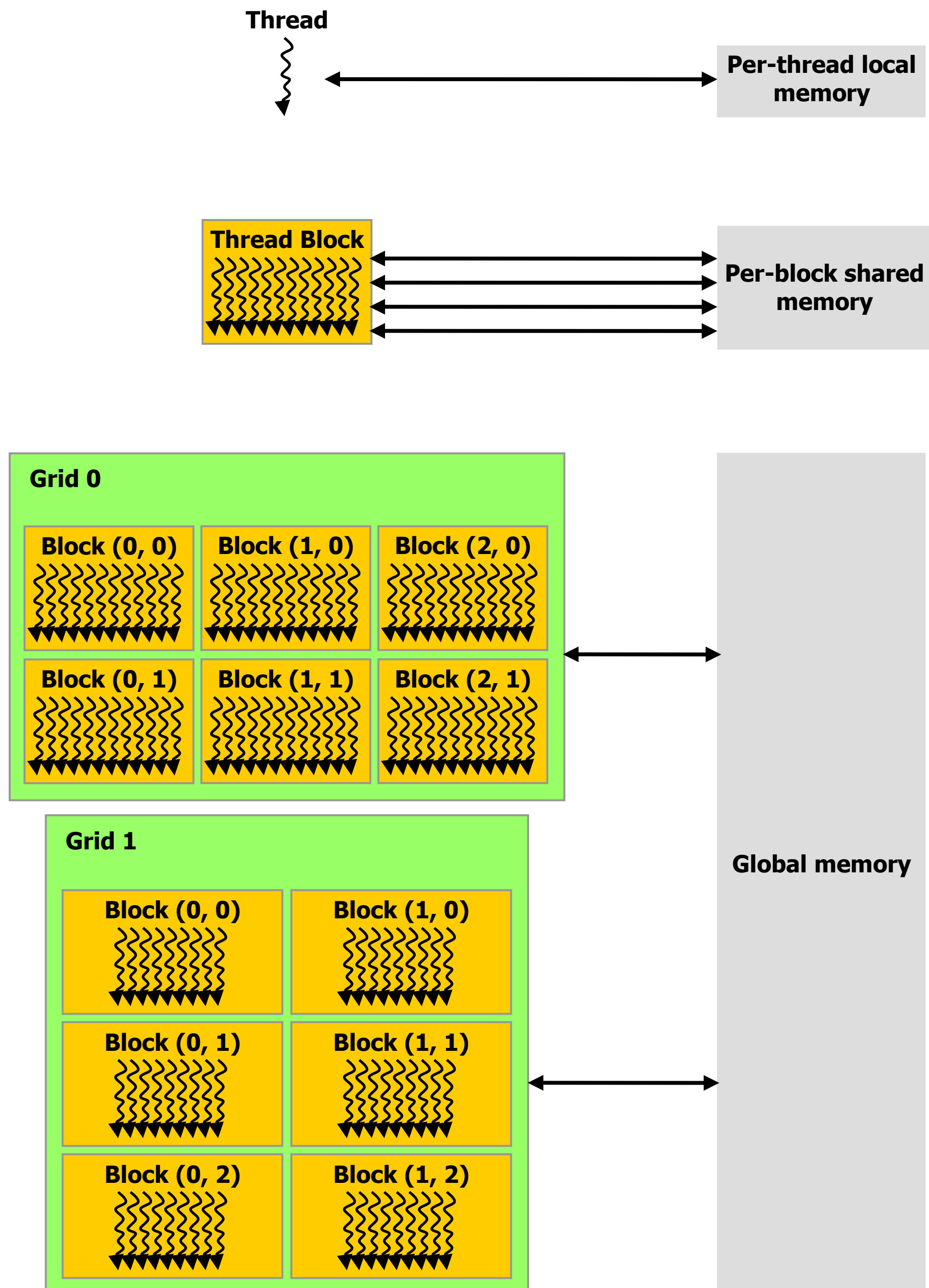


Figure 4.4: CUDA memory hierarchy. Picture taken from [20].

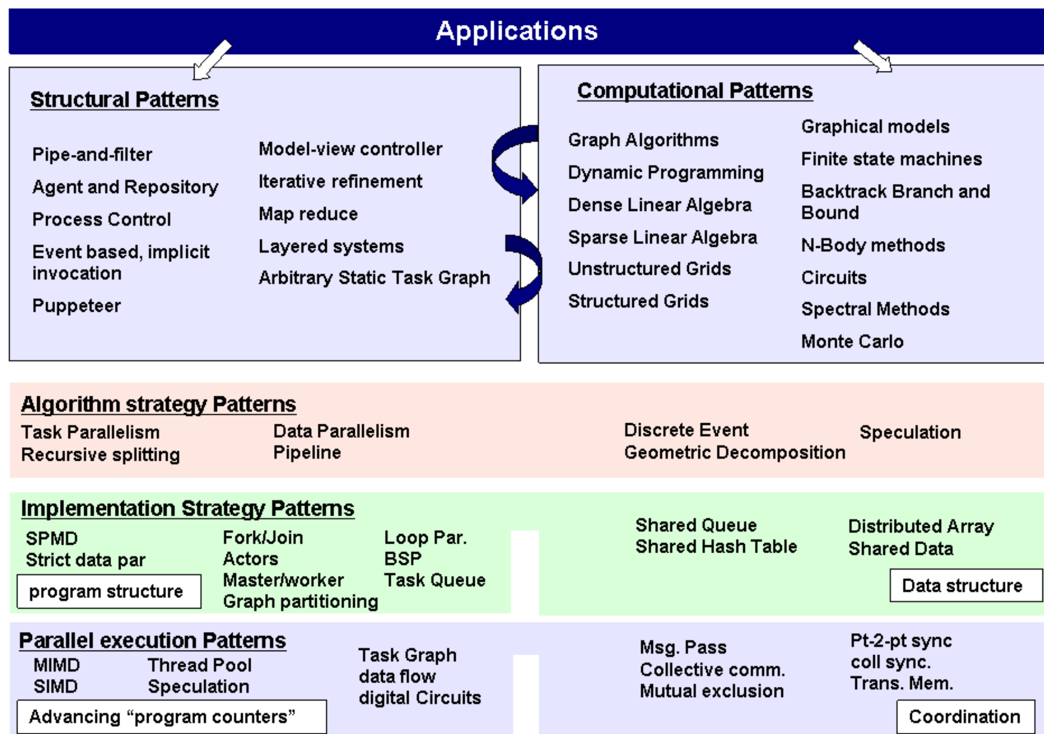


Figure 4.5: Structure of OPL. Picture taken from [14].

4.3.1 Sparse matrix-vector products

The sparse matrix-vector product (SpMV) is an important component in iterative methods for sparse linear systems. Both algorithms presented in Chapter 3, CG and AMG, are examples of this; both an iteration of CG and a V-cycle requires SpMVs. Furthermore, the other operations are either dot-products or axpys ($\alpha x + y$, where α is a scalar and x and y are vectors). These are both $O(n)$-operations that can be performed efficiently using BLAS [6]. As consequence, the key to good performance of both the inner loop in CG and the V-cycle, is the implementation of the SpMV. This is typical for both Krylov subspace methods and multigrid methods, which CG and AMG are examples of.

Efficient implementations of the SpMV kernel relies on exploiting knowledge of the matrix. In some cases, it is possible to write highly specialized kernels that are optimized for a particular application. One example is using an explicit formula for the matrix entries if the matrix comes from a known discretization scheme. The obvious drawback of this approach is that the kernel will be application specific. A more general approach is to use a sparse storage format to represent the matrix. A sparse matrix format is a data structure for storing the non zeros (and possibly some zeros) with corresponding indices of a sparse matrix. This can be done both

implicitly and explicitly.

The non zero structure of the matrix is the determining factor in choosing an appropriate storage format represent a matrix [4]. A good matrix format should fit the non zero structure of the matrix. In this context, it is also important to consider the architecture of the platform the SpMVs are computed on, as it is desirable that SpMV kernels are memory bound [4], that is the limiting factor for the running time is the time spent transferring data from slow memory. For parallel platforms such as GPUs, it is also important to choose a data structure where the work can be distributed evenly among processors or threads. For GPUs in particular, it is essential to achieve fine grained parallelism.

Given a $n \times n$-matrix A with nnz non-zero entries, the simplest storage format is the coordinate format (COO). In COO, A is represented using the three arrays, `values`, `row_indices`, and `column_indices`, all of length nnz.

A commonly used matrix format is compressed sparse row format (CSR). It uses three arrays to store the matrix. Each of the values of the non-zero matrix entries is stored in an array of length nnz, say `values`, in row-wise order. The column index of each entry is stored in the array `column_indices`, in the same order as the values in `values`. The row indices are stored implicitly in the array `row_offsets`, which has length $n + 1$. The non-zero entries of row i are stored at index `row_offsets[i]` through `row_offsets[i+1]` $- 1$ in `values`. The compressed sparse column format (CSC) is similar to CSR, but interchanges the roles of the two arrays for representing the indices of non-zero entries, using row index and column offset arrays.

The ELLPACK/ITPACK format (ELL) is a format more specialized for efficient execution of the SpMV kernel, especially for target architecture such as GPUs. It stores the A using two $n \times k$-arrays, `values` and `column_indices`, where k is the maximal number of non-zeros in any of the rows of A. The non-zeros values of row i are stored in order in row i of `values`. The corresponding column indices are stored in `column_indices`. If row i has less than k non-zeros, the remaining entries are zero-padded.

The ELL format is clearly inappropriate if the maximal number of non-zeros in a row in A is much greater than the average number of non-zeros per row, as this will mean most of the values stored are zero-padding. The hybrid format (HYB) aims to make use of the benefits of ELL, while avoiding this situation by using the flexibility of the COO format. This is done by letting k be the "typical" number of non-zeros per row, and storing the k first non-zeros of any row using ELL. The remaining non-zeros are stored using COO. In some cases an optimal value for k can be chosen using knowledge of the matrix, however, it must in general be determined using heuristics.

In [4], Bell and Garland gives a revision of storage formats for sparse linear

algebra, with detailed experiments exhibiting the performance differences of the formats in various test cases. For structured and semistructured matrices, they show that ELL and the DIA format, which we will not consider because of its memory consumption, generally is faster than COO and CSR. For unstructured grids, they conclude that their implementation of HYB is the fastest format for a large range of matrices. We refer to [4] for more details.

4.3.2 CUSP

CUSP [5] is a sparse linear algebra library for Nvidia GPUs implemented in CUDA. It provides seven iterative solvers based on Krylov subspace methods, one of which is CG. In addition, it offers AMG with smoothed aggregation, approximate inverse and diagonal preconditioners to speed up convergence. To support this, CUSP comes with data structures for storing vectors and sparse matrices, with functionality for converting between different sparse formats and transferring data between the CPU and GPU. Furthermore, as SpMVs are at the core of both Krylov subspace methods and the solve phase of AMG, CUSP has implementations of the matrix formats described in Section 4.3.1, using techniques laid out in [4]. CUSP also supports other sparse linear algebra functionality, such as a BLAS implementation and the possibility to write black-box SpMV to implement matrix-free versions of the iterative solvers.

CUSP makes extensive use of the template library THRUST [13], both for its data structures and algorithms. THRUST is a part of the official CUDA toolkit released by Nvidia. It offers data structures on both host and device and algorithms such as sorting, reductions, and copying. The interface is aimed at being similar to functionality found in the C++ Standard Template Library's `vector` class and the Boost libraries. This makes parallel data structures and basic algorithms available for the user in a way that abstracts away implementation details. The data structures for vectors and matrices in CUSP stores their values using THRUST's `host_vector` and `device_vector` classes. Hence, we can access these using THRUST in addition to the algorithms implemented in CUSP.

One of the most attractive features of CUSP for our purposes is it's parallel smoothed aggregation-based preconditioner. In the solve phase, the V-cycle is inherently sequential, however, pre- and post-smoothing, and restriction and interpolation at each level are done using SpMVs, which, as discussed, can be done efficiently parallel. The setup stage is also sequential in the sense that the levels in the V-cycle must be constructed in order, as aggregation on level k depends on the matrix on level $k-1$. However, the aggregation procedure described in Section 3.2.4 does not - with its greedy, sequential approach - exhibit the fine grained parallelism required for an efficient GPU implementation [3]. It should be pointed out that Algorithm 4 can be parallelized by splitting up the domain, Ω^h, in

smaller subregions, running the algorithm on each subregion in parallel, and using a sequential algorithm to resolve the boundaries between subregions. However, this creates a coarse-grained form of parallelism, and hence it is not suitable for efficient GPU implementation. Instead, CUSP uses an aggregation strategy based on distance-2 maximal independent sets (MIS(2)). A MIS(2) is, loosely speaking, a subset of a graph, such as Ω^h, where any two nodes are at least at a distance 2 apart in the graph, and no further nodes can be added to the subset without violating this property. In [3], Bell. et al. describes how this can be used to create aggregates that mimics the result from a sequential algorithm, using THRUST.

CUSP's smoothed aggregation implementation supports anisotropic aggregation using a filtering parameter, as described in Section 3.2.4. However, θ cannot vary between levels in the setup phase. Furthermore, using the filtered matrix in the prolongator smoother, (3.22), is not supported. For more details regarding the implementation of CUSP's preconditioner, we refer to [3].

Chapter 5

Implementation

With the theoretical background presented in Chapter 2, 3, and 4, we will now describe the implementation of our linear solver. We will first, however, describe MEX-interfaces, which allows us to integrate CUSP code, written in CUDA C, and MRST, written in Matlab.

5.1 External interfaces in Matlab

Clearly, some parts of a typical MRST program, such as some of the grid processing routines and the linear solver, will be quite computationally costly and dominate the running time of the entire program. Although these parts may be easy to implement in a high level language such as Matlab, it might be difficult to exploit lower level details, such as the structure of computation or memory management, which potentially could have given a significant speed-up.

This is obviously not a concern that is specific for MRST, but may occur in any Matlab application. However, with MEX-files Matlab provides an interface for implementing functions in C/C++ or Fortran, in order to address this concern. Using the MEX API, code written in either language can be called as built-in Matlab functions, making it possible to exploit existing software or speed up parts of Matlab code.

A C/C++ MEX-file is created by compiling a MEX source file using Matlab's `mex` compile program, which creates a file that is used like a regular Matlab built-in function. The interaction between the MEX-file and the Matlab environment is managed using the MEX function library. This consists of functions for communicating with the Matlab environment and operate on data. Matlab data are stored using the data type `mxArray`, which are created accessed and manipulated with MEX library functions.

Every MEX source file must have a gateway function. This is a function with

the signature

```
void mexFunction( int nlhs, mxArray
*plhs[], int nrhs, const mxArray *prhs[] );
```

and it is the interface between Matlab and the MEX-file. Whenever a MEX-file is called from Matlab, `mexFunction` is invoked. The input parameters `nrhs` and `nlhs` will be the number of input and output arguments in the Matlab function call, while each entry of `prhs` and `plhs` will be a pointer to a `mxArray` corresponding to the input and output arguments of the Matlab function. Using the MEX function library, one can get C pointers to the underlying data in order to do computations.

Figure 5.1 shows some of the basic components of a MEX program. The example code does a simple vector addition of two real-valued $m \times 1$-vectors.

5.2 General setup

As described in Section 2.3, MRST allows the user to provide a function handle to a linear solver for the Schur complement system, (2.39) to `solveIncompFlow`. Hence, by combining MEX-files with CUDA, it is possible to use GPUs to accelerate the computation.

MRST expects the solver function, say `solver`, passed to `solveIncompFlow` to have the calling sequence `x = solver(A,b)`. The arguments `solveIncompFlow` passes to `solver` will be the matrix and the right hand side vector in (2.39). Because this implies that the MEX-file will access the matrix directly, we will store the matrix in a sparse storage format, and not consider a matrix-free implementation.

As Matlab uses the CSC (Compressed Sparse Column) format for storing sparse matrices, we will initially have access to the matrix in this format. However, as long as the matrix is symmetric, the CSC and CSR (Compressed Sparse Row) formats are identical, if we only rename the row index array to the column index array, and vice versa. Since CUSP supports the CSR format and has functionality for format conversion, we can easily try different matrix storage formats for our solver letting CUSP do format conversions.

This suggests organizing the solver with a MEX-file which implicitly converts the matrix to CSR format, passing it, together with the right-hand-side-vector, to a computational function. This transfers the matrix to the GPU in a desired storage format using CUSP, and solves the linear system on the GPU. In order to compile this into a single MEX-file, we compile the MEX source code using the MEX compile program and the GPU code using NVCC into object files, and link them together with the MEX compile program.

```
#include <mex.h>

void vecAdd(mwSize m, double *x, double *y, double *z)
{
  int i;
  for (i = 0; i < m; i++)
    z[i] = x[i]+y[i];
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const ...
    mxArray *prhs[])
{
  mwSize m;
  double *x;
  double *y;
  double *z;

  m = mxGetM(prhs[0]);
  plhs[0] = mxCreateDoubleMatrix(m,1,mxREAL);

  z = mxGetPr(plhs[0]);
  x = mxGetPr(prhs[0]);
  y = mxGetPr(prhs[1]);

  vecAdd(m,x,y,z);

}
```

Figure 5.1: Vector addition MEX-file

5.3 Linear solver using CUSP

As described in Section 4.3.2, CUSP provides iterative Krylov solvers with preconditioners. We will use CUSP's conjugate gradients with smoothed aggregation for our linear solvers. Since, as mentioned above, we can use CUSP to transfer the matrix onto the GPU in different storage formats, we will test different versions storing the matrix in the CSR, HYB, and ELL formats. Because we expect COO to be slow, we will not consider this format.

Since the pressure equations, (2.9) and (2.10), may have strongly anisotropic coefficients in realistic scenarios, we will consider constructing aggregates for the corresponding discretization matrix as discussed in Section 3.2.4. As described in Section 4.3.2, this is supported in CUSP, and done by specifying a value for the

filtering parameter θ in (3.18). The default value for θ is 0.

Based on experience from their numerical expirements, Vaněk et. al. [26] suggest

$$\theta = 0.08 \left(\frac{1}{2}\right)^{l-1}, \tag{5.1}$$

where $l = 1, \dots, L$ are the levels of the AMG V-cycle, but does not give any guidance on how to systematically choose a good value. As the value in (5.1) is experimentally determined, we expect it to be dependent on the specifics of the problem it is chosen for, and not necessarily the best choice for our application. It does however suggest that θ might be changed between levels, relaxing the condition for what is considered a strong connection on coarser levels. CUSP does not support changing θ between levels as in (5.1), however, it is straight forward to change the source code to allow dividing θ by 2 between levels. We will consider both constant θ and dividing θ by 2 between levels in our numerical experiments. We will not consider other of the - infinite - possibilities there are in varying θ between levels, partly to contain complexity, partly as we do not have any theoretical guidance to any other specific choices.

Because CUSP does not support using the filtered matrix for the prolongator smoother, we will have to modify CUSP in order to test the effect of this technique. However, we can exploit the way CUSP creates anisotropic aggregates to this end. Given a fine level matrix A_h and a filtering parameter θ, aggregation in CUSP is implemented by creating a strength matrix C, given by

$$c_{ij} = \begin{cases} a_{ij} & \text{if } j \in N_i(\theta) \\ 0 & \text{if } j \notin N_i(\theta). \end{cases} \tag{5.2}$$

As C is the matrix whose entries are precisely the strong couplings in A_h, aggregation using C without a filtering parameter will produce the same set of aggregates as anisotropic aggregation with A_h and θ. C is constructed with fine-grained parallelism using THRUST algorithms [3]. Furthermore, this means detection of strong connection is separated from the aggregation procedure.

Since $i \in N_i(\theta)$ by construction, the only difference between C and A_h^F is that matrix entries in A_h corresponding to weak couplings are not added to the diagonal of C as for A_h^F. In particular, the two matrices have the same sparsity pattern. For simplicity of implementation, we have used C instead of A_h^F for the prolongator smoother. Hence, instead of (3.22), we have used

$$M_H = I - \omega D_h^{-1} C. \tag{5.3}$$

As mentioned in Section 3.2.4, adding weak couplings to the diagonal of A_h^F is necessary for guaranteeing convergence of AMG. Hence, there is a concern that

this will make the convergence properties of the V-cycle deteriorate. As we use AMG as a preconditioner for CG, this does not mean that convergence of the solver is not guaranteed, because the convergence analysis of CG is unaffected. However, recalling from Section 3.1.2 and Section 3.1.3 that the convergence rate depends of $\kappa(M^{-1}A)$, where M^{-1} is the preconditioner, it may affect the convergence rate.

Chapter 6

Numerical Results

Having described our implementation in Chapter 5, it is time to test our linear solver. In this chapter, we present and discuss the numerical experiments we have undertaken.

6.1 Test setup

It is important to decide which parameters to consider for testing and what measures to use. Ultimately, the most important questions are whether the produced answer is correct, and how long time it takes to produce. Hence, our measures fall into to categories; performance and correctness.

6.1.1 Performance

The primary measure for performance will, as mentioned above, be the run-time of the solvers. This is a natural consequence of our goal of providing a black-box solver for MRST. It also means that we will use secondary performance measures mostly in order to analyze our results and suggests improvements.

One secondary measure is the number of CG iterations required to reach convergence. This can give information about the quality of the AMG preconditioner, as good convergence properties of the preconditioner leads to fewer iterations. Another quantity related to the preconditioner is the operator complexity, defined as

$$c = \frac{\sum_{i=2}^{L} \mathrm{nnz}(A_i)}{\mathrm{nnz}(A_1)}, \tag{6.1}$$

where $A_1, \ldots, A_L$ are the matrices on the L multigrid levels, and $\mathrm{nnz}(A_i)$ is the number of non-zeros in the matrix on level i. As we recall from Algorithm 3, both the pre- and post-smoothing steps at each level in a V-cycle requires a SpMV.

Since the run-time of SpMVs are dependent on the number of non-zeros of the matrix involved in the product, a higher operator complexity will results in slower V-cycles.

6.1.2 Correctness

As both the GPU solvers and AGMG are based on iterative methods, it is necessary provide a stopping criteria for when to finish iteration. Choosing a stopping criteria is essentially problem specific, where there is a trade off between getting accurate enough solutions and speed. One of the most common ways is to specify the stopping criteria as a tolerance for the relative reduction of the residual. In our test we will use a relative tolerance of 10^{-9}, which admittedly is somewhat arbitrary.

Since the solvers return the solution to (2.39), a natural primary measure for correctness is the relative norm of the error of the vector $\boldsymbol{\pi}$ computed. Computing the error requires the exact answer, we will use the answer from using Matlab's standard linear solver `mldivide` for this purpose. There is of course, strictly speaking, an error related even to the result from `mldivide`, as with any linear solver representing numbers with finite precision. However, as we in general cannot expect the relative norm of the error to be reduced any more than the relative norm of the residual, the error of `mldivide` will be several orders of magnitude less than the result from the iterative solvers. Consequently, when compared with the answer produced by the iterative solvers, the answer from `mldivide` will be indistinguishable from the true exact answer for practical purposes.

In addition to the norm of the error, an important secondary measure for the quality of the solution computed is whether the flux-field corresponding to $\boldsymbol{\pi}$ fulfills mass conservation. To measure this, we use the maximal cell divergence relative to the maximal face-flux, that is

$$\frac{\max_{\Omega_j \in \Omega} \left\{ \left| \sum_{i \in E_j} \boldsymbol{v_i} \right| \right\}}{\max_{\Omega_j \in \Omega} \{ \max_{k \in E_j} \{ |\boldsymbol{v_k}| \} \}}, \tag{6.2}$$

where Ω is the entire grid, the Ω_j's are the grid cells and E_j is the set of fluxes over the faces of Ω_j.

6.1.3 Reference solvers

For benchmarking, we use Matlab's matrix left division operator, `mldivide`, and the CPU-based AGMG [17, 18] solver as CPU references. Since `mldivide` is a direct solver, it is based on completely different mathematical principles compared to the combination of Krylov subspace and AMG techniques in our iterative method. As such, one can discuss the relevance of comparing the general purpose `mldivide`

with our special purpose GPU-accelerated solver. However, it is the default linear solver used by MRST, meaning that in practice, it is one of the alternatives most easily available. Furthermore, in a practical setting, it may also be illustrative to have a direct solver as a reference point when evaluating the performance of the iterative solvers. Additionally, we will use `mldivide` to verify correctness, as discussed above.

AGMG is an implementation of aggregation-based algebraic multigrid written in Fortran, which can be called from Matlab through a MEX-interface. AGMG can be used both as standalone algebraic multigrid or as conjugate gradients with algebraic multigrid as preconditioner. As AGMG implements ”pure” aggregation, it is not a mathematically equivalent CPU implementation of the smoothed aggregation algorithm which CUSP implements on the GPU.

6.1.4 Test platform

The tests have been performed on a computer with

- 8 GB RAM and 4 GB swap space
- Intel i7-2600 3.40 GHz CPU
- Nvidia GeForce GTX 570 GPU

The operating system is Ubuntu 10.04 LTS. We have been using Matlab R2011b, Nvidia Toolkit 4.1 with THRUST v1.5.1 and CUSP v0.3.0.

6.2 Test cases

We wish to test both realistic reservoir simulation scenarios and idealized cases in which we examine how the solvers behave when solving problems with specific properties. To this end, we have used the grid processing functionality in MRST to generate test cases.

6.2.1 Box

As our GPU solvers are intended to be used with MRST, we will be interested in testing how the physical phenomena that typically occur in reservoir simulation affects the behaviour of the solvers. In order to test this systematically, we will generate tests based on a box shaped grid. By changing the properties of the grid, such as the permeability properties of the rock or the aspect ratio of the grid cells, this allows for isolating the effects of a single phenomena, and then examine how the solvers perform on the resulting matrices.

As a starting point and reference, our simplest test case will be based on a cubic domain consisting of $n \times n \times n$ grid cells. The rock will be heterogeneous and isotropic, with a scalar permeability of 1 mD in every grid cell. To close the problem, we will specify pressure boundary conditions giving an exact solution of the pressure in (2.5) on the form

$$p(\boldsymbol{x}) = \boldsymbol{a} \cdot \boldsymbol{x} + b, \tag{6.3}$$

where $\boldsymbol{a}/|\boldsymbol{a}| = -(\sqrt{2}/2, \sqrt{2}/2, 0)^T$. We use MFD with `IP_SIMPLE` to obtain the discretized hybrid system, (2.38), and its corresponding Schur form, (2.39). Since MFD is exact for linear pressure fields, (6.3) should be the pressure computed by the GPU solvers. We will use these boundary conditions for all tests described in this section, except for the test involving wells. The pressure field for the isotropic test case is shown in Figure 6.1.

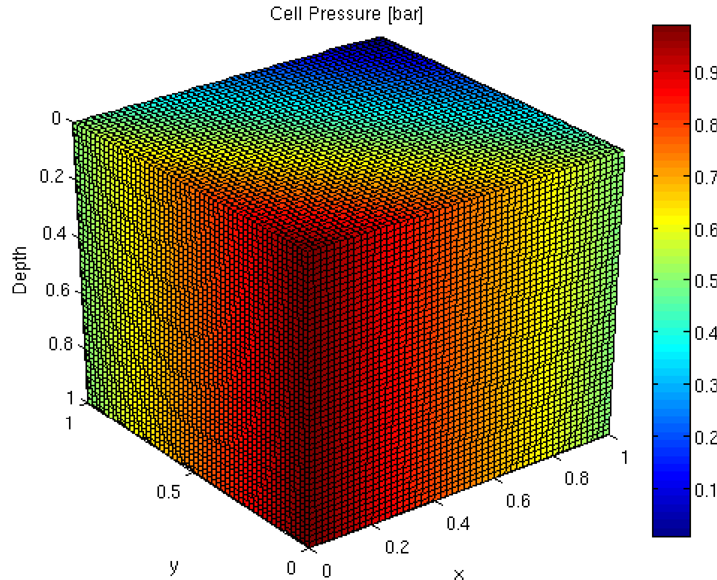


Figure 6.1: Visualization of test case 1.

As described in Section 2.1, the rock in real world reservoirs are described by a permeability tensor that may be anisotropic and discontinuous, with the numerical values varying several orders of magnitude. A related effect is the aspect ratio of the grid cells. Realistic reservoir grid cells may have physical dimensions which differ with several orders of magnitude in the different spatial directions. This may have an effect similar to anisotropic permeability. As described in Section 3.2.4,

anisotropies in the underlying PDE may need special treatment by AMG. We will discuss this in more detail in Section 6.3.2.

To test the effect of anisotropic permeability, we will use change the permeability tensor from the isotropic case into

$$\mathbf{K} = \begin{bmatrix} 10^4 & 0 & 0 \\ 0 & 10^4 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{6.4}$$

In order to generate a test case with heterogeneous permeability, we use MRST's function `logNormLayers` to generate the permeability field. This gives a scalar permeability field where the permeability for each grid cell is drawn from a log-normal probability distribution. For the test of cells with large aspect ratio, we use the homogeneous, isotropic permeability field from the first test case, but change the physical dimensions of each grid cell to $10^2 \times 10^1 \times 10^{-1}$ m. This gives an aspect ratio on an order of magnitude that may be present in a realistic reservoir simulation scenario.

Table 6.1: Overview of box test cases.

Number	Effect	Description
1	Isotropic	Cubic domain, isotropic and homogeneous permeability
2	Anisotropic	Permeability tensor $\mathbf{K} = \text{diag}(1, 1, 10^4)$ mD
3	Aspect ratio	Physical dimensions of grid cells $(10^4, 10^2, 1)^T$ m
4	Heterogeneous	Heterogeneous permeability generated by `logNormlayers`
5	Fault	Fault inserted in the y-direction at $x = 0.5$
6	Wells	Injection well and production well, no flow boundary conditions

In addition to the permeability of the rock and aspect ratio of the grid cells, we will be interested in testing how faults and wells affect the solvers. To test faults, we will insert a single fault in the z-direction into the first test case, parallel with the y-axis at $x = 0.5$. Wells are tested by adding an injection well and a production well as shown in Figure 6.2. As wells allow fluid to enter and exit the domain, we impose no flow boundary conditions in this case.

Whenever wells are present, extra couplings are introduced between well cells. Although it is in principle depending on the specifics of the problem, this generally results in the number of non zeros in the corresponding row of the linear system to be much higher than the average number of non zeros per row of the matrix. Since

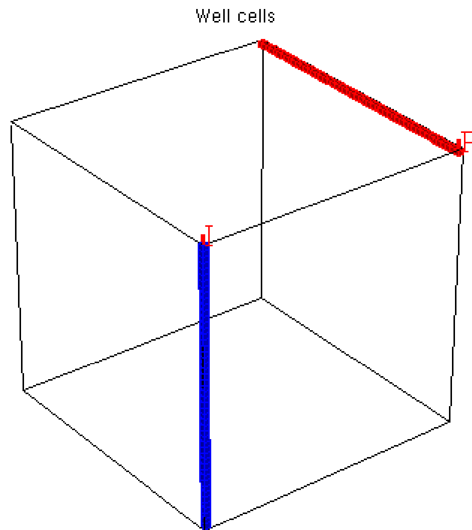


Figure 6.2: Visualization of well cells in test case 6.

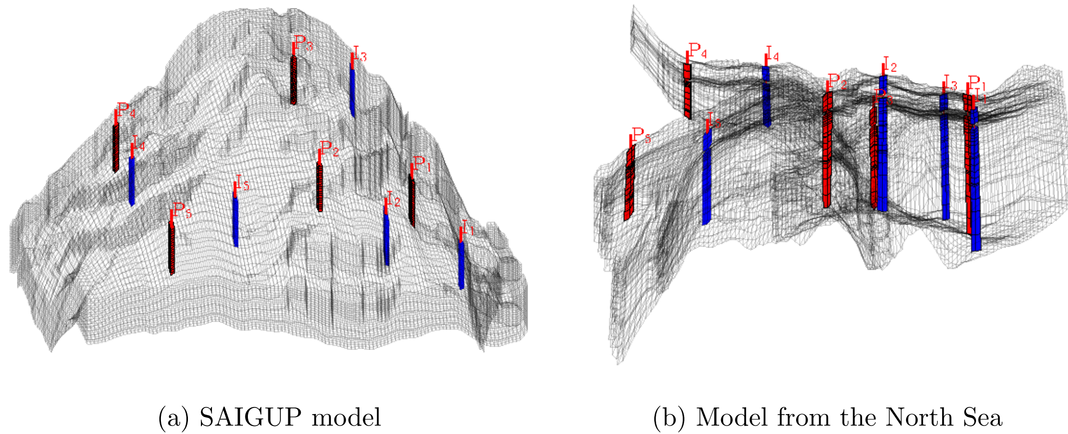
the ELL sparse matrix format (Section 4.3.1) stores the non zero values in a full, zero-padded $n \times k$-array, where n is the number of rows and k the maximal number of non zeros in a single row in the matrix, most of the memory used for storing the matrix will wasted storing zeros. Hence, the ELL format is inappropriate for scenarios involving wells, as it may well exhaust the GPU memory. Table 6.1 gives a summary of the test cases described in this section.

6.2.2 Realistic scenarios

In addition to the idealized test cases described in the previous section, we will also consider two cases using realistic geometry and physical properties. The first is a synthetic model created by the SAIGUP project [24], and contains faults, wells, and anisotropic and heterogeneous permeability. The other is based on a model with a grid using data from the North Sea. For this model, we add wells and generate anisotropic and heterogeneous permeability using the functionality of MRST. The permeability field consists of layers with values drawn from a lognormal distribution, ranging from 20 to 2000 mD. To make the test scenario anisotropic, the z-component of the permeability in all cells is scaled by a constant factor of 0.0001.

The SAIGUP model has approximately 78,000 cells, while the North Sea model has approximately 45,000 cells. However, using MRST, we refine the model by

splitting each cell into two in the z-direction, doubling the number of cells. This increases the aspect ratio of the grid cells. A visualization of the two models is shown in Figure 6.3.



(a) SAIGUP model

(b) Model from the North Sea

Figure 6.3: Visualization of grids in realistic models.

6.3 Results

6.3.1 A note on memory consumption and problem size

It is obvious that it is desirable to run the solvers on as large linear systems as possible. However, given the finite amount of CPU or GPU memory available, it is also clear that we cannot solve arbitrarily large systems. This is in the end bound by the hardware, however for an AMG-based solver, not only the dimensions or number of non-zeros of the matrix, but also the matrix structure affects the memory consumption of the solver, as it determines the coarse level matrices, which also need to be stored. Because CUSP require both the matrix itself and the preconditioner to be stored on the GPU throughout the solve, the GPU solvers can only handle problems small enough to hold these in memory.

In the context of reservoir simulation, it may be more natural to use the number of grid cells, rather than matrix dimensions and number of non-zeros, as parameter for the problem size. For the grid consisting of $n \times n \times n$ cells described above, we will use $n = 50$, meaning $125,000$ cells, for our primary benchmarks. This is close to the maximal n for which `mldivide` can solve all test cases with our hardware. Since the input parameters for the GPU solvers, in particular the filtering parameter θ, affects the memory consumption as well as the specifics of

the different problems do, the actual maximal size available for the GPU solver may vary. We have not been able to establish a priori tests for determining whether a particular scenario is to large to be solved on the GPU. Hence, we cannot conclude that the GPU solver is able to solve larger problems than `mldivide` in general on our hardware.

It should be pointed out that the above numbers obviously are limited by the specifics of the hardware used for testing. However, as the GPU we have used has 1280 MB memory as opposed to the 12 GB of CPU memory and swap space - disregarding resources used by the operating system - it is clear that `mldivide`, consumes more memory in absolute terms in the test cases we have considered.

6.3.2 Performance

Matrix format

Figure 6.4 shows a histogram of the running times for the run-times of the test cases described in Section 6.2.1 with $n = 50$, i.e. 125,000 cells. As in the previous section, we show the results for the GPU solver when using different matrix storage formats. The preconditioner is constructed using $\theta = 0$, and as such, it does not take features like anisotropy or aspect ratio of grid cells into account.

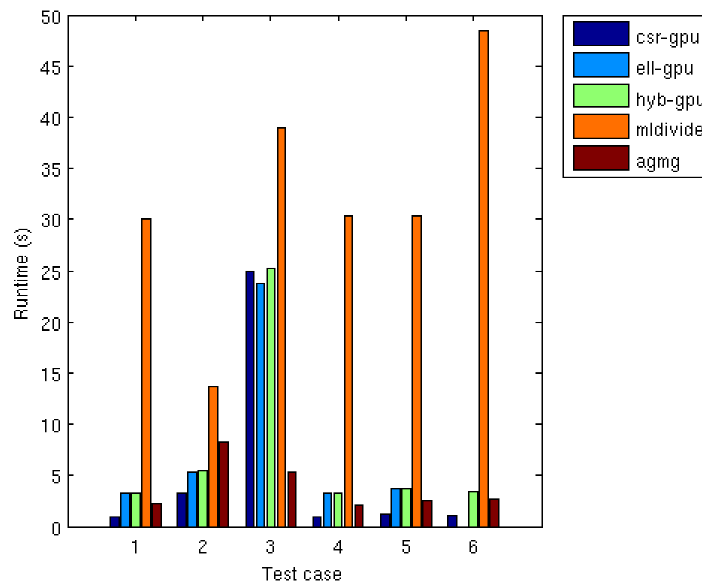


Figure 6.4: Run times for box test cases, $50 \times 50 \times 50$ cells.

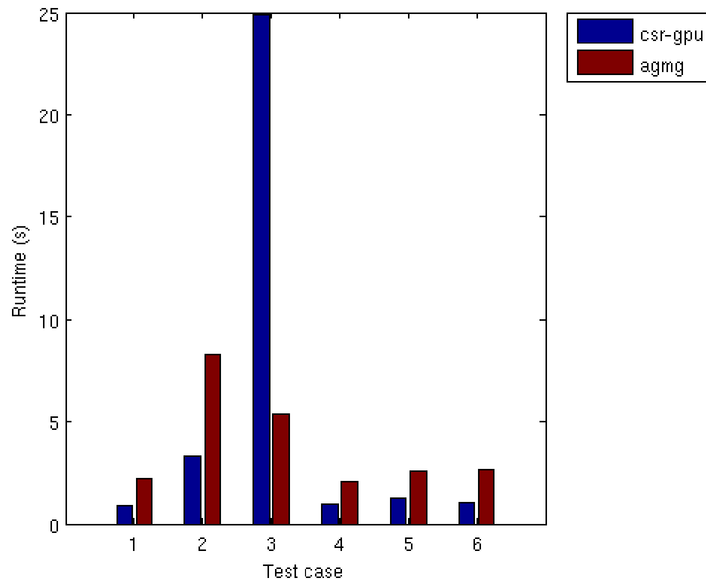


Figure 6.5: Run times for box test cases, GPU solver with CSR and AGMG.

One clear trend is that the solver using the CSR storage format is faster than both the ones using HYB and ELL in all cases except case 3. This illustrates the trade-off when choosing formats that shows better SpMV performance in tests [4], such as HYB and ELL, and the time it takes to convert the matrix into these formats from CSR, in which it is originally stored. The convergence properties of the preconditioner is obviously important in this context, as better convergence of the multigrid V-cycle yields fewer iterations of CG, meaning that the performance difference between storage formats becomes less important relative to the costs of doing the format conversion. As such, choosing the correct matrix format, depends on knowing, in advance, how successful the preconditioner will be in terms of convergence. For our purposes, the results for different storage formats in Figure 6.4 indicates that performing a format conversion might not be worthwhile. Although this is not clear for test case 3, it is, as we will see, possible to improve the convergence properties of the preconditioner using the techniques outlined in Section 3.2.4. This means fewer iterations will be needed to reach convergence, hence shifting the balance in favor of not performing a format conversion.

The realistic scenarios exhibits the same behaviour with respect to matrix format. Figure 6.6 compares the run time of the GPU solver when using CSR and HYB. As we can see, there is a significant increase in run time for the version using HYB, indicating that the cost of format conversion outweighs the potential

gains in the solve phase of the solver.

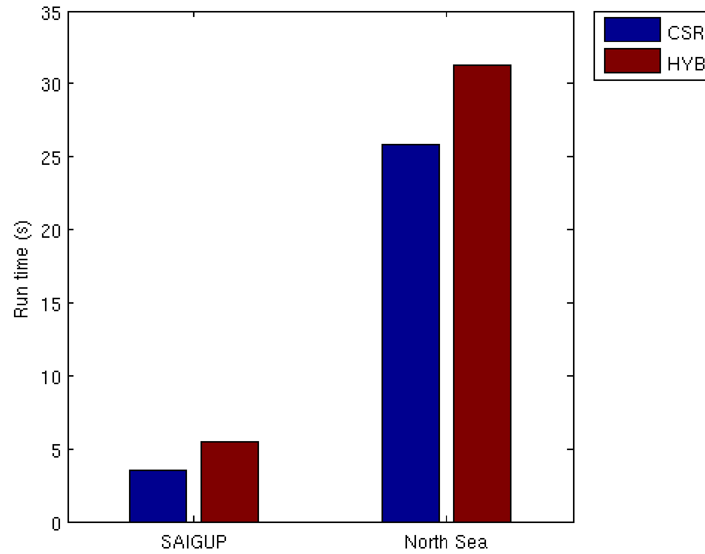


Figure 6.6: Run times for GPU solver with CSR and HYB matrix formats on realistic scenarios.

In the following, all results for the GPU solver will be using CSR to store the matrix.

Anisotropic aggregation

Figure 6.5 shows the results from Figure 6.4 for the GPU solver using CSR and AGMG. This gives a clearer presentation of how the GPU-solver compares with a CPU-based implementation of CG preconditioned with AMG. Except from test case 3, using the GPU gives a speedup of $2.1 - 2.6$x.

Test case 3 shows significantly worse performance for the GPU solvers, both when comparing to the other test cases, and with AGMG. This comes from the fact that the preconditioner is constructed using $\theta = 0$ to determine which couplings are strong. In other words, every non-zero matrix entry is considered as a strong coupling between nodes in the corresponding set of degrees of freedom.

As a consequence, the algorithm only considers the sparsity pattern of the matrix when choosing aggregates for the preconditioner. Hence, the preconditioner essentially does not take the aspect ratio of the grid cells into account. Since the sparsity pattern of the matrices in test case 1 and 3 are identical, it should not

be surprising that the performance is worse in case 3, where the aspect ratio is different from 1.

This argument also applies to test case 2, however the effect is much less pronounced. Heuristically, this can be viewed in light of Equation (3.16). This shows that, in the case of a five-point stencil, the isotropic stencil discretized with different step sizes in the spatial directions, is equivalent to an anisotropic equation, discretized with aspect ratio equal to 1. However, ε in the corresponding anisotropic equation - which for our purposes can be viewed as the permeability in the y-direction, normalized after the permeability in the x-direction - is the square of the aspect ratio. Hence, the aspect ratio effects an equivalent permeability which scales as the square of the aspect ratio.

The same principle applies to the case of MFD. This can most easily be seen by recalling Equation 2.31, which gives a condition on the inner product $\boldsymbol{M}$ in a single cell,

$$\boldsymbol{M} = \frac{1}{|E|}\boldsymbol{C}^T\boldsymbol{K}^{-1}\boldsymbol{C} + \boldsymbol{M}_2.$$

Recalling that the i row of $\boldsymbol{C}$, $\boldsymbol{c}_i$, is the vector pointing from the cell centroid to the centroid of the ith face of the cell, we can use the norm of $\boldsymbol{c}^j = (\boldsymbol{c}_{1j}, \ldots, \boldsymbol{c}_{N_f j})$, with $j = 1, 2, 3$ as generalized step sizes in the three spatial directions. Hence, we can rewrite Equation 2.31 as

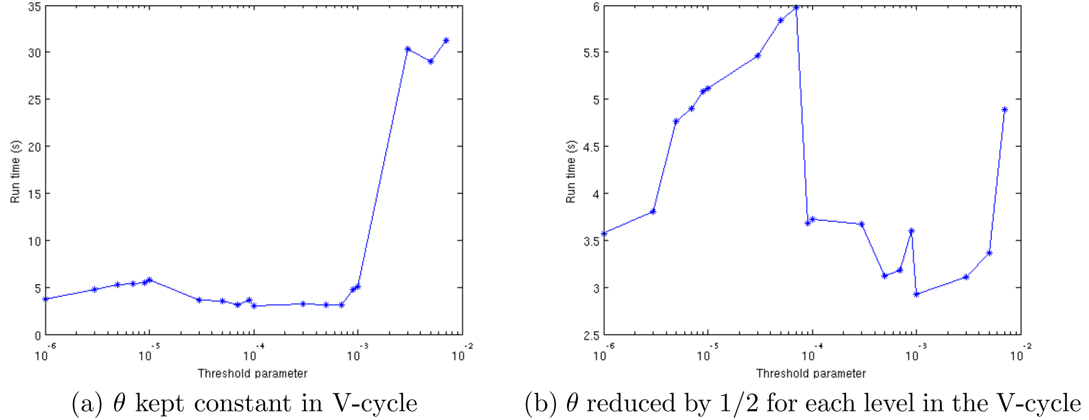
$$\boldsymbol{M} = \frac{1}{E|}\tilde{\boldsymbol{C}}^T\mathrm{diag}(\|\boldsymbol{c}^i\|^2)\boldsymbol{K}^{-1}\tilde{\boldsymbol{C}} + \boldsymbol{M}_2, \tag{6.5}$$

where $\tilde{\boldsymbol{C}} = \mathrm{diag}(\|\boldsymbol{c}^j\|)^{-1}\boldsymbol{C}$ is the matrix for a cell with aspect ratio equal to 1. By using $\tilde{\boldsymbol{K}}^{-1} = \mathrm{diag}(\|\boldsymbol{c}^i\|^2)\boldsymbol{K}^{-1}$ as the effective permeability, we see that the effective relative permeability changes as the square of the aspect ratio. This might help explain why the performance in test case 3 is severely worse than test case 2.

As described in Section 3.2.4 and Section 5.3, the standard way to deal with this issue is letting $\theta \neq 0$. When choosing θ, there is a trade-off between the convergence of the V-cycle on one side, and the run-time of a single V-cycle, and the operator complexity of the preconditioner on the other. As θ increases, the coarsening will tend to go in the direction of strong connections, which should improve the convergence factor of a V-cycle. However, since only degrees of freedoms that are strongly coupled are allowed to be aggregated together, this also means that the coarsening process becomes less aggressive, which can give more multigrid levels with Galerkin products where the sparsity structure of the Galerkin products become more complex [10], something which can be measured using operator complexity, as defined in Section 6.3.2.

Figure 6.7 illustrates the effect of different values of θ on the run-time of the CSR GPU solver on test case 3 with $n = 25$, with θ ranging from 0 to 0.007.

Higher values for θ exhausts the memory of the GPU, causing the solver to return without producing an answer. In Figure 6.7a, θ has been kept constant between levels in the V-cycle of AMG, whereas θ has been reduced by 1/2 for each level in Figure 6.7b. The values for θ in Figure 6.7b refers to the value at the finest level, $l = 1$.

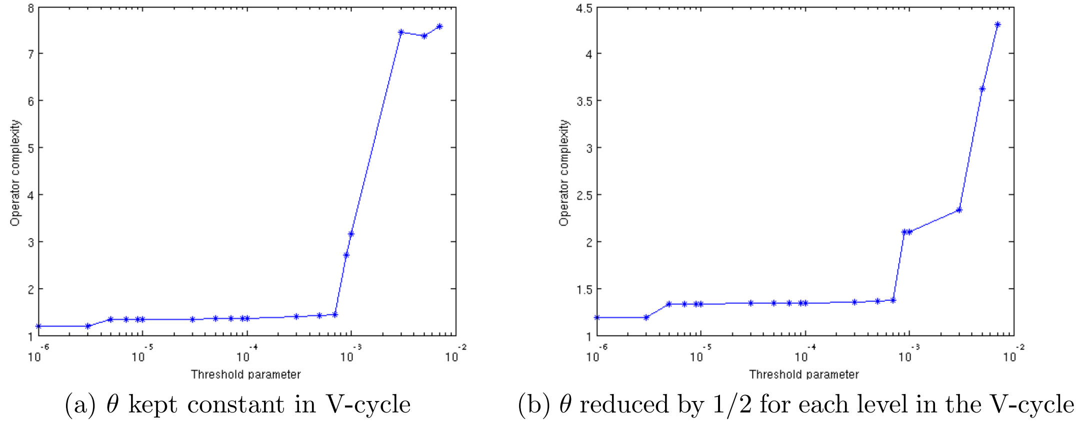


(a) θ kept constant in V-cycle (b) θ reduced by 1/2 for each level in the V-cycle

Figure 6.7: Effect of θ on run-time on test case 3 with $25 \times 25 \times 25$ cells.

As we can see, it seems possible to gain some speed up through right choice of θ, at least in principle. However, there are several issues the two tests reveal. First, the best speed ups in both tests are rather moderate, 1.12x and 1.23x, respectively. Furthermore, in both cases, different values of θ might both speed up or slow down the solvers, which suggests thorough testing would be required to find a value of θ robust to changes in the parameters of the problem, if such a value can be found at all. The greatest issue, however is related to the memory consumption of the solver. This is reflected in the spikes in run time for the highest values of θ in both figures, but can be more easily seen in Figure 6.8. This shows the operator complexity, as defined in (6.1), for the same tests. As we can see, the operator complexity increases greatly for bigger values of θ. This means the matrices on coarser levels become relatively denser, which has the twin effect that each V-cycle becomes more expensive and that the memory required to store the preconditioner increases. It also explains the fact that using $\theta > 0.007$ exhausts GPU memory. Another indication of how the coarsening fails in producing an efficient preconditioner can be seen from the fact that for $\theta = 0.007$ with θ kept constant, the preconditioner has 7 levels, with the matrices on the 4 coarsest levels being full.

As discussed in Section 3.2.4, fill-in is likely to occur on the coarse level ma-



(a) θ kept constant in V-cycle (b) θ reduced by 1/2 for each level in the V-cycle

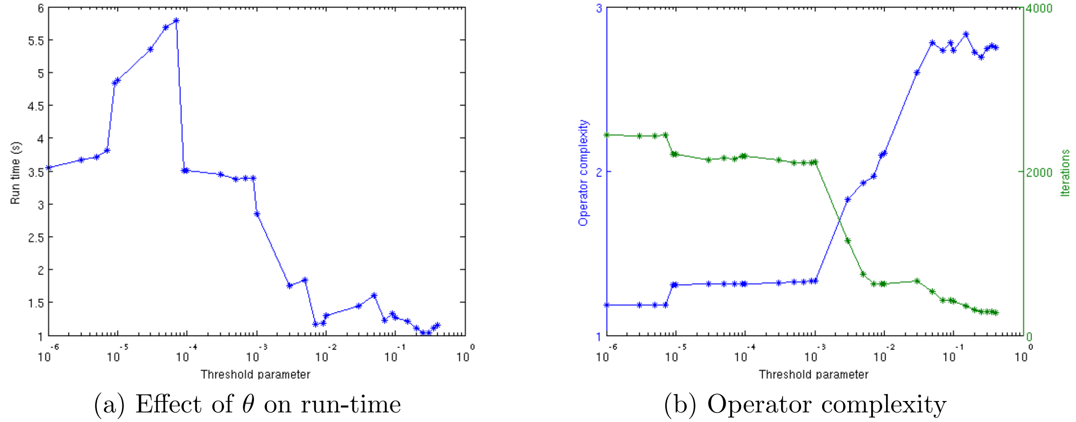
Figure 6.8: Operator complexity as function of θ, test case 3 with $25 \times 25 \times 25$ cells.

trices when anisotropic aggregation according to Algorithm 4 is combined with a prolongator smoother, (3.13), in which the unfiltered fine level matrices are used. The results above indicate that this is in fact happening for our test case.

Consequently, the setup phase of the preconditioner must be changed to use the filtered fine level matrices, (3.23), in the prolongator smoother, (3.22). As discussed in Section 5.3, our implementation is, on each level, really using a matrix whose off-diagonal elements are equal to that of the filtered matrix, but we do not add weak couplings in the unfiltered matrix to the diagonal, which is required in the convergence analysis.

Figure 6.9 shows the run time and operator complexity for test case 3 with $n = 25$, and θ reduced by 1/2 for each V-cycle level. In contrast to the test presented in Figure 6.7 and Figure 6.8, we are able to let θ range up to 0.4. Because the coarse level matrices are kept sparse, we do not exhaust GPU memory for higher values for θ as in our previous test. This can be seen in Figure 6.9b, which shows that the increase in operator complexity is much more modest than in Figure 6.8. We can also see that the number of CG iterations needed to reach convergence is significantly reduced. As the run time of each V-cycle depends on the sparsity of the coarse level matrices, the improved convergence properties for higher values of θ does not seem to be offset by the cost of the V-cycle as in Figure 6.7b.

Although Figure 6.8 shows that the convergence properties of the preconditioner are improved by increasing θ, there is still reason to question the overall quality of the preconditioner. This can be seen from Table 6.2, which compares



(a) Effect of θ on run-time

(b) Operator complexity

Figure 6.9: Test case 3, $n = 25$, θ reduced by 1/2 for each level in the V-cycle, filtered matrix in prolongator smoother.

Table 6.2: Preconditioner statistics for test case 3, $n = 25$.

	θ			AGMG
	0	0.08	0.3	
Operator complexity	1.18	2.73	2.75	1.77
CG iterations	2452	418	281	23

the operator complexity and number of CG iterations for the GPU solver with AGMG. AGMG converges in less than a tenth of the best number of iterations for the GPU solver, with a lower operator complexity.

To perform a more comprehensive test of the performance and robustness of this approach, we need to decide how to determine θ. As described in Section 5.3, Vaněk et al. [26] suggests

$$\theta = 0.08 \left(\frac{1}{2}\right)^{l-1}, \quad l = 1, \dots, L, \tag{6.6}$$

based on their numerical experiments. From Figure 6.9 and our further experiments, this seems like a reasonable choice. It should be pointed out that there is nothing special about this value in Figure 6.7, where all values of θ greater than roughly 0.001 seems to significantly reduce the run time. We have not carried out comprehensive tests attempting to determine an optimal value for θ.

To illustrate how using (6.6) for anisotropic aggregation and creating filtered matrices, Figure 6.10 shows the results for the same test cases as Figure 6.5. The

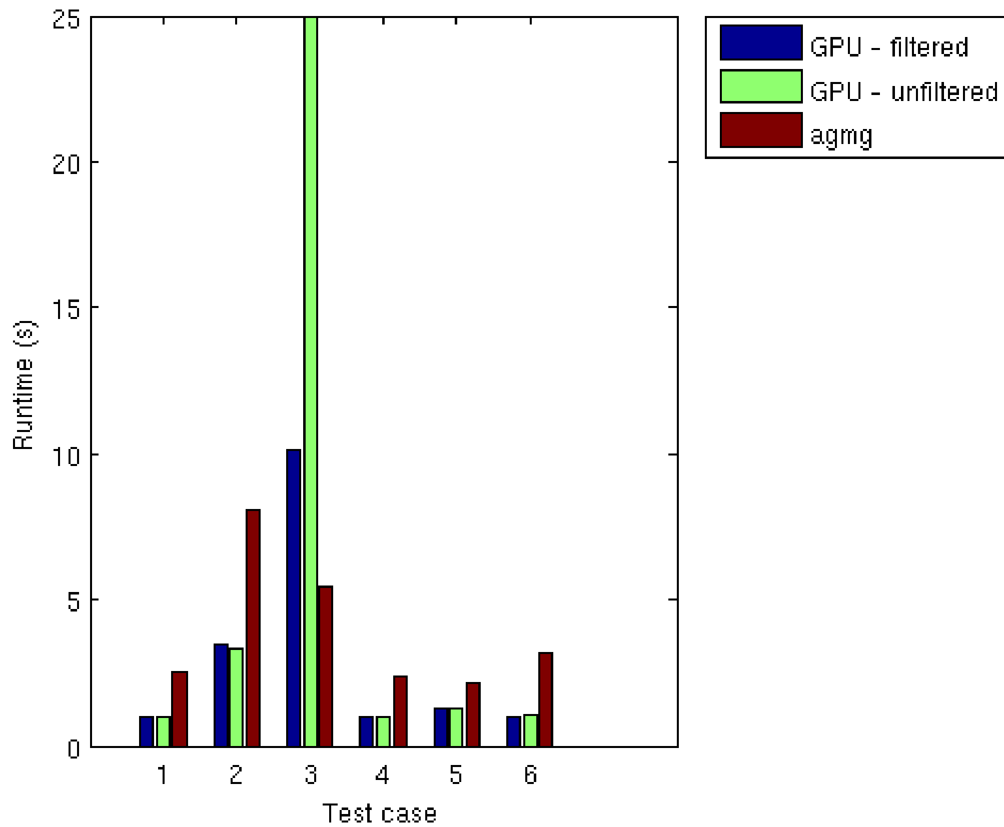


Figure 6.10: Running times for GPU solver with and without filtered matrix, and AGMG, all box test cases, $n = 50$.

GPU solver using $\theta = 0$ and AGMG are shown as references. Note that anisotropic aggregation does not seem to have a significant effect in other cases than test case 3.

Table 6.3 shows the behaviour of the preconditioner for all test cases. It is worth noting that anisotropic aggregation seems to have negligible effect on the convergence properties in test case 2, where the rock permeability is strongly anisotropic. Furthermore, the number of iterations required by AGMG is less than one tenth of that of the GPU solver with anisotropic aggregation for test case 2 and 3.

Table 6.3: Preconditioner properties all test cases, $n = 50$.

	$\theta = 0$ Op. comp./Iterations	$\theta = 0.08$ Op. comp./Iterations	AGMG Op. comp./Iterations
1	1.19/31	1.95/24	2.00/30
2	1.19/406	1.96/405	2.81/40
3	1.19/3760	3.44/735	2.14/65
4	1.19/34	1.20/26	1.99/24
5	1.19/50	1.33/44	1.78/21
6	1.19/39	1.20/33	1.89/30

Preconditioner performance for realistic scenarios

Based on the results for the $n \times n \times n$-grid, we have tested the GPU solver on the realistic scenarios using both $\theta = 0$, and anisotropic aggregation with $\theta = 0.08$ and the filtered matrix in the prolongator smoother. Figure 6.11 shows the run time for both scenarios compared to AGMG and `mldivide`.

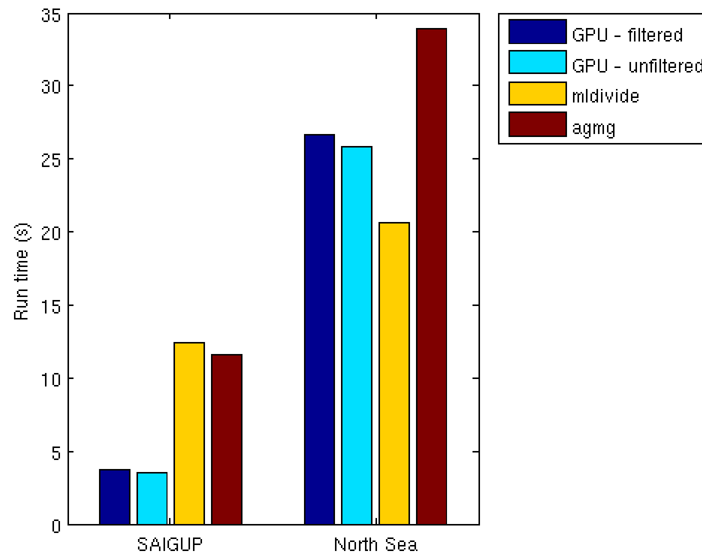


Figure 6.11: Run time for realistic scenarios.

When considering the run times only, the GPU solver compares well for SAIGUP. For the North Sea scenario, it is faster than AGMG, however, the fact that

`mldivide` is faster should be a concern. When considering the operator complexity and number of iterations to reach convergence, shown in Table 6.4, we see the same trends as in Table 6.2 and 6.3. In both cases, the convergence properties are significantly better for AGMG. This is especially true for the North Sea model.

Table 6.4: Preconditioner properties, realistic models.

	$\theta = 0$ Op. comp./Iterations	$\theta = 0.08$ Op. comp./Iterations	AGMG Op. comp./Iterations
SAIGUP	1.18/391	1.42/387	1.88/177
North Sea	1.19/2801	1.21/2876	2.31/374

6.3.3 Correctness

Table 6.5: 2-norm of relative error $\times 10^{-9}$ for iterative solvers compared to `mldivide` for box test cases, $n = 50$.

Test case	Solver				
	CSR, $\theta = 0$	CSR, $\theta = 0.08$	ELL	HYB	AGMG
1	1.051	0.687	1.051	1.051	2.806
2	52.46	52.43	52.46	52.46	6.081
3	1.853	1.404	1.851	1.851	0.236
4	3.524	2.095	3.524	3.524	3.951
5	3.315	1.771	3.315	3.315	0.756
6	0.135	0.068	-	0.135	0.206

Table 6.6: 2-norm of relative error $\times 10^{-7}$ for iterative solvers compared to `mldivide` for realistic scenarios.

Test case	Solver			
	CSR, $\theta = 0$	CSR, $\theta = 0.08$	HYB	AGMG
SAIGUP	2.007	1.502	2.007	3.974
North Sea	0.452	1.128	0.456	2.911

The correctness results for the test cases are presented in Table 6.5, Table 6.5, and Table 6.7. For the GPU solver, we list the results for the different storage formats discussed in 5.3.

Table 6.7: Maximal absolute value of relative divergence per cell $\times 10^{-7}$ for box test cases, $n = 50$.

Test case	Solver				
	CSR, $\theta = 0$	CSR, $\theta = 0.08$	ELL	HYB	AGMG
1	1.455	4.903	1.455	1.455	1.817
2	1.144	1.062	1.144	1.145	0.784
3	0.248	0.752	0.244	0.244	1.374
4	0.742	1.195	0.742	0.742	1.298
5	0.230	0.245	0.230	0.230	0.171

Table 6.5 shows the 2-norm of the relative error of $\boldsymbol{\pi}$ computed by the solvers. As the stopping criteria is a relative reduction of the residual of 10^{-9}, the results seem satisfactory. In Table 6.7, the maximal relative divergence is shown for all solvers.

It is worth noting that the correctness results for the GPU solver is independent of the storage format.

Chapter 7

Conclusions and Further Work

In this chapter, we will summarize our work, and discuss our findings.

7.1 Conclusion

In the introduction, we posed the following three questions:

1. Can we use CUSP to create a GPU-based linear solver for MRST implementing conjugate gradients with algebraic multigrid as preconditioner?

2. Are we able to use GPUs to speed up the solution of linear systems in MRST compared to existing CPU-based alternatives?

3. Can we propose and implement improvements to CUSP's linear solver, making it better suited for our application in reservoir simulation?

The first question is, of course, a prerequisite to the two following. In Chapter 5 we demonstrate how we are able to use CUSP for MRST using MEX-interfaces. As we have seen in Section 6.3.3, we have verified the correctness of the answers, and shown that they have the same quality as the CPU-based AGMG. We have also argued that the GPU solver uses less memory than `mldivide`, however since the GPU has a separate memory space from the CPU, the practical outcome of this is, of course, hardware dependent.

As to our second question, we must consider the results in Section 6.3.2. They do not, however, give conclusive answers. While most of the idealized test cases and the SAIGUP model are solved faster using the GPU solver, this is not the case for the test case with high aspect ratio of grid cells and the North Sea model. The preconditioner statistics in Table 6.2, 6.3, and 6.4 explains much of this behavior. As we can see, the quality of CUSP's preconditioner deteriorates significantly

when the underlying reservoir model has strongly anisotropic permeability or the grid cells have high aspect ratios. Although the preconditioner of AGMG also is affected by these phenomena, it is on a completely different scale. For the test cases with isotropic characteristics, we observe that the preconditioner implemented in CUSP has convergence properties that are comparable to AGMG. Hence, it seems reasonable to conclude that the there is significant room for improvement of the mathematical properties of CUSP's preconditioner when facing some of the challenges arising in reservoir simulation. On the other side, it should be emphasized that CUSP outperforms the CPU based solvers we have used in several test cases. Furthermore, even in the cases where AGMG is faster, it is clear from the preconditioner statistics that the GPU-based solver is able to perform PCG iterations faster than AGMG. This demonstrates the strength of the GPU in terms of computing power in a practical setting.

This brings us over to a our third question. As it seems like CUSP is able to make the GPU perform PCG iterations efficiently, we believe the most room for improvement is found in improving the properties of the AMG preconditioner. As discussed in Section 3.2.4 and Section 5.3, we have proposed using the filtered matrix for the prolongator smoother to improve properties of the preconditioner. Strictly speaking, we have not implemented this, as we have not been able to modify the diagonal of A_h^F in accordance with (3.23). Hence, we should not be to conclusive when evaluating the success of this approach. However, from our experiments, it seems like although this technique seems to have a significant effect in the pathological test case 3, it has negligible effect in the other scenarios we have tested. Since the convergence of the preconditioner may have been negatively affected by our implementation, we can not rule out that a correct implementation of the approach described in Section 3.2.4 can give further improvements. However, when we compare with the preconditioner statistics of AGMG, which uses a type of AMG preconditioner, it is natural to rise the question whether other approaches to AMG could be more successful.

7.2 Further work

Given the limitations in time and scope, there are of course topics we have not dealt with in this thesis. In this section we will point out some of these.

Test run time complexity. An important question when evaluating programs in numerical linear algebra - and computing in general - is how the run time scales with the input size. In our tests, we have focused on how the solver handles properties and effects that frequently arises in reservoir simulation, and we have not found time to test the run time complexity. For a more complete overview of

the solver, this is an aspect that should be considered.

Implement a correct version of filtered matrix prolongator smoother. As discussed, the implementation of the filtered prolongator smoother is not consistent with the definition in Section 3.2.4. A correct implementation should be feasible using THRUST's functionality.

Investigate alternative approaches to AMG preconditioning. From a mathematical point of view, it would make sense to use a preconditioner that aims specifically to deal efficiently with anisotropic problems. One possible approach is described in [10], where Gee et al. introduces a new preconditioner which is designed to perform more efficiently on anisotropic problems. Their approach, basis shifted smoothed aggregation (BSSA), is based on smoothed aggregation, but modifies the prolongator from standard smoothed aggregation. In their numerical experiments, they are able to improve convergence properties without sacrificing operator complexity compared to the approaches considered in this thesis.

This raises, however, some design and implementation issues. As mentioned in Section 4.3.2, CUSP performs not only the solve phase of the linear solver, but even the setup phase on the GPU. This does, however require a version of AMG that can be constructed with fine-grained parallelism. To our knowledge, it is not clear whether it is possible to implement the BSSA algorithm efficiently on a GPU.

A more general point can made from this. The design decision to construct the preconditioner on the GPU makes sense given that it enables CUSP to offer a complete black box GPU solver, however, this excludes preconditioners that are not feasible for efficient setup on GPUs. However, given CUSP's and THRUST's functionality for generic data structures and copying between host and device, an interesting question is whether it is possible to construct the preconditioner on the CPU and transfer it to the GPU for the solve phase of the algorithm. If so, one could consider aggregation schemes that have been successfully implemented for CPUs, such as the ones used in AGMG [17], or even classical AMG [25], possibly allowing approaches more tailored to anisotropic problems. This requires, however, that the preconditioner can be constructed in a way that is compatible with the data structures used in CUSP and allows for efficient V-cycle execution.

We leave the questions raised in this paragraph open.

Bibliography

[1] Jørg E. Aarnes, Tore Gimse, and Knut-Andreas Lie. An introduction to the numerics of flow in porous media using Matlab. In *Geometric modelling, numerical simulation, and optimization: applied mathematics at SINTEF*, pages 265–306. Springer, Berlin, 2007.

[2] Ivar Aavatsmark. Bevarelsesmetoder for elliptiske differensialligninger. Forelesningsnotater ved Universitetet i Bergen, 2002.

[3] Nathan Bell, Steven Dalton, and Luke Olson. Exposing fine-grained parallelism in algebraic multigrid methods. NVIDIA Technical Report NVR-2011-002, NVIDIA Corporation, June 2011.

[4] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proc. Supercomputing '09*, November 2009.

[5] Nathan Bell and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2011. Version 0.2.0.

[6] L. Susan Blackford and et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software*, 28(2):135–151, 2002.

[7] Franco Brezzi, Konstantin Lipnikov, and Mikhail Shashkov. Convergence of the mimetic finite difference method for diffusion problems on polyhedral meshes. *SIAM J. Numer. Anal.*, 43(5):1872–1896 (electronic), 2005.

[8] Bryan Catanzaro. An introduction to CUDA/OpenCL and manycore graphics processors. Lecture notes CS 267, UC Berkeley, 2011.

[9] James W. Demmel. *Applied numerical linear algebra.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1st edition, 1997.

[10] Michael W. Gee, Jonathan J. Hu, and Raymond S. Tuminaro. A new smoothed aggregation multigrid method for anisotropic problems. *Numer. Linear Algebra Appl.*, 16(1):19–37, 2009.

[11] gpgpu.org. General-Purpose Computation on Graphics Hardware.

[12] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *J. Research Nat. Bur. Standards*, 49:409–436 (1953), 1952.

[13] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2012. Version 1.5.0.

[14] Kurt Keutzer and Tim Mattson. A design pattern language for engineering (parallel) software.

[15] Knut-Andreas Lie, Stein Krogstad, Ingeborg Ligaarden, Jostein Natvig, Halvor Nilsen, and Bård Skaflestad. Open-source matlab implementation of consistent discretisations on complex grids. *Computational Geosciences*, pages 1–26, 2011. 10.1007/s10596-011-9244-4.

[16] Scott MacLachlan. *Improving Robustness in Multiscale Methods*. PhD thesis, University of Colorado, 2004.

[17] Yvan Notay. Aggregation-based algebraic multilevel preconditioning. *SIAM J. Matrix Anal. Appl.*, 27(4):998–1018 (electronic), 2006.

[18] Yvan Notay. User's guide to AGMG, 2008.

[19] NVIDIA. CUDA C Best Practices Guide, 2011. Version 4.0.

[20] NVIDIA. NVIDIA CUDA C Programming Guide, 2011. Version 4.0.

[21] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[22] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[23] Jonathan Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. http://www.cs.berkeley.edu/ jrs/jrspapers.html, August 1994.

[24] SINTEF Applied Mathematics. MATLAB Reservoir Simulation Toolkit, 2011. Version 2011a.

[25] K. Stüben. Algebraic multigrid (AMG): An introduction with applications. Appeared as an appendix in the book: Multigrid“ by U. Trottenberg; C.W. Oosterlee; A. Schüller, Academic Press, pp. 413-532, 2001.

[26] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996. International GAMM-Workshop on Multi-level Methods (Meisdorf, 1994).

[27] Petr Vaněk. Acceleration of convergence of a two-level algorithm by smoothing transfer operators. *Appl. Math.*, 37(4):265–274, 1992.