Norwegian University of
Science and Technology

Mimetic Finite Difference Method on GPU

Application in Reservoir Simulation and Well Modeling

Gagandeep Singh

Master of Science in Physics and Mathematics
Submission date: July 2010
Supervisor: Trond Kvamsdal, MATH
Co-supervisor: Knut-Andreas Lie, SINTEF Oslo

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

The primary purpose of this thesis is to exploit the excessive floating-point capabilities of a Graphical Processing Unit (GPU) to implement a mathematical model and get the most performance benefits compared to an (existing) CPU implementation. Among the few available GPU programming architectures, we have chosen to make the implementations using NVIDIA's CUDA on an NVIDIA GPU.

We study a well-known mathematical model in the form of an elliptic partial differential equation (PDE) that describes the incompressible single-phase fluid flow through a porous medium. The model is based on mass conservation, Darcy's Law and the physical concepts of porosity and permeability of a spatial body. The application of this model is made in oil reservoir simulation, where the reservoir is the porous medium. The reservoir model is most generally represented by corner-point grids that capture the main features of a real reservoir, such as faults. The model also incorporates injection and production wells in the model.

We solve this PDE numerically and discretise the model using the numerical technique of Mimetic Finite Difference (MFD) method. This results in a linear system that with physically correct boundary conditions provides an symmetric positive definite system. The linear solver has been chosen to be the Jacobi-preconditioned conjugate gradient method. The system is built by assembling the local stiffness cell matrices.

During the primary discussions there has not been any well-defined plan as to how we proceed, but we intend to start with a method where we avoid assembling the linear system and try to solve it using only the cell matrices.

Assignment given: 15. February 2010
Supervisor: Trond Kvamsdal, MATH

Abstract

Heterogeneous and parallel computing systems are increasingly appealing to high-performance computing. Among heterogeneous systems, the GPUs have become an attractive device for compute-intensive problems. Their many-core architecture, primarily customized for graphics processing, is now widely available through programming architectures that exploit parallelism in GPUs. We follow this new trend and attempt an implementation of a classical mathematical model describing incompressible single-phase fluid flow through a porous medium. The porous medium is an oil reservoir represented by means of corner-point grids. Important geological and mathematical properties of corner-point grids will be discussed. The model will also incorporate pressure- and rate-controlled wells to be used for some realistic simulations. Among the test models are the 10th SPE Comparative Solution Project Model 2. After deriving the underlying mathematical model, it will be discretised using the numerical technique of Mimetic Finite Difference methods. The heterogeneous system utilised is a desktop computer with an NVIDIA GPU, and the programming architecture to be used is CUDA, which will be described. Two different versions of the final discretised system have been implemented; a traditional way of using an assembled global stiffness sparse matrix, and a Matrix-free version, in which only the element stiffness matrices are used. The former version evaluates two GPU libraries; CUSP and THRUST. These libraries will be briefly decribed. The linear system is solved using the iterative Jacobi-preconditioned conjugate gradient method. Numerical tests on realistic and complex reservoir models shows significant performance benefits compared to corresponding CPU implementations.

Acknowledgements

This thesis is submitted as the final work required to complete the Master's Degree programme in Numerical Mathematics at the Norwegian University of Science and Technology (NTNU). The thesis was written in the period from February to July 2010.

I would like to thank Professor Trond Kvamsdal who acted as my supervisor for this thesis and the co-supervisor Professor of Mathematics at the University of Bergen (UiB) and Research Scientist Knut-Andreas Lie at SINTEF, who by inspiring discussions raised my initial interest in the fascinating topic of Mimetic Finite Difference Methods for Partial Differential Equations and discrete vector and tensor analysis, and who actually proposed the main topic for the thesis. A special thanks to him also for his generous help in securing an office and computer resources at SINTEF's office in Oslo for my use during the writing of this thesis.

I would like to express my deepest gratitude to Research Scientist Bård Skaflestad at SINTEF's office in Oslo for always being available and generous with his time and for the invaluable discussions that helped me understand the mathematical ideas and methods needed for my thesis in a profound manner that boosted both my interest and enthusiasm. No student has received more valuable help or more inspiring support for a Master's Thesis than I have received from Bård. Thank you!

I also extend a thank to Trond R. Hagen and Andre R. Brodtkorb for lending me GPUs for testing purposes.

A particular note of thanks is also given to Bjørn Jahnsen for proofreading this report and giving me useful suggestions for language improvements.

Gagandeep Singh
June 2010

Contents

1 Introduction **1**
- 1.1 Outline 4

2 A Brief Note on Mathematical Modeling **5**
- 2.1 Mathematical Modeling Through Differential Equations 8

3 Reservoir Modeling **9**
- 3.1 Reservoir Description 9
  - 3.1.1 Reservoir Porosity 10
  - 3.1.2 Permeability 11
- 3.2 Incompressible Single-Phase Flow 12
  - 3.2.1 Darcy's Law 12
  - 3.2.2 Elliptic Pressure Equation 12

4 Discretisation of Elliptic Pressure Equation **14**
- 4.1 Mimetic Finite Difference Method (MFD) 14
  - 4.1.1 Mixed Formulation 16
  - 4.1.2 Hybrid Formulation 17
  - 4.1.3 Discrete Formulation 19
  - 4.1.4 An MFD Formulation 19
  - 4.1.5 Schur-complement Reduction 20
  - 4.1.6 Motivation for the Approximate Bilinear Form 21
- 4.2 Well Modeling 25
- 4.3 Corner-point Grids 28

5 Conjugate Gradient Method **31**
- 5.1 Theory and Background 32
  - 5.1.1 Rate of Convergence 34
- 5.2 Preconditioned CG-Algorithm 36
  - 5.2.1 Practical Preconditioners 36

6 GPGPU - General-Purpose Computing on GPUs **38**
6.1 Contrast to CPUs 40
6.2 GPUs and Scientific Computing 41
6.3 Heterogeneous Computing Model 42
6.4 GPU Computing with CUDA 44

7 CUDA Programming Model **46**
7.1 GPU Device Architecture Overview 46
7.2 Introducing Fermi 48
7.3 CUDA Architecture 50
7.4 CUDA Memory Model 52
7.4.1 Memory Hierarchy 52
7.5 CUDA Execution Model 53
7.6 Shared Memory 54
7.7 Memory Coalescing 56

8 Implementation **57**
8.1 MRST - MATLAB Reservoir Simulation Toolbox 59
8.2 Matrix-free Implementation 59
8.3 Full-matrix Implementation 63
8.4 Sparse Matrix-Vector Multiplication on CUDA 64
8.4.1 Sparse Matrix Formats 65
8.4.2 The THRUST Library 68
8.4.3 The CUSP Library 70

9 Numerical Experiments **73**
9.1 Verification of the Solvers 73
9.2 Reservoir Models 74
9.2.1 SAIGUP Reservoir Model 28 74
9.2.2 SPE10 Reservoir Model 2 76
9.2.3 BIG 1 Reservoir Model 79
9.3 Speedup Measurements 82
9.3.1 Comparision with Solvers from SINTEF 85
9.3.2 Discussion of the Results 86

10 Conclusions and Summary **90**
10.1 Further Work 92

Bibliography **94**

Chapter 1

Introduction

Mathematical modeling is an important scientific discipline applied by scientists to simulate physical phenomena which are difficult to experiment on in practice. In combination with computer science's continuing progress in computational power, the underlying mathematical models, often driven by scientific research and industry, are growing more complex and accurate. The demand for computational power from such models is ever-expanding.

In combination with the CPU's relatively slow development in terms of floating-point operations, a new paradigm shift is about to be established in computational science. The now almost obsolete way of performing calculations sequentially is beginning to be taken over by computing systems that exploit parallelism[1]. There are supercomputers with thousands of CPUs, and integrated many-core systems in a single device also exist. The latter model is well-known in computer graphics and gaming industry, namely the Graphical Processing Unit, GPU for short. Gaming industry's enormous growth has pushed the prices of GPUs to a manageable level, and the GPU has evolved, and continues to evolve, at a pace that now seems to be over for the CPU. This disparity in performance can be attributed to fundamental architectural differences: CPUs are optimized for high performance on sequential code, so many of their transistors are dedicated to supporting non-computational tasks such as branch prediction and caching. On the other hand, the highly parallel nature of graphics computations enables GPUs to use additional transistors for computation, achieving higher arithmetic intensity with the same transistor count[2].

The computational power and memory bandwidth of GPUs have significantly overwhelmed CPU specifications. For example, an Intel Core i7-965 Extreme

Edition CPU has a theoretical peak double precision performance of 53.28 GFlops[1], with a memory bandwidth of maximum 25.6 GB/s. The recent NVIDIA GTX 480 GPU has a theoretical single precision peak performance of 1344.96 GFlops[2], with a memory bandwidth of 177.4 GB/s (GTX 480 has 480 cored each capable of doing 1 multiply-add operation per cycle. This gives 480x2x1401 = 1344.96 GFlops). GTX 480 performs at half the speed in double precision, giving 672.48 GFlops. Currently, in July 2010, the price of GTX 480 is half of the price of the CPU.

However, it has not been possible to exploit the GPU's computational power primarily because it is fixed-function[2]. Until recently, there has not been any programming interface to access GPUs beyond its strong graphics context. Several hetereogeneous architectures have emerged during this decade. GPU manufacturers such as NVIDIA and AMD/ATI have introduced new dedicated APIs to general-purpose computations on GPUs[3, 1]: The Compute Unified Device Architecture (CUDA) from NVIDIA and the ATI Stream SDK from AMD/ATI. These APIs provide low-level or direct access to GPUs, exposing them as large arrays of parallel processors. Future applications will require heterogeneous processing[1].

The problem we shall examine in this thesis is an existing mathematical model that describes the incompressible single-phase fluid flow through a porous medium. The implementation will focus on reservoir simulation with an oil reservoir as the porous medium. We will take this one step further and add production and injection wells in the model and simulate some realistic reservoir models. This part of the problem involves a mathematical model. The other half of the problem includes the implementation of this model on a desktop computer with a GPU. We will implement the solution using NVIDIA's GPUs and their programming architecture CUDA. The choice of using CUDA has been based on the experience SINTEF has accrued using CUDA and my own modest experience with CUDA from my last year's thesis project.

The mathematical model is in the form of an elliptic partial differential equation. After deriving the model, we will describe how we discretise it using the Mimetic Finite Difference method (MFD) with a hybrid formulation technique. The reservoir model, the underlying porous medium, is represented using corner-point grid, the properties of which will be described. The resulting linear system will be solved on GPU using two techniques; the traditional technique of representing the global stiffness

[1]http://www.intel.com/support/processors/sb/cs-023143.htm

[2]http://www.nvidia.com/object/product_geforce_gtx_480_us.html

matrix in various sparse matrix formats (Full-matrix version), as well as a Matrix-free implementation which never actually assembles the matrix (Matrix-free version).

The linear system $\mathbf{A}x = \mathbf{b}$ resulting from the mimetic discretisation on general corner-point grids is sparse and unstructured. The dimension of this system on realistic reservoir models does not prevail the use of direct solvers. It is common practice to apply iterative solvers on large sparse systems. The implementation in this thesis uses the Jacobi-preconditioned iterative conjugate gradient method [4] to solve the system. This method has a much smaller memory footprint than direct solvers such as Gaussian elimination , and can be applied to very large sparse systems. This method is aslo easier to parallelize and implement on the GPU. The method will be described to approriate details.

The Full-matrix version stores the sparse matrix **A** using different sparse matrix formats. These include CSR (*Compressed Sparse Row*), HYB (Hybrid format), COO (*Coordinate Format*), and ELLPACK. We shall describe these formats. The Full-matrix version evaluates two CUDA libraries developed by people in NVIDIA; the libraries CUSP and THRUST. THRUST is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). CUSP is a library for sparse linear algebra and graph computations on CUDA. CUDA SDK provides an optimised GPU implementation of BLAS called CuBLAS, but currently it only supports dense matrices. CUSP implements these sparse formats and important matrix operations such as matrix-vector multiplication. CUSP also includes Jacobi-preconditioned conjugate gradient and biconjugate gradients stabilized method. The more advanced AMG preconditioner is currently under development.

The extent to which it is a question of implementation on the GPU, the main purpose is speedup compared with existing CPU implementations. We also want to test how the two implementations, Full-matrix and Matrix-free, perform relative to each other. The systems we solve in this project are impossible to solve with single floating-point precision, and it has been taken into account that modern GPUs adapted to scientific computing now have support for double precision. Only double precision has been used, even though it is posssible to switch to single precision when solving smaller systems.

1.1 Outline

The report is outlined as follows. Chapter 2 is a brief note on mathematical modeling. Chapter 3 introduces briefly the most important concepts used in the derivation of the mathematical model, such as reservoir description (Section 3.1), reservoir porosity (Section 3.1.1) and permeability (Section 3.1.2). Section 3.2 discusses and derives the elliptic partial differential equation (PDE) describing the incompressible single-phase fluid flow through porous medium. The most important results referred to in this chapter are the conservation of mass and the well-known Darcy's Law. Chapter 4 begins with an introduction of the MFD method, which will be used to discretise the PDE. Subsequent sections discuss the mixed formulation and the hybrid formulation, which is the point of departure for the final discretisation. Section 4.1.6 derives the approximate bilinear form used in the discretisation, motivating the primary difference between MFD and mixed finite element methods. Section 4.2 describes modeling of wells. Wells are modeled as Dirichlet and Neumann boundary conditions, and we discuss a slight modification of the linear system without wells from Section 4.1.4, and add wells to this system. Finally, Section 4.3 provides a brief description of corner-point grids. Chapter 5 is an introduction to the conjugate gradient method, which is described as a line search method. Chapters 6 and 7 are dedicated to GPU computing and the CUDA programming model. Chapter 6 discusses the role of GPU and heterogeneous systems in scientific computing and explains the primary differences between the GPU and CPU. Chapter 7 begins with an overview of the GPU architecture in the context of CUDA, and also explains some of major new features of the recent next-generation NVIDIA GPU architecture named Fermi. The rest of this chapter introduces the CUDA programming model. Chapter 8 describes the most important implementation details. Section 8.2 is a description of the Matrix-free implementation, followed by a description of the Full-matrix version in Section 8.3. Section 8.4 is primarily a reference to papers [5] and [6], and explains the data structure of the sparse matrix formats. A concise introduction to CUSP and THRUST is given in Sections 8.4.2 and 8.4.3, respectively. Chapter 9 includes a verification of the implementations, as well as simulation of realistic test reservoir models. Section 9.2 gives a description of the test models, and in Section 9.3 we discuss the CPU and GPU timing results. This section also includes some timing results from existing solvers at SINTEF. Chapter 10 gives a summary and conclusion of the report.

Chapter 2

A Brief Note on Mathematical Modeling

To define the context within which this work can be placed, scientifically speaking, let us start by referring to some words on applied mathematics and mathematical modeling. After that we will continue with the introduction of reservoir modeling, discuss the different assumptions and the mathematical model for single-phase incompressible fluid flow.

To a certain extent, mathematics is about explaining organized structures. Different patterns in nature, for example spiral galactic patterns, demand an investigation, not only for reasons of aesthetics, but the patterns offers an important clue to the fundamental laws of nature.

Applied mathematics is the sub-discipline of mathematics which examines the real-world through mathematical models. It is guided by the spirit of and belief in the *interdependence* of mathematics and the sciences. Historically, the development of mathematics and physics had a very close connection. Classical examples may be found in the works of Newton, Gauss, Euler, Cauchy, and others. Under the contemplation of applied mathematics, we think about mathematical models formulated through hypothetical assumptions and empirical observations. Natural sciences, as well as social sciences, take advantage of mathematical models. Important application fields include engineering, physics, geology, physiology, ecology, chemistry. Instead of undertaking experiments in the real world, a modeler simulates the experiment using mathematical representations. There are thousands of mathematical models which have been successfully developed and applied to gain insight into tens of thousands of situations. The

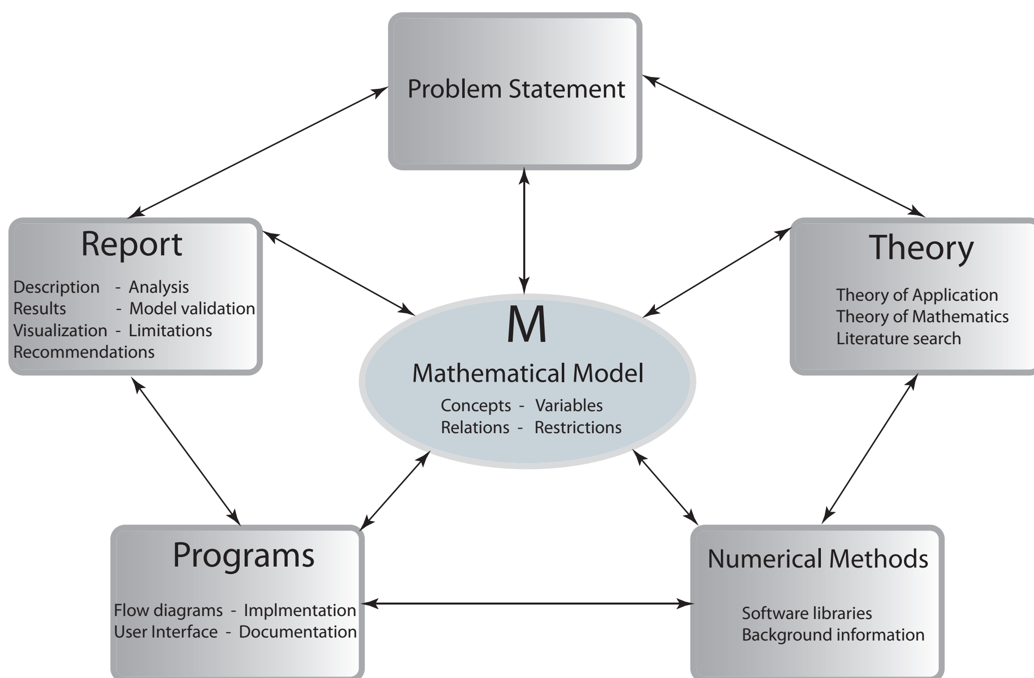


Fig. 2.1: The modeling diagram. The nodes represent information to be collected, sorted, evaluated and organized. The arrows represent two-way communication.

development of powerful computers has enabled a much larger number of situations to be mathematically simulated. Medical imaging and weather modeling are two well-known examples. Moreover, it has been possible to make more realistic models and to obtain better agreement with observations.

Let us attempt to give a brief description of the objectives and the methodology of applied mathematics. The purpose of applied mathematics is to elucidate scientific concepts and describe scientific phenomena through the use of mathematics, and to stimulate the development of new mathematics through such studies. The process of using mathematics for increasing scientific understanding can be conveniently divided into the following three steps:

- The *formulation* of the scientific problem in mathematical terms.
- The *solution* of the mathematical problems thus created.
- The *interpretation* of the solution and its empirical verification in scientific terms.

Thus, applied mathematics is concerned with the construction, analysis,

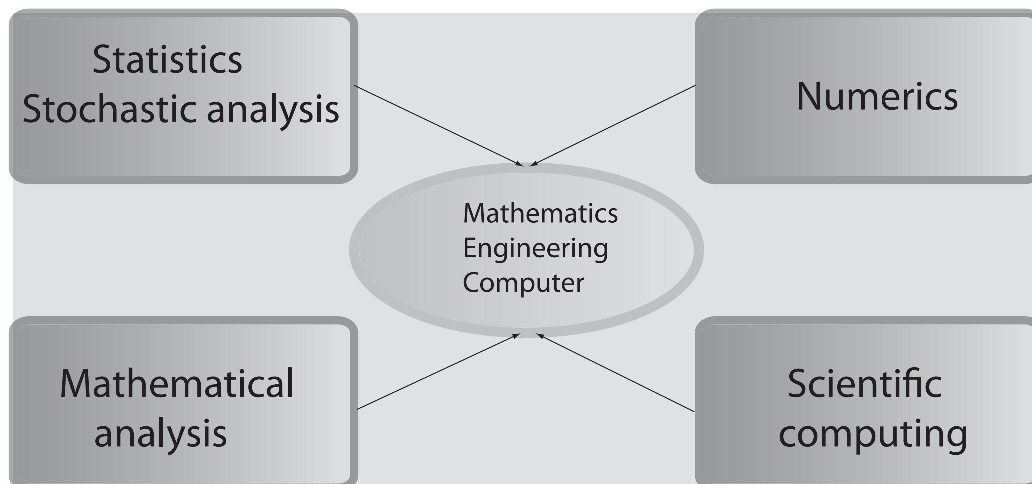


Fig. 2.2: The impact from computers in omnipresent.

and interpretation of mathematical models that shed light on significant problems in the natural sciences. Generally speaking, all three steps are equally important. In a given class of problems, one step might stand out as more important or more difficult than another.

Realism of mathematical models is always considered. We want a mathematical model to be as realistic as possible and to represent reality as closely as possible. However, if a model is very realistic, it may not be mathematically tractable. In making a mathematical model, there has to be a trade-off between tractability and reality. A mathematical model is said to be robust if small perturbations in the parameters lead to small changes in the behavior of the model. The challenge in mathematical modeling is not to produce the most comprehensive descriptive model, but to produce the simplest possible model that incorporates the major features of the phenomenon of interest. Some of the main principles often used in formulating mathematical models are[1]:

- Balance and conservation
- Flow, transport and logistics
- Equilibrium, stability, bifurcations
- Optimisation and variational principles
- Input - Filter - Output

[1]http://www.math.ntnu.no/ hek/MatMod2009/IntroduksjonTilFaget.pdf

- Feedback and control
- Kinetics, growth and decay
- Invariance principles

2.1 Mathematical Modeling Through Differential Equations

Mathematical modeling in terms of differential equations arises when the situation modeled involves some continuous variable(s) varying with respect to some other continuous variable(s) and we have some reasonable hypotheses about the *rates of change* of dependent variable(s) with respect to independent variable(s).

When we have one dependent variable x (say population size) depending on one independent variable (say time t), the mathematical model is expressed in terms of *ordinary* differential equation of the *first* order, if the hypothesis is about the of rate dx/dt. If the hypothesis involves the rate of change of dx/dt, the model will be in terms of an ordinary differential equation of the *second* order .

If there are a number of dependent continuous variables and only one independent variable, the hypothesis may give a mathematical model in terms of a system of first or higher order ordinary differential equations. If there is one dependent continuous variable (say pressure p) and a number of independent continuous variables (say space coordinates x, y, z and time t), the mathematical model is expressed in terms of a *partial differential equation*, PDE for short. Similarly we can have situations where we have a system of partial differential equations.

Chapter 3

Reservoir Modeling

The primary objective of an oil reservoir study is to predict future performance of a reservoir in terms of production rate, production charactercics, reservoir parameters, etc[7, 8]. Generally, a reservoir model consists of basic physical laws that govern fluid flow through porous media, conservation of mass law and the diffiusivity equation in addition to the fluid behavoir. A computer implementation that incorporates this model relative to some reservoir data constitues a reservoir model.

The following section gives a brief explanation of the key terms to be used in the description of the mathematical model. These include description of the reservoir formation, porosity, and permeability. A detailed and lucid overview of general reservoir parameters and derivation of the mathematical model is given in [7] and [9]. The most relevant parts will be extracted.

3.1 Reservoir Description

Millions of years ago the oceans were awash with countless tiny plants and animals which died and fell to the seabed. Mud and sand from rivers covered the plant and animal remains and over time more and more layers were added. As the old layers were burried deeper by new layers, the remains were subjected to pressure and heat. Over millions of years the remains of dead plants and animals decomposed into petroleum[1] - crude oil and natural gas. Petroleum as it occurs in nature is a mixture of organic materials composed only of carbon and hydrogen.

[1] from Greek word *petra* (rock) and Latin word *oleum* (oil).

The depoists of petroleum were trapped by the movements of the earth's crust when the flat layers became bent into folds and broken faults. The petroleum moved upwards through sandstone and other porous materials until it was trapped by barriers of dense/impermeable rock which it could not get through. There are several types of traps in which petroelum has accumulated. The most common are domes and faults. The pressures are usually high (10000 psi) and fluids are able to flow within these reservoirs, if the pressure gradient can be eshablished. This causes hydrocarbons to exist in both fluid and gas state. The trapped gas and liquids will separate into three layers inside the porous rock; on top a layer of gas, then a layer of liquid petroleum and at the bottom a layer of water. A volume of porous rock containing petroelum is called the "reservoir".

The geologists look for oil by studying the earth for signs of ancient seabeds and typical traps, such as a dome or a fault. By analysing the information from seismic data, the geophysicists construct a map of the underground rock formations (for instance by tomographic reconstruction). When they locate a possible trap in the form of a dome or a fault, a drilling rig is moved in and a hole drilled from the surface to the trap. Only by drilling is it possible to find out if there is oil in the identified trap. Drilling can be done both on land and in the sea.

Drilling a well can take from a few days up to several months. Oil wells are typically 1 to 5 km deep. Modern wells can be drilled at an angle to reach areas up to 6 to 8 km away from the drilling rig. It is even possible to drill horizontally and such wells may reach a total length of up to 10 km.

When the valves installed at the top of an oil well are opened, the oil flows to the surface by itself. Production resulting from reservoir pressure is called the *primary* production. This is because the pressure in the reservoir rock is higher than the pressure created by the weight of the oil in the well. After a period of production, the pressure in the reservoir falls and the oil eventually stops flowing naturally. Various methods for increasing the recovery factor (enhanced production) can be utilised to increase the production, for example by water injection or gas injection, and pumping.

3.1.1 Reservoir Porosity

The rock porosity, denoted by ϕ, is the void volume-fraction of a volume, and $0 \leq \phi < 1$. The rock porosity usually is affected by the pressure. The

rock is *compressible*, and the rock *compressibility* is defined by:

$$c_r = \frac{1}{\phi}\frac{d\phi}{dp}, \tag{3.1}$$

where p is the overall reservoir pressure. For simplified models it is customary to neglect the rock comressibility and assume that ϕ only depends on the spatial coordinates. If compressibility cannot be neglected, it is common to use a linearization so that:

$$\phi = \phi_0(1 + c_r(p - p_0)). \tag{3.2}$$

Since the dimension of the pores is very small compared to any interesting scale for reservoir simulation, one normally assumes that porosity is a piecewise continous spatial function.

3.1.2 Permeability

The *permeability* $\mathbf{K}$ is a measure of the rock's ability to transmit a single fluid at certain conditions. Since the orientation and interconnection of the pores are essential for flow, the permeability is not necessarily proportional to the porosity, but $\mathbf{K}$ is normally strongly correlated to ϕ. A rock with well-connected or large pores which transmits fluids readily is described as *permeable*. Formations with smaller, fewer, or less interconnected pores are described as *impermeable*. The SI-unit for permeability is m^2, but it is commonly represented in Darcy (D). One Darcy is approximately $0.987 \cdot 10^{-12}$ m^2.

Since permeability is dependent on the direction in which the fluid flows, in general $\mathbf{K}$ is a tensor. The medium is described as isotropic (as opposed to anisotropic) if $\mathbf{K}$ can be represented as a scalar function, e.g., if the horisontal permeability is equal to the vertical permeability. Due to seismic fractures in reservoirs, permeability distribution in a reservoir may vary rapidly over several orders of magnitude. This is the case for the test reservoir models we will use in numerical tests (See Chapter 9 on page 73).

Having good estimates of $\mathbf{K}$ gives a better characterization of the flow problem. On small scales, the permeability is a diagonal tensor. Still, though, full tensor permeabilities occur when an upscaling of a reservoir is performed. Upscaling is an important tool in reservoir simulation to solve large systems by means of coarsening the reservoir grid, thereby using small scale permeabilities to construct a tensor which represents a reasonable picture of coarse scale [9, 10, 11]. In this thesis, we only consider diagonal permeability tensors aligned with the Cartesian coordinate directions.

3.2 Incompressible Single-Phase Flow

In this report, we will only consider single-phase flow. Multi-phase flow models are treated in [7].

If we look at a closed volume Ω of a porous medium, we know from the law of conservation of mass that the fluid mass is conserved. A change in the fluid mass within this volume is a result of flux through the boundary $\partial\Omega$ of Ω, or a variation in fluid density or medium porosity. The law of conservation thus gives

$$\frac{\partial}{\partial t}\int_{\Omega}(\phi\rho)dV + \int_{\partial\Omega}(\rho v)\cdot\hat{n}dA = \int_{\Omega} qdV. \quad (3.3)$$

Here, q models sources or sinks, that is, outflow and inflow per volume at desginated well locations. Mass density is denoted by ρ, $\hat{n}$ is the outward-pointing normal vector of the volume, and v is the average flux, also called flow velocity.

Using the Divergence Theorem, we can also write Eq. 3.3 in differential form

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla\cdot(\rho v) = q. \quad (3.4)$$

In addition to Eq. 3.4, which contains the flux velocity v, we need to consider the well-known Darcy's law which relates v to the pressure gradient ∇p and gravity forces.

3.2.1 Darcy's Law

Empirical experiments done by the French engineer Henry Darcy (in 1856), showed that the average flux v through a cross-section of a porous medium is linearly proportional to a combination of the gradient of the fluid pressure p and pull-down effects due to gravity. It is given by

$$v = -\frac{\mathbf{K}}{\mu}(\nabla p + \rho g\nabla z). \quad (3.5)$$

In Eq. 3.5, μ is the fluid viscosity, g the gravitional constant, and z is the upward vertical direction. Darcy's law is valid under non-turbulent fluid flow, that is, when the fluid velocity is low.

3.2.2 Elliptic Pressure Equation

If we assume that the porosity ϕ is constant in time and the flow is incompressible (which means ρ is constant in time), then the first term in

Eq. 3.4 on the preceding page vanishes. Compined with Eq. 3.5 on the previous page, we obtain

$$\nabla \cdot v = \nabla \cdot \left[-\frac{\mathbf{K}}{\mu} (\nabla p + \rho g \nabla z) \right] \equiv \nabla \cdot \left[-\frac{\mathbf{K}}{\mu} \nabla u \right] = \frac{q}{\rho}. \tag{3.6}$$

We will equivalently write Eq. 3.6 as the system (of first order)

$$\begin{aligned} v &= -\frac{\mathbf{K}\nabla}{p}\mu \\ \nabla \cdot v &= f \equiv \frac{q}{\rho}. \end{aligned} \tag{3.7}$$

In 3.7, we have set $\nabla u = \nabla p$, including in p the total driving force. To close the model, we must specify boundary conditions, which may be of Dirichlet type, where p is given at the boundary, or of Neumann type, which specifies the flux $v \cdot \hat{n}$ through the boudary. In the following, we go with the assumption that our reservoir is a closed system, so we specify *no-flow boundary condition* by imposing $v \cdot \hat{n} = 0$ on the reservoir boundary $\partial\Omega$. The only way to put the reservoir out of equilibrium is by adding wells into it, which will be considered in Section 4.2.

Chapter 4

Discretisation of Elliptic Pressure Equation

The elliptic pressure equation in the system 3.7 on the preceding page and no-flow boundary condition on the reservoir boundary will be our initial model. Additional requirements, such as Dirichlet boundary conditions, will be discussed as and when needed. The only technique applied to discretise this model in this report is Mimetic Finite Difference method (MFD). This chapter begins with a discussion of the MFD method. Then we will lead our PDE through the strong and the more general mixed (weak) formulation. After discussing the mixed and the hybrid discretisation principles, we will formulate the final discretized system $\mathbf{Ax} = \mathbf{b}$. We will then modify the system slightly and add wells to the system, both pressure- and rate-constrained wells. This will primarily involve multiple links between non-adjacent cells. Finally, we shall refer to some properties of corner-point grids.

4.1 Mimetic Finite Difference Method (MFD)

Let $\Omega \in \mathbb{R}^3$ be a polyhedral domain, and let n be the outward unit normal to $\partial\Omega$. We restate Eq. 3.7 on the previous page as the strong form:

$$\begin{aligned} v &= -\mathbf{K}\nabla p \quad && \mathbf{x} \in \Omega \\ \nabla \cdot v &= f \quad && \mathbf{x} \in \Omega \\ v \cdot n &= 0 \quad && \mathbf{x} \in \partial\Omega \end{aligned} \tag{4.1}$$

The general idea in MFD is to split the strong formulation of a PDE into a system of first-order PDEs (which Eq. 4.1 already is) and to discretise

this system by giving discrete analog of the usual first-order continuum differential operators, such as gradient ∇, curl $(\nabla\times)$, divergence $(\nabla\cdot)$, etc. These analogs are constructed by *mimicking* important properties of the underlying gemometrical, mathematical and physical models; the continuum operators such as conservation, operator symmetrices, kernels of operators, basic theorems of vector calculus, such as Gauss, Stokes, Green, etc[1].

Motivation behind MFD is that it creates locally conservative schemes, works on unstructured computational grids, and higher order methods for unstrcutured meshes. There are no constraints on the number of vertices of a cell, nor on the angles[2]. Polyhedral meshes naturally arise in the treatment of complex solution domains and heterogeneous materials, e.g. reservoir models[12, 13]. This allows for a large degree of freedom in using unstructured grids consisting of general polyhedral cells to model complex geology[11].

The MFD method was designed to provide accurate approximation of differential operators on general meshes. These meshes may include degenerate elements, as in adaptive mesh refinement methods, non-convex elements, as in moving mesh methods, and even elements with curved edges near curvilinear boundaries[14].

The MFD method has many similarities with a low-order finite element (FE) method. Both methods try to preserve fundamental properties of physical and mathematical models. Various approaches to extend the FE method to non-simplicial elements have been developed over the last decade. Construction of basis functions for such elements is a challenging task and may require extensive analysis of geometry. Contrary to the FE method, the MFD method uses only boundary representation of discrete unknowns to build stiffness and mass matrices. Since no extension inside the mesh element is required, practical implementation of the MFD method is simple for general polygonal meshes[3].

In the discretisation of Eq. 4.1 on the previous page, the scalar function p is represented by one unknown; its average value in each grid cell. The flux v is represented, in each cell, by one unknown on each cell face/edge. Face pressure, denoted by π, on each grid face is also represented by one unknown.

[1] http://www.ima.umn.edu/talks/workshops/5-11-15.2004/shashkov/talk.pdf

[2] http://math.lanl.gov/Research/Highlights/PDF/homfdmmesh.pdf

[3] http://math.lanl.gov/Research/Highlights/mimetic-stokes.shtml

We have

$$\begin{aligned} p \in U(\Omega) : &\quad \text{scalar "pressure" space} \\ v \in H_0^{\text{div}}(\Omega) : &\quad \text{vector "velocity" space,} \end{aligned} \tag{4.2}$$

where U and H_0^{div} are appropriate function spaces. The primary property we intend to mimic is the integration by parts formula

$$\int_\Omega \nabla p \cdot v = -\int_\Omega p \nabla \cdot v + \int_{\partial\Omega} pv \cdot ndS \tag{4.3}$$

We can write this in terms of inner-products:

$$(\nabla p, v)_{H_0^{\text{div}}(\Omega)} = (p, -\nabla \cdot v)_U + \int_{\partial\Omega} pv \cdot ndS \tag{4.4}$$

Given discrete subspaces $U^h \subset U$ and $H_0^{\text{div}^h} \subset H_0^{\text{div}}$, we seek discrete operators $(\nabla\cdot)^h$ and ∇^h such that

$$\left(\nabla^h p, v\right)_{H_0^{\text{div}^h}} = \left(p, -\nabla^h \cdot v\right)_{U^h} + \int_{\partial\Omega} pv \cdot ndS, \tag{4.5}$$

for meaningful inner-products $(\cdot,\cdot)_{H_0^{\text{div}^h}}$ and $(\cdot,\cdot)_{U^h}$.

4.1.1 Mixed Formulation

The name "mixed" is applied to a variety of finite element methods which have more than one approximation space. Typically one or more of the spaces play the role of Lagrange multipliers which enforce constraints. One characteristic of mixed formulations is that not all choices of finite element spaces will lead to convergent approximations. Standard approximation alone is not sufficient to guarantee success. For a mathematical treatment of the well-posedness of mixed methods, see [15].

To discretise Eq. 4.1 on page 14, let us partition Ω into a set of polyhedral-like cells $\mathcal{T} = \{T\}$. We want each component of flux vector $v = (v_x, v_y, v_z)$ to be well-behaved on each cell, and we want it to be continous across cell boundaries. We impose this by making v and $\nabla \cdot v$ square-integrable on each cell, as well as on Ω. This can be expressed by the following function spaces:

$$\begin{aligned} H^{\text{div}}(T) &= \left\{ v \in L^2(T)^d \ : \ \nabla \cdot v \in L^2(T) \right\} \\ H_0^{\text{div}}(\mathcal{T}) &= \left\{ v \in H^{\text{div}}(\cup_{T\in\mathcal{T}} T) \ : \ v \cdot n = 0 \ \text{ on } \ \partial\Omega \right\} \\ H_0^{\text{div}}(\Omega) &= H_0^{\text{div}}(\mathcal{T}) \cap H^{\text{div}}(\Omega) \end{aligned} \tag{4.6}$$

Let us multiply the first equation in 4.1 on page 14 with an arbitrary function from $H_0^{\text{div}}(\Omega)$, the second with an arbitrary function from $L^2(\Omega)$, and integrate on Ω. Integrating by parts the first equation results in the weak form:

$$\begin{aligned} \sum_{T\in\mathcal{T}} \int_T q_1 \cdot v - \sum_{T\in\mathcal{T}} \int_T \mathbf{K}\nabla q_1 \cdot p &= 0 \qquad \forall q_1 \in H_0^{\text{div}}(\Omega) \\ \int_\Omega q_2 \nabla \cdot v &= \int_\Omega q_2 f \quad \forall q_2 \in L^2(\Omega) \end{aligned} \tag{4.7}$$

We introduce three bilinear forms:

$$\begin{aligned} (\cdot,\cdot) &: L^2(\Omega) \times L^2(\Omega) \mapsto \mathbb{R} \\ b(\cdot,\cdot) &: H_0^{\text{div}}(\Omega) \times H_0^{\text{div}}(\Omega) \mapsto \mathbb{R} \\ c(\cdot,\cdot) &: H_0^{\text{div}}(\Omega) \times L^2(\Omega) \mapsto \mathbb{R}. \end{aligned} \tag{4.8}$$

In the general mixed formulation of 4.1 on page 14, one seeks a pair of functions $(p, v) \in L^2(\Omega) \times H_0^{\text{div}}(\Omega)$ such that

$$\begin{aligned} b(q_1, v) - c(q_1, p) &= 0 \qquad \forall q_1 \in H_0^{\text{div}}(\Omega) \\ c(v, q_2) &= (f, q_2) \quad \forall q_2 \in L^2(\Omega) \end{aligned} \tag{4.9}$$

This is the *point of departure* for both Mixed Finite Element method and Mimetic Finite Difference method. The solution of 4.9 is a saddle point of the Langrange functional

$$L(v, p) = \frac{1}{2} b(v, v) - c(v, p) + (p, f). \tag{4.10}$$

This is apparent from variational analysis setting $dL = \frac{\partial L}{\partial v} dv + \frac{\partial L}{\partial p} dp = 0$, and then letting the variation $q_1 = dv$ and $q_2 = dp$ be arbitrary. Looking at the determinant of the Hessian matrix $\Delta L = \frac{\partial^2 L}{\partial v^2}\frac{\partial^2 L}{\partial p^2} - \left(\frac{\partial^2 L}{\partial v \partial p}\right) = -c(\vec{1}, 1)^2$, we see that the stationary point is a saddle point. Consequently, discretisation based on mixed formulation leads to indefinite linear system. Indefinite systems results in a linear system with both positive and negative eigenvalues. Such linear systems require special solvers and are often considerably harder to solve. We will therefore introduce a different technique that involves a positive definite system.

4.1.2 Hybrid Formulation

Continuity of flux through the cell faces in 4.9 is reflected in the function space $H_0^{\text{div}}(\Omega)$. In the alternative *hybrid formulation*, we allow this space

to also include discontinuous functions across the cell boundaries, which means that we now consider v to be in $H_0^{\text{div}}(\mathcal{T})$. Instead, a slight change is imposeed on the mixed formulation by adding an extra set of equations. These additional equations impose the continuity of the flux through the cell faces by the Langrange multiplier method, where the pressure at the cell faces plays the role of Lagrange multipliers. This method preserves both v and p, but incorporates pressure values at cell faces in the linear system.

We replace the mixed formulation 4.9 on the previous page with the following hybrid formulation: find $(v, p, \pi) \in H_0^{\text{div}}(\mathcal{T}) \times L^2(\Omega) \times L^2(\partial\mathcal{T}/\partial\Omega)$ such that

$$\begin{aligned} b(u,v) - c(u,p) + d(u,\pi) &= 0 & \forall u \in H_0^{\text{div}}(\mathcal{T}) \\ c(v,q) &= (f,q) & \forall q \in L^2(\Omega) \\ d(v,\mu) &= 0 & \forall q \in L^2(\partial\mathcal{T}/\partial\Omega) \end{aligned} \tag{4.11}$$

In 4.11, the bilinear form $d(v,\pi) : H_0^{\text{div}}(\Omega) \times L^2(\partial T) \mapsto \mathbb{R}$ is given by

$$d(v,\pi) = \sum_{T \in \mathcal{T}} \int_{\partial T} \pi v \cdot n_T dS \tag{4.12}$$

The Lagrange function for the hybrid formulation is now

$$L(v,p) = \frac{1}{2} b(v,v) - c(v,p) + (f,p) + (d,\pi), \tag{4.13}$$

and the Hessian is a 3×3 matrix, the eigenvalues of which should be greater than or equal to 0. The system is still positive semi-definite since we have not assigned any Dirichlet boundary conditions to the system. We write b, c, and d, for $b(\vec{1}, 1)$, $c(\vec{1}, 1)$, and $d(\vec{1}, 1)$, respectively. The Hessian then expresses

$$\Delta L = \begin{vmatrix} b-\lambda & c & d \\ c & -\lambda & 0 \\ d & 0 & -\lambda \end{vmatrix} . = \lambda((b-\lambda) - c^2 + d^2) = 0,$$

which implies that λ is nonzero or

$$\lambda = \frac{b}{2} \pm \sqrt{\frac{b^2}{4} - (d^2 - c^2)}. \tag{4.14}$$

Since $b(\cdot,\cdot)$ is positive definite and $c = 0$, the eigenvalues are non-negative.

To discretise 4.11, one selects finite-dimensional subspaces $V \subset H_0^{\text{div}}(\mathcal{T})$, $U \subset L^2(\Omega)$, and $\Pi \subset L^2(\partial\mathcal{T}/\partial\Omega)$, and seeks $(v, p, \pi) \in V \times U \times \Pi$ such that 4.11 holds for all $(u, q, \mu) \in V \times U \times \Pi$. The space V is often what is required in mixed FE methods (which requires the definition of V's basis functions), but the MFD method approaches the discretisation by a slight twist, which is what makes MFD easier on general grids than mixed FE.

4.1.3 Discrete Formulation

In MFD, the discretisation is introduced already in the variational formulation in the form of a discrete inner-product. Mathematically, this means that the subspace V in $H_0^{\text{div}}(\mathcal{T})$ is replaced by a discrete subspace of $L^2(\partial\mathcal{T})$, and the associated bilinear form $b(\cdot,\cdot)$ is replaced by a bilinear form that acts on $L^2(\partial\mathcal{T}) \times L^2(\partial\mathcal{T})$. This means that instead of seeking an unknown velocity field v that acts on each cell T, one seeks a set of fluxes defined over cell faces ∂T. This is similar to Finite Difference Methods, where (only) differential operators are discretized.

4.1.4 An MFD Formulation

The subspace U in which the cell pressures p reside consists of cell-wise constant functions:

$$U = \text{span}\left\{\chi_m : T_m \in \mathcal{T}\right\}, \quad \chi_m(x) = \begin{cases} 1 & \text{if } x \in T_m \\ 0 & \text{otherwise} \end{cases}$$

For $v \in H_0^{\text{div}}(\mathcal{T})$ and $p \in U$, this implies

$$c(v,p) = \sum_m p_m \int_{T_m} \nabla \cdot v = \sum_m p_m \int_{\partial T_m} v \cdot n_{T_m} ds. \tag{4.16}$$

The last sum shows that an explicit representation of flux v inside each cell is not needed, only values on the cell boundaries. Face pressure π is also located on cell boundaries, and thus the same is true for $d(v,\pi)$ for any $\pi \in \Pi \subset L^2(\partial\mathcal{T})$. The subspace Π consists of face-wise constant functions:

$$\Pi = \text{span}\left\{\pi_j^i : |\gamma_j^i| > 0, \gamma_j^i = \partial T_i \cap \partial T_j\right\}, \quad \pi_i^j(x) = \begin{cases} 1 & \text{if } x \in \gamma_j^i \\ 0 & \text{otherwise} \end{cases}$$

The third subspace we need to consider is $V \subset H_0^{\text{div}}(\mathcal{T})$. In mixed/hybrid finite element methods ones usually discretizes the space in which V is located, and the differential operators remain untouched; computation of $b(\cdot,\cdot)$ requires explicit representations of the flux velocity in each cell. This is not trivial for irregular grids, such as general corner-point grids. MFD finds instead a replacement $m(\cdot,\cdot)$ that mimics $b(\cdot,\cdot)$ (and avoids $H_0^{\text{div}}(\mathcal{T})$).

For the moment, let us assume that a "velocity" basis function ψ_i^m is associated with each face γ_i^m of cell T_m. Remember that there is only one face

pressure π associated with each cell face, while there are two flux velocities for internal cell faces, and a single value for boundary faces, which is zero as a result of no-flow boundary conditions. MFD with the hybrid formulation 4.11 on page 18 implies the following linear system:

$$\begin{bmatrix} \mathbf{B} & -\mathbf{C}^T & \mathbf{\Pi}^T \\ \mathbf{C} & \mathbf{0} & \mathbf{0} \\ \mathbf{\Pi} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} v \\ p \\ \pi \end{bmatrix} = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \tag{4.18}$$

The matrices $\mathbf{C}$ and $\mathbf{\Pi}$ are now given by

$$\mathbf{C} = [c(\psi_i^m, \chi_n)] \quad \text{and} \quad \mathbf{\Pi} = [d(\psi_k^m, \mu_j^i)], \tag{4.19}$$

and the matrices $\mathbf{B}$ is given by

$$\mathbf{B} = [m(\psi_i^m, \psi_j^n)]. \tag{4.20}$$

Note that $b(\cdot, \cdot)$ has been replaced by an approximative bilinear form $m(\cdot, \cdot)$. This will be explained soon.

4.1.5 Schur-complement Reduction

The hybrid system 4.21 is indefinite, but $b(\psi_i^m, \psi_j^n)$ and $m(\psi_i^m, \psi_j^n)$ are nonzero only for $n = m$, that is, for the same cell, and by numbering the flux velocities on a cell-by-cell basis, the matrix $\mathbf{B}$ becomes block-diagonal, where each block has the same dimension as the number of faces the corresponding cell has. $\mathbf{B}$ is invertible since the system is positive definite, and a Schur-complement reduction with respect to $\mathbf{B}$ eliminates v and results in the system

$$\begin{bmatrix} \mathbf{D} & -\mathbf{F}^T \\ \mathbf{F} & -\mathbf{\Pi}\mathbf{B}^{-1}\mathbf{\Pi}^T \end{bmatrix} \begin{bmatrix} p \\ \pi \end{bmatrix} = \begin{bmatrix} g - \mathbf{C}\mathbf{B}^{-1}f \\ h - \mathbf{\Pi}\mathbf{B}^{-1}f \end{bmatrix}, \tag{4.21}$$

where $\mathbf{D} = \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^T$ and $\mathbf{F} = \mathbf{\Pi}\mathbf{B}^{-1}\mathbf{C}^T$. $\mathbf{D}$ is a diagonal matrix resulting from the fact that $c(\psi_i^m, \chi_n) = 0$ for $n \neq m$. A Schur-complement with respect to $\mathbf{D}$ eliminates p and gives us the following semi-definite linear system

$$\mathbf{S}\pi = \mathbf{R} = h - \mathbf{F}\mathbf{D}^{-1}(g - \mathbf{C}\mathbf{B}^{-1}f) - \mathbf{\Pi}\mathbf{B}^{-1}f \tag{4.22}$$

The system is only semi-definite because no Dirichlet face pressure boundary conditions has been specified yet. Here, $\mathbf{S} = \mathbf{F}\mathbf{D}^{-1}\mathbf{F}^T - \mathbf{\Pi}\mathbf{B}^{-1}\mathbf{\Pi}^T$. We actually solve $-\mathbf{S}\pi = -\mathbf{R}$, because 4.22 gives a negative semi-definite system[4]. Once π is computed, one can easily compute p and v by solving a diagonal and a block-diagnonal system, respectively. In 4.21, f in the right-hand side is the gravity effects. For simplicity, we have set $f = 0$.

[4] Pointed out to me by my supervisor Bård Skaflestad

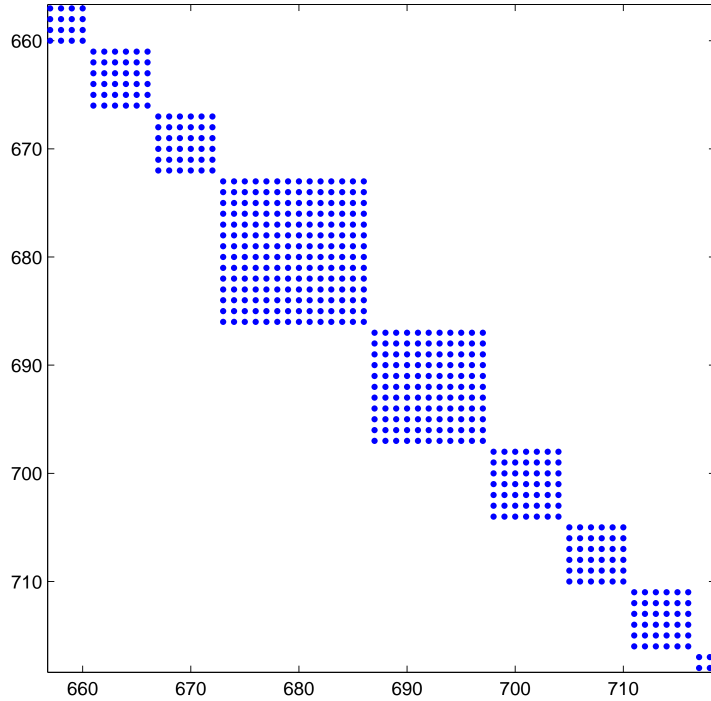


Fig. 4.1: The matrix $\mathbf{B}$ is block-diagonal, where each block corresponds to a single cell and the dimension is equal to the number of cell faces. In a corner-point grid, the number of cell faces is variable.

4.1.6 Motivation for the Approximate Bilinear Form

To this end, denote by $\mathcal{F}_m$ the set of cell faces of cell T_m and expand v and u in the basis $\{\psi_i^m : F_m \in \mathcal{F}_m, T_m \in \mathcal{T}\}$:

$$v = \sum_{i,m} v_i^m \psi_i^m \quad \text{and} \quad v = \sum_{i,m} u_i^m \psi_i^m. \tag{4.23}$$

Since $b(\psi_i^m, \psi_j^n)$ is nonzero only if $n = m$, we may write

$$b(u, v) = \sum_{T_m \in \mathcal{T}} u_m^T \mathbf{B}_m v_m. \tag{4.24}$$

As from 4.21 on the previous page, $\mathbf{B}_m$ is the matrix in the block-diagonal $\mathbf{B}$ associated with cell T_m (See Figure 4.1), and $v_m, u_m \in \mathbb{R}^{N_m}$ where N_m

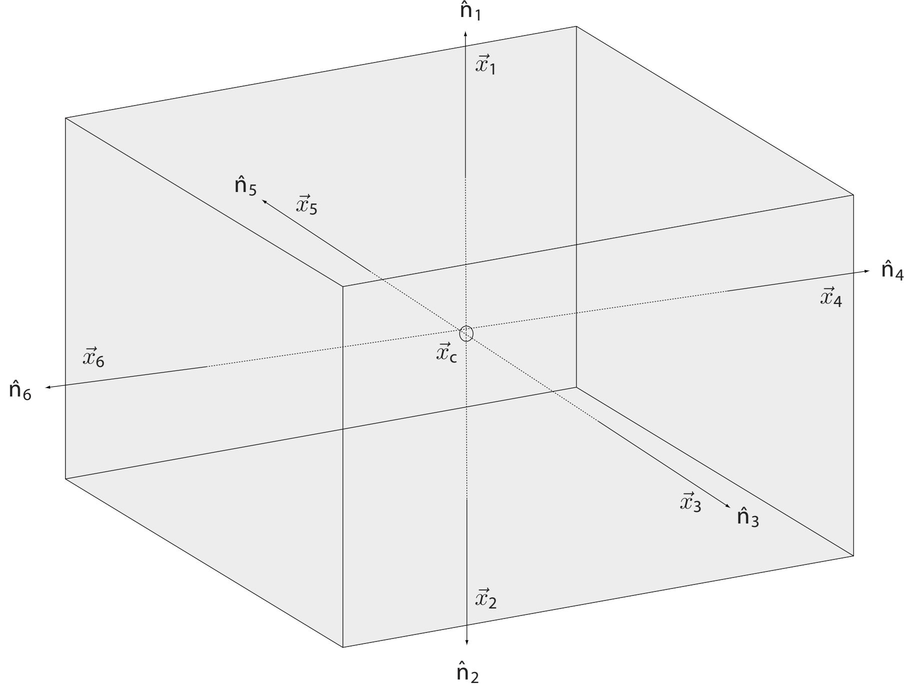


Fig. 4.2: Figure showing a Cartesian cell with six faces. The pressure p and π at the cell center and cell boundaries, respectively, are supposed to be linearly dependent.

is the number of faces of T_m. We have already mentioned some of the reasons why MFD is appropriate on general grids; mainly imposed by the difficulty of finding appropriate subspaces of $H_0^{\text{div}}(\mathcal{T})$. In the following, we will motivate the form of the discrete analog of $b(\cdot,\cdot)_m$ on each cell, equivalent to $\mathbf{B}_m$ (as well as $\mathbf{C}$ and $\mathbf{\Pi}$). Since 4.21 on page 20 uses the symbol $\mathbf{B} = \text{diag}\{\mathbf{B}_1, \mathbf{B}_2, \cdots\}$, we will continue to use $\mathbf{B}_m$ also for the approximate matrix. We present a geometric approach to find the approximate $\mathbf{B}_m$. We will use Finite Difference method to obtain a system of equations in the same form as 4.21 on page 20. This system gives the exact solution of the hybrid form if p is linear and $\mathbf{K}$ is constant in each cell.

Consider the three-dimensional illustration of a Cartesian grid cell e in Figure 4.2. The flux is only used at the cell faces. We can write the flux $v|_i$ that goes through face i as a scalar v_i times an average unit normal vector $\hat{n}_i$

$$v|_i = v_i\hat{n}_i = \int_{\gamma_i} \vec{v} \cdot \hat{n} \tag{4.25}$$

From the equation system 4.1 on page 14, we know that the flux v is a linear transformation of ∇p. Assume therefore that the fluxes through each face i can be written as a linear transformation of the drops in pressure at the face compared to the mass-center of the cell. Let us create a vector v_f containing the values v_i at the n faces of the cell; $v_f = [v_1, v_2, \cdots, v_n]^T$, then there is a matrix $\mathbf{B}_m^{-1}$ such that

$$v_f = \mathbf{B}_m^{-1}(p_c - \pi_f), \tag{4.26}$$

where the right-hand side gives the drops in driving force from the mass-center $\vec{x}_c$ of the cell to the *centroid* of each face (p_c here is a vector of the same length as v_f repeating the cell centroid pressure). Notice that we use two different symbols, π and p, to distinguish face pressure from the centroid pressure, respectively. Although MFD is able to handle curved surfaces, we will henceforth consider only planar cell faces.

If we assume that the pressure is linear internally to each cell with $(p_c - \pi_i) = (\vec{x}_i - \vec{x}_c)\vec{a}$, we get,

$$v_i = -\mu^{-1} A_i \hat{n}_i^T \mathbf{K}_e \nabla p = -\mu^{-1} A_i \hat{n}_i^T \mathbf{K}_e \vec{a}, \tag{4.27}$$

where A_i is the area of face i. This gives two equations for v_f in each cell and we get

$$v_f = \mathbf{B}_m^{-1} \underbrace{\begin{bmatrix} \vec{x}_1 - \vec{x}_c \\ \vec{x}_2 - \vec{x}_c \\ \vec{x}_3 - \vec{x}_c \\ \vdots \\ \vec{x}_n - \vec{x}_c \end{bmatrix}}_{\mathbf{X}} \vec{a} = \mu^{-1} \underbrace{\begin{bmatrix} A_1 \hat{n}_1^T \\ A_2 \hat{n}_2^T \\ A_3 \hat{n}_3^T \\ \vdots \\ A_n \hat{n}_n^T \end{bmatrix}}_{\mathbf{N}_e} \mathbf{K}_e \vec{a} \Rightarrow \mathbf{B}_m^{-1}\mathbf{X} = \mu^{-1}\mathbf{N}_e\mathbf{K}_e. \tag{4.28}$$

Lemma 1. *We find a solution of* 4.28 *by setting* $\mathbf{B}_m^{-1} = \frac{\mu^{-1}}{|e|}\mathbf{N}_e\mathbf{K}_e\mathbf{N}_e^T + \mathbf{T}_2$, *where* $\mathbf{T}_2\mathbf{X} = \mathbf{0}$ and $|e|$ is the cell volume.

Proof. This is clearly the case if $\mathbf{N}_e^T\mathbf{X} = |e|\mathbf{I}_3$ for the three-dimensional identity. We then check index z_{ij} in the matrix $\mathbf{N}_e\mathbf{X} = [z_{ij}]$, Writing $\mathbf{N}_e = [n_1|n_2|n_3]$ and $\mathbf{X} = [x_1|x_2|x_3]$ and using the superscript $\hat{n}_k^{(i)}$ to represent the i-th Cartesian coordinate of $\hat{n}_k$ gives

$$z_{ij} = n_i^T x_j = \sum_{k=1}^{n} A_k n_k^{(i)} (\vec{x}_k - \vec{x}_c)^{(j)}. \tag{4.29}$$

We can extend $n_k^{(i)}$ to be written as $n_k^{(i)} = \hat{e}_i \hat{n}_k$, where $\hat{e}_k$ is the Cartesian unit vector in the i-th direction. Since $(\vec{x}_k - \vec{x}_c)$ is the average unit vector from

the center to a cell face, we can use the Divergence Theorem to conclude

$$\begin{aligned} z_{ij} &= \sum_{k=1}^{n} A_k \hat{e}_i \cdot \hat{n}_k \frac{1}{A_k} \int_{\gamma_k} (\vec{x} - \vec{x}_c)^{(j)} dA \\ &= \sum_{k=1}^{n} \hat{e}_i \cdot \int_{\gamma_i} (\vec{x} - \vec{x}_c)^{(j)} \cdot \hat{n}_k dA \\ &= \hat{e}_i \cdot \int_{\Omega} (\nabla \cdot \vec{x} - \nabla \cdot \vec{x}_c)^{(j)} dV \\ &= \hat{e}_i \cdot \hat{e}_j V = \delta_{ij} V, \end{aligned} \tag{4.30}$$

and the proof is complete. □

Equation 4.26 on the preceding page gives the following expression for each cell in the grid:

$$\mathbf{B}_m v_f - p_c + \pi_f = 0 \tag{4.31}$$

This equation together with the other two equations according to the hybrid formulation 4.11 on page 18, assembles the system 4.21 on page 20. However, $\mathbf{B}_m^{-1}$ is positive semi-definite and not strictly positive definite. We are therefore not guaranteed that $\mathbf{B}_m^{-1}$ is invertible for any $\mathbf{T}_2$ satifying Lemma 1. The following Theorem is presented by F. Brezzi in [12] and gives a recipe for choosing $\mathbf{T}_2$ in a way that will ensure positive definiteness.

Theorem 1. *Let* $\mathbf{F}$ be an $n \times (n-d)$ *matrix whose columns span the null space of the matrix* $\mathbf{X}^T$*, so that* $\mathbf{F}^T\mathbf{X} = \mathbf{0}$*. Then for every* $(n-d) \times (n-d)$ *symmetric positive definite matrix* $\mathbf{U}$ *we can set*

$$\mathbf{B}_m^{-1} = \frac{\mu^{-1}}{|e|} \mathbf{N}_e \mathbf{K}_e \mathbf{N}_e^T + \mathbf{F}\mathbf{U}\mathbf{F}^T, \tag{4.32}$$

which makes $\mathbf{B}_m^{-1}$ *symmetric and positive definite, satisfying* $\mathbf{B}_m^{-1}\mathbf{X} = \mu^{-1}\mathbf{N}_e\mathbf{K}_e$

Proof. Lemma 1. with $\mathbf{T}_2 = \mathbf{F}\mathbf{U}\mathbf{F}^T$ states that $\mathbf{T}\mathbf{X} = \mathbf{N}\mathbf{K}_e$ since $\mathbf{T}_2\mathbf{X} = \mathbf{F}\mathbf{U}\mathbf{F}^T\mathbf{X} = \mathbf{0}$. The matrix $\mathbf{B}_m^{-1}$ is positive definite by construction, and we only need to show that it is non-singular.

If we assume that there exists a nonzero vector $v \neq 0$ such that $\mathbf{B}_m^{-1}v = 0$, then we have

$$|| \frac{\mu^{-1/2}}{\sqrt{|e|}} \sqrt{\mathbf{K}_e} \mathbf{N}_e^T v||_2^2 + ||\sqrt{\mathbf{U}}\mathbf{F}^T v||_2^2 = 0. \tag{4.33}$$

This implies that $\mathbf{N}_e^T v = \vec{0}$ and $\mathbf{F}^T v = 0$, since both $\mathbf{K}_e^{1/2}$ and $\mathbf{U}^{1/2}$ are positive definite. This means that $v \in \ker(\mathbf{F}^T) = \{\mathrm{im}(\mathbf{F})\}^T = \{\ker(\mathbf{X})\}^T =$

im($\mathbf{X}$), and we can write v as a linear combination of the columns of $\mathbf{X}$ on the form $v = \mathbf{X}w$ for a vector w. Hence, $\mathbf{N}^T v = \mathbf{N}^T \mathbf{X} w$. From Lemma 1, $\mathbf{N}^T \mathbf{X} = |e| \mathbf{I}_d$, which implies that w has to be zero since $\mathbf{N}^T v = 0$. Thus, v is also zero, which is a contradiction to our assumption of v. Hence, the matrix $\mathbf{B}_m^{-1}$ is SPD. □

Structure of cell matrices C and Π is simple according to Eq. 4.31 on the previous page; on each cell T_e, $\mathbf{C}_e$ is a matrix of dimension $1 \times N_m$, and Π_e is the identity matrix of dimension$N_m \times N_m$. N_m is the number of cell faces.

This is the cell matrix we have used in this report:

$$\mathbf{B}_e^{-1} = \frac{1}{|e|}(\mathbf{N}_e \mathbf{K}_e \mathbf{N}_e^T + \frac{6}{d}\text{trace}(\mathbf{K}_e)\mathbf{D}_A(\mathbf{I}_{N_m} - \mathbf{Q}_e \mathbf{Q}_e^T)\mathbf{D}_A) \tag{4.34}$$

Matrix $\mathbf{D}_A$ is the diagonal matrix containing face areas, and $\mathbf{I}_{N_m}$ is the identity with dimension N_m. Matrix $\mathbf{Q}_e$ is an orthonormal matrix, composed by applying (for instance) the Modified Gram-Schmidt orthogonalization algorithm to the matrix $\mathbf{D}_A \mathbf{C}_e$.

4.2 Well Modeling

As stated earlier, the linear system 4.21 on page 20 is semi-definite because we have not imposed any Dirichlet conditions on the system; the pressure is floating. Before we modify this system to include wells, let us see how the system will look like when we impose Dirichlet conditions. Our hybrid system looks like

$$\begin{bmatrix} \mathbf{B} & -\mathbf{C}^T & \mathbf{\Pi}^T \\ \mathbf{C} & \mathbf{0} & \mathbf{0} \\ \mathbf{\Pi} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} v \\ p \\ \pi \end{bmatrix} = \begin{bmatrix} 0 \\ g \\ h \end{bmatrix}, \tag{4.35}$$

where we set $f = 0$ explicitly. The right-hand side h is the Neumann bondary conditions. To incorporate pressure (Dirichlet) boundary conditions, we split the vector $\pi^T = (\pi_{I,N}^T \quad \pi_D^T)$ and the matrix $\Pi = (\Pi_{I,N} \quad \Pi_D)$ in two parts where the first corresponds to the interior and Neumann faces, and the second corresponds to the Dirichlet faces. That is, the Dirichlet contribution is shifted into the right-hand side. Then the system reduces to

$$\begin{bmatrix} \mathbf{B} & -\mathbf{C}^T & \mathbf{\Pi}_{I,N}^T \\ \mathbf{C} & \mathbf{0} & \mathbf{0} \\ \mathbf{\Pi}_{I,N} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} v \\ p \\ \pi_{I,N} \end{bmatrix} = \begin{bmatrix} -\mathbf{\Pi}_D^T \pi_D \\ g \\ h \end{bmatrix}, \tag{4.36}$$

and the final SPD system is obtained according to Schur-complement reduction as in 4.22 on page 20 (with respect to $\mathbf{B}$ and then $\mathbf{D}$).

In the following, we will represent wells as boundary conditions. Hence, a well w with boundary γ_w is conceptually represented as a *hole* in our domain Ω. We consider both pressure- and rate-constrained wells. Pressure-constrained wells result in Dirichlet conditions, while rate-constrained wells give Neumann conditions. A well w consists of a number of perforations located at different locations in the reservoir grid model. We assume that wells do not intersect, and that each perforation goes through the centroid of a grid cell.

Since typical wells in reservoirs have small diameters compared to the size of the cells, it is common to employ a well index or productivity index WI to relate the local flow rate q to the difference in well pressure (bottom hole pressure) p_w and numerically computed pressure p_E in the perforated cell by

$$-q = \lambda_t(s_E)WI(p_E - p_w), \tag{4.37}$$

where $\lambda_t(s)$ is the totalt mobility. Total mobility is a measure of how easily fluid flows with respect to each other. Total mobility is important when we have multi-phase equations. It is a positive number in each cell through which a well is perforated. Depending on units of measurement, it often ranges between 0.001 and 1000. Since we consider only single-phase fluid, we hereby assume $\lambda_t = 1$

Productivity index WI is a parameter that is part of the Peaceman's well model. It measures how strongly the flow in a well affects the flow of cells passing through the well (and vice versa). It is a positive number in each perforation (assuming that the permeability is nonzero). For a vertical well in a Cartesian cell with dimensions $\Delta X \times \Delta Y \times \Delta Z$ is given by

$$WI = \frac{2\pi k \Delta z}{\ln(r_0/r_w)}. \tag{4.38}$$

For isotropic media, k is given by $\mathbf{K} = k\mathbf{I}$, and $r_0 = 0.14(\Delta x^2 + \Delta y^2)^{1/2}$. Here, r_0 is the *effective well-cell radius*, and can be interpreted as (ideally) the radius at which the actual pressure equals the numerically computed pressure. For validity of the index and other considerations, see [16].

Consider a system containg N_w wells $w_1, \dots, w_{N_w}$. For a well w_k, let n_k be the number of cells perforated by the well, and denote these cells by $E_{k_i}, i = 1, \dots, n_k$. Furthermore, let WI_i^k be the well index corresponding to the perforation of well w_k in cell E_{k_i}. The set of equations for all wells is the

given, for $k = 1, \ldots, N_w$

$$
\begin{aligned}
-q_i^k &= \lambda_t(s_{k_i}) W I_i^k (p_{E_{k_i}} - p_{w_k}), \quad i = 1, \ldots, n_k \\
p_{\text{tot}}^k &= \sum_{i=1}^{n_k} q_i^k.
\end{aligned}
\tag{4.39}
$$

Assuming no-well boundary conditions (our reservoir is a closed system) except at wells and that there are no additions sources (we set $g = 0$ in 4.21 on page 20), the system 4.21 on page 20, coupled with 4.39, results in an expanded linear system

$$
\begin{pmatrix}
\mathbf{B} & \mathbf{0} & -\mathbf{C}^T & \mathbf{\Pi}^T & \mathbf{0} \\
\mathbf{0} & -\mathbf{B}_w & -\mathbf{C}_w & \mathbf{0} & \mathbf{D}_w \\
\mathbf{C}^T & -\mathbf{C}_w^T & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{\Pi} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0}^T & -\mathbf{D}_w^T & \mathbf{0} & \mathbf{0} & \mathbf{0}
\end{pmatrix}
\begin{pmatrix} v \\ q_w \\ p \\ \pi \\ p_w \end{pmatrix}
=
\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -q_{w,\text{tot}} \end{pmatrix}.
\tag{4.40}
$$

This system is a direct result of incorporating the equation system 4.39 into 4.21 on page 20. Here, each well perforation results in an additional face at the centroid of the cell (theoretically the perforation could be anywhere in the cell), and the linear system 4.40 treats these perforations separately from the other cell faces, giving rise to $\mathbf{B}_w$, $\mathbf{C}_w$ and $\mathbf{D}_w$. Here, these matrices has the form

$$
\mathbf{B}_w = \begin{pmatrix} \mathbf{B}_1 & & \\ & \ddots & \\ & & \mathbf{B}_{N_w} \end{pmatrix},
\mathbf{C}_w = \begin{pmatrix} \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{N_w} \end{pmatrix},
\mathbf{D}_w = \begin{pmatrix} \mathbf{d}_1 & & \\ & \ddots & \\ & & \mathbf{d}_{N_w} \end{pmatrix}.
\tag{4.41}
$$

Matrix $\mathbf{B}_k$ is $n_x \times n_k$ diagonal matrix containging well indices WI for all perforations for well w_k, according to the numbering we use for these additional perforation faces. Matrix $\mathbf{C}_w$ is the sparse $n_w \times N$ matrix having unit entries in positions $(i, k_i) = 1, \ldots, n_k$, and $\mathbf{d_w}$ is a $n_k \times 1$ vector with all entries equal to one. The vectors q_w, p_w and $q_{w,\text{tot}}$ contain local well rates, well pressure, and total well rates, respectively. A well is pressure-constrained when p_{w_k} is given, and rate-constrained when q_{tot^k} is given. For pressure-constrained wells, the system reduces according to 4.36 on page 25.

In the actual implementation, we have merged the matrix $\mathbf{B}_w$ into $\mathbf{B}$, $\mathbf{D}_w$ into $\mathbf{\Pi}$ and the matrix $\mathbf{C}_w$ into $\mathbf{C}$. This can be done because each well perforation is modeled as an additional face at the centroid of the corresponding cell. A well introduces a system coupling non-adjacent cells, and all perforations of a given well represent the same pressure unknown; well pressure, also called

bottom hole pressure. Each perforation has its own perforation flux. In the reduced system 4.22 on page 20, the dimension of the matrix $\mathbf{S}$ increases by the same number as there are wells.

If a well goes through cell E, the cell matrix $\mathbf{B}_E$ in 4.21 on page 20 look like

$$\mathbf{B}_E = \begin{bmatrix} \mathbf{B}_E & 0 \\ 0 & WI^{-1} \end{bmatrix}, \tag{4.42}$$

that is, the dimension of $\mathbf{B}_E$ increases by one. Note that the local numbering of cell faces puts the well perforation at the end (even though any numbering can be used). If N is the number of global faces, global numbering of well perforations begin at $N+1$. The dimension of $\mathbf{C}_E$ and $\mathbf{\Pi}_E$ is also increased by one.

4.3 Corner-point Grids

In reservoir engineering, the reservoir is modelled in terms of a three-dimensional grid, in which the layered structure of sedimentary beds in the reservoir is reflected in the geometry of the grid cells [10, 11]. The physcial properties of the reservoir rock (porosity and permeability) are represented as constant values inside each grid cell. Due to the highly heterogeneous nature of porous rock formations, geomodels tend to have strongly irregular geometries and very complex hydraulic connectivities.

To model such geological structures, a standard approach is to introduce what is called a corner-point grid. A corner-point grid consists of a set of hexahedral cells that are aligned in a logical Cartesian fashion. One horisontal layer in the grid is then assigned to each sedimentary bed to be modelled. In its simplest form, a corner-point grid is specified in terms of a set of vertical or inclined pillars defined over an Cartesian 2D-mesh in the lateral direction. Each cell in the volumetric corner-point grid is restricted by four pillars and is defined by specifying the eight corner points of the cell, two on each pillar. The corner-point format easily allows for degeneracies in the cells and discontinuities, for example, fracture and faults, across faces. Fractures are cracks in the rock, across which there has been no movement. Faults are fractures across which the layers in the rock have been displaced. Faults are usually modeled as hyperplanes, i.e., as surfaces. Across fault-faces, the corner-point grids are generally non-conforming, having non-matching interfaces.

Hence, using corner-point format it is possible to construct very complex geological models that match the geologist's perception of the underlying

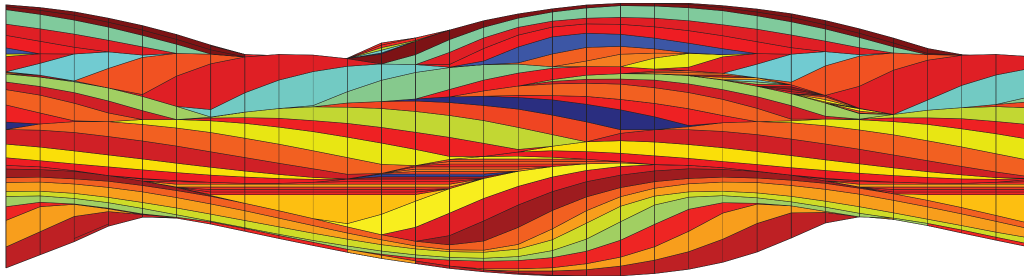


Fig. 4.3: Side-view of a corner-point grid showing vertical pillars and different sedimentary layers showing complex strucutre.

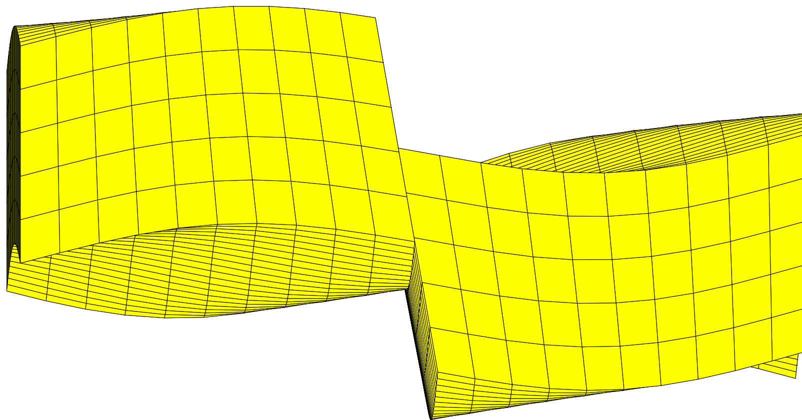


Fig. 4.4: Figure showing a simple corner-point grid model with single seismic fault. The fault creates a *narrow passage*, creating a narrowing of the permeable area.

rock formations. Figure 4.3 shows a side-view of such a corner-point grid. Notice the occurence of degenerate cells with less than eight non-identical corners where the beds are partially eroded away. Some cells also disappear completely and hence introduce new connections between cells that are not neighbours in the underlying logical Cartesian grid.

Corner-point cells may have zero volume, which introduces coupling between non-adjacent cells and gives rise to discretisation matrices with complex sparsity patterns. The presence of degenerate cells, in which the corner-points collapse in pairs, means that the cells will generally be polyhedral and possibly contain both triangular and bilinear faces. This calls for a very flexible discretisation that is not sensitive to the geometry of each cell or the

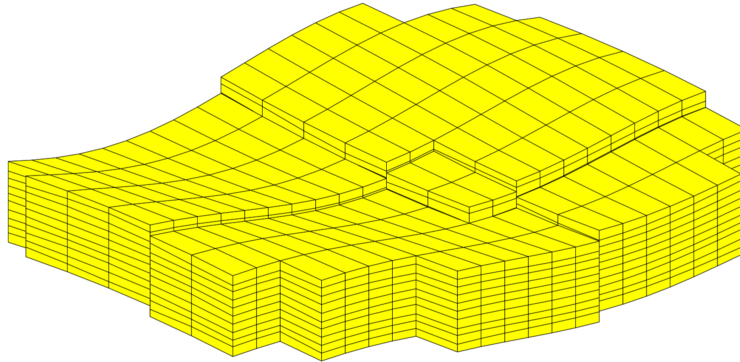


Fig. 4.5: Figure showing a simple corner-point grid model showing two faults. This is an example of *fault-crossing*, which gives high condition numbers of the linear system.

number of faces and corner points.

Chapter 5

Conjugate Gradient Method

In this brief chapter, we discuss the linear conjugate gradient method (CG) to solve the SPD linear system we have discussed in the previous chapters. We will also refer to important convergence results, and then discuss preconditioning techniques used for improvement of the convergence rate.

The conjugate gradient method of Hestens and Stiefel was orignially developed as a direct method designed to solve an $n \times n$ SPD linear system[17]. As a direct method it is generally inferior to Gaussian elimination with pivoting since both methods require n steps to determine a solution, and the steps of the CG method are more computationally expensive than in Gaussian elimination. However, the CG method is particularly advantageous when employed as an iterative method for large sparse systems[4]. Gaussian and other direct methods are not very effective on sparse matrices; they alter the coefficient matrix themselves when solving the system, which causes fill-in. Iterative techniques work with the original matrix and rely on repeatedly applying the matrix on vectors, and can therefore fully exploit the sparsity of a linear system. The CG method sometimes approaches the solution quickly, as we will discuss.

The systems resulting from partial differential equations are usually very sparse, and realistic models result in millions of unknowns, making them improper for direct methods. Fortunately, the larger the system, the more impressive the CG method becomes since it significantly reduces the number of iterations required.

5.1 Theory and Background

CG is well-known as an iterative solver, but can also be studied as a type of *line search method*[4]. Line search metods are, in addition to for example trust-region methods, used to find solutions of problems such as $\min_x f(x)$ where the domain is constrained or unconstrained. In line search strategy, the algorithm applies some technique for choosing a direction p_k and searches along this direction from the current iterate x_k for a new iterate $x_{k+1} = x_k + \alpha p_k$ such that $f(x_{k+1}) < f(x_k)$. The distance to move along p_k can be found by approximately solving the following minimization problem to find a step length α:

$$\min_{\alpha > 0} f(x_k + \alpha p_k). \tag{5.1}$$

The success of a line search method depends on effective choices of the the direction p_k and the step length α.

Solving the linear SPD system $\mathbf{A}x = b$ is equivalent to solving the quadratic minimization problem

$$\min \phi(x) = \frac{1}{2} x^T \mathbf{A} x - b^T x. \tag{5.2}$$

The gradient of ϕ equals the *residual* of the linear system:

$$\nabla \phi(x) = \mathbf{A}x - b = r(x), \tag{5.3}$$

so in particular at $x = x_k$ we have $\mathbf{A}x_k - b = r_k$.

One of the remarkable properties of the CG method is its ability to generate, in a very economical fahsion, a set of vectors with a property known as *conjugacy*. A set of nonzero vectors $\{p_0, p_1, \ldots, p_l\}$ is said to be *conjugate* with respect to an SPD matrix $\mathbf{A}$ if

$$p_i^T \mathbf{A} p_j = 0 \quad \forall i \neq j \tag{5.4}$$

In fact, this set also shows to be linearly independent. The importance of conjugacy lies in the fact that we can minimize $\phi(\cdot)$ in n steps by *successively* minimizing it along the individual directions in a conjugate set (given that we have n conjugate vectors with repsect to $\mathbf{A}$). This can be shown very easily, but here we just refer to two Theorems. The successive iterates are set by

$$x_{k+1} = x_k + \alpha_k p_k, \tag{5.5}$$

where α_k is given by

$$\alpha_k = -\frac{r_k^T p_k}{p_k^T \mathbf{A} p_k}.^1 \tag{5.6}$$

Theorem 1. *For any $x_0 \in \mathbb{R}^n$ the sequence $\{x_k\}$ generated by applying formulas* 5.5 on the preceding page and 5.6 *converges to the solution x^* of the linear system $\mathbf{A}x = b$ in at most n steps.*

Proof. See [4]. □

Theorem 2. *Let $x_0 \in \mathbb{R}^n$ be any starting point and suppose that the sequence $\{x_k\}$ is generated by the method of Theorem 1.* Then

$$r_k^T p_i = 0 \quad \forall i = 0, 1, \ldots, k-1, \tag{5.7}$$

and x_k is the minimizer of $\phi(x)$ over the set

$$\{x | x = x_0 + \text{span}\{p_0, p_1, \ldots, p_{k-1}\}\}. \tag{5.8}$$

Proof. See [4]. □

So far, we have referred to the conjugate directions, but the question we pose next is how we find this set of n vectors? Theoretically, they can be based on any procedure giving a conjugate set $\{p_0, \ldots, p_{n-1}\}$. For instance, eigenvectors $v_1, v_2, \ldots, v_n$ of $\mathbf{A}$ are mutually orthogonal as well as conjugate. For large sparse matrices, however, finding them requires an unacceptable amount of computation. It is also possible to modify the Gram-Schmidt orthogonalization to produce conjugate set. However, the Gram-Schmidt is also expensive, and requires that the entire direction set is stored.

It is here the conjugate gradient method manifests itself; it can generate a new conjugate direction p_k by using only the previous vector p_{k-1}. The direction p_k is conjugate to all the previous directions. This property makes the conjugate gradient extremely scalable with respect to memory requirements. Direction p_k is given by

$$p_k = -r_k + \beta_k p_{k-1}, \tag{5.9}$$

[1] If f is a convex quadratic, $f(x) = \frac{1}{2}x^T \mathbf{A} x - b^T x$, its one-dimensional minimizer along the ray $x_k + \alpha p_k$ can be computed analytically and is given by

$$\alpha_k = -\frac{\nabla f_k^T p_k}{p_k^T \mathbf{A} p_k}.$$

where the scalar β_k is to be determinded by the requirement that p_{k-1} and p_k must be conjugate with respect to $\mathbf{A}$. By premultiplying 5.9 on the preceding page by $p_{k-1}^T\mathbf{A}$ and imposing the condition $p_{k-1}\mathbf{A}p_k = 0$ gives

$$\beta_k = \frac{r_k^T\mathbf{A}p_{k-1}}{p_{k-1}^T\mathbf{A}p_k}. \tag{5.10}$$

The first iteration often starts with choosing an initial guess x_0, and the first direction is chosen to be $-\nabla\phi(x_0) = b - \mathbf{A}x_0$, which gives the steepest descent direction. Before we state the final conjugate gradient algorithm, we should state an important theorem:

Threom 3. *Suppose that the k-th iterate generated by the conjugate gradient method is not the solution point x^*. Then the following four properties hold:*

$$r_k^T r_i = 0, \quad \forall i = 0, 1, \dots, k-1, \tag{5.11}$$

$$\text{span}\{r_0, r_1, \dots, r_k\} = \text{span}\left\{r_0, \mathbf{A}r_0, \dots, \mathbf{A}^k r_0\right\}, \tag{5.12}$$

$$\text{span}\{p_0, p_1, \dots, p_k\} = \text{span}\left\{r_0, \mathbf{A}r_0, \dots, \mathbf{A}^k r_0\right\}, \tag{5.13}$$

$$p_k^T\mathbf{A}p_i = 0, \quad \forall i = 0, 1, \dots, k-1. \tag{5.14}$$

Therefore, the sequence $\{x_k\}$ convergence to x^* in at most n steps.

Proof. See [4]. □

It is approriate to mention that the space $\mathcal{K}(r_0; k) = \text{span}\left\{r_0, \mathbf{A}r_0, \dots, \mathbf{A}^k r_0\right\}$ is called the *Krylov subspace of degree k* for r_0. The CG method is categorized under so-called *Krylov space methods*.

A slightly more economical form of the CG method, using Eq. 5.7 on the previous page, 5.9 on the preceding page, and 5.11, uses the follwing expressions for α_k and β_{k+1}:

$$\alpha_k = \frac{r_k^T r_k}{p_k^T\mathbf{A}p_k} \quad \beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}. \tag{5.15}$$

By using these expressions, we obtain the following standard conjugate gradient algorithm.

5.1.1 Rate of Convergence

As we have seen, CG is guarranted to convergence after at most n iterasjons. However, the number of iterations used by CG is dependent of $\mathbf{A}$. The rate at

Algorithm 1 Conjugate Gradient Method

$r_0 = b - \mathbf{A}x_0$, $p_0 = r_0$
For $k = 0, 1, 2, ..$until convergence
$\alpha_k = \frac{{r_k}^T r_k}{{p_k}^T \mathbf{A} p_k}$
$x_{k+1} = x_k + \alpha_k p_k$
$r_{k+1} = r_k - \alpha_k \mathbf{A} p_k$
$\beta_k = \frac{{r_{k+1}}^T r_{k+1}}{r_k^T r_k}$
$p_{k+1} = r_{k+1} + \beta_k p_k$
End

which the CG method converges to the solution depends on the distribution of eigenvalues of $\mathbf{A}$. Firstly, if $\mathbf{A}$ has only r distinct eigenvalues, then the CG iteration will terminate at the solution in at most r iterations. This is a special case of the following result; if $\mathbf{A}$ has eigenvalues $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$, we have

$$||x_{k+1} - x^*||_{\mathbf{A}}^2 \leq \left(\frac{\lambda_{n-k} - \lambda_1}{\lambda_{n-k} + \lambda_1} \right)^2 ||x_0 - x^*||_{\mathbf{A}}^2, \quad (5.16)$$

where x^* is the theoretical solution of $\mathbf{A}x = \mathbf{b}$. We note that if the eigenvalues are uniformly distributed, then each iteration contributes to decrease the error by the same factor. This results in a slow and uniform convergence. If the eigenvalues are distributed in r clusters of numerically close eigenvalues (where the cluster may be a repetition), then we would expect the error to fall sharply each time the iteration number k went from one cluster to another.

Another, more approximate, convergence expression for CG is based on the Euclidean *condition number* of $\mathbf{A}$, which is defined by

$$\kappa(\mathbf{A}) = ||\mathbf{A}||_2 ||\mathbf{A}^{-1}||_2 = \lambda_n / \lambda_1. \quad (5.17)$$

It can be shown that

$$||x_k - x^*||_{\mathbf{A}} \leq 2 \left(\frac{\sqrt{\kappa(\mathbf{A})} - 1}{\sqrt{\kappa(\mathbf{A})} + 1} \right)^k ||x_0 - x^*||_{\mathbf{A}}. \quad (5.18)$$

Matrix $\mathbf{A}$ is said to be ill-conditioned if the condition number is big. This bound gives a large overestimate of the error, but it can be useful when it is the only information we have about the eigenvalues of $\mathbf{A}$. The matrices resulting from the mimetic discretisation, when we use highly heterogeneous permeability fields and complex geometry, are very ill-conditioned, and the CG method is useless without precondition techniques for convergence rate improvement, which we discuss next.

5.2 Preconditioned CG-Algorithm

When the distribution of the eigenvalues of $\mathbf{A}$ is not favourable and the matrix is ill-conditioned, number of iterations for big systems can reach several hundred thousands. The method of preconditioning transforms or *preconditions* the linear system such that we can make the eigenvalue distribution more favourable and improve the convergence of the method significantly. No single preconditioing strategy is "best" for all conceivable types of matrices. In general, the reliability of iterative techniques depends much more on the quality of the preconditioner than on the iterative technique.

A preconditioner is a matrix $\mathbf{M}$ of the same dimension as $\mathbf{A}$ and is supposed to be SPD. The additional requirement is that it is should be inexpensive to solve the system $\mathbf{M}y = r$ (compared to $\mathbf{A}x = b$), because each iteration requires the solution of such a system. In the preconditioned CG version, we solve the system

$$\mathbf{M}^{-1}\mathbf{A}x = \mathbf{M}^{-1}b \tag{5.19}$$

The preconditioner has the original form as $\mathbf{M} = \mathbf{C}^T\mathbf{C}$ to preserve symmetry, because $\mathbf{M}^{-1}\mathbf{A}$ is not symmetric. $\mathbf{M}$ results from a change of variable $\hat{x} = \mathbf{C}x$, which results in the system $\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}\hat{x} = \mathbf{C}^{-1}b$.

The convergence rate will depend on the eigenvalues of $\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}$. $\mathbf{C}$ and the preconditioner can be attempted chosen such that the condition number is much smaller than that of $\mathbf{A}$ or that the eigenvalues of $\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}$ are clustered, also reducing the number of iterations.

To obtain the preconditioned CG-algorithm, we apply the CG method without preconditioing to the new system $\mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1}\hat{x} = \mathbf{C}^{-1}b$, and then invert $\hat{x}$ to x. This results in the following algorithm which uses only $\mathbf{M}$.

5.2.1 Practical Preconditioners

Often, the preconditioner $\mathbf{M}$ is defined in such a way that the system $\mathbf{M}y = r$ amounts to a simplified version of the original system $\mathbf{A}x = b$. In the case of a linear system representing a discretisation of a partial differential equation, $\mathbf{M}$ could come from a coarser discretisation. These types of preconditioners are often categorized under *multi-grid* preconditioners, where we may use knowledge of the underlying grid (*geometric multigrid*) or approximate this from the structure of the matrix $\mathbf{A}$ (*algebraic multigrid*).

Algorithm 2 Preconditioned Conjugate Gradient Method

$r_0 = b - \mathbf{A}x_0$, $y_0 = \mathbf{M}^{-1}r_0$, $p_0 = y_0$
For $k = 0, 1, 2, ..$until convergence
  $\alpha_k = \frac{{r_k}^T y_k}{{p_k}^T \mathbf{A} p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k \mathbf{A} p_k$
  $y_{k+1} = \mathbf{M}^{-1} r_{k+1}$
  $\beta_k = \frac{{r_{k+1}}^T y_{k+1}}{r_k^T y_k}$
  $p_{k+1} = y_{k+1} + \beta_k p_k$
End

Such preconditioners are difficult to implement, and we have not considered them as possible candidate preconditioners in this report.

General-purpose preconditioners using only the matrix $\mathbf{A}$ include Jacobi, symmetric successive overrelaxation (SSOR), incomplete Cholesky, sparse approximate inverse (SPAI). The Jacobi preconditioner is the simplest one among these, in which the preconditioner is chosen to be the diagonal of the matrix $\mathbf{A}$; $\mathbf{M} = \mathrm{diag}(\mathbf{A})$. In SSOR, $\mathbf{M}$ is given by

$$\mathbf{M} = \left(\frac{\mathbf{D}}{\omega} + \mathbf{L}\right) \frac{\omega}{2-\omega} \mathbf{D}^{-1} \left(\frac{\mathbf{D}}{\omega} + \mathbf{U}\right), \tag{5.20}$$

where $\mathbf{U} = \mathbf{L}^T$ is the upper triangluar part of $\mathbf{A}$. The parameter ω is between 0 and 2. Incomplete Cholesky is probably the most effective in general. The basic idea is simple; we follow the Cholesky procedure, but instead of computing the exact Cholesky factor $\mathbf{L}$ such that $\mathbf{A} = \mathbf{L}^T\mathbf{L}$, we compute an approximate factor $\tilde{\mathbf{L}}$ sparser than $\mathbf{L}$. We then choose $\mathbf{C} = \tilde{\mathbf{L}}$. There are several other considerations in this preconditioner, but we leave them out.

The implementation in this report implements only the Jacobi preconditioner. For a detailed discussion of the preconditioner mentioned above, see [18].

Chapter 6

GPGPU - General-Purpose Computing on GPUs

GPGPU stands for General-Purpose Computation on GPUs. GPUs are high-performance many-core processors capable of high computation and data throughput. The GPU was traditionally designed for use in computer graphics, where 2D images of 3D triangles and other geometric objects are *rendered*[2, 1]. Each element in the output image is referred to as a pixel, and the GPU used a set of processors to compute the color of such pixels in parallel. Recent GPUs are more general, with rendering only as a special case. The theoretical performance of GPUs is now close to three teraflops (tera = 1×10^{12}), making them attractive for high-performance computing[1]. Today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C.

The term GPGPU was coined by Mark Harris in 2002[3]. Interest in GPGPU has since been on the rise. Initial attempts at running non-graphics related software on GPUs were largely proof-of-concepts, and often relied on clever use of the hardware and a substantial knowledge of the inner workings of the GPU. GPUs could only be programmed using graphics APIs such as OpenGL. General-purpose stream-computing was achieved by mapping stream elements to pixels, and that required a through understanding of the inner workings of the GPU[2, 1]. In an attempt to appeal to a broader audience of scientific communities, and to provide a more direct access to the GPU, AMD released Stream SDK and NVIDIA CUDA. CUDA Toolkit provides the means to program CUDA-enabled NVIDIA GPUs, and is available on Windows, Linux and Mac OS X. CUDA consists of a runtime

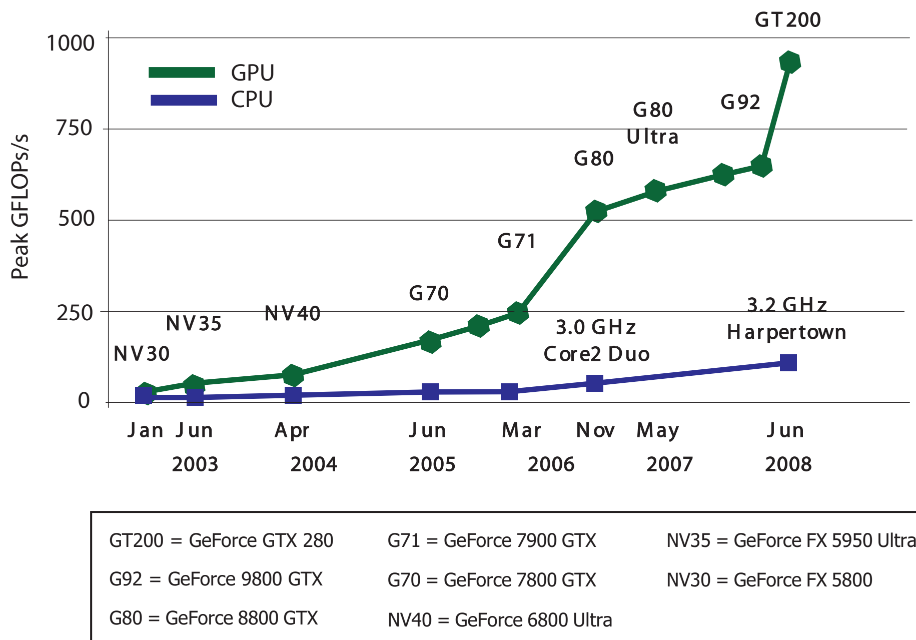


Fig. 6.1: GPUs are scaling with a rate much greater than CPUs. The Figure shows how some of the modern CPUs compare with modern GPUs in producing flops. From [19]

library and a set of compiler tools, and exposes an SPMD programming model where a large number of threads, grouped into blocks, run the same kernel.

This recent emergence of techniques of General-purpose Computing on GPUs has caused a breakthrough in computational science[2]. The performance in terms of FLOPs today offered by GPUs was for not long ago only accessible through supercomputers[20]. Numerical simulation does very clearly benefit from this new architecture, as shown by many successful experiments[21, 22, 23, 24, 25, 26]. The only drawback is that porting applications to heterogeneous architechtures often require complete algorithm redesign, as some programming patterns, which in principle are trivial, require great care for efficient implementation on a heterogeneous platform. It often takes time to develop and redesign algorithms, and there is almost no automization in the process. Issues with traditional approach often involve significant code change, and progress is often tedious and error-prone[1, 27]. In addition, the GPU architecture requires an extensive manipulation of array indices and data structures for management and optimisation. Attempts have been made by others to eliminate or simplify mechanical steps in the process of

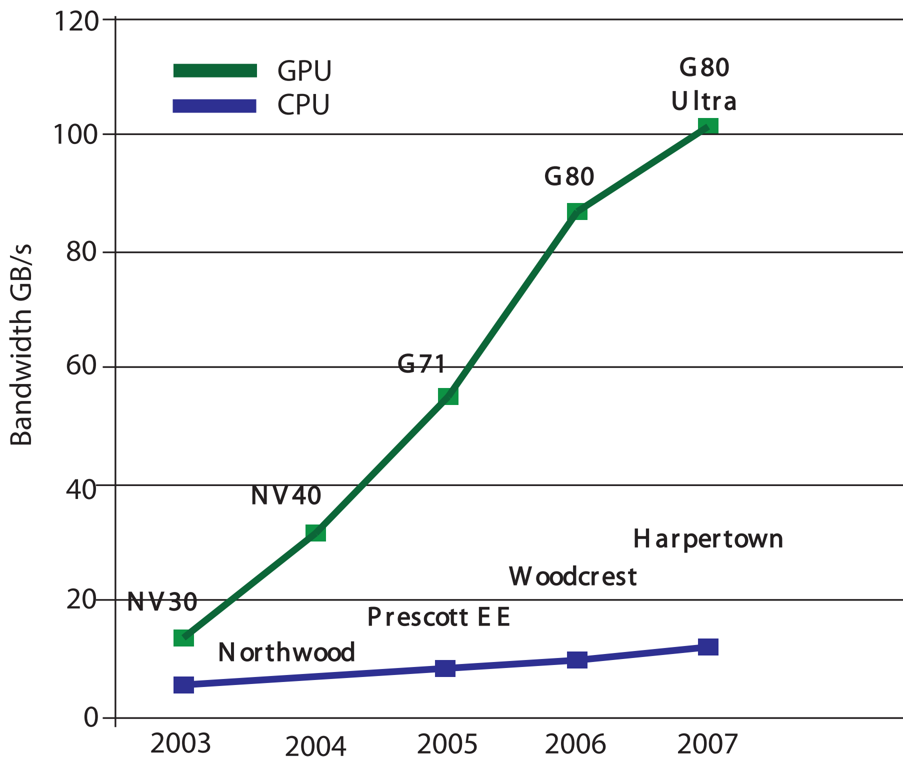


Fig. 6.2: Memory bandwidth for the CPU and GPU. From [19]

converting single-threaded programs to CUDA programs[28]. Still, though, the development activity for GPU libraries is high, and there exist a number of commerical and non-commerical libraries for GPU computing which makes the interface between CPU and GPU fairly intuitive[29, 30, 31, 32, 33, 34]. As stated earlier, two of the libraries currently undergoing high development activity has a major role in the implmentations in this thesis; the CUSP [35] and THRUST [36] libraries. A relevant description of them will be included when we discuss implementation.

6.1 Contrast to CPUs

The reason why the GPU has manifested itself in scientific disciplines has to do with the evolution of CPUs. Not long ago CPUs ceased to evolve with the scalability that is often desirable among science's increasingly insistent demands for more computational power. It is not long ago we got news about the newer CPUs with more and more powerful cores. Such news has now

begun to be infrequent, and newer CPUs are also fairly expensive.

Traditionally, the processor frequency closely followed Moore's law[37]. However, physical contraints have stopped and even slightly reversed this exponential growth in frequency rate[1]. A key physical constraint is power density, often referred to as the *power wall*. The relationship between power density P in watts per unit area, the total capacitance C, the transistor density ρ, the processor frequency f, and supply voltage V_{dd} is given by

$$P = C\rho f V_{dd}^2. \tag{6.1}$$

The frequency and supply voltage are related, as higher supply voltages allow transistors to charge and discharge more radiply, enabling higher clock frequencies. The power density in processors has already surpassed that of a hot plate, and is approaching the physcial limit of what silicon can withstand with current cooling techniques. GPUs, however, have continued to evolve in pace with the enormous development of the gaming industry. Figures 6.1 on page 39 and 6.2 on the previous page shows this development and the dividing gap between the CPU and GPU.

We can think of a GPU as a kind of primitive or simpler CPU, but with a force that greatly exceeds the CPU power. CPU is intended for a task that is not as specific as what the GPU is designed for. A CPU will generally be able to run any problem without any additional consideration. CPU is much more complex in terms of design, and a much larger part of it is occupied by the control unit and the cache. GPU, however, is intended for a very specific task, which is graphics. A pixel do not differ much from other nearby or far left pixels; they often undergo the same calculations, and this can be done in parallel without any form of advanced control since the same task is performed on many indepedent units. The GPU therefore has a much larger space devoted to data processing rather than data caching and flow control.

6.2 GPUs and Scientific Computing

The appeal of using GPUs in scientific computing is in taking advantage of a hardware architecture that provides a high level of parallelism and computational power for a relatively small amount of money. Using GPUs for scientific applications such as numerics does, however, often pose different requirements to the hardware than the graphics programs the GPUs traditionally have been developed for. One of the most important such aspects is double precision. Double precision is important both for exact results, as well as rapid convergence in numerical solvers. On modern CPUs

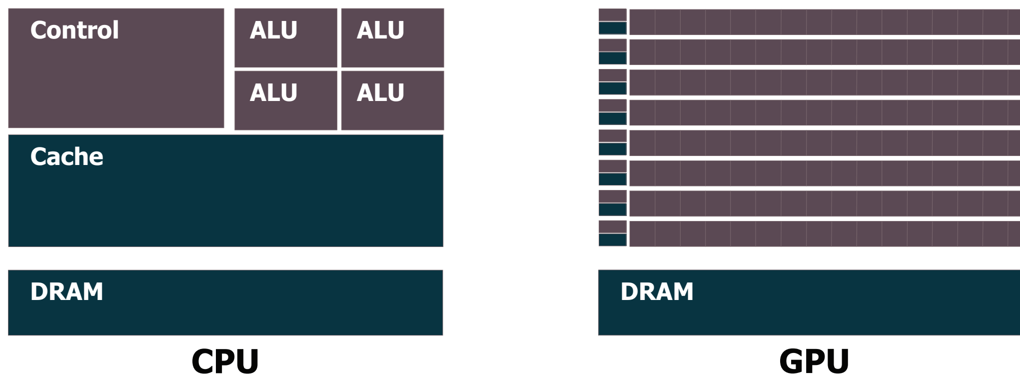


Fig. 6.3: The GPU has a much a larger space devoted to arithmtic units. The CPU has more of control and cache. From [19]

this basically comes for free, but on the current generation of NVIDIA GPUs single precision floating-point units outnumber double precision 8 units to 1. This means that the speedup one usually obtains for less-precision dependent programs is much more limited when requiring double precision. This has, however, given the GPU industry something to think about, and the next-generation GT300 GPU architecture from NVIDIA, code-named “Fermi”, has some interesting features for the scientific community.

The double precision performance has improved, now running at half the speed of single precision[1]. Another important aspect of programming on GPU is the need for decent development and debugging tools. Both CUDA from NVIDIA and OpenCL are frameworks developed for making development of more general applications easier. NVIDIA will also launch its new GPU development tool NVIDIA Parallel Nsight to be integrated with Microsoft Visual Studio[38, 39].

6.3 Heterogeneous Computing Model

In order to achieve higher and higher computing performance, current and future computer architechtures are based on parallelism. However, even modern multi-core CPUs (2-8 cores) use up to 80% of its resources on non-computational tasks. At the same time, dedicated stream accelerator cards, which contain hundreds of lightweight cores, are available. Such stream accelerators are designed for high computational throughput, while sacrificing complexity, thus disabling their ability to run operating systems.

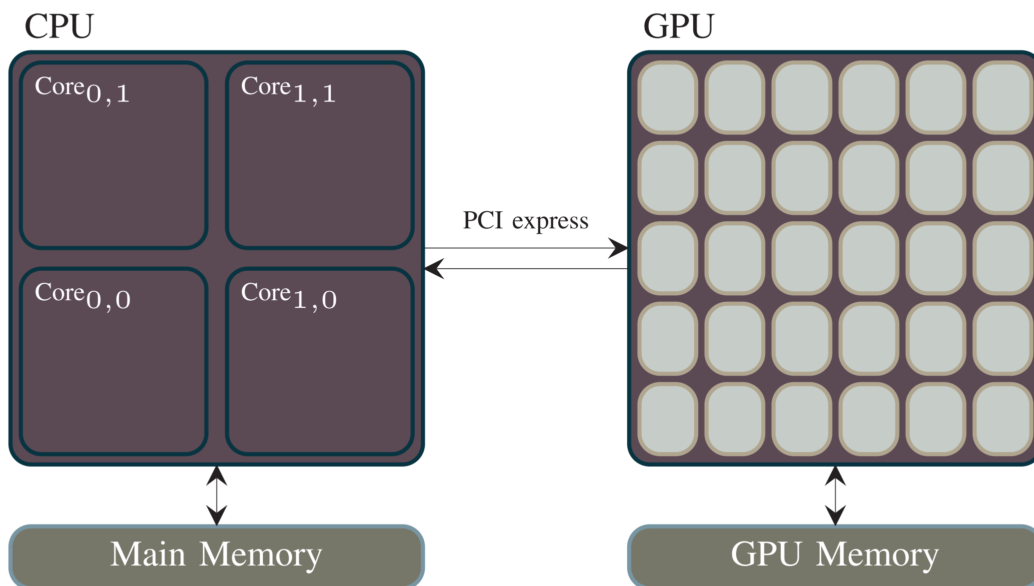


Fig. 6.4: A CPU in combination with a GPU is a heterogeneous system. From [1]

They are typically managed by traditional cores to offload resource-intensive operations. Most applications contain a mixture of tasks, some are best suited for multi-cores and others for streaming accelerators, and will ultimately perform best on heterogeneous architectures[1]. The concept of heterogeneous computing is the use of a GPU to do general-purpose scientific and engineering computing. The sequential part of the application runs on the CPU and the computationally-intensive part runs on the GPU. From the user's perspective, the application just runs faster because it is using the high-performance of the GPU to boost performance. In literature, CPU is often called the host and GPU is called the device. The terms processor and co-processor are also used. The application developer has to modify the serial application by mapping compute-intensive parts of the program to the GPU. The rest of the application remains on the CPU. The CPU is desgined as a general-purpose processor capable and designed for handling advanced flow control and data caching. The GPU on the other hand is developed for handling code that can run in parallel, is computational intensive and that requires a high memory bandwidth. This is typical for the graphical operations performed in games, with high-resolution textures and 3D-models with a high polygon count. In a typical program the CPU will handle the overall program execution and serial code, while the GPU performs tasks with a high degree of parallelization. The bottleneck in heterogenous computing such as GPU programming is very often the transfer of data between the

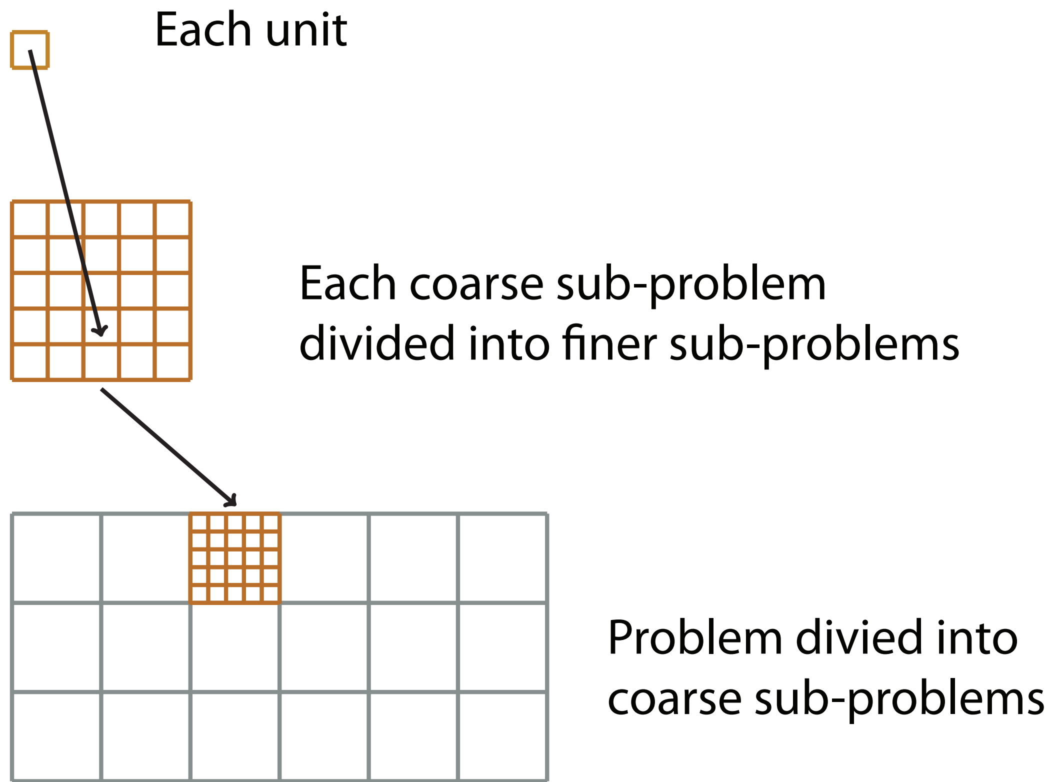


Fig. 6.5: Schematic illustration of sub-division of a problem into coarser and fine grain sub-problems.

two devices. For the CPU and GPU, all communication must go through the PCI-express interface (which is slow). A programmer wants to transfer as much data as possible in as few transfers as possible, in order to reduce overhead costs. We also want to design our program such that it does as many computations as possible on the data each device has locally before transfering more data. Figure 6.4 on the preceding page shows a GPU with 30 highly multi-threaded SIMD (Single Instruction Multiple Data) accelerator cores in combination with a standard multi-core CPU. The GPU is optimised for running SPMD (Single Program Multiple Data) programs with little or no synchronization.

6.4 GPU Computing with CUDA

CUDA is the name of a general-purpose parallel computing architecture of modern NVIDIA GPUs. In the litterature, CUDA refers to both hardware architecture of the GPU, and the software components used to program the

hardware[19]. The reason why CUDA is associated with general-purpose computing on GPUs is that the programmer does not need to have any extensive knowledge about the primary GPU architecture or traditional GPU programming, such as OpenGL. CUDA is an abstraction which gives the programmer a very clearly defined interface which maps nicely to problems that can be parallelized both on a coarse and fine grained level. The programmer can choose to express the parallelism in high-level languages such as C, C++, Fortran or driver APIs such as OpenCL and DirectX-11 Compute. CUDA is a fairly new technology but there are already many examples in the literature and on the Internet highlighting significant performance boosts using GPU hardware, cf. [21, 22, 23, 24, 25, 26].

The CUDA parallel programming model guides programmers to partition the problem into coarse sub-problems that can be solved independently in parallel. Fine grain parallelism in the sub-problems is then expressed such that each sub-problem can be solved cooperatively in parallel (See Figure 6.5 on the previous page).

Chapter 7

CUDA Programming Model

7.1 GPU Device Architecture Overview

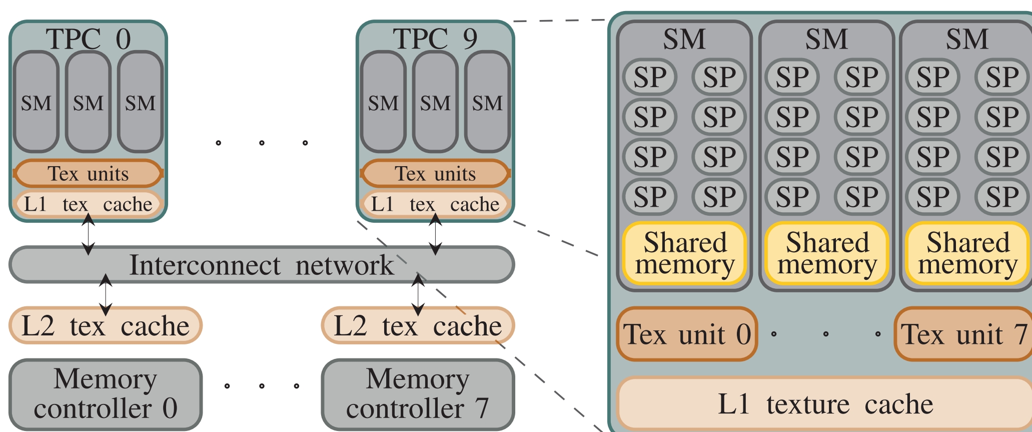


Fig. 7.1: The NVIDIA GT200 architecture. The abbreviations have the following meaning: TPC - texture processing cluster; SM - streaming multiprocessor; Tex unit - texture unit; Tex cache - texture cache; SP - scalar processor. From [1]

Before we explain the basic CUDA for the further purpose of the algorithm, it is perhaps appropriate to mention some few words about the GPU architecture from the point of view of CUDA. Simultaneously, an attempt will be made to introduce the new GT300 Fermi architecture and compare its main features with the previous GT200.

Figure 7.1 shows the NVIDIA GT200 GPU architecture. It has 10 TPCs, texture processing units, each containging 3 so-called streaming

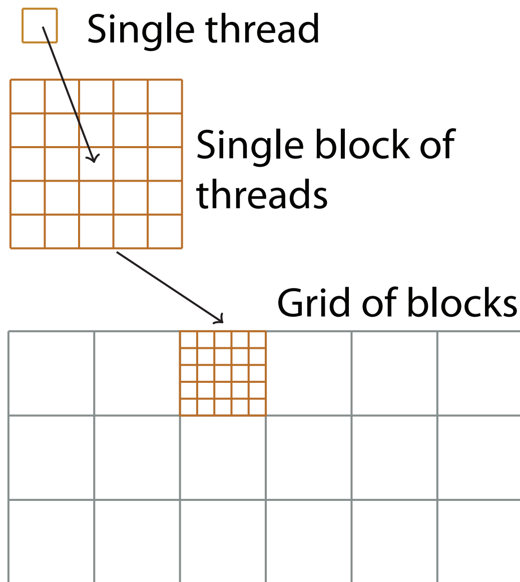


Fig. 7.2: Schematic illustration of a 2D grid containing 2D blocks.

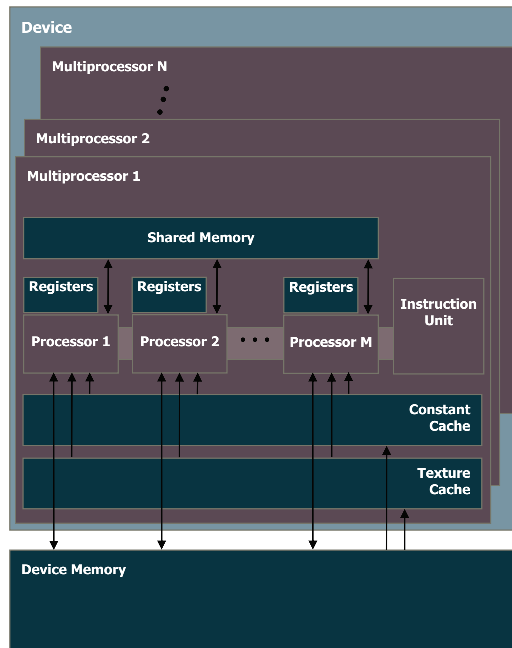


Fig. 7.3: Schematic illustration of an SM and the different memory levels. From [19].

multiprocessors (SM). Each SM has 8 scalar processors (SP). SPs execute individual threads. Hence, the total number of threads that can run in parallel are 240 (the G80 architecture before GT200 had two SMs per TPC, enabling 128 threads). Along with the SPs, each SM has its own *shared memory* which is shared by all the SPs in the SM. In addition to the SPs, it contains two Special Function Units (SFUs) and a *double precision core*. Each SP is fully pipelined arithmtic-logic unit capable of integer and single precision floating-point operations, while the SFUs contain four arithemtic-logic units, mainly used for vertex attribute interpolation in graphics and available to handle transcendental and other special operations such as sin, cos, exp, and rcp (reciprocal). The SMs have an SIMD (Single Instruction Multiple Data) architecture. That is, at each clock-cycle, the SP of each SM executes the same instruction on potentially different data. Each SM has 16,382 of 32-bit registers divided among all SPs. The shared memory is of size 16 KB. Each SM has 8 KB of read-only constant cache that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory. Only the host has write access to the constant memory. The GT200 has 8 KB of read-only

texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory. The on-chip shared memory has virtually no latency and is not limited by the bandwidth of the memory bus. When accessing the device memory (off-chip) there is a latency of about 400 to 600 clock cycles[19]. A GPU is built up by connecting a number of SMs through the RAM (random access memory). An SM is a type of sub-system, and the GPU connects these independent sub-systems through device memory.

7.2 Introducing Fermi

NVIDIA's first GPU supporting general-purpose computing, the G80-based GeForce 8800 introduced in November 2006, could provide up to 500 single precision GFlops. The fastest CPU at the time could not provide more than 20 GFlops. G80 architecture replaced separate vertex and pixel processors from NVIDIA GPUs at that time with unified computing processors that could be programmed in C. This, along with other innovations, enabled the GPU's enormous floating-point computing capacity in non-graphical applications. In 2008, NVIDIA launched their GT200-based GeForce 280. GT200 nearly doubled the performance compared to G80 reaching up to 900 single precision GFlops. GT200 also added support for double precision floating-point operations (FP64), which G80 lacked. FP64 is vital for many scientific and engineering programs, and unnecessary for 3D graphics.

Again, the agendas of the autumn 2009 and spring 2010 have been marked by the launch of NDIVIA's new Fermi architecture. At a high level, Fermi does not look much different from GT200. Fermi has three billion transistors distributed among the 480 (the GTX 480 GPU) cores/SPs (GT200 had 1.4 billion transistors on 240 cores). Despite the similarities, large parts of the architecture have evolved[39].

Fermi is based around the same concepts as GT200, with some major improvements. The number of SPs has doubled. The double precision performance has also improved dramatically, now running at half the speed of single precision. The memory space is also unified, so that shared memory and global memory use the same address space, thus enabling execution of C++ code directly on the GPU[1].

All of the processing done at the core level is now IEEE-specific. That is, IEEE-754 2008 for floating-point math and a full 32-bit integer. In the past 32-bit integer multiplier had to be emulated; the hardware could only do 24-bit integer muls. Fused Multiply Add (FMA) is also included. Each core can

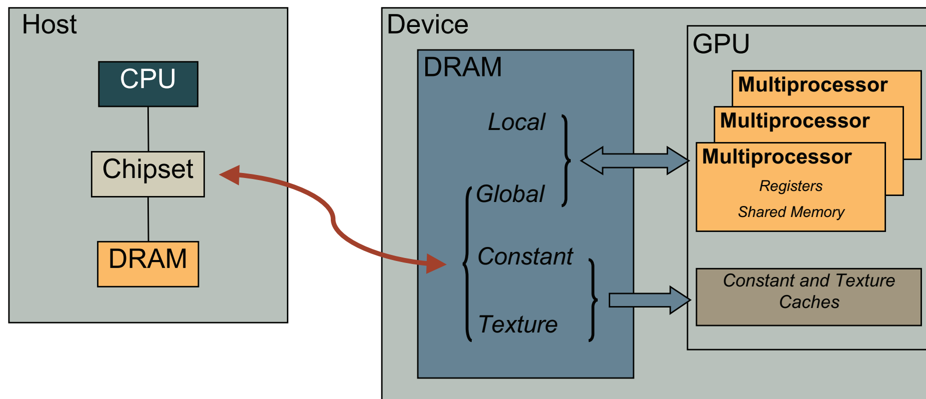


Fig. 7.4: Illustration of different memories of the device. The arrow direction shows reading access. Arrow both way means read and write access. The host can write to device memory, as well as Constant and Textures. From [40].

perform one FMA operation in each clock cycle and one double-precision FMA in two clock cycles. FP64 performance is improved tremendously. Peak FP64 execution rate is now half of 32-bit floating-point (FP32); it is 1/8 on GT200. In GT200 double precision processing was handled by a single dedicated unit per SM with much lower throughput. G80 and GT200 group eight cores into one SM. Fermi puts 32 cores per SM. In addition to the cores, each SM has a Special Function Unit (SFU), available to handle transcendental and other special operations such as sin, cos, exp, and rcp (reciprocal). In GT200 this SFU had two pipelines, in Fermi it now has four.

Each SM in GT200 had 16 KB of shared memory that could be used by all of the cores. This shared memory is not cache, but rather software-managed memory. The application would have to knowingly move data in and out of it. The benefit here is predictability; you always know if something is in shared memory because it is explicitly put there. The downside is it does not work so well if the application is not very predictable.

Each SM in Fermi has 64 KB of configurable memory. It can be partitioned as 16 KB/48 KB or 48 KB/16 KB; one partition is shared memory, the other partition is an L1 cache. The 16 KB minimum partition means that applications written for GT200 that require 16 KB of shared memory will still work just fine on Fermi. GT200 did have an L1 texture cache, but the cache was mostly useless when the GPU ran in compute mode.

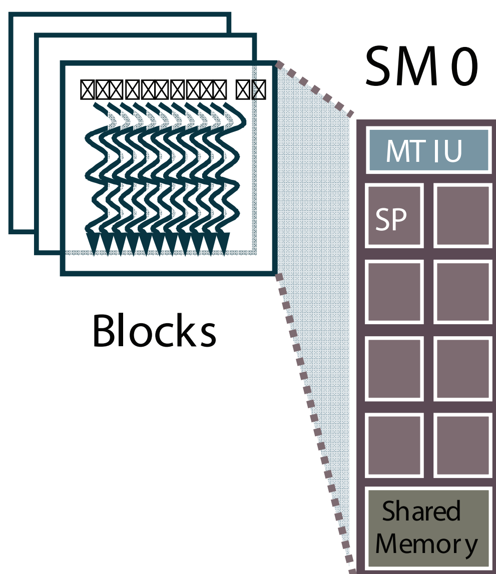
7.3 CUDA Architecture

The foundation of the CUDA software stack is CUDA PTX, which defines a virtual machine for parallel thread execution. This provides a stable low-level interface decoupled from specific target GPU. The set of threads within a *block* is referred to as a co-operative thread array (CTA) in PTX terms. All threads of a CTA run concurrently, communicate through the software-managed shared memory, and synchronize using barrier instructions. Multiple CTAs run concurrently, but communication between CTAs is limited to atom operations on global memory. Each CTA has a position within the *grid*, and each thread has a position within a CTA, which threads use to explicitly manage input and output of data. A kernel is compiled into a PTX program, and executed for all threads in a CTA[19, 1].

A thread block can be understood conceptually as a type of thread aggregate in which threads are aware of each other in terms of data dependecncy and synchronization (see Figure 7.2 on page 47). A thread block in CUDA is software-mangaged and gives a convenient interface to the programmer to partition the problem into independent subproblems. From the hardware point of view, CUDA is implemented by organizing the GPU around the concept of SM (see Figure 7.1 on page 46, 7.3 on page 47 and 7.5 on the following page).

A modern NVIDIA GPU, such as GTX 285, contains tens of SMs. Each SM consists of multiple SPs (see Figures 7.1 on page 46 and 7.5 on the following page), each capable of executing an independent thread. So the GPU runs many threads in parallel on different SMs. When we write parallel code using CUDA, we essentially write a sequential code for one thread. When executed, the same code runs on different threads (on SPs). This is called single instruction multiple thread (SIMT), and is similar to single instruction multiple data, SIMD, since each thread can work on its own data (each SM is SIMD). In that sense, it is similar to writing the code for one single CPU in, for instance, MPI[41]. Just as in MPI, the threads can also branch into different paths, even though this is never a recommendation based on the fact that branching implies serialization of instructions [19].

Whenever we launch threads on a GPU, we actually launch all the threads in a two level hierarchy. We have a collection of blocks called a grid, and each block is a collection of threads (See Figure 7.2 on page 47). There are essentially two reasons why we have two levels of threads. Each block is an independent unit which represents a coarse grained parallelism and all the threads in a block can cooperate and synchronize with each other. Thread



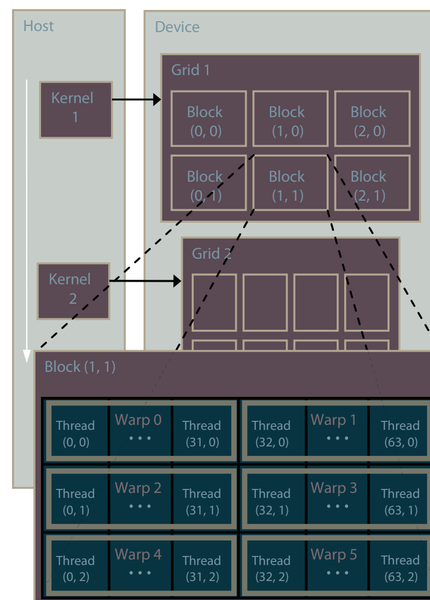


Fig. 7.5: A thread block runs on a single SM. Multiple independent blocks are often scheduled on one SM. From [42].

Fig. 7.6: A thread block runs on a single SM and is scheduled in a collection of 32 threads called warp.

synchronization here means that we can set up a barrier at a point in the thread code such that right at the point of the barrier, all the threads in the block will wait until each thread has arrived at this stage during the execution. Threads within a block run on a single SM (each thread running on single SP). This is exactly what gives them the possibility to synchronize and share information efficiently using the on-chip shared memory of the SM. Thus, we can understand the naming convention shared memory (See Figure 7.3 on page 47). Shared memory will be discussed in Section 7.6.

The CUDA programming model is based upon the concept of a *kernel*. A kernel is a function that is executed multiple times in parallel, each instance running in a separate thread. The threads are organized into one-, two- or three-dimensional blocks, which in turn are organized into one- or two-dimensional grids. The blocks are completely independent of each other and can be executed in any order. Figures 7.2 on page 47 shows an illustration. As opposed to different blocks of one grid, two grids represented by two kernels can have an dependency (See Figure 7.7 on the following page). In this project, we have used one-dimensional grids and blocks. Two-dimensional blocks are approriate when using for example Finite Difference Method on structured grids.

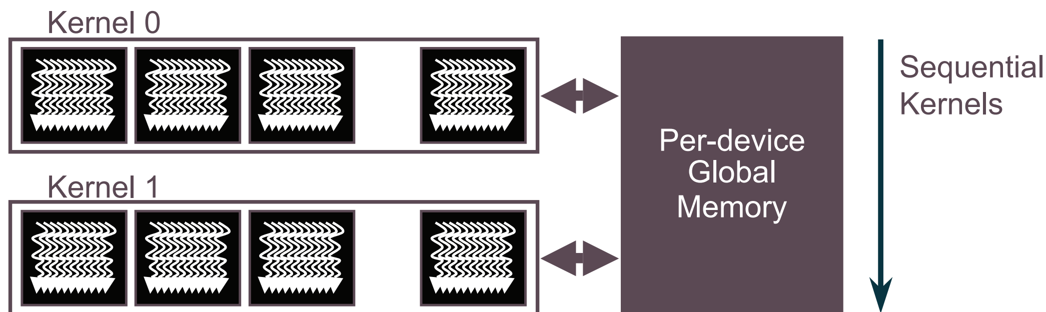


Fig. 7.7: Two kernels run sequenctially and can depend on each other. From [43]

7.4 CUDA Memory Model

Figure 7.4 on page 49 shows how the host and the device co-operate with each other. The device has read-only access to the constant and texture memory, to which only the host can write. In the case an SM runs out of registers, the threads can spill to the per-thread local memory which resides in global memory.

7.4.1 Memory Hierarchy

The memory structures are part of the CUDA architecture. Threads in a block will in most programs have to exchange data, and the shared memory allows for this. The shared memory will have the same lifetime as the the block it is attached to, while the global memory is persistent throughout the program, and hence can be used for communication between different kernels. Two blocks of the grid cannot communicate. They represent parallel units.

In addition to the private, shared and global memory spaces, threads have read access to the texture memory and the constant memory. The constant memory is cached and, so it is extremely fast. It is, however, very limited in space, only 64 KB, hence of limited use. Constant memory cannot be alloced dynamically, even from host. The host and the device will maintain different memory spaces, and pointers from one cannot be dereferenced in the other. Before executing a kernel function, the necessary memory must be allocated on the GPU, and data must be transferred from the host system. This transfer goes over the PCI-express interface. Table 7.8 on the following page gives a summary of access level and lifetime of different memory spaces.

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	Block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Fig. 7.8: Table showing access level and lifetime of different memories on the GPU.

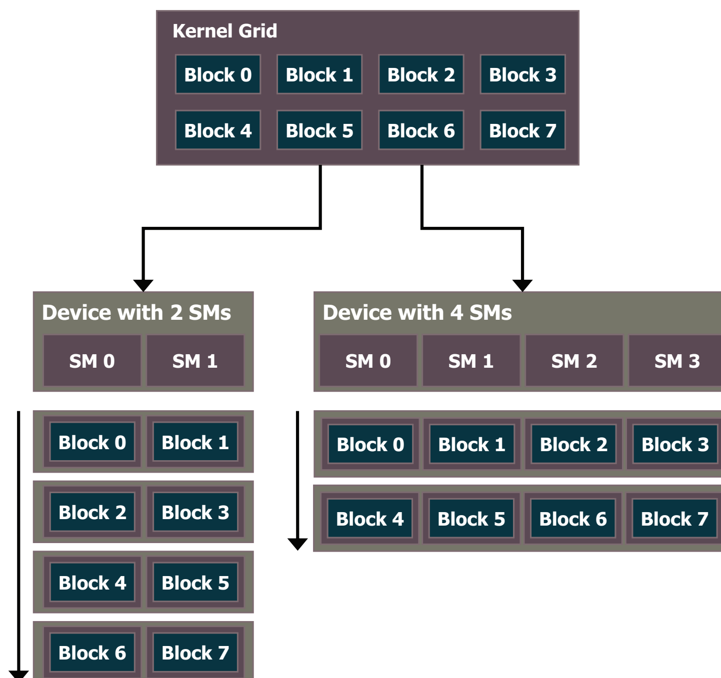


Fig. 7.9: Scalability of a CUDA program. From [44]

7.5 CUDA Execution Model

The GPU is designed as a stream processor, which means that it takes a large set of data (a stream) and applies a series of operations (a kernel function) to each element in the stream. The processing of each element must be independent of each other so that they can be executed in parallel. Another way of seeing it is as if we have a large set of threads, each executing the same kernel function in parallel on its respective element of the dataset.

As we know, in the CUDA architecture, not all threads need to be independent. Whereas each block has to be completely independent of each

other, the threads within the blocks can share data and synchronize their execution if neccessary. This is possible because each block is executed on only one SM. This fact also implies that the resources each block uses determines the total number of blocks that can run per MP. This puts a limit on the number of blocks we can run. Another point is scalability of a CUDA program. Since a CUDA program is not bound to one particular GPU, we would expect roughly halving in execution time if running on a GPU with twice the number of SMs, given that the grid size is large enough. This is indeed an important point. Figure 7.9 on the previous page shows a grid with 8 blocks. If run a GPU with only two SMS, 4 blocks must be sheduled on each SM. If run with 4 SMs, only two blocks are run on each SM.

The *thread execution manager* schedules the blocks for execution by passing it to one of the SMs (see Figure 7.5 on page 51). Each SM can run up to 8 blocks at a time. The blocks that are running are called active blocks. When all the threads of an active block have finished, a new block is scheduled. This goes on until there are no more blocks in the grid. A new kernel can only be launched when all the blocks of the previous launch have finished.

On GT200, since block size can be up to 512 threads large [19], and the number of SPs in each SM is only 8, not all the threads in a block can run at the same time. The blocks are automatically split into *warps* consisting of 32 consecutive threads (See Figure 7.6 on page 51). The blocks are scheduled to the SMs at runtime, and each warp in executed in SIMD manner. Every 4th clock cycle, the *warp scheduler* (instruction dispatch unit) selects one of the active warps that are ready to go (all operands are available), and loads the next instruction for that warp into the SPs. That instruction is then executed on all 8 SPs for 4 clock cycles (once for each of the 32 threads of the warp). Divergent code flow between threads is handled in hardware by automatic thread masking and serialization, and thus, divergence withing a warp reduces performance, while divergence between warps has on impact[1]. Active warps are scheduled in a way that maximizes utilization and try to hide the delays associated with memory loads.

7.6 Shared Memory

Shared memory is a type of user-managed cache in a couple of considerations when we are making use of that. The amount of shared memory is very limited in comparison to the total global memory available on a GPU. Its advantage lies in the access time. It is roughly 100x faster than accessing

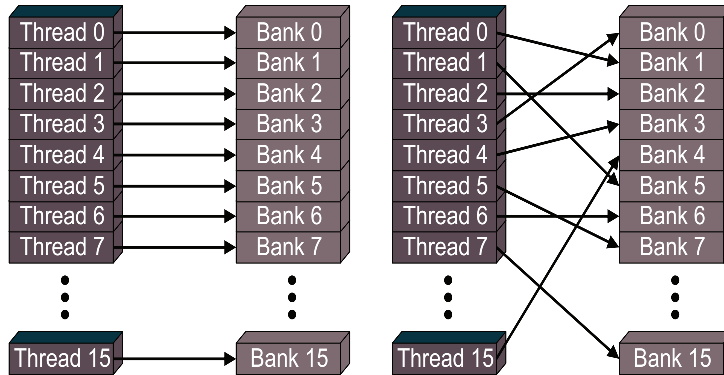


Fig. 7.10: No bank conflicts. From [40].

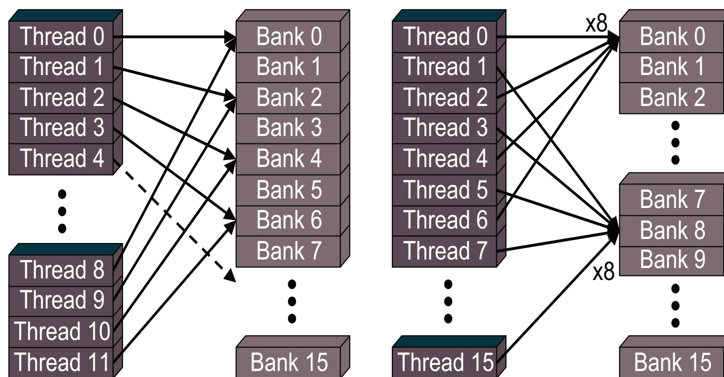


Fig. 7.11: Bank conflicts occur here. From [40].

the GPUs global memory. Threads can co-operate their data accesses and computations via shared memory. For example it is possible to use one or a few threads to load data from global memory into shared memory that is going to be used by all the threads, thereby negating the need to go to the global memory[19].

Let us first say a few words about the architecture of the shared memory. The shared memory is divided into banks of successive 32-bit. Each 32-bit word is assigned to successive banks. Each bank can service one single address per clock cycle. Hence, shared memory can service as many one-cycle accesses as it has banks. Multiple simultaneous accesses to one bank result in what is called bank conflict. If two threads look up the same bank, they are conflicting, and so one has to wait for the other, making the two accesses several cycles long. Bank conflicts therefore results in serialization of these shared memory requests. Figure 7.10 shows an access pattern which is a very orderly pattern. There are no bank conflicts and each access is served

in one cycle. It is appropriate to note that the bank accesses do not need to be ordered in any sense. As the simultaneous accesses increases, we get more and more degradation in access time, bank conflicts of higher order (see Figure 7.11 on the previous page). On the level of half-warp, we can have from 1 to 16-way bank conflict, in the latter case all 16 threads accessing the same bank. To summarize, to maximize usage, we want each thread in a half-warp to get shared memory at the same time. This can only happen in certain situation, similar in concept to global memory coalescing.

7.7 Memory Coalescing

The threads of a warp are also free to use arbitrary addresses when accessing on-chip memory with load/store operations. Accessing scattered locations results in memory divergence and requires the processor to perform one memory transaction per thread. On the other hand, if the locations being accessed are sufficiently close together, the per-thread operations can be coalesced for greater memory efficiency. Global memory is conceptually organized into a sequence of 128-byte segments. Memory requests are serviced for 16 threads (a half-warp) at a time. The number of memory transactions performed for a half-warp will be the number of segments touched by the addresses used by that half-warp. If a memory request performed by a half-warp touches precisely one segment, we call this request *fully coalesced*, and one in which each thread touches a separate segment we call *uncoalesced*. If only the upper or lower half of a segment is accessed, the size of the transaction is reduced[19, 5].

Chapter 8

Implementation

The subsequent sections in this chapter discuss important details and procedures in the implementations. Both the Full-matrix and the Matrix-free version use the conjugate gradient method for solving the resulting linear system. Matrix-free only requires that the local cell matrices are calculated, while the Full-matrix version requires an additional assembling of them to create the global stiffness matrix. To the extent it is a question of implementation, that is essentially the main difference between these versions.

The CG-algorithms in the previous chapter show that the preconditioned CG method requires, during each iteration, one matrix-vector multiplication, two inner-products, two *saxpy* operations, one vector scaling ($bp_k = \beta_k p_k$), and one vector addition ($p_{k+1} = y_{k+1} + bp_k$). The operations $\mathbf{M}^{-1} r_{k+1}$ in the case of Jacobi preconditioing is trivial. Except for the matrix-vector multiplication, all operations are computationally inexpensive and straightforward to implement, both on host and device. The challenging and expensive operation is the matrix-vector multiplication, which is possibly also one of the most important numerical operations in numerical mathematics. It is this operation that ultimately make up our primary focus when it comes to optimisation.

Our primary goal has been to create an iterative linear solver that works on general corner-point grids using the MFD discretisation we discussed in Chapter 4. In a matrix resulting from MFD discretisation, the number of nonzero elements in a row is one less than the total number of cell faces in the two neighboring cells that share the face representing the given row (given that we no wells). For a 3D Cartesian grid without wells, each row

representing an internal face has 11 nonzero elements, and each face residing on the boundary gives only 6 nonzero entries in the given row. For a Cartesian grid with three million cell faces, the number of nonzero elements in the matrix corresponds to approx. $3.7e^{-4}$ % nonzeros compared to total entries. Thus, this poses an inherent requirement that the global matrix is represented in a sparse format. In this regard, we will discuss various sparse matrix formats and say a few words about which ones are eligible for the host and the device.

A library for sparse matrices is not yet included in the current CUDA SDK. This will most probably change in the future releases[1]. A thesis was written last year at SINTEF which treated the same theme as this thesis, that is, a mimetic implementation (in addition to other methods) on corner-point grids[45]. The author of this thesis computed all the global matrices in MATLAB using a toolbox developed by SINTEF (described next) and imported the matrix (and the right-hand side) from MATLAB to the CUDA-program. The system $\mathbf{S}\pi = \mathbf{R}$ was then solved using different sparse matrix formats, created by converting the matrices imported from MATLAB. The implementations was on based [5] and used the same formats that are discussed here.

In this thesis, we read the grid and geometry data from two files created using the MATLAB toolbox; a file containging grid information and well data, and the second file containing permeability data. Local cell matrices are then calculated using this data. Right-hand side is also calculated. For the Full-matrix version, the cell matrices are assembled to the global matrix $\mathbf{S}$ in 4.22 on page 20. In addition, this thesis evaluates the CUDA-libraries CUSP and THRUST, which were made available recently. CUSP is a library for sparse linear algebra and graph computations on CUDA and is based on[5]. CUSP provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Thrust provides a flexible high-level interface for GPU programming that greatly enhances developer productivity. The primary purpose has been not to repeat the material from the thesis mentioned. Kernel implementations for different sparse matrix formats are also open source. Moreover, using CUSP and THRUST implies a significant improvement with respect to performance and the interface between the host and device.

[1]According to one of the authors of [5], and the designer of CUSP, on NVIDIA forums, future CUDA SDK releases may include the sparse matrix-vector multiplication kernels implemented in CUSP.

8.1 MRST - MATLAB Reservoir Simulation Toolbox

The geometry and grid representation of coner-point grids is a difficult task. For that reason, a GNU MATLAB Reservoir Simulation Toolbox (MRST) developed by SINTEF has been used to compute the required grid and permeability information. The toolbox consists of a comprehensive set of routines and data structures for reading, representing, processing, and visualizing unstructured grids, with particular emphasis on the corner-point format used within the petroleum industry. It also contains a wide range of finite-dimensional discretisation schemes, such as MFD, Mixed Finite Element, two-point flux approximation TPFA, as well as direct solvers for resulting sytemes. All the visualization has been done using MRST functions `plotCelldata(...)` and `plotFacedata(...)`.

The MFD requires that the following geometry information is available for all cells in grid: cell volumes, cell mass centers, face mass centers, face areas, normal vector for each face. In addition to that, the implementation requires the grid representation which is given by the following tables; `cellFaces, cellP, faceP, faceNodes, nodes, neigbhours`. All of these are one dimensional arrays. Information about these arrays and additional information about corner-point grid can be found in the MATLAB file `grid_structure.m` of MRST[46].

8.2 Matrix-free Implementation

The first thing to be said about the Matrix-free version is that it is very specific. Before we say anything else about that version, let us see how the algorithm looks. The following algorithm shows the preconditioned CG-algorithm for the Matrix-free version. The algorithm starts by computing r_0, y_0 and p_0 (cf. Section 5.2). The one-dimensional array $\mathbf{CM} = [S_1, S_2, \ldots, S_M]$ (**C**ell **M**atrices) contains the the upper symmetric portion of the cell matrices (a total of M cells). Cell matrix S_i is stored row-wise, the first row storing all entries, and the last one storing only the last entry. The functions `cell_mvp(...)` and `reduce(...)` in the algorithm defines the Matrix-free functionality of the algorithm. The rest is identical to Full-matrix version. These will be explained.

Algorithm 3 Preconditioned Matrix-free CG Method

$r_0 = b - \mathbf{S}x_0$, $y_0 = \mathbf{M}^{-1}r_0$, $p_0 = y_0$
For $k = 0, 1, 2, ..$until convergence
$\mathcal{R}_k = \text{cell_mvp}(\mathbf{CM}, p_k)$
$\mathcal{L}_k = \text{reduce}(\mathcal{R}_k)$
$\alpha_k = \frac{{r_k}^T y_k}{{p_k}^T \mathcal{L}_k}$
$x_{k+1} = x_k + \alpha_k p_k$
$r_{k+1} = r_k - \alpha_k \mathcal{L}_k$
$y_{k+1} = \mathbf{M}^{-1} r_{k+1}$
$\beta_k = \frac{{r_{k+1}}^T y_{k+1}}{r_k^T y_k}$
$p_{k+1} = y_{k+1} + \beta_k p_k$
End

Each cell in the grid is assigned to a single thread. That is, the matrix-vector product $S_i p_k^i = \mathcal{R}_k^i$, where p_k^i is the vector of elements from p_k corresponding to the global faces of cell i, is computed by a single thread, and the result $\mathcal{R}_k^i$ is stored in device memory. This means that a total of M threads are needed to process the whole grid. These threads are organised into a number of thread blocks. Knowing that two threads in two different blocks cannot synchronize or co-operate, this also results in blocks of non-communicating cells (cf. Section 7.5). Hence, two cells in two different blocks which share a common face has to be done independent somehow.

Usually, one partitions or divides the grid into a set of connected sub-grids, for example using metis[47], see Figure 8.1 on the next page. On a supercomputer, for instance using MPI, one can assign each sub-grid to different CPUs with their own random access memory. The interface between the sub-grids still requires global reduction. Sometimes the problem may nesscitate such a partitioning for parallelization. For a problem, for example The Poisson Equation on an unstructured triangular grid, where the unknowns are located at the nodes, and not on the faces, we may not have the knowledge of how many cells/elements contribute to a given node. This often changes throughout the grid (See Figure 8.1 on the following page). For an effective implementation, such a partitioning is essential. Using MPI, a sub-problem can be assigned to a single CPU where the sub-problem is solved serially. On CUDA, a sub-problem is assigned to a single block, and the sub-problem is solved using several co-operating threads. Then only a global reduction is needed for the interface nodes. In MPI this can be realized in multiple ways. On CUDA, we can split the problem into two kernels. Notice that the GPU has only one RAM, so a partitioning of the grid just gives a

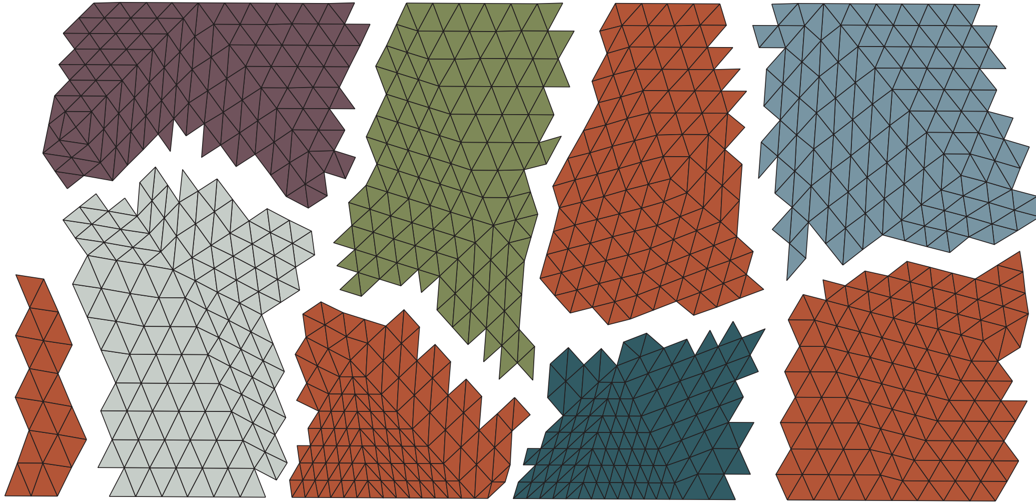


Fig. 8.1: An example showing partitioning of an unstructured triangular grid. Each node recieves contribution from a variable number of triangle elements, and this information is usually not accessible. Partitioning is the only way to parallelize such grids where the unknowns are located on grid nodes.

permutation of the cells; one sub-partition is stored continously in memory, and a block knows where to read its partition by means of given offsets.

On the other hand, when the unknowns are located on the cell faces, such as the face pressure π we have in the system $\mathbf{S}\pi = \mathbf{R}$, we can infer that a given face only receive contribution from two cells. That is, the ones sharing that face. We can use this information to avoid partition of the grid. Partitioning of a grid often requires an excessive amount of preprocessing and often makes the problem more difficult to implement and debug. Since we are solving the system $\mathbf{S}\pi = \mathbf{R}$, the partitioning has been avoided.

In summary, the Matrix-free version assumes that only two cells can be connected. It is correct when we look at the grid; each cell face is shared by maximum two cells, and each row of $\mathbf{S}$ contains only contribution from maximum two cells. That changes if we include wells into the system. A well usually goes through multiple non-adjacent cells and thereby introduces connections between multiple cells. For that reason, the Matrix-free version does not support wells[2].

[2]It was relatively late in the project that my supervisor Bård Skaflestad showed interest in adding wells to the system 4.21 on page 20. The choice was between implementing other discretizations, but the choice ultimately fell on the wells. That is when I discovered that my assumption that only two cells share an internal cell face made it impossible to add wells. Any partitioning would still demanded that all the well cells ended up in the same

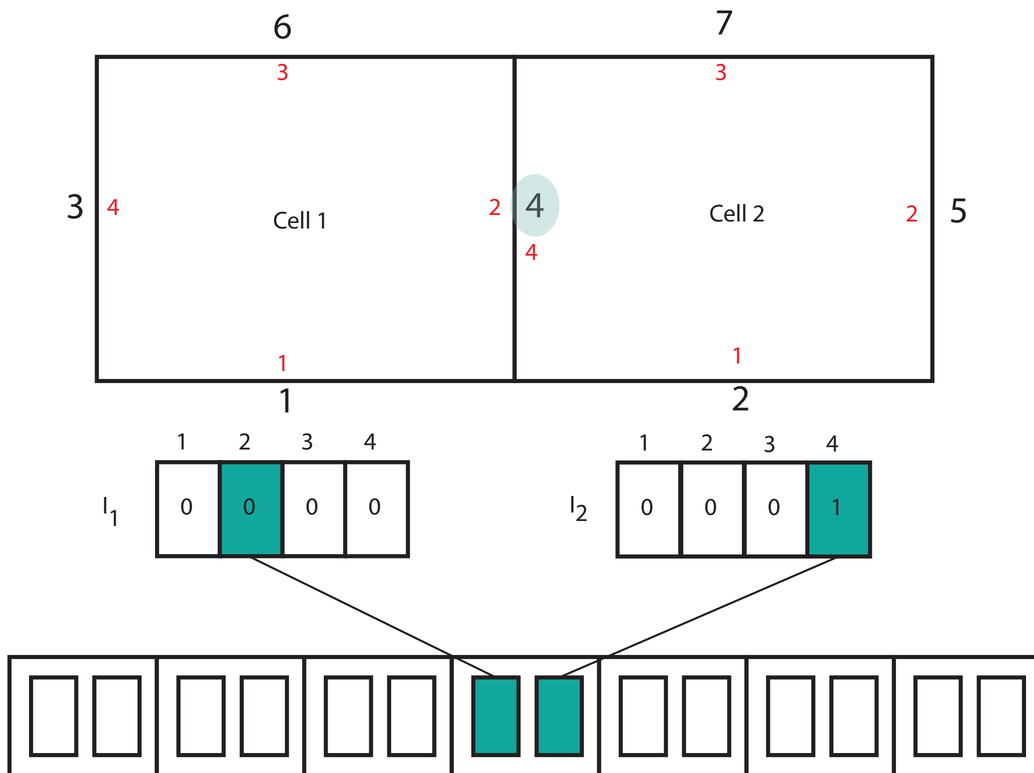


Fig. 8.2: This figure shows an example of how partitioning of the grid is avoided based on the information that the unknowns are located on the cell faces. An struct array is created with two doubles, and the integer arrays I_1 and I_2 contain the information which tells each cell where to store its contributions for each of its faces. Component-wise sum of the struct array is the result of a matrix-vector multiplication.

One thread has been assigned to one cell, and the global thread number i is associated with the global cell number i. That is, we may have adjacent cells associated with threads in different blocks, because there is no logical ordering of cells (which is essentially what partitioning does). If two cells share a face, this face receive exactly two contributions. If the associated threads live in different blocks, we need to store these contributions at different places in global device memory. This problem has been solved by telling each thread where to store its contributions in an `double2`[3] array, which is a struct containing two doubles. Figure 8.2 shows a schematic

partition. On a supercomputer, that would be easier. The GPU introduces additional constraints. I think that would be quite troublesome to achieve, given that the time was limited.

[3] `double2` is an in-built struct in CUDA

illustration of a grid with two cells. The global face 4, which is local face number 2 for cell 1 and local face number 4 for cell 2, is shared by the two cells. Each cell has its own index array, which contains the same number of elements as the number of cell faces. If a face is shared, one of the cells has entry 1 at the corresponding local face index. This index array is stored as $I = [I_1, I_2, \ldots, I_M]$.

Let us now explain the functions `cell_mvp(...)` and `reduce(...)`. From the discussion over, $\mathcal{R}_k$ is an one-dimensioanl `double2` array. This array is initialised to $(0, 0)$ prior to the CG iterations, which after each invocation of `cell_mvp(...)` contains the contributions from grid cells. Both the first and second entries of the struct `double2` will be different from zero for all shared faces. Notice that this array is N long, where N is the total number of global faces. $\mathcal{R}_k$ is then added component-wise using the `reduce(...)` function. This function updates $\mathcal{L}_k$ after each iteration. We then understand that $\mathcal{L}_k = \mathbf{S}p_k + = \sum_{i=1}^{M} S_i p_k^i$. The rest of the algorithm is equivalent to the Full-matrix version. Except `cell_mvp(...)` and `reduce(...)`, all the other operations in the CG method has been done using CUBLAS. CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA driver[48].

It may be appropriate to mention that the function `cell_mvp(...)` does not contain any `_syncthreads()`. Thus, each thread in the whole grid is a completely independent unit. This follows directly from the index array I.

A further discussion of the Matrix-free will be made when we compare the two versions in Chapter 9.

8.3 Full-matrix Implementation

The Full-matrix version is the traditional CG-algorithm where the matrix $\mathbf{S}$ is stored in a sparse format. Instead of splitting the matrix-vector multiplication in two kernel functions as in the Matrix-free version, the Full-matrix version computes $\mathbf{S}p_k$ directly using a single kernel. A single thread in this case can, for instance, compute the contribution from a single row in the matrix.

First, the cell matrices $\mathbf{S}_i$, $i \in 0, 1, \ldots, M$, are computed on the device. This is done using multiple kernels. After reading the grid, well and permeability data from files, the cell matrices $\mathbf{C}_i$, $\mathbf{Q}_i$ and $\mathbf{N}_i$ are computed in the given order for each cell i. The matrix $\mathbf{Q}_i$ is an orthonormalization of the matrix $\mathbf{D}_i\mathbf{C}_i$ computed using the Modified Gram-Schmidt algorithm. These matrices

are then used to compute $\mathbf{B}_i^{-1}$, which is then used to compute the final cell matrix $\mathbf{S}_i$. Since the dimensions of $\mathbf{B}_i^{-1}$ and $\mathbf{S}_i$ are equal, $\mathbf{B}_i^{-1}$ is overwritten by $\mathbf{S}_i$. Finally, we assemble $\mathbf{S}_i, i \in 0, 1, \ldots, M$ to $\mathbf{S}$ in sparse CSR format. This format will be described in the next section. The process of assembling is done on CPU using `std::map`. That is, after computing $\mathbf{S}_i$ for all cells, we copy them to host and assemble them into the CSR format.

Computation of the right-hand side $\mathbf{R}$ is also done on CPU. In the case of Dirichlet boundary conditions π_D, the matrix is assembled twice. We follow this process.

1. Assemble $\mathbf{S}$ from $\mathbf{S}_i, \quad i \in 0, 1, \ldots, M$.
2. Assemble $\mathbf{R}$ for Neumann boundary conditions and, possibly, sources/sinks.
3. Set $\mathbf{R} = \mathbf{R} - \mathbf{S}\pi_D$.
4. Set $\mathbf{R}(k) = \mathbf{S}(k, k)\pi_D(k)$ $\forall$ Dirichlet faces k.
5. Set $\mathbf{S}_i(:, j) = 0$ $\forall$ local Dirichlet faces j, except $\mathbf{S}_i(j, j)$.
6. Set $\mathbf{S}_i(j, :) = 0$ $\forall$ local Dirichlet faces j, except $\mathbf{S}_i(j, j)$.
7. Assemble $\mathbf{S}$ from $\mathbf{S}_i, \quad i \in 0, 1, \ldots, M$ (again).
8. Solve $\mathbf{S}\pi = \mathbf{R}$ using CG-algorithm on host or device.

Steps 5 and 6 are done in parallel on the device, which is very fast, and then copied back to the host to be assembled to the final $\mathbf{S}$.

8.4 Sparse Matrix-Vector Multiplication on CUDA

We generally talk about two types of matrices; dense and sparse. Dense matrices can both be structured and unstructured, and mostly have $\mathcal{O}(n^2)$ elements, and the number of nonzero elements in each row are uniformly distributed. Sparse matrices, however, have far fewer nonzero elements and each row may have variable or fixed number of nonzero elements. Matrices resulting from PDE discretisations on structured/unstructured grids are often (highly) irregular. MFD discretisation always gives unstructured sparse matrices, even though nonzeros in each row can be of almost constant size throughout the grid, as they are in Cartesian grids. Without wells, Cartesian grids result in matrices with only 6 and 11 nonzeros per row, mostly 11 (on big systems).

Implementing dense matrix-vector multiplication operation on CUDA is relatively simple. This operation is limited by *arithmetic intensity* (the number of floating-point operations per loaded data from global memory), and it is possible to approach to the theoretical memory bandwidth of GPUs because each warp reads from a continuous memory segment, and this is often done through a single memory transaction (cf. Section 7.7 on page 56). Sparse matrix-vector multiplication (SpMV), however, has one advantage: we need far fewer floating-point operations. SpMV is therefore not limited by the floating-point throughput. If we are able to access the matrix elements quickly enough, i.e., by utilising the memory bandwidth of the GPU efficiently, we can increase the performance of this operation significantly compared to an optimised CPU implementation.

The irregular nature of sparse matrices makes it difficult to exploit the memory bandwidth of the GPUs. This obviously has to do with the very constrained regularity in access patterns from global memory. GPUs require regular access pattern on a fine-grained level, and sparse matrix operations are typically much less regular in their access patterns and consequently are generally limited purely by bandwidth; the MPs are idle a significant amount of time just waiting for threads because it takes several memory transcations to load data from the device memory. This happens because each warp jumps in multiple memory segments, which requires the same number of transcations, each transaction taking several hundreds of clock cycles. optimisation of SpMV is therefore a data structure problem. With an efficient and clever representation of nonzero elements of a sparse matrix, we can exploit the bandwidth significantly, and thereby boost performance.

The following section is mostly a reference to [5] and [6]. These recent papers discuss data structures and algorithms for SpMV that are efficiently implemented on the CUDA platform for the fine-grained parallel architecture of the GPU. Given the memory-bound nature of SpMV, the authors emphasise memory bandwidth efficiency and compact storage formats. Both regular and irregular matrices are considered.

8.4.1 Sparse Matrix Formats

There are a multitude of sparse matrix representations, each with different storage requirements, computational characteristics, and methods of accessing and manipulating entries of the matrix. We will only be concerned with static sparse matrix formats, as opposed to those suitable for rapid insertion and deletion of elements. The CUSP library is based on [5] (and implemented by the authors) and we start by introducing sparse matrix formats that we

$$S = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \qquad \begin{aligned} \texttt{row} &= [0 \ \ 0 \ \ 1 \ \ 1 \ \ 2 \ \ 2 \ \ 2 \ \ 3 \ \ 3] \\ \texttt{col} &= [0 \ \ 1 \ \ 1 \ \ 2 \ \ 0 \ \ 2 \ \ 3 \ \ 1 \ \ 3] \\ \texttt{data} &= [1 \ \ 7 \ \ 2 \ \ 8 \ \ 5 \ \ 3 \ \ 9 \ \ 6 \ \ 4]. \end{aligned}$$

Fig. 8.3: Coordinate representation of the sparse matrix S.

have used in our implementation.

Coordinate Format

The *coordinate* (COO) format stores a sparse matrix by three one dimensional arrays; the arrays `row`, `col` and `data` store the row indices, column indices and the nonzero matrix entries, respectively. This format represents a *general* sparse format since it can store any sparsity pattern, and the required storage is scalable. In order to store the rows continuously, the `row` array is sorted (sorted COO). Figure 8.3 shows a simple example of the sorted COO format.

Compressed Sparse Row Format

Notice that the `row` array in the sorted COO format will mostly have the row indices repeated. The *compressed sparse row* format (CSR) changes this array (while keeping the others) with a new array `row_offsets`. It has dimension $M + 1$, where M is the number of rows. Row i starts at index `row_offsets`$[i]$ ends at `row_offsets`$[i] - 1$. Similarly, we have *compressed sparse column* format (CSC) where the `data` is sorted after column indices. Figure 8.4 on the next page show an example of CSR format. Convertion between COO and CSR is straight-forward.

ELLPACK Format

This (ELL) format is interesting. Given an $M \times N$ sparse matrix having a maximum of K nonzeros entries per row, we create a dense $M \times K$ matrix, where all the rows having less than K nonzeros elements are padded with zeros. ELL stores the matrix using two one-dimensional array; `data` and `indices` containing matrix entries (sorted by rows) and column indices, respectively. When implementing ELL SpMV, we can either use some sentinel value in the `indices` array to check whether we are at a padded or a real nonzero entry, or we can use the `data` array and check whether the current entry is nonzero. Figure 8.7 on page 69 shows an example of this format. In ELL, `data` and `indices` are stored in column-major order, for

$$S = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \qquad \begin{aligned} \texttt{row_offsets} &= [0 \quad 2 \quad 4 \quad 7 \quad 9] \\ \texttt{indices} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \quad 3 \quad 1 \quad 3] \\ \texttt{data} &= [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4]. \end{aligned}$$

Fig. 8.4: CSR representation of the sparse matrix S.

reasons soon explained. When the maximum number of nonzeros per row does not substantially differ from the average, the ELL format is an efficient sparse matrix format for the GPU, though not for the CPU, as we will see in the Chapter 9. Figure 8.5 on the next page shows a modified version of the SpMV ELL kernel. The access pattern for both `data` and `indices` is fully coalesced, which is the reason why the matrix entries and indices are stored in a column-major order.

Figure 8.6 on page 69 compares the reading patterns of CSR and ELL for a 3×3 dense matrix. Note that ELL is identical to CSC (in the case of dense matrix). As can be seen, each warp of threads reads from adjacent 128-byte memory segments. In the CSR reading pattern, where each row is also assigned to a single row, each thread reads continously in memory, and each memory read requires the warp to read from multiple 128-byte segments (not exactly in this Figure, but if we increase the matrix dimension to for example 1000×1000), thus making the MPs idle. Coalescing works perfectly with column-major order, but fails miserably with row-major.

Hybrid Format

In reservoir grid models we have used in this thesis, the number of nonzeros in each row is uniformly distributed. As mentioned earlier, on Cartesian 3D grids, the number of nonzeros per row is mostly 11. MFD on (highly) unstructured corner-point grids also gives well-suited sparse matrices for the ELL format [4]. The idea of the Hybrid format (HYB) is to represent a sparse matrix initially ill-suited for ELL with a mixture of ELL and COO formats. The rows with the most frequent number of nonzeros are (easily) extracted and stored in the ELL format, and the rest is stored in the COO format.

[4] My knowledge of corner-point grids is limited, both in terms of their creation and variety, but from talks with my supervisor Bård Skaflestad, I have the impression that the number of cell faces may change drastically from cell to cell. On grids where that is the case throughout many cells, ELL format is ill-suited since we have to zero-pad a lot of rows. For corner-point grids I have been exposed to, the ELL format is very flexible, in addition to the fact that this format is extremely easy to implement.

```
__global__ void
spmv_ell_kernel ( const int num_rows,
                  const int num_cols,
                  const int num_cols_per_row,
                  const int *indices,
                  const float *data,
                  const float *x,
                  dobule *y)
{
        int row = blockDim.x * blockIdx.x + threadIdx.x ;
        if(row < num_rows ){
        float dot = 0;
        for ( int n = 0; n < num_cols_per_row ; n ++){
          int col = indices [ num_rows * n + row ];
          float val = data [ num_rows * n + row ];
          if( val != 0)
                dot += val * x[col ];
        }
        y[row] = dot;
        }
}
```

Fig. 8.5: SpMV kernel for the ELL sparse matrix format.

If not given a priori, this typical number is determined from the matrix. The authors of the referred paper computes a histogram of the row size (in CSR format this number given by $\texttt{row_offsets}[i+1] - \texttt{row_offsets}[i]$ for row i) and determines the largest number K such that using K columns per row in the ELL portion of the HYB format is best fitted to performance ratio between ELL and COO. The ELL format is roughly three times faster than COO [nathan]. All the nonzero entries of rows which exceed this number are placed in the COO portion. See Figure 8.8 on page 70 for an example of HYB format. The matrix-vector multiplication $\mathbf{S}_k = \mathcal{L}_k$ using the HYB format is done in two steps, first for the ELL part and then updated with the remaining COO format; $\mathcal{L}_k = \mathbf{S}_{\text{ELL}} p_k + \mathbf{S}_{\text{COO}} p_k$.

8.4.2 The THRUST Library

The implementations in this thesis depends extensively on THRUST and CUSP. THRUST library is an STL analog on CUDA, and offers an intuitive and clearly defined interface to the device. Among the modules of THRUST, we can mention Iterator, Container Classes, Algorithms (including search, copy, reduction, sorting, transform), Memory Management, and Random

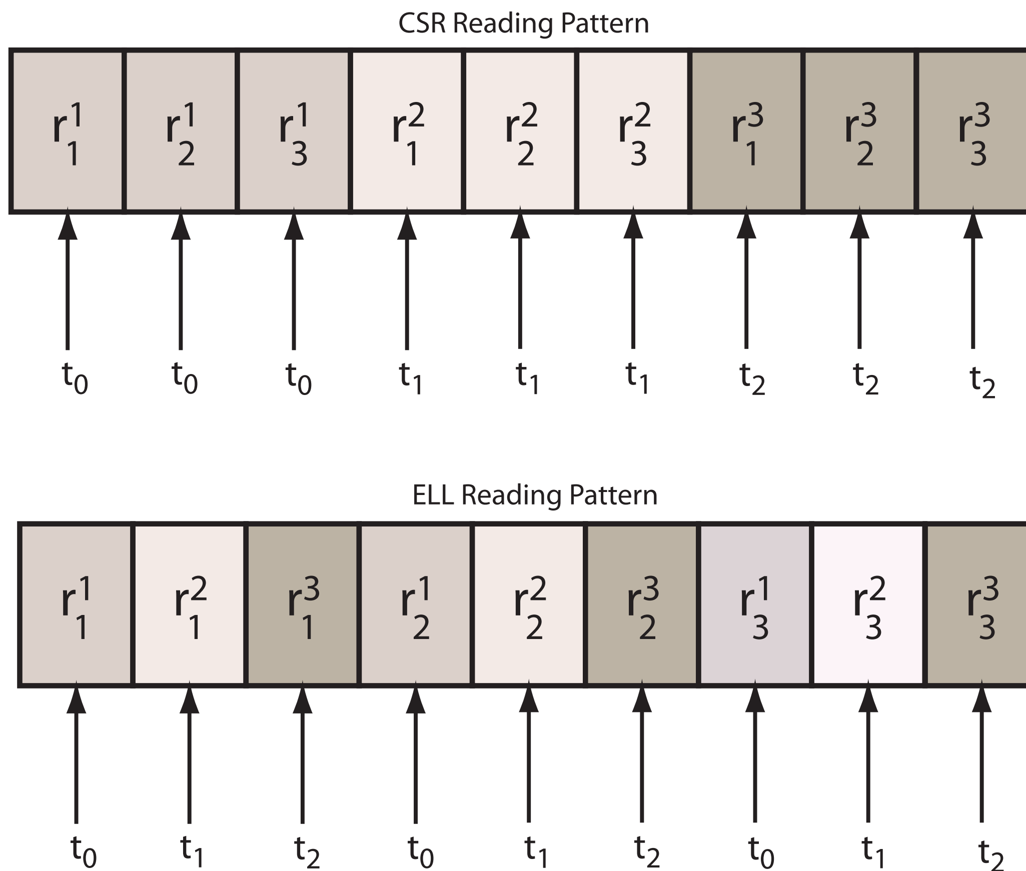


Fig. 8.6: Comparison between the reading patterns of CSR and ELL on GPU. In ELL, consequtive threads read elements from consequtive memory segments. In CSR, each thread reads data from consequtive segments, implying that consequtive threds read data from multiple segments.

$$S = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \qquad \begin{aligned} \texttt{indices} &= [0 \ \ 1 \ \ 0 \ \ 1 \ \ 1 \ \ 2 \ \ 2 \ \ 3 \ \ * \ \ * \ \ 3 \ \ *] \\ \texttt{data} &= [1 \ \ 2 \ \ 5 \ \ 6 \ \ 7 \ \ 8 \ \ 3 \ \ 4 \ \ 0 \ \ 0 \ \ 9 \ \ 0]. \end{aligned}$$

Fig. 8.7: ELL representation of the sparse matrix S. Here, we can either use any unique flag ($*$) in `indices` to check for padded values, or directly check whether the value itself is zero.

Number Generation.

The Container Classes provide two generic vector[5] containers, `host_vector`

[5] See: http://en.wikipedia.org/wiki/Vector_(C++)

$$S = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \qquad \begin{aligned} \texttt{ELL_indices} &= [1\ \ 2\ \ 5\ \ 6\ \ 7\ \ 8\ \ 3\ \ 4] \\ \texttt{ELL_data} &= [0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 2\ \ 2\ \ 3] \\ \texttt{COO_data} &= 9 \\ \texttt{COO_row} &= 2 \\ \texttt{COO_col} &= 3. \end{aligned}$$

Fig. 8.8: HYB representation of the sparse matrix S.

and `device_vector`, residing in host and device memory, respectively. These behave just like `std::vector`, and are able to support any data type dynamically. There is no need for `cudaMalloc` or `cudaMemcpy`; the allocation is automatic. Copying data between these two containers is very easy. Raw pointers to these vector containers can be optained and are easy to send to function kernels for read/write operations. For example,

```
thrust::device_vector<int> D(10, 1);
```

allocates storage for and initializes ten integers of a device vector to 1. This vector can be copied to a vector on host using the overloaded = operator

```
thrust::host_vector<int> H = D;
```

All algorithms have implementations for both host and device. Many of the algorithms have direct analogs in the STL, and we distinguish them by using the namespace `thrust`, such as `thrust::transform`. Figure 8.9 on the following page shows some simple examples.

8.4.3 The CUSP Library

CUSP builds upon THRUST, and implements all the sparse matrix formats mentioned above, both on host and device. Hence, it is easy to run the same code on host or device by storing the data on host or device, respectively. CUSP makes it easy to transfer data between host and device. It supports convertions between different matrix formats on host. For example,

```
cusp::csr_matrix<int,double,cusp::host_memory> A(5,8,12);
```

allocates storage for a CSR matrix on the host with 5 rows, 8 columns, and 12 nonzero values. After initializing `A.row_offsets`, `A.values`, and `A.column_indices`, we can transfer the matrix to device with a single line of code,

```
cusp::csr_matrix<in,double,cusp::device_memory> B = A;
```

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/replace.h>
#include <thrust/functional.h>
#include <iostream>

int main(void)
{
   // allocate three device_vectors with 10 elements
   thrust::device_vector<int> X(10);
   thrust::device_vector<int> Y(10);
   thrust::device_vector<int> Z(10);

   // initialize X to 0,1,2,3, ....
   thrust::sequence(X.begin(), X.end());

   // compute Y = -X
   thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>());

   // fill Z with twos
   thrust::fill(Z.begin(), Z.end(), 2);

   // compute Y = X mod 2
   thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());

   // replace all the ones in Y with tens
   thrust::replace(Y.begin(), Y.end(), 1, 10);

   // print Y
   thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));

   return 0;
}
```

Fig. 8.9: Simple examples of usage of host and deivce thrust vector containers.

or transparently convert it to another format (conversion is done on host)

```
cusp::ell_matrix<in,double,cusp::device_memory> E = A;
cusp::hyb_matrix<in,double,cusp::device_memory> H = A;
cusp::coo_matrix<in,double,cusp::device_memory> C = A;
```

CUSP contains algorithms `cusp::multiply` and `cusp::transpose` for sparse matrix-vector multiplication and matrix transpose, respectively. It provides CG and BICGSTAB solvers `cusp::krylov::cg` and `cusp::krylov::bicgstab`. Both runs on host or device depending on the memory location of the matrix. Vector inner-products in these algorithms uses collection of level-

```
#include <cusp/hyb_matrix.h>
#include <cusp/gallery/poisson.h>
#include <cusp/krylov/cg.h>

int main(void)
{
   // create an empty sparse matrix structure (HYB format)
   cusp::hyb_matrix<int, double, cusp::device_memory> A;

   // create a 2d Poisson problem on a 10x10 mesh
   cusp::gallery::poisson5pt(A, 10, 10);

   // allocate storage for solution (x) and right hand side (b)
   cusp::array1d<double, cusp::device_memory> x(A.num_rows, 0);
   cusp::array1d<double, cusp::device_memory> b(A.num_rows, 1);

   // set stopping criteria:
   //  iteration_limit    = 100
   //  relative_tolerance = 1e-6
   cusp::verbose_monitor<double> monitor(b, 100, 1e-6);

   // setup preconditioner
   cusp::precond::diagonal<double, cusp::device_memory> M(A);

   // solve the linear system A * x = b with the Conjugate Gradient method
   cusp::krylov::cg(A, x, b, monitor, M);

   return 0;
}
```

Fig. 8.10: Example code illustrating the usage of preconditioned CG on CUSP.

1 BLAS routines located in `cusp::blas::*`. Figure 8.10 shows an example of how to solve a linear system on device using the HYB format. Currently, CUSP provides only the diagonal/Jacobi preconditioners through `cusp::precond::diagonal`. A smoothed aggregation preconditioner is currently under development. A draft based on smoothed aggregation is already out in CUSP repository. It has still limited functionality, and we would not expect it to solve anything beyond isotropic Poisson on structured grid[6]. Our use of these libraries makes the code more progressive and relatively soon we can adopt a better preconditioning.

[6] See cusp/precond/smoothed_aggregation.h

Chapter 9

Numerical Experiments

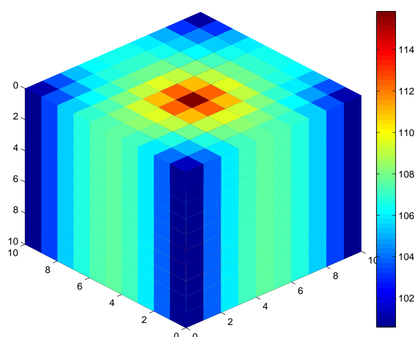
This chapter starts with the verification of the Full-matrix and Matrix-free versions. For this purpose we have chosen a simple 3D Cartesian grid. Since the Matrix-free cannot handle wells, we have made two different tests, one with wells and one without. This is followed by a description and simulation of three reservoir models. These include BIG 1, SPE10 Model 2, and a SAIGUP model together with their permeability fields. We have run simulations on three different GPU models. Test platform specifications are shown Table 9.1. Then we present time and iteration measurements followed by a discussion and interpretation of the results with respect to the two methods.

Table 9.1: Test platform specifications. GTX 480 is based on Fermi GT300.

GPU 1	NVIDIA GeForce GTX 480 (480 cores)	GT300
GPU 2	NVIDIA GeForce GTX 285 (240 cores)	GT200
GPU 3	NVIDIA Quadro FX 4800 (192 cores)	GT200
CPU	Intel Core 2 Quad	
Memory	3×2GB DDR2-6400	
OS	Ubuntu 9.04	
CUDA	CUDA 3.0 (Beta)	
Host Compiler	GCC 4.2.4	

9.1 Verification of the Solvers

To test the two solver versions for correctness, we have chosen a small $10 \times 10 \times 10$ Cartesian 3D grid. To test the Full-matrix version, we add five wells



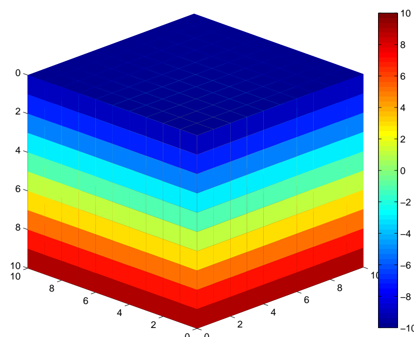


Fig. 9.1: Figure showing face pressures with five wells. The central well is rate-constrained, and corner wells are producers.

Fig. 9.2: Figure showing the solution with face pressure Dirichlet conditions at top and bottom. The solution is a linear pressure fall.

to the model, one injector in the center and four producers at the four corners of the grid. The permeability has been set to $\mathbf{K} = \mathbf{I}$ in each cell, where $\mathbf{I}$ is the identity matrix. The other model specifies face pressure boundary conditions at the top and bottom of the grid, that is, at $z = 0$ and $z = 10$. This with $\mathbf{K} = \mathbf{I}$ gives linear pressure fall. The Full-matrix version runs both of these model, while the Matrix-free version tests only the latter. Figure 9.1 and 9.2 plots solution of the models. These solutions are identical to the solutions MRST's `solveIncompFlow(...)` gives with MFD hybrid.

9.2 Reservoir Models

All of the reservoir models we discuss below are corner-points grids with very complex geometry. This results in a very ill-conditioned linear system which is hard to solve. The SPE10 Model 2 is only solved on GTX 480 and Quadro FX 4800 because of insufficient memory on GTX 285.

9.2.1 SAIGUP Reservoir Model 28

SAIGUP is an abbreviation for *Sensitivity Analysis of the Impact of Geological Uncertainties on Production forecasting in clastic hydrocarbon reservoirs*, and is an international research project funded through the Energy Project ENK6-2000-2007.

Estimates of recovery from oil fields are often found to be significantly in error. This can be caused by the uncertainty of the geological description

within the simulation model[49]. The multidisciplinary SAIGUP project was designed to analyse the sensitivity of estimates of recovery due to geological uncertainty in a synthetic suite of shallow-marine reservoir models. These sedimentological models were combined with structural models sharing a common overall form but consisting of three different fault systems with variable fault density and fault permeability characteristics and a common unfaulted end-member[50].

The purpose of the project is stated as:

- to quantify objectively the sensitivity of geological complexity on production forecasts, as a function of generic aspects of both the sedimentological architecture and faulted structure of shallow-marine hydrocarbon reservoirs, and
- to validate these results using real-case reservoirs and production data.

In this project we consider one of the many SAIGUP models [1], specifically model realisation number 28. Figure 9.3 on the next page shows this model with configuration wells. The x- and z-permeability components for this model are shown in Figure 9.4 on page 77 and 9.5 on page 77. The y-permeability is similar to x-permeability. The plots show that SAIGUP has a highly heterogeneous permeability field. The red areas are permeable, and the blue areas are impermeable or less permeable. This model has 78720 cells and 264305 faces. The approximate condition number of the matrix resulting from MFD discretisation has been found to be 2.3×10^{17}, which gives a very ill-conditioned system. The maximum and minimum number of cell faces is 20 and 6. Only about 16% of the cells have more than 6 faces, and less than 4% has more than 10 faces, making it an appropriate grid for ELL and/or HYB format.

Figures 9.6 on page 77 and 9.7 on page 77 plot, respectively, the cell pressure p and face pressure π. Figure 9.8 on page 78 shows a plot of a parameter describing "flux intensity" given per cell. This value is derived from the face fluxes v of a cell, and reduced to a single value for the cell. The plot emphasizes both low and high flux regions. As we can see, the pressure difference is extremely low. This is because the permeability values are very small, as can be seen from the permeability plots (logarithmic scale). Another permeability field would have given another pressure distribution.

[1] It was given to me by my supervisor Bård Skaflestad, together with the permeability and well data

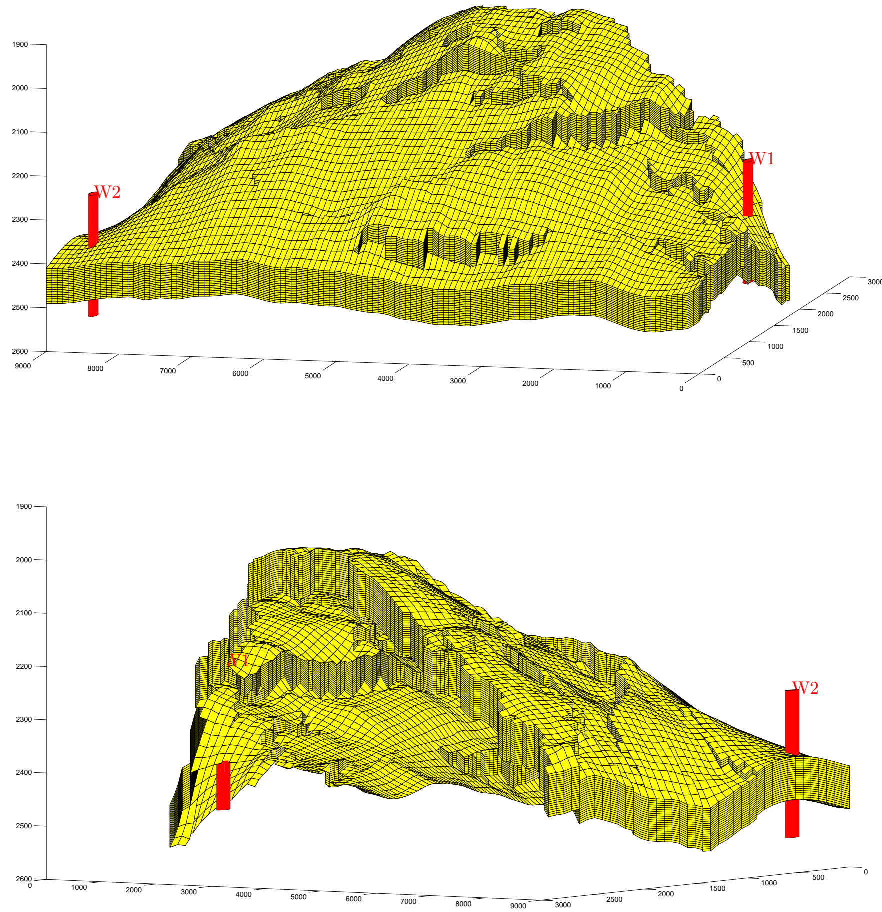


Fig. 9.3: The SAIGUP model. The model streches about 9 kilometers in the x-direction, 3 kilometers in the y-direction, and 600 meters in the vertical direction. The model also has two vertical wells used for simulation in this thesis.

9.2.2 SPE10 Reservoir Model 2

This model is part of the *2001 SPE Comparative Solutions Project* and is provided by SPE, Society of Petroleum Engineers. The purpose of the projects has been to provide bechnmark datasets which can be used to compare the performance of different simulators or algorithms. The synthetic reservoirs have been generated on the basis of how present shoreface depositions are.

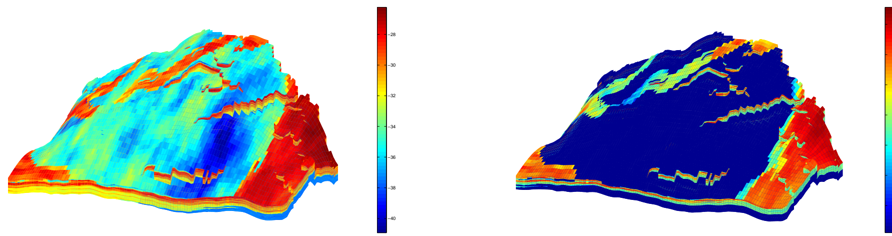


Fig. 9.4: Plot of x-permeability field of the SAIGUP model on a logarithmic scale.

Fig. 9.5: Plot of z-permeability field of the SAIGUP model on a logarithmic scale.

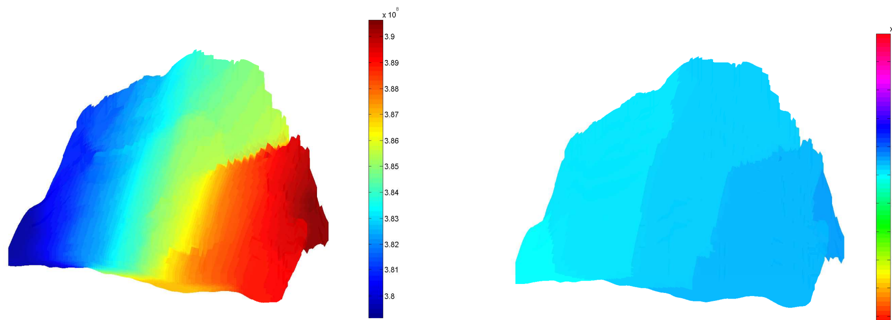


Fig. 9.6: Figure showing the cell pressure p of the SAIGUP model.

Fig. 9.7: Figure showing face pressure π of the SAIGUP model.

Reservoir Description

Model 2 is a million cell dataset where the CPU time to use classical pseudoisation would be excessive. This model has a sufficiently fine grid to make use of classical pseudoisation methods almost impossible. The model has a simple geometry without any top structure or faults. At the geological model scale, the model is described on a regular Cartesian grid and has dimension $1200 \times 2200 \times 170$ (ft^3). The top 70 ft, corresponding to 35 layers, represent the shallow-marine *Tarbert formation*, where the permeability is relatively smooth, and the bottom 100 ft, corresponding to 50 layers, represents fluvial *Upper Ness* permeability. Both formations are characterised by large permeability variations, 8–12 orders of magnitude. The fine scale cell size is $20 \times 10 \times 2$ ft^3. The fine scale model size is $60 \times 220 \times 85$ cell making up $1,122,000$ cells and $3,366,000$ faces. Since each cell has 6 faces, this Cartesian model is appropriate for the ELL/HYB formats.

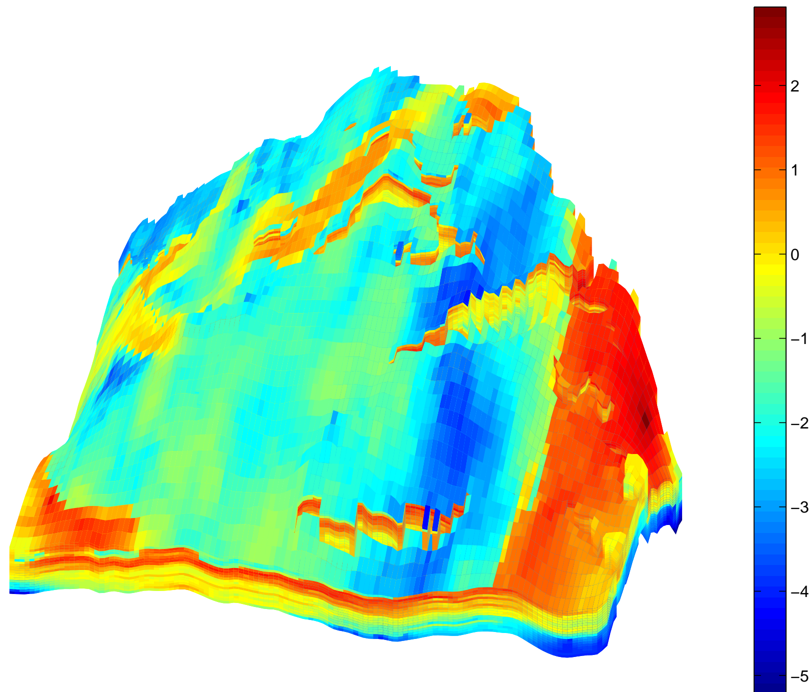


Fig. 9.8: Figure showing the flux intensity of the SAIGUP model.

Well Configuration

All wells are vertical, and the model contains five wells. Figure 9.9 on the next page shows a scaled version of the grid with the well locations. The central well is an injector with an injection rate of 5000 bbl/day. Max injection bottom hole pressure is $10,000$ psi. The four producers at the four corners with a bottom hole pressure of 400 psi.

Figure 9.9 on the following page shows a upscaled version of SPE10 to illustrate the well locations. Figures 9.10 on the next page and 9.11 on the following page show the permeability field of SPE10 on logarithmic scale. The two permeability layers dividing the reservoir is clear. The ratio between the maximum and minimum permeability in the grid is of order 10^{12}. The linear system resulting from the MFD discretisation has a condition number of approximately the same order, making the matrix very ill-conditioned. Figures 9.12 on page 80 and 9.13 on page 80 show the cell pressures and flux

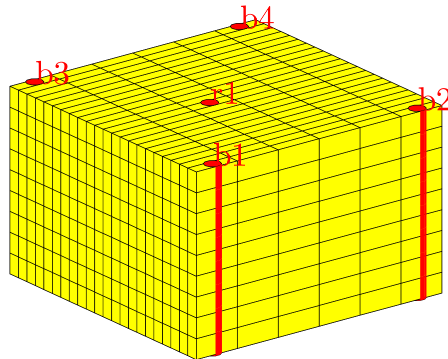


Fig. 9.9: Figure showing an upscaled version of the SPE10 model to illustrate the well locations. The central well is an injector (rate-controlled) and the the four wells at the corners are pressure-controlled producers.

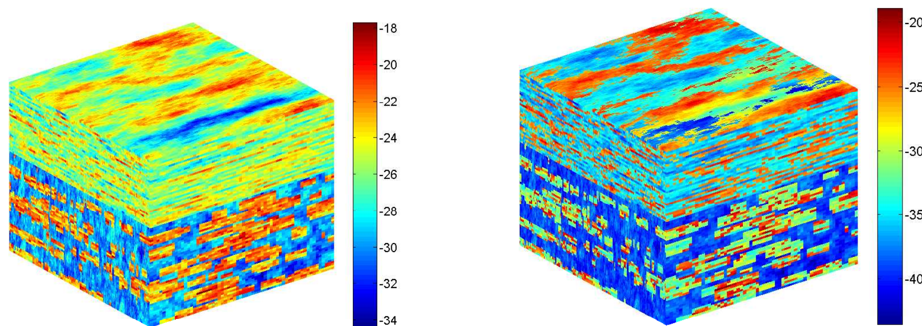


Fig. 9.10: Plot of x-permeability field of SPE10 on a logarithmic scale.

Fig. 9.11: Plot of z-permeability field of SPE10 on a logarithmic scale.

intensity of the grid, respectively.

9.2.3 BIG 1 Reservoir Model

This is yet an unfinished reservoir model developed by SINTEF[2]. The geometry of the grid looks very complex when perceived. Basically, BIG 1 consists of two layers; the upper portion of the model is a regular Cartesian grid, joined with an irregular corner-point grid. Figure 9.14 on the next page shows the model. It has $538,765$ cells and $1,550,505$ faces. Each

[2] I have no knowledge about this model other than what I have been told by my supervisor. The model remains to be upscaled.

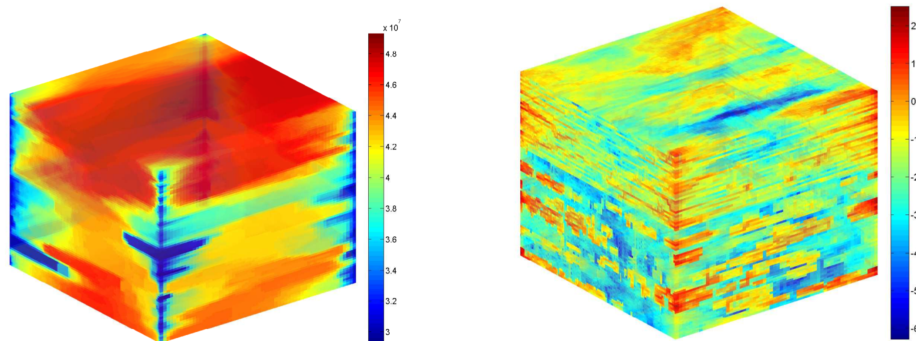


Fig. 9.12: Figure showing the cell pressure of the SPE10 model.

Fig. 9.13: Figure showing the flux intensity of the SPE10 model.

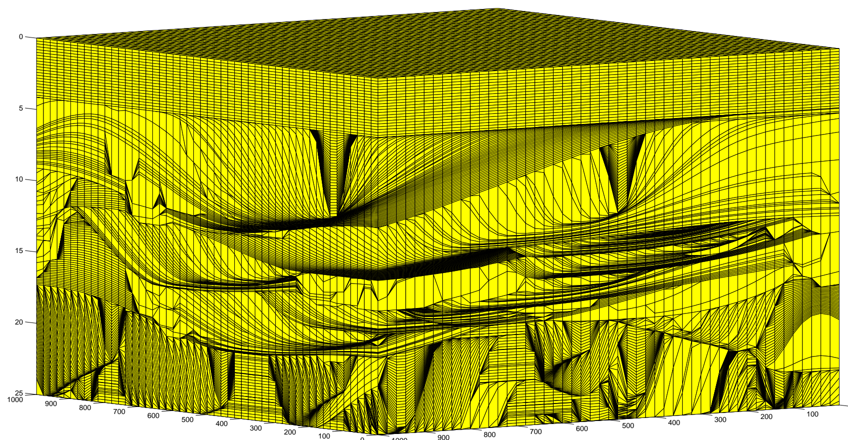


Fig. 9.14: Side-view of the BIG 1 model. The upper portion of the model is Carteisan, jointed with a complex lower corner-point grid.

cell has faces between six and four, and apprx. 79% of the cells has six cell faces. This makes BIG 1 a suitable grid for the ELL/HYB format. Figures 9.15 on the following page and 9.16 on the next page shows the x- and z-permeability fields, respectively. The y-component has a similar distribution. The permeability distribution is heteregeneous, giving an ill-conditioned system.

Since SAIGUP and SPE10 models contain wells, they can only be solved using the Full-matrix version. BIG 1 has no well, so this is the only model

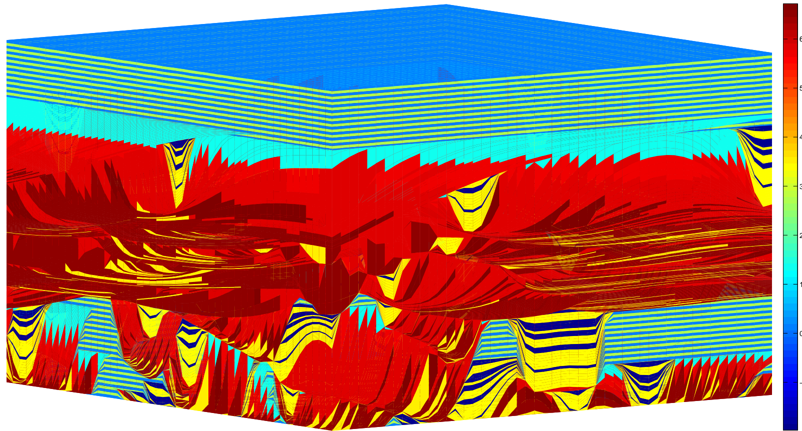


Fig. 9.15: Plot of y-permeability field of BIG 1 on a logarithmic scale.

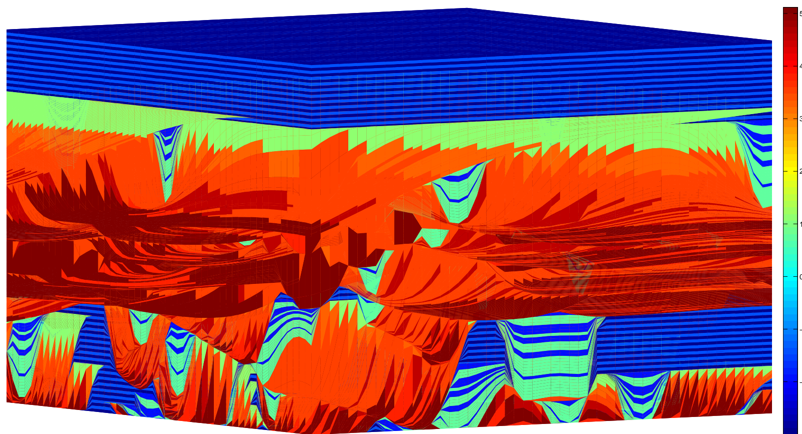


Fig. 9.16: Plot of x-permeability field of BIG 1 on a logarithmic scale.

that has been used to test the Matrix-free version. This model includes simple boundary conditions with a source and a sink at global cell numbers 1 and M, respectively. The rate of these sources was set to 10 and -10, which does not have any physical motivation.

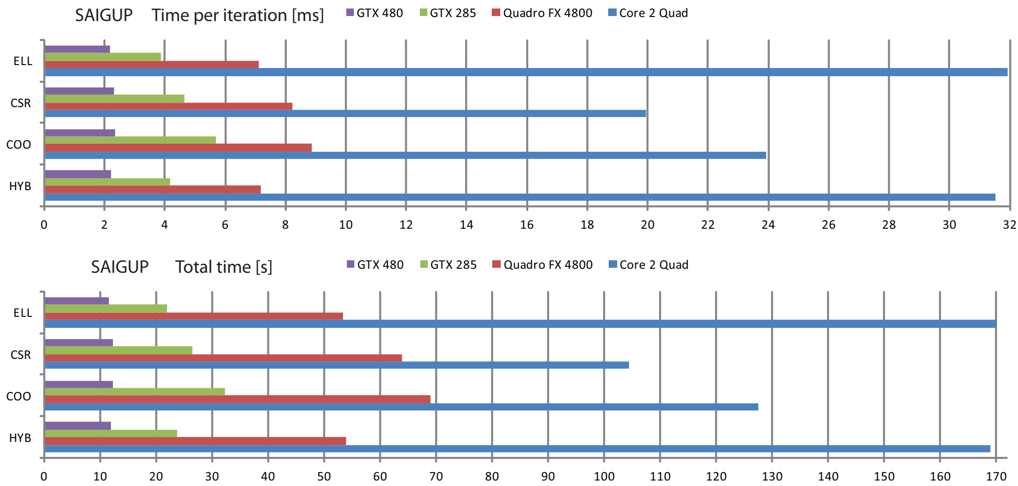


Fig. 9.17: Timings for SAIGUP: The top Figure shows time in milliseonds per CG-iteration for the different sparse matrix formats on the three GPU models and the CPU. ELL and HYB perform fastest on the GPUs, being slowest for the CPU. CSR performs best on the CPU, being slowest for the GPUs. The bottom Figure shows total time in seconds for different sparse matrix formats.

9.3 Speedup Measurements

As mentioned above, we have used three different NVIDIA GPU models in our tests. The three reservoir models were run separately on the individual GPUs. To compare the running time of the GPUs with a CPU, the models were simulated on the Intel Core 2 Quad CPU using the Full-matrix version. The relative error tolerance for BIG 1, SAIGUP, and SPE10 has been set to, respectively, 5.0×10^{-11}, 5.0×10^{-11}, and 4.0×10^{-7}.

Figures 9.17, 9.18 on the next page, and 9.19 on page 85 plots a comparison of the timing results for SAIGUP, SPE10, and BIG 1, respectively. It has been appropriate to compare the total time needed for convergence to a given relative tolerance, and the time per a single CG-iteration. The total run time is shown in seconds, and the time per iteration is given in milliseconds. The timing results compare run time for each sparse matrix format on the three GPUs and the CPU. The general trend from the tests indicates that ELL and HYB formats are the fastest formats on all GPUs. For the models we use, there is essentially no performance deviation between HYB and ELL. HYB will most probably provide better performance for models that are large and contain several large wells resulting in a significant variation in the number of nonzeros per row. HYB will also be more flexible on grids that have an

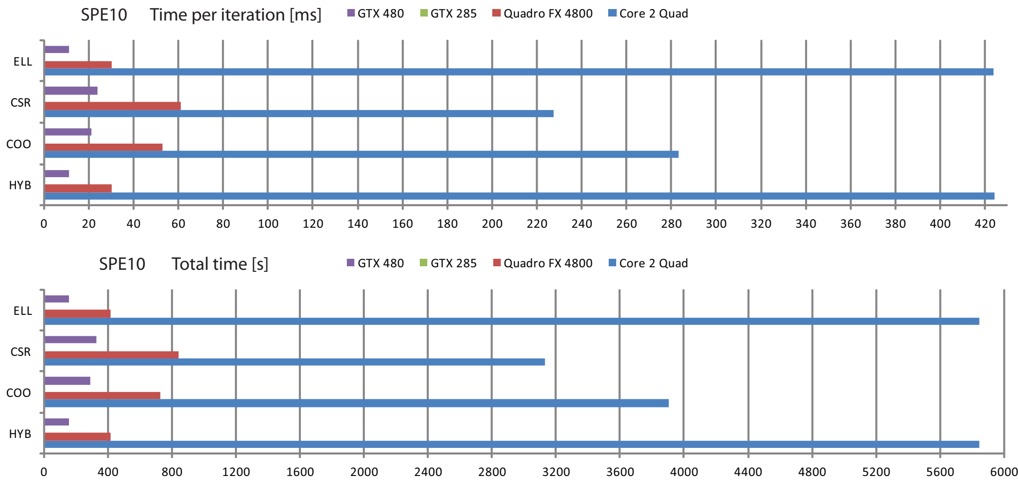


Fig. 9.18: Timings for SPE10: The top Figure shows time in milliseonds per CG-iteration for different sparse matrix formats on the three GPU models and the CPU. ELL and HYB are fastest on the GPUs, being slowest for the CPU. CSR performs best on the CPU, being slowest for the GPUs. The bottom Figure shows total time in seconds for different sparse matrix formats.

uneven distribution of the number of faces per cell. CSR does not perform well on any GPU, and the reasons has already been discussed in Section 8.4.1; CSR suffers from non-coalescing, resulting in wasted bandwidth. It is still the superior format for the CPU. It is nicely cached because of its regular structure in memory. ELL and HYB are highly irregular for the CPU, becaue each row is essentially scattered in the memory. It is interesting to learn that the highly optimised data structures for the CPU are equally unoptimised for the GPU, and vice versa.

Tables 9.2 on the next page gives a summary of the results. For each combination of the models and the platforms, it shows the fastest sparse matrix format, and gives the toal number of iterations, time[ms] per iteration and the total time[s]. HYB and ELL perform equally well on each platform and reservoir model. From this Table, it is interesting to note that all the GPUs deliver better performance on bigger models. SAIGUP is the least model, giving a maximam speedup of 9.0 on GTX 480. SPE10 and BIG 1 approach 21. SPE10 is almost twice as big as BIG 1, so the results also indicate that there is an upper limit of the speedup factor.

Table 9.3 on the following page shows timing results for the Matrix-free version. If we compare the performance of each GPU with respect to the two versions (only BIG 1), we see that the Full-matrix version is about 1.38 times

Table 9.2: Summary of the timing results. Each entry shows the fastest performing format, the total number of iterations, time[ms] per iteration, and the total time needed for convergence, in the same order. HYB and ELL formats are significantly identical, so for the models used, both are optimal. Speedup is computed with respect to the CSR times on Intel Core 2 Quad.

	BIG 1	**SAIGUP**	**SPE10**
Intel Core 2 Quad	CSR 45829 100.8 4621	CSR 5235 19.94 104.4	CSR 13784 227.3 3139
Quadro FX 4800	ELL (HYB) 45732 (45732) 18.24 (18.24) 834 (834)	ELL (HYB) 5692 (5691) 7.44 (7.78) 42.24 (44.3)	ELL (HYB) 13777 (13778) 30.25 (30.19) 416.7 (416)
GTX 285	ELL (HYB) 45771 (45771) 10.29 (10.25) 470.8 (469.2)	ELL (HYB) 5692 (5691) 3.87 (4.00) 22 (22.85)	
GTX 480	ELL (HYB) 45734 (45754) 4.83 (4.83) 221 (221)	ELL (HYB) 5320 (5320) 2.18 (2.23) 11.6 (11.89)	ELL (HYB) 13775 (13778) 11.18 (11.25) 154 (155)
Speedup (FX 4800)	5.54	2.47	7.53
Speedup (GTX 285)	9.81	4.74	
Speedup (GTX 480)	20.91	9.0	20.38

Table 9.3: Timing results of BIG 1 on the Matrix-free version. Speedup indicates how fast each Matrix-free iteration is running with respect to the fastest CPU Full-matrix version.

Matrix-free	**Iterations**	**Time[s]**	**Time [ms/itr]**	**Speedup**
Quadro FX 4800	38127	958	25.13	4.0
GTX 285	38124	616-73	16.18	6.21
GTX 480	38118	508	13.32	7.54

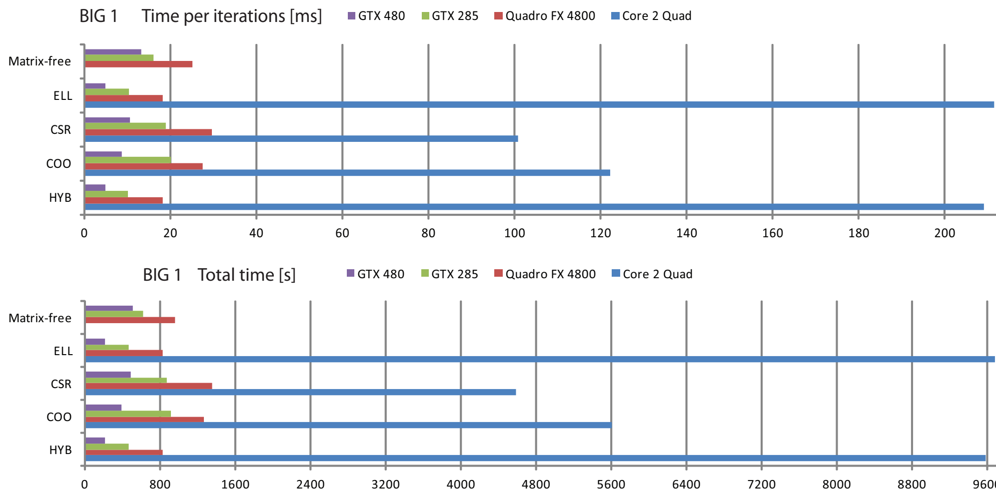


Fig. 9.19: Timings for BIG 1: The top Figure shows time in milliseonds per CG-iteration for different sparse matrix formats on the three GPU models and the CPU. ELL and HYB are fastest on the GPUs, being slowest for the CPU. CSR and COO performs best on the CPU, being slowest for the GPUs. The bottom Figure shows total time in seconds for different sparse matrix formats.

faster on Quadro FX 4800, 1.58 times faster on GTX 285, and 2.77 times faster on GTX 480. It is interesting to note this increase in percentage as we move to better GPU. We will discuss these results in subsequent sections.

9.3.1 Comparision with Solvers from SINTEF

It was desirable to compare the fastest GPU run times (GTX 480) with existing iterative solvers at SINTEF. It should be noted that the GPU solvers in this thesis are only Jacobi-preconditioned. SINTEF has only iterative solvers with more advanced preconditioning, such as ILU and AMG. This means that the comparison is only intended to compare the running time, although the methods cannot be directly compared because of differences in preconditioners.

Table 9.4 on the next page shows results obtained from solvers at SINTEF with the same configuration of boundary conditions and convergence tolerance criteria as used for the tests described in Section 9.3. Solver 2 is AMG-preconditioned CG, and Solver 1 has another preconditioner.

We notice that these advanced serial solvers have relatively heavy iterations of several milliseconds. These require, however, very few iterations to converge

Table 9.4: Test results from two preconditioned CPU iterative solvers developed by SINTEF. The models have the same boundary and convergence conditions as those run on the GPUs. The desktop computer running ubuntu OS has this configuration; Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz, 12 GB RAM.

	TOL	**Iterations**	**Time[s]**	**Time[ms/itr]**
SPE10	4.6×10^{-7}	Solver 1: 87	113	1299
SAIGUP	5.0×10^{-11}	Solver 2: 188	20	106.4
BIG 1	5.0×10^{-11}	Solver 1: 726	380	523.4
	5.0×10^{-11}	Solver 2: 45	47.7	1060

compared with the Jacobi-preconditioned CG on GPU. Solver 1 requires only 113 iterations for SPE10. This is impressing compared to the 13,775 iterations with Jacobi on GTX 480. Each iteration on GTX 480 is still very fast compared to these serial solvers; SPE10 iterations are ca. 116 times faster, SAIGUP iterations are ca. 49 times faster, and each BIG 1 iteration is 108 faster than Solver 1 and 219 faster than Solver 2. In terms of convergence time, SPE10 is approx. 36% faster with Solver 1. BIG 1 with Solver 2 is approx. 4.6 times faster. Solver 1, on the other hand, is approx. 72% slower on CPU, and the same is SAIGUP with solver 2.

The main disadvantage with the GPU solver is that it requires too many iterations because of the trivial preconditioner. This indicates that support for advanced preconditioners on GPUs can significantly lower the iteration number and boost performance. The developers of the CUSP library are working on an AMG preconditioner.

9.3.2 Discussion of the Results

The Full-matrix version has already been discussed as far as the SpMV is concerned, as well as different SpMV realisations through different sparse matrix representations. The rest of this section discusses the primary reasons for the slow performance of the Matrix-free version.

Based on the results we have from time measurements, it seems that the only advantage of the Matrix-free version is that it uses less memory because the cell matrices stores only the symmetric upper half portion. We will now discuss some of the reasons for this speedup deviation between the two versions, and also see if there is any possibility at all to optimise the Matrix-free version.

It is hard to say with certainty how many memory accesses and floating-point

operations these versions make compared to one another, but we make a little heuristic reasoning to find the average case. On a Cartesian grid of dimension 100^3 with one million cells and three million faces, we find that the Matrix-free has approx. 15% more memory accesses, and 9% more floating-point operations. It is difficult to say to what extent this contributes to extend the time per iteration, but it certainly is a factor worth mentioning. We could actually have lowered the number of memory accesses by only reading the upper part of the matrix through shared memory, but that is very hard because the amount of shared memory per block is very limited, even on the new Fermi. With a block size of 64 threads (which is the minimum size recommended for sheduling enough threads to avoid idle processors [19]), each block would require more than 13 KB of shared memory (each cell matrix is 6×6. We also need to store the right-hand side 6-element vector for each thread, giving $(21+6) \cdot 64 \cdot 8 = 13824$ KB memory), making only four blocks run concurrently on Fermi using 48 KB shared memory. That would rather degrade the performance. In the current Matrix-free implementation, the matrix elements are not cached, as in the ELL kernel, but read directly from global memory. A total of N_i^2 reads are carried out per thread, where N_i is the number of faces of cell i.

The most important factor is, however, not that the Matrix-free does a little more work in each iteration. The essential factor is due to the irregular and scattered memory accesses. Memory coalescing was mentioned in Section 7.7 on page 56, and during the discussion of sparse matrix-vector multiplication in Section 8.4 on page 64, the main focus was to create sparse matrix data structures that would enable coalsecing. The HYB and ELL formats are fully coalesced and hence optimised with respect to memory accesses, which is essential on GPUs. The HYB and ELL utilises upto 63% of the memory bandwidth[5].

In the early phase of the project I discussed with my supervisor implementation of a Matrix-free version that never actually assembles the full stiffness matrix. During the first month, I implemented a Matrix-free version for Cartesian grids and then modified it to run on general corner-point grids. At that time I became more and more aware that any optimisation of this version would be difficult, if worth optimizing at all, and certainly time consuming. No immediate idea came into mind about any type of optimisation.

We have seen that the CSR format is difficult to optimise on GPU (cf. Section 8.4 on page 64,[5] and timing results). The most straight-forward way to process CSR on GPU is to let one thread take care of one row. Since each row is stored continuously in memory, each warp reads far outside of the

memory segment which would allow for coalescing and make all 16 readings in a single transcation. For instance, having 11 nonzeros per row (in doubles), each row would read 11 8-byte words from a continuous segment of memory, and so the whole warp would read 176 words from a continous segment, which in terms of a 128-byte segments are 11 segments. That is, the 16 reads in a single warp takes 11 transcations, wasting more than 90% of the total memory bandwidth of the GPU.

The authors of [5] also implemens a CSR-kernel called csr-vector implementation. This assigns each warp to a single row. This version is slightly better than assigning single thread to a row, but coalescing is still only partial, and the disadvantage is that occupancy of the device remains low because many threads are doing what only a single thread does in the other version. From the similarity between the CSR and Matrix-free version, speedup could be improved, though not significantly, by assigning each matrix to a single warp of thread. This could not be accomplished primarily because of the limited time.

It is difficult to think of any way of reading CSR format such that the reads are coalesced. The only very difficult and cumbersome way of doing this would be to scatter the nonzero row elements at different places in the `data` array. Given that the number of nonzeros varies across rows, this type of way would be difficult to generalize.

The Matrix-free version suffers from the same type of irregular access pattern and thereby wasted bandwidth. Each cell matrix can be compared to a single CSR row. Cell matrices are stored consecutively in memory, which on CPU is the most reasonable and straight-forward structure (remember that CSR is fastest on CPU). If each cell has 6 faces, each cell matrix has 21 entries, and a single warp reads in 21 different 128-byte memory segments, wasting more than 95% of the total bandwidth. Given that the sparse matrix-vector multiplication is bandwidth limited, this is disappointingly unoptimised.

It is very hard to think of any logical or reasonable data structure which would eable each warp to read in fewer transcations. This would again mean a scattering of cell matrix entries in such a way that the each warp could read from nearby segments. In the Matrix-free case, this is even harder to think of any such structure than CSR, essentially because we have two levels of information stored in the cell matrix array; we store a large number of cell matrices, located consecutively, and each cell matrix has multiple rows (if the full cell matrix is stored, or, as in our case, row i has $N_i - i$, the first row having N_i elements and the last having only one). Another point worth mentioning is that if one thread is assigned to a single cell, then in order to

read from a single segment, each matrix must only have 8 double elements, which is not the case. Given that we have minimum 21 entries, each warp still has to read in three segments.

The CSR Full-matrix versions on Quadro FX 4800, GTX 285, and GTX 480, are, respectively, 39%, 46%, and 55% slower compared to ELL/HYB results given in Table 9.2 on page 84. This is comparable to the Matrix-free speedups given in Table 9.3 on page 84. Apart from miner implementation details, these versions have identical structure and access the memory segments in a similar way.

The conclusion we draw based on the discussion above is based on how the GPU works with respect to global memory access. Given the data structure we use to store the cell matrices in succession, it is almost impossible to optimise it for regular reading pattern. Although this was the case, we do not exploit a significant percentage of the memory bandwidth.

Chapter 10

Conclusions and Summary

This thesis has studied a mathematical model that describes a single-phase incompressible fluid flow through porous medium. The application has been made in reservoir simulation with the porous medium as an oil reservoir. Pressure- and rate-controlled wells was also considered. This is a well-known model and is based on the conservation law and Darcy's law that combines the pressure gradient and fluid flux through porous media. The model takes the form of an elliptic partial differential equation, which is discretised by the Mimetic Finite Difference method using the hybrid technique. The hybrid Mimetic Finite Difference method can handle advanced and complex meshes in a simple manner, compared to, for instance, Finite Element methods. The discretisation results in an SPD linear system of the form $\mathbf{S}\pi = \mathbf{R}$.

The matrix $\mathbf{S}$ is sparse, and the system has been solved using the iterative Jacobi-preconditioned conjugate gradient method. The CG method is appropriate for large problems and has low storage requirements. In addition, it is well-suited for parallellisation.

The matrix $\mathbf{S}$ is the global matrix assembled from local cell matrices S_i from each cell i in the grid. We also implement a second version of the CG-algorithm that does not assemble $\mathbf{S}$, but works directly from the cell matrices. Instead of assembling the matrices, the matrix-vector product $\mathbf{S}p$ needed in each iteration of the CG is computed by calculating the contribution $L_i = S_i p$ from each cell i. L_i is then assembled in the global $\mathbf{L} = \mathbf{S}p$. This version is named Matrix-free. The other method is named Full-matrix. Because of implementation-specific details, the Matrix-free method cannot handle wells.

The matrix $\mathbf{S}$ has been represented in various sparse matrix formats. These include CSR, COO, ELL, and HYB. Both the Full-matrix and Matrix-free

versions make extensive use of CUSP and THRUST, two CUDA libraries. THRUST is an STL analog in CUDA, and makes the interface to GPU much more intuitive. CUSP is based on THRUST, and implements the sparse matrix formats mentioned here. It includes operations such as sparse matrix-vector multiplication, matrix transpose, matrix conversions, and wraps these operations in a clearly set out library containing the CG- and BICGSTAB-algorithm. Switching the code to run between host and device is very easy.

The primary purpose of the thesis has been to take advantage of a CPU-GPU heterogeneous computing system and exploit GPU's parallelism to increase performance of the iterative solvers. Modern GPUs are no longer function-specific targeted only for graphics processing. Nor do they require exclusively any high-level shading languages or an intimate knowledge of the hardware architecture to access their enormous capacity of performing TFLOPs and memory bandwidth. Performance of GPUs in terms of FLOPs and their programmability has increased, and high-performance computing has benefited largely from GPUs during recent years. The implementations in this thesis have used CUDA. CUDA is an extension to C++, but is also a software and hardware architecture reflecting NVIDIA's modern GPUs.

The underlying conclusion that can be drawn from numerical tests is that the performance increase has been significant compared to the corresponding CPU implementations. Given that we follow the optimisation techniqiues for GPUs, porting of serial algorithms to heterogeneous systems by making them parallel provide remarkable increase in performance.

The implementation starts by reading two files containing grid geometry and permeability for the reservoir. This is used to compute the cell matrices on the GPU. The Full-matrix method assembles the global matrix $\mathbf{S}$ on CPU, and then solves the final system on the GPU. Only double precision is used.

For numerical experiments, we chose three reservoir models. Because of heterogeneous permeability fields and complex reservoir geometry, the linear systems resulting from each of these models are ill-conditioned with very high condition numbers. SAIGUP, the least model, has an approximate condition number of 2.3×10^{17}, and for SPE10, the biggest model, the condition number is of order 10^{12}. Such matrices require many iterations to converge. Trivial preconditioners such as Jacobi can decrease this number somewhat, but still requires an order of 30 - 1000 times more iterations compared to more advanced preconditioners such as ILU and AMG.

The Full-matrix version simulates the three models on three different GPUs with various specifications. The latest among these is the GTX 480 based

on the recent Fermi-architecture from NVIDIA. Compared to the Jacobi-preconditioned CPU (Intel Core 2 Quad) run time, GTX 480 is about 21 times faster on the biggest SPE10 model, and 9 times faster on the least SAIGUP model. The intermediate model, BIG 1, is about 20.38 times faster. Given that we use only double precision, this is remarkable.

SAIGUP and SPE10 are both configured with wells, so the only model tested on the Matrix-free version is BIG 1. Matrix-free is about 7.54 times faster on GTX 480 than the Full-matrix CPU version, which is about 2.7 times slower than the corresponding Full-matrix time on GTX 480.

Matrix-free is therefore significantly slower than Full-matrix, and the reason has been discussed to be suffering from non-coalescing. Memory coalescing on GPU is the most important optimisation. The data structure used for storing the cell matrices on GPU does not allow for coalesced memory reads, and essensially poses the same optimisation challenges as the CSR format. This is because each CSR row is equivalent to a cell matrix in terms of reading pattern. Any optimisation of a general CSR format would require padding almost every row to equalize the number of words in memory for each row storage, but this is done effectively and cleverly by the ELL and HYB formats. Optimising the Matrix-free version is even more difficult because each cell matrix stores more information than a CSR row; each cell matrix stores several rows. Another disadvantage with the Matrix-free version is that it may pose difficulties when implementing non-trivial preconditioners.

Finally, the numerical tests demonstrate that using only double precision on recent GPUs is not neccessarily a disadvantage, given that the double precision is invariant to problem sizes and problem difficulties in numerical methods. We also observe that bigger problems on GPUs has clear benefits in terms of performance.

10.1 Further Work

Models we have used in numerical tests are very complex in terms of solving the resulting linear system effectively without good preconditioners. The CG method is almost useless witout good preconditioners on such system. Implementation of a non-trivial preconditioner in combination with what has been implemented would increase the performance benefits compared to any CPU implementations. Comparisions with advanced preconditioned iterative solvers developed by SINTEF also shows that these trivial GPU implementations can compete with these advanced solvers because of fast iterations, so any non-trivial preconditioning on GPU would make these

implementations very attractive.

The current mimetic method does not add the gravitational effects in the model, which should be added to make the model more realistic. In addition to that, other discretisation methods, such as TPFA, the O-method, and Mixed Finite Element methods, would benefit from such implementations. It could also be interesting to compare these methods to each other. An MRST-like library, based on CUSP, THRUST, and some other libraries, such as CUDPP, implementing all these numerical methods, where flexibility, maintainability, and performance are the keywords, could give enormous benefits.

Bibliography

[1] T. R. Hagen J. M. Hjelmervik A. R. Brodtkorb, C. Dyken and O. O. Storaasli. State of the art in heterogeneous computing. 2010.

[2] Naga Govindaraju Mark Harris Jens Kruger Aaron E. Lefohn John D. Owens, David Luebke and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. pages 21–51, August 2005.

[3] GPGPU. http://gpgpu.org/.

[4] Nocedal Wright. *Numerical Optimization*. Springer, 2 edition.

[5] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda, Desember 2008.

[6] Rajesh Bordawekar Muthu Manikandan Baskaran. Optimizing sparse matrix-vector multiplication on gpus. April 2009.

[7] K.-A. Lie G. Hasle and E. Quak, editors. *Geometrical Modeling, Numerical Simulation and Optimisation: Industrial Mathematics at SINTEF*, pages 265–306. Springer Verlag, 2007.

[8] Pierre Donnez. *Essentials of Reservoir Engineering*. Editions Technip, 2007. ISBN: 2710808927.

[9] P. Lotstedt B. Engquist and O. Runborg, editors. *Multiscale Modeling and Simulation in Science*, volume 66, pages 3–48. Springer Verlag, 2008.

[10] S. Krogstad J. E. Aarnes and K.-A. Lie. Multiscale mixed/mimetic methods on corner-point grids. *Computational Geosciences*, 12(3):297–315, 2008.

[11] K.-A. Lie E. Aarnes, S. Krogstad and V. Laptev. Multiscale mixed methods on corner-point grids: mimetic versus mixed subgrid solvers, Desember 2006. SINTEF Report A578.

[12] K. Lipnikov F. Brezzi and V. Simoncini. A family of mimetic finite difference methods on polygonal and polyhedral meshes. *Math. Models Methods Appl. Sci.*, 15:1533–1551, 2005.

[13] M.Shashkov F. Brezzi, K. Lipnikov and V. Simoncini. A new discretization methodology for diffusion problems on generalized polyhedral meshes. *Comput. Methods Appl. Mech. Engrg.*, 196:3682–3692, 2007.

[14] A. Buffa F. Brezzi and K. Lipnikov. Mimetic finite differences for elliptic problems. *M2AN: Math. Model.*, 2008.

[15] Susanne C. Brenner and L. Ridgeway Scott. *The Mathematical Theory of Finite Element Methods*. Spring, 2 edition.

[16] B. Skaflestad and S. Krogstad. Multiscale/mimetic pressure solvers with near-well grid adaption, September 2008. Proceedings of ECMOR XI, Bergen, Norway.

[17] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6), Desember 1952.

[18] Ke Chen. *Matrix Preconditioning Techniques and Application*. Cambridge University Press, 1 edition, 2005.

[19] NVIDIA Corporation. Nvidia cuda programming guide, 2010. Version 3.0.

[20] M. Januszewskia and M. Kostur. Accelerating numerical solution of stochastic differential equations with cuda. August 2009.

[21] Sheetal Lahabar and P. J. Narayanan. Singular value decomposition on gpu using cuda. pages 1–10, 2009.

[22] Eitan Grinspun Peter Schroder Jeff Bolz, Ian Farmer. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. 3:917–924, July 2003.

[23] Yoichiro Kawaguchi Takahiro Harada, Seiichi Koshizuka. Smoothed particle hydrodynamics on gpus. 2007.

[24] Markus Gross Matthias Muller, David Charypar. Particle-based fluid simulation for interactive applications. pages 154 – 159, July 2003.

[25] G. Amador and A. Gomes. Cuda-based linear solvers for stable fluids. pages 1–8, April 2010.

[26] S. Sengupta D. Luebke C. Lauterbach, M. Garland and D. Manocha. Fast bvh construction on gpus. *EUROGRAPHICS*, 28(2), 2009.

[27] K.-A. Lie G. Hasle and E. Quak, editors. *Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF*, pages 123–161. Springer Verlag, 2007.

[28] Tarek S. Abdelrahman Tianyi David Han. hicuda: a high-level directive-based language for gpu programming. 383:52–61, 2009.

[29] OpenCurrent GPU-library. http://code.google.com/p/opencurrent/. OpenCurrent is an open source C++ library for solving Partial Differential Equations (PDEs) over regular grids using the CUDA platform from NVIDIA.

[30] CUDPP GPU-library. http://code.google.com/p/cudpp/. CUDA Data Parallel Primitives Library.

[31] MAGMA GPU-library. http://icl.cs.utk.edu/magma/. Matrix Algebra on GPU and Multicore Architectures.

[32] CULAtools GPU-library. http://www.culatools.com/. GPU-accelerated linear algebra library.

[33] GPULib GPU-library. http://www.txcorp.com/products/GPULib/. GPU-Accelerated Computing for Very High-Level Languages.

[34] MATLAB ACCELERATION. http://www.nvidia.com/object/matlab_acceleration.html. Use GPU-libraries such as CUDA FFT and CUBLAS besides writing CUDA functions for key kernels.

[35] CUSP GPU-library. http://code.google.com/p/cusp-library/. Generic Parallel Algorithms for Sparse Matrix and Graph Computations.

[36] THRUST GPU-library. http://code.google.com/p/thrust/. Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL).

[37] Intel Corporation. Excerpts from a conversation with Gordon Moore: Moore's Law, 2005.

[38] NVIDIA Corporation. http://developer.nvidia.com/object/nsight.html.

[39] NVIDIA Corporation. Nvidia's next generation cuda compute architecture, 2010. Whitepaper.

[40] NVIDIA Corporation. http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf. Memory Optimizations.

[41] Message Passing Interface. http://en.wikipedia.org/wiki/Message_Passing_Interface. MPI is an API specification that allows computers to communicate with one another. It is used in computer clusters and supercomputers.

[42] Asplos cuda tutorial. http://gpgpu.org/static/asplos2008/ASPLOS08-1-intro-overview.pdf, 2008.

[43] Asplos cuda tutorial. http://gpgpu.org/static/asplos2008/ASPLOS08-3-CUDA-model-and-language.pdf, 2008.

[44] Asplos cuda tutorial. http://gpgpu.org/static/sc2008/M02-02_CUDA.pdf, 2008.

[45] Audun Torp. Sparse linear algebra on a gpu. NTNU Trondheim, Norway, June 2009. Master's thesis.

[46] Matlab reservoir simulation toolbox is developed by sintef. http://www.sintef.no/Projectweb/MRST/.

[47] Karypis Lab. Serial Graph Partitioning and Fill-reducing Matrix Ordering.

[48] Cuda implementation of blas. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ CUBLAS_Library_3.1.pdf.

[49] Carter Jonathan N. Stephen Karl D. Zimmerman Robert W. Skorstad Arne Manzocchi Tom Howell John A. Matthews, John D. Assessing the effect of geological uncertainty on recovery estimates in shallow-marine reservoirs: the application of reservoir engineering to the saigup project. *Petroleum Geoscience*, 1:35–44, 2008.

[50] A. Skorstad B. Fjellvoll K. D. Stephen J. A. Howell J. D. Matthews J. J. Walsh M. Nepveu C. Bos J. Cole P. Egberts S. Flint C. Hern L. Holden H. Hovland H. Jackson O. KolbjÃ¸rnsen A. MacDonald P. A. R. Nell K. Onyeagoro J. Strand A. R. Syversveen A. Tchistiakov C. Yang G. Yielding T. Manzocchi, J. N. Carter and R. W. Zimmerman. Sensitivity of the impact of geological uncertainty on production from faulted and unfaulted shallow-marine oil reservoirs: objectives and methods. *Petroleum Geoscience*, 1:3–15, February 2008.