NTNU
Norwegian University of
Science and Technology

Sparse linear algebra on a GPU

with Applications to flow in porous Media

Audun Torp

Master of Science in Physics and Mathematics
Submission date: June 2009
Supervisor: Trond Kvamsdal, MATH
Co-supervisor: Knut-Andreas Lie, SINTEF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

This master thesis evaluates how Graphical Processing Units (GPUs) may be utilized to increase the speed when performing numerical simulations. A physical problem is solved in parallel on a normal desktop computer with Nvidia graphics processors.

The physical phenomenon simulated in this report is flow of a single fluid in a porous media. We discuss simple mathematical models describing fluid reservoirs as well as some more complex ones. The simple model is aimed towards Cartesian grids while the more elaborate models are designed for general grids.

In this assignment we will perform the following:

1. Analyze mathematical models describing the reservoir.
   (a) Explain the two point flux approximation method (TPFA) used in the specialization project.
   (b) Discretize flow in porous media with the mimetic method, mixed element method, or the O-method.
2. Create a conjugate gradient (CG) solver for a linear system of equations in the CUDA programming language.
   (a) Improve the specialization project to support double precision.
   (b) Analyze numerical errors of the results.
3. Look at different implementations of sparse matrices in CUDA.
   (a) CUDA data parallel primitives (CUDPP)
   (b) How CUBLAS can exploit matrix structure.
      i. Positive definite matrices.
      ii. Symmetric matrices.
      iii. Band matrices.
4. If time allows it, look into:
   (a) The connection between Matlab and CUDA.
   (b) How to simulate the behavior of a reservoir over time.

Assignment given: 27. January 2009
Supervisor: Trond Kvamsdal, MATH

Abstract

We investigate what the graphics processing units (GPUs) have to offer compared to the central processing units (CPUs) when solving a sparse linear system of equations. This is performed by using a GPU to simulate fluid-flow in a porous medium. Flow-problems are discretized mainly by the mimetic finite element discretization, but also by a two-point flux-approximation (TPFA) method. Both of these discretization schemes are explained in detail. Example-models of flow in porous media are simulated, as well as CO_2-injection into a realistic model of a sub-sea storage-cite.

The linear algebra is solved by the conjugate gradient (CG) method without a preconditioner. The computationally most expensive calculation of this algorithm is the matrix-vector product. Several formats for storing sparse matrices are presented and implemented on both a CPU and a GPU. The fastest format on the CPU is different from the format performing best on the GPU. Implementations for the GPU is written for the compute unified driver architecture (CUDA), and C++ is used for the CPU-implementations. The program is created as a plug-in for Matlab and may be used to solve any symmetric positive definite (SPD) linear system.

How a GPU differs from a CPU is explained, where focus is put on how a program should be written to fully utilize the potential of a GPU. The optimized implementation on the GPU outperforms the CPU, and offers a substantial improvement compared to Matlab's conjugate gradient method, when no preconditioner is used.

Preface

This thesis is my submission for a masters degree in the field of numerics (TMA4910) at the Norwegian University of Science and engineering (NTNU). The topic is proposed by SINTEF, as a continuation of the specialization project TMA4500. My assignment is to use a graphics card to solve linear algebra while broaden my knowledge of numerical methods.

This topic fascinates me, and inspired me to change direction of specialization. I was studying cryptography and number theory in Japan last year, but have now switched to numerics. Getting the numerical knowledge I wanted demanded a lot of work and delayed my start, but now it feels like it was worth the effort.

My supervisor Trond Kvamsdal has provided me with valuable input throughout this thesis. Being able to meet on a regular basis have helped me through many of the small problems that appeared along the way. His good mood, in spite of being busy most of the time, has made our meetings pleasant and constructive, and I would like to thank him for his support.

The individuality that is required to write a thesis like this one, can make the daily life very lonely. Having mathematical consultancy and contact with my co-students have been vital to me. I would especially thank my girlfriend Ingrid G. Dragset for being positive to my suggestion of meeting once a week to exchange our progress in a mathematical and structured fashion. However, our exchange of loving words and support outside the mathematical universe has of-course been more important.

I would also like to thank Knut-Andreas Lie. He has taught me how a report should be structured, and helped me set up the Matlab reservoir simulation toolbox (MRST). This toolbox has been a useful bridge between the linear algebra solver and the mathematics of discretizing flow in porous media, so I am happy that such a software is available for free.

Stein Krogstad has a profound knowledge of the discretization techniques used in this thesis. I am grateful for the input he has given to deepen my mathematical understanding. I have at several occasions been able to visit him, Knut-Andreas Lie, and the other researchers at SINTEF in Oslo, and thank NTNU for covering my travelling expenses as well as providing me with an awesome graphics card for my home computer.

Audun Torp
Trondheim, June 2009.

Contents

1 Introduction **1**
- 1.1 Knowing the hardware . . . 1
- 1.2 Characterization of the problem . . . 3
- 1.3 Outline of the thesis . . . 3

2 Hardware **5**
- 2.1 What separates a GPU from a CPU? . . . 5
  - 2.1.1 Movement towards similar properties . . . 6
  - 2.1.2 Why use the GPU? . . . 7
- 2.2 The similarities of graphics and numerics . . . 8
- 2.3 A brief history of GPU programming . . . 8
- 2.4 Programming in CUDA . . . 9
- 2.5 Hardware structure . . . 10
  - 2.5.1 Structuring threads for a kernel-launch . . . 11
  - 2.5.2 Latency hiding . . . 11
  - 2.5.3 Coalescing . . . 12
  - 2.5.4 Constant cache and texture cache . . . 12
  - 2.5.5 Output from profiling . . . 13
- 2.6 Compiling a CUDA program . . . 13

3 Physical background **15**
- 3.1 Flow in porous media . . . 15
- 3.2 Conservation of mass . . . 16
- 3.3 Darcy's law . . . 17
- 3.4 Assembling the elliptic PDE . . . 18

4 Numerical methods **19**
- 4.1 Two point flux approximation scheme . . . 19

4.2 The O-method . . . 21
4.3 Mixed formulation . . . 23
4.4 Hybrid formulation . . . 24
4.4.1 Building the linear system . . . 26
4.4.2 Schur-complement reduction . . . 27
4.5 Mimetic finite element method . . . 27

5 Linear algebra solvers **33**
5.1 Choosing an appropriate solver . . . 33
5.2 Overview of the CG method . . . 34
5.3 Iterative smoothers and multi-grid . . . 36

6 Implementations **38**
6.1 Finding the kernels of the program . . . 38
6.2 The CG algorithm . . . 39
6.2.1 Additions to the CG code . . . 40
6.2.2 Solving in mixed precision . . . 41
6.3 Sparse representations . . . 43
6.3.1 Sparse matrix libraries for CUDA . . . 43
6.3.2 General sparse formats . . . 45
6.3.3 Problem-specific sparse formats . . . 46
6.3.4 Hybrid representation . . . 47
6.4 Implementation of the matrix-formats . . . 48
6.5 The Matlab reservoir simulation toolbox . . . 49
6.6 Data transfer between Matlab and C++ . . . 50
6.7 Compiling the program . . . 51
6.8 Implementing a kernel for CUDA . . . 52
6.9 Using the CUDA profiler . . . 54

7 Numerical Experiments **58**
7.1 Two-dimensional verification model . . . 59
7.2 Models from the Matlab Reservoir Simulation Toolbox . . . 60
7.2.1 The narrow passage example . . . 60
7.2.2 The fault-crossing example . . . 61
7.3 The Johansen data set . . . 62
7.4 Analyzing the matrix-structures . . . 63
7.4.1 TPFA discretization . . . 63
7.4.2 Mimetic discretization . . . 64
7.5 Assembling the linear systems . . . 66
7.6 Speed of the sparse formats . . . 67

7.7 Solving the Johansen formation 71
7.8 Using the mixed precision solver 75
7.9 Summary of the results 76

8 Conclusions **78**
8.1 Conclusions in more detail 79
8.2 Further work 80

Chapter 1

Introduction

For many decades, the performance of processors has been dictated by Moore's law [20], which states that the number of transistors per chip doubles roughly every second year, and the performance is doubled around every 18th month. Until recently, we have seen this performance growth in the form of higher clock rates, and larger cache-sizes. However, the generation of heat and increasing power consumption have stopped the clock frequency of processors from rising much above 4 GHz. Improvements are now realized as an increasing number of processor-cores on the central processing units (CPUs).

Computer games and gaming consoles has rapidly market. The sales of gaming hardware have doubled year after year. Nearly 41 million game consoles were sold in 2008 [44], and the demand for high-end computer graphic cards is following the same trend. The race to improve realism in computer games has pushed the Graphics Processing Units (GPUs) up to performances of Terra-flops (trillion floating point operations per second). This performance is achieved by focusing on the ability to perform pure calculations of parallel nature. GPUs outperform CPUs on such tasks (see Figure 1.1), but CPUs are also turning more parallel. Certainly, programmers have to write parallel code in order to make benefit of current and coming hardware.

1.1 Knowing the hardware

The switch from sequential to parallel computing offers new challenges to a programmer. Not all tasks are easy to solve in parallel, and a parallel implementation has often a different approach than its sequential counterpart. Some problems even have a sequential nature, making them unsuitable for parallel implementation. It is therefore becoming more important for a pro-

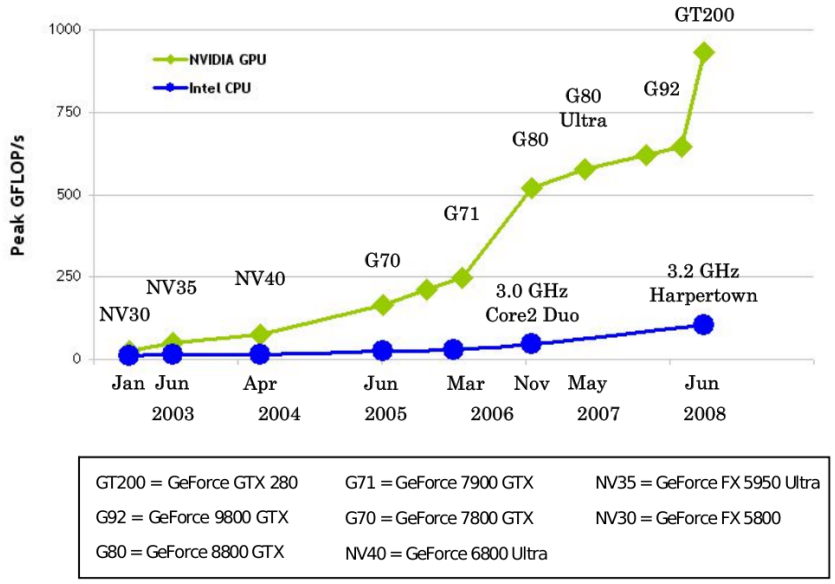


Figure 1.1: Performance of recent GPUs compared to CPUs in floating point operations per second (flops), taken from the CUDA programming guide [29].

grammer to classify his problem, and be aware of the costly computations, or the *kernels*, of his program.

A kernel is a (computational expensive) part of a program, which significantly contributes to its runtime. Kernels define the limits and possibilities when it comes to optimizing a program for speed, since an improvement in the runtime of a kernel process gives a noticeable change in runtime for the whole program.

At the same time, a programmer needs to be aware of different available hardware and their characteristics. Available hardware can range from personal computers, gaming consoles, and supercomputers, down to micro controllers and cellular phones. The choices of hardware vary in price, accessibility and hardware layout.

Where processors are located on the hardware and how the memory is structured around them plays an important role in deciding how well the hardware performs on a certain task. Knowledge of the design of the hardware at hand is also vital when writing parallel code for it. At present, different designs of parallel processors require different programming, and even different languages, but a universal language OpenCL is released, and compilers for different hardware is under development. Even though compilers take over some of the optimization in the future, knowing the hardware layout is crucial when implementing for top performance.

A group of researcher met at Berkeley in 2006 to discuss the upcoming challenges related to parallel computing. They categorized different computational tasks into what they called *dwarfs* [3]. Such a dwarf is a classification of problems relying on the same type of calculations, and should run well on the same type of hardware. Problems belonging to the same dwarf may also be solved using the same type of kernel-implementation. Usually, however, problem-specific kernels achieve better performance.

1.2 Characterization of the problem

In this thesis, the aim is to simulate fluid-flow in a porous medium when a grid-structure and the flow-properties of the medium (the permeability) are given. Fluid flow in porous media can be reduced to an elliptic partial differential equation (PDE). We search for an approximate solutions to this equation, satisfying a set of criteria on each cell in the grid. Then, the PDE can be solved by a linear system of equation.

How the linear system is built from the PDE is called the discretization. Such discretizations vary in complexity, but is generally calculated quickly on a computer. The computational expensive operation is to solve the linear system, and this is the kernel of our problem. This task is well described by the second dwarf called sparse linear algebra, which the research group presented in [3],

The challenge of this thesis, is to bridge the gap between the kernel of solving sparse linear systems with the hardware of Nvidia's programmable graphics cards. (see illustration in Figure 1.2). This is achieved by writing appropriate algorithms that are solving the kernels, while making the most out of the hardware. Here, we need to match the problem of flow in porous media with the hardware that is available for this thesis, which is Intel's Core 2 Quad 6600 with a GeForce 260 graphics card.

1.3 Outline of the thesis

This thesis is organized as follows. First, we look at the characterizations of a graphics card in Chapter 2. We explain the major differences between a GPU and a CPU, and outline the important hardware properties that a programmer needs to be aware of when programming for speed on a GPU.

Chapter 3 presents the physical problem, and we derive the elliptic equation describing flow through a porous medium. Techniques for discretizing the elliptic equation are explained in Chapter 4. We put extra em-

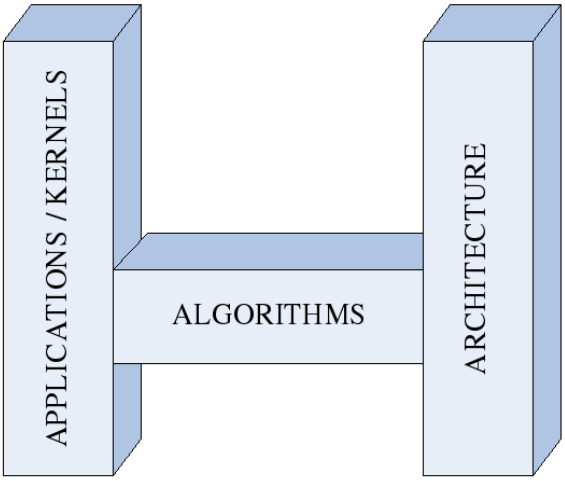


Figure 1.2: The challenge of parallel computing is illustrated by a bridge. We have to bridge the gap between the kernel process at hand and its suitable hardware. In this thesis, the kernel is to solve sparse linear algebra by the conjugate gradient method and the hardware is a programmable graphics card supporting CUDA. The picture is inspired by a group of researchers who met at Berkeley [3].

phasis on the mimetic finite element discretization and the two-point flux-approximation (TPFA) method. A TPFA discretization results in a relatively small system of linear equations, and is therefore solved quickly. The advantage of the mimetic formulation is that it is able to discretize models with complex geometries in a straight-forward manner.

The conjugate gradient (CG) method is a method to solve the system of linear equation. We explain the method in Chapter 5, and implement it for the GPU in Chapter 6. The linear system is represented by using a sparse matrix-format. Several sparse matrix-formats exist, and the format which is best suited for the CPU is not necessary the best format to implement for the GPU.

In Chapter 7, we analyze the speed of the different sparse formats. We create linear systems by mimetic discretizations of example models, as well as a simplified model of an off-shore cite for CO_2 deposit. These systems are solved by the CG algorithm by using the sparse matrix-formats on both the CPU and the GPU. We draw the conclusions of the result in Chapter 8.

Chapter 2

Hardware

In order to fully utilize the computational potential of GPUs, one have know the hardware layout. In this thesis, we look at graphics cards with support for the compute unified driver architecture (CUDA). Memory communication is usually the bottleneck of the applications running on such devices. We therefore present the most important methods to maximize the memory throughput in Section 2.5, but before that we explain the characterizations of a GPU.

2.1 What separates a GPU from a CPU?

CPUs in modern computers are optimized for high performance on multiple programs running sequential code. The main purpose of a CPU is to be compatible with numerous instructions and perform well on general code. The exponential growth in performance from both the CPU and the GPU is not matched by the same increase in speed of the internal memory (RAM) on the motherboard [23]. Because of this, a CPU has to employ methods for handling memory in an efficient way. Many transistors on the CPU are therefore devoted to advanced communication and administration techniques like *out-of-order execution* and *branch prediction* [17].

Branch prediction is a method used to guess the memory address of following instructions. With good guessing, a CPU can request the instructions from memory long before they are executed. This decreases the time the CPU wastes on waiting for the memory to arrive.

Out-of-order execution is a queuing system for instructions on the CPU. Supported instructions have their respective resources with a physical location. If a processor is supporting out-of-order execution, instructions can line

up for their resource and run as soon as it becomes available. The results are coordinated and synchronized automatically afterwards. Techniques of handling memory together with the support of task switching makes a CPU perform well when running operation systems.

In contrast to a CPU, a GPU focuses mainly on floating-point multiplication and addition, together with the ability to execute many threads in parallel. Memory is located physically closer to the processors and memory transfer has a large bandwidth within the graphics card (111.9 GB/s on a GeForce 260). Bandwidth is the maximum amount of data that can be transferred per second. It should not be confused with memory latency, which is the time spent waiting from data is requested until it is received. A graphics card has a particular structure of its memory which we discuss in Section 2.5. However, it does not have the a system choosing how the memory should be used for optimal performance [29]. Increased performance can therefore be attained by programming specifically where and when memory should be stored on the graphics card.

GPUs outperform CPUs on some tasks, but the arithmetic power of a GPU results from a highly specialized architecture. Many applications exists for which GPUs are not well suited, and probably never will be. A classical example of such a program is a word processor, which is a pointer chasing application, dominated by memory communication. Such a program is hard to parallelize and performs better with access to advanced memory instructions [31].

Other less suited applications for the GPU are cryptographic protocols and programs working with integers. Current Nvidia GPUs do not support logical operators and modulus calculations. Compilers have to translate these instructions to a combination of supported operations that give the same result [29].

2.1.1 Movement towards similar properties

Double precision (64-bit) floating points have until recently also been replaced at compile time, but graphics cards are starting to support additional instructions. The newest generation of graphics cards from Nvidia; the GeForce 200-series, has full support for double precision [29], and GPUs are showing a trend of becoming more general.

CPUs, on the other hand, have a tendency of becoming more parallel. MMX [32] is a CPU implementation of an operation called *single instruction multiple data* (SIMD), which is a basic parallel instruction. A SIMD operation is a technique for sending the same instruction to a multiple of processing threads. The instruction is executed in parallel, increasing the

amount of data that is processed per second.

Vector addition is a typical example of an instruction which is highly parallel and runs well on SIMD machines. A standard implementation of vector addition on a SIMD machine gives each thread the instruction to sum its own index of the two input vectors. In this way multiple threads work simultaneously, and several indexes get summed in one clock cycle. MMX was released with integer support in 1997 by Intel. AMD answered by releasing 3DNow! in 1998, which is a SIMD instruction set supporting floating points [21]. These have been followed by several SIMD implementations, giving wider support for parallel implementations on CPUs.

A GPU code-named Larrabee is currently under development by Intel [36]. This GPU is basically a multi-core CPU with tens of cores, supporting the same instruction set as a normal CPU. This decreases the distinction between CPUs and GPUs even more.

2.1.2 Why use the GPU?

Modern CPUs are operating at around 3 GHz, while most GPUs use clock frequencies close to 600 MHz. As processors are pushing the limits of clock frequencies the power consumption increase substantially [22]. Even though GPUs are currently surpassing CPUs as the most power-consuming component in a personal computer [39], GPUs can offer a higher performance per watt. The cost of supplying power to a supercomputer is getting close to the cost of the supercomputer itself. Performance per watt is therefore becoming a more important benchmark of supercomputers. There is an official ranking of the supercomputers delivering the most performance per Watt [10]. It is created to encourage energy-efficient super-computing.

Even though both the CPU and the GPU enjoy the same technology of semiconductors, the GPU is able to focus its technology on pure calculations. In this way, GPUs achieve a yearly growth in terms of flops that have significantly outnumbered Moore's law [31]. Comparing Nvidia's top model graphics cards with each other, reflects this. A new generation is released roughly every 12 to 15 month and consists of a series of graphics cards based on the same processor. The arithmetic throughput was almost tripled between both the 6- and 7-series, and the 7- and 8-series, while the improvement between the 8- and 9-series was not as high. The 9-series achieved almost doubling, rising from 387 Gflops for the GeForce 8800 Ultra to 768 Gflops on the GeForce 9800 gx2 card [43]. The CPU is also staying a small step ahead of Moore's law at the moment, but as we can see in Figure 1.1, it is lagging behind the GPU at calculation-expensive tasks.

2.2 The similarities of graphics and numerics

Creating graphical realism in games revolves around a set of standard operations defined mainly by the interfaces of OpenGL (developed by Silicon Graphics SGI [35]) and Direct3D (by Microsoft [5]). A technique of making graphics in these frameworks is to create 3D-models consisting of triangular surfaces. These triangles are scaled, rotated, and translated into what is displayed on the screen. Manipulations of a triangle are conducted as a matrix-vector product and have analogues to methods for solving linear systems. Using GPUs for solving linear algebra has therefore been an active field of research since the appearance of the first graphics cards [15].

Creating effects of light and shadow on a 3D surface is called *shading*. These effects can be calculated independently of other surfaces in the same scene, and the manipulations on a triangle are also well suited for parallel implementation. Ray-tracing is the second most common way of creating 3D-graphics, and consists also of many independent calculations [40]. Rendering 3D images is therefore regarded as a highly parallel operation. Thus, in order to utilize this, graphics cards have evolved into special processors capable of executing a large number of instructions in parallel.

Lately, these processors have become easier to program for more general calculations other than just graphics. Competition in the market is pushing the prices down, and many researchers have caught interest in using this inexpensive and efficient processing power in new areas of computation.

2.3 A brief history of GPU programming

Initially, the GPU was fixed-function pieces of silicon, created to fulfill a standard instruction interface defined mainly by OpenGL [35] and Direct-3D [5]. Numerical calculations could be executed through the instructions of these interfaces, but the process was often tedious. Numbers had to be represented as textures, and instructions were limited to the known graphical instructions. Nevertheless, general purpose programming for the GPU (GPGPU) has been an active field of research for for many years.

The first programmable GPU was introduced in 2001, accepting programs written in an assembly-like language. However, higher level languages followed. Nvidia collaborated with Microsoft to create Cg [25] (C for graphics) in 2003. Microsoft called it High Level Shading Language (HLSL), and specialized it for their Direct3D interface [11]. The language is the same, giving control of the GPU through C language.

Nvidia's competitor ATI, worked on an abstraction layer for their GPUs

under the name Close-To-Metal [18] (CTM). It was supported by Brook+ [19], which is an open source compiler modified by ATI to support their GPUs. CTM did not get an official release. Instead, another language called Stream SDK [2] was released in 2007, with support for Brook+. Stream SDK is still the best tool for high-level and low-level access to ATI's, and AMD's graphics hardware.

When Nvidia released their GeForce 8-series in 2007, they included a more general and simple language for programming their GPUs. It was called compute unified device architecture (CUDA), and has the syntax of C with a few extensions to handle parallel code [29]. CUDA enables full control of the GPU and comes with a complete software development kit. An important feature that is available in CUDA, but not in previous frameworks, is high-speed synchronization and data sharing between processor threads on the GPU. Although CUDA only supports Nvidia products, it aims to be a generalized framework for many-core systems.

Another framework for multi-core processors that was released recently is the open computing language (OpenCL) [30]. At present, code written for a supercomputer, a GPU, or a parallel CPU has to be written in different languages. OpenCL was defined to unify the languages of parallel computing into one platform. The structure of OpenCL is quite similar to CUDA, aiming to support various types of hardware. All the largest producers of processors (AMD, Intel, and Nvidia) are collaborating on the OpenCL project [26], and most parallel hardware, including GPUs and CPUs, will probably support OpenCL in the future. However, at present there exists only one OpenCL compiler [30]. It is built upon CUDA and supports only hardware from Nvidia.

2.4 Programming in CUDA

CUDA is a good choice of language for parallel programming on a GPU, especially for those who are new to GPGPU. The difference between CUDA code and normal C code is easy to grasp if the programmer is familiar with parallel programming. CUDA has also an emulation mode simulating the behavior of a GPU on a CPU. This enables debugging, making it a lot easier to check the code for errors. The programmer can therefore focus on creating parallel code which is optimized for the hardware.

Graphics cards supporting CUDA has the ability to report back how well the hardware structure was used on a given kernel-launch. This is called *profiling*, and gives the programmer a hint on how the kernel may be changed to better exploit its structure.

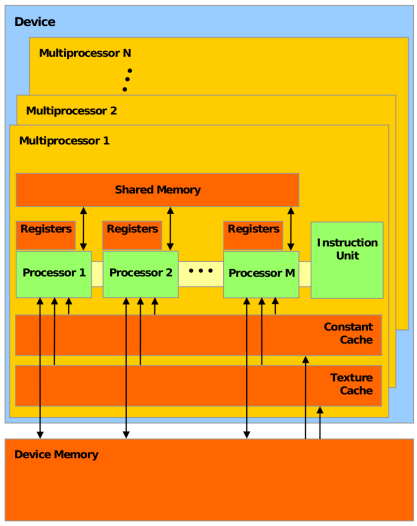


Figure 2.1: Hardware layout of a modern Nvidia graphics card with CUDA support. A multiprocessor consist of several *stream processors*, and in current high-end GPUs each multiprocessor is able to run 1024 concurrent threads [29].

2.5 Hardware structure

A device supporting CUDA consists of one or several multiprocessors as illustrated in Figure 2.1. Each multiprocessor creates, manages, and executes concurrent threads in the hardware without any scheduling overhead. This allows us to do parallelism on a fine scale without loosing efficiency. For example assigning each thread to do only one small operation, like we do for the axpy calculation we present in Chapter 6, does not necessarily lead to an inefficient implementation [29].

The multiprocessors share a common memory located on the graphics card. If a CPU wants to communicate with the card, it has to read from or write to this *device memory*. The device can structure its memory into a read-only part and a global memory for both read and write.

In addition to this, each multiprocessor has its own *shared memory*, which is situated closer to its processors as illustrated in Figure 2.1. Accessing this shared memory is as fast as accessing the processor's registers as long as there are no bank conflicts as we describe in Section 2.5.5. Operating on the shared memory has then approximately hundred times lower latency than accessing the global device memory [29]. It is therefore often clever to load all the data into the shared memory, before running calculations on it [14].

2.5.1 Structuring threads for a kernel-launch

Code running on the GPU needs to specify how many processing threads it will execute simultaneously. The threads have to be structured in *blocks*. All threads within a block run at the same time on the same multiprocessor [29]. The threads are typically executing the same instruction on different data or with different effect. Threads within a block are therefore structured along one, two, or three dimensions, and get their own x, y, and z-coordinate within the block. We use this coordinate to specify the correct data for the thread, and to distribute the workload.

When an instruction for a block is sent to a graphics card, it is handled by one of the multiprocessors. The multiprocessor distributes the work using an architecture called single instruction multiple threads (SIMT). It is doing basically the same as SIMD (see Section 2.1.1), by sending the same instructions to each thread in the block. The difference from SIMD is that threads in a SIMT machine do not need to execute the same instructions at exactly the same time.

Instead, processor-threads are divided into packs of 32 threads each, called *warps* [24], and the workload is distributed among these warps. Every warp in a block starts at the same place in the code, but is free to progress independently of the other warps. We have the possibility to make them wait for each other on a code-line by using the command **__syncthreads()**. Inside a warp however, every instruction is sent to all the threads simultaneously. So if the threads diverge, they will have to wait for each other to finish. As a worst case, this leads to sequential execution within a warp.

Each multiprocessor on a graphics card supporting CUDA has the ability to execute at least 768 concurrent threads in parallel. Multiprocessors on the GeForce 200 series and newer cards can execute 1024 threads [27]. The maximum number of threads per block is 512. It is therefore necessary to execute multiple blocks on each multiprocessor for efficiency. We have the ability to organize the blocks in a grid, so threads has also a three-dimensional grid-coordinate.

The multiprocessor optimizes the distribution of workload automatically, by employing warps as soon as they become idle [29]. Warps become idle either by finishing their instruction queue or while waiting for memory transfers.

2.5.2 Latency hiding

It is often an advantage to load all the data at the same time. We then let the multiprocessor ask for more data while waiting for the requested memory

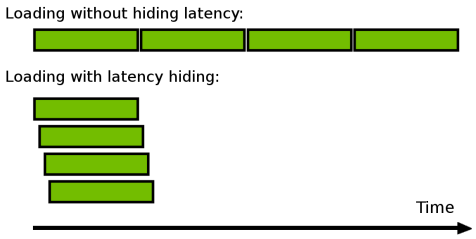


Figure 2.2: Illustration of how memory latency is hidden, when loading more data at once. After requesting data from device memory, threads execute instructions which are independent of the incoming data, while waiting to receive it. These instructions may for example be requests for more data.

to arrive as we illustrate in Figure 2.2. The GPU may perform any operation while waiting to receive data, as long as the operations do not involve the memory it is waiting for [29]. Utilizing the processor while waiting for data to arrive is called *latency hiding*, since it decreases the negative effect of memory latency.

2.5.3 Coalescing

The global memory space is not cached, so it is important to follow the right access pattern to get maximum memory bandwidth, especially given how costly accesses to device memory are. Memory bandwidth is most efficiently used if the simultaneous memory access of half a warp (16 threads) can be *coalesced* into a single transaction of 32, 64, or 128bytes [14].

Recent graphics cards have different requirements for when the memory can be coalesced than older models. In our test we use a card of the GeForce 200-series (with compute capability 1.3). For a memory transaction to be coalesced for this card (or later models), the data that is accessed has to lie in one segment of twice the size of the requested memory [29]. For example, if each thread loads a word with the size of a byte, the requested memory for the half-warp has to lie in a segment of 32 bytes size. Similarly we require segments of 64 bytes for two-byte loads, and 128 bytes for load-sizes of four or eight bytes. If the requested memory is spread over n segments of the respective size, n loads are performed.

2.5.4 Constant cache and texture cache

As Figure 2.1 illustrates, we have a constant cache and a texture cache located on every multiprocessor. They are speeding up data-fetching from

two constant-memory areas of the device memory called constant memory and texture memory respectively. Accessing textures is highly optimized, and reading from device memory through texture-fetching presents several benefits [29].

Both texture fetching and reading through constant memory is cached, potentially giving higher bandwidth when there is locality in the memory fetch. The penalties for unstructured reads from global memory do not apply to textures, as we discuss in Section 2.5.3. Also, memory latency is hidden better for texture fetches and this might improve performance even more. In Section 6.9 we analyze how the texture cache may be utilized for our kernels.

2.5.5 Output from profiling

How efficiently the multiprocessors are used on a kernel launch is reported when profiling is activated. This is given as a occupancy factor telling the fraction of the total available threads that is actually used. From the profile, we may also read out the number of divergent branches we get when threads diverge. These numbers are useful to consider when we look at which part of a program we can change to avoid wasting time on idle threads.

Another important counter from CUDA profiling is the number of warps which are serialized due to bank conflict. A bank conflict happens when threads from the same block tries to access a shared memory in a wrong way. There are two ways that the threads may use the shared memory correctly. The access is performed for 16 threads (a half-warp) at the time, and they may either all read the same value or access one distinct value each. If this is not the case, the 16 operations are broken down into combinations of legal access patterns. The access is therefore performed as several accesses instead, and we have a bank conflict. Bank conflict may also happen if different warps tries to access the same shared data at the same time.

2.6 Compiling a CUDA program

Code written for CUDA consists of three major parts: the GPU kernels, CUDA specific code, and normal C or C++ code. The kernels are the functions that run on the GPU. Code specific for CUDA are the instructions for setting up block sizes, grid sizes, and calling the kernel functions. Both of these types of code have to be compiled with Nvidia's nvcc-compiler and can be linked with the rest of the code by any C compiler. Both CUDA specific code and normal C code are accepted by nvcc along with the majority of C++ syntax including objects and templates. An extra compiler is therefore

not mandatory unless unsupported C++ code needs to be included.

Exception handling is one of the few features that are not supported by the nvcc compiler. Several of the standard C++ libraries have built-in exceptions, and can only be included in source code which is compiled with a normal C++ compiler. The library iostream is an example of a commonly used library implementing exceptions. It is therefore customary on large programs, to separate the CUDA specific code from the rest of C++ code in order to utilize such libraries.

Chapter 3

Physical background

To utilize the computational power of a GPU on a physical problem, we look at how fluid flows in a porous medium. The porous medium can be any substance, where there are pores formed in such a way that liquid or gas is able to propagate through them. Finding how fast the fluids move through the medium, and in which directions it propagates has several applications.

3.1 Flow in porous media

We briefly present some of the areas where simulation of flow in a porous media is used.

Applications in medicine

Knowing how nutrients and blood flows through the body is useful. For example, calculating the effect the blood-flow gets from blood-clot and other abnormalities may increase the analysis and understanding of different diseases. Looking at how medicine in water propagates through the tissue of the body, may also be helpful for example to be able to more accurately choose the dozes and to better simulate their effect.

Water-flow in tissue and in bones and blood-flow in capillary veins are the flow-problems of the body which are best suited for the model we present here. This is because the pores are relatively small and evenly distributed. We can therefore look at their average behavior.

Analyzing oil-production

Simulating how oil behaves in rock formations is important to maximize the outcome of oil production from reservoirs lying underground. To drill a well is expensive, and the profit of extra production is high. A large amount of research have therefore been conducted on numerical simulation of oil-flow in reservoirs.

A reservoir consists of large formations of rock, formed over millions of years, containing layers with organic material. These formations have been twisted and shaped by the cracking of continental plates and volcanic activities, forming a complex porous rock structure in which oil is trapped The pressures are usually high and fluids are able to flow within these sub-sea reservoirs. This causes hydrocarbons to exist as both fluid and gas, and makes parts of the oil shoot up by itself if a well is drilled. Such an automated production is called the primary production. Getting more out of the oil-reservoir after this primary production, is where the simulation of the reservoir becomes handy. We can then analyze the effect of different approaches used to squeeze out more oil. Methods may for example be to drill new wells, or to inject water or gas into the reservoir.

Determining ground-water flow

Finding how ground-water propagates in soil, earth, and rock can predict how a possible release of hazardous waste in the ground-water will spread to areas nearby. It gives indications on where water-polluting industry is best located.

Predicting the behavior of CO_2 storage

Injecting CO_2 into rock formations is being used as a way to avoid polluting the air when fossil fuel is used in for example power-plants. We simulate the behavior of such a deposit in Section 7.3, were we look at how the CO_2 propagates in a sub-sea site called the Johansen formation.

3.2 Conservation of mass

A porous medium is defined as a material volume consisting of a solid with a interconnected void filled with a fluid or a gas. Porosity ϕ is given as the ratio of void space.

Mass cannot appear or disappear. So if we look at a fixed volume Ω of the porous medium, we know that a change in the fluid mass within this volume

is a result of a flux through the boundary $\partial\Omega$ of the volume, or a variation in fluid density or medium porosity. This gives the relation

$$\frac{\partial}{\partial t}\int_{\Omega}(\phi\rho)\,dV + \int_{\partial\Omega}(\rho\vec{v})\cdot\hat{n}\,dA = \int_{\Omega} q\,dV, \tag{3.1}$$

where ρ is the mass density of the fluid, $\hat{n}$ is the outward-pointing normal vector of the control-volume, and q is the mass injected or ejected through sources or sinks. In order to use information about the pressure in a porous medium to solve this equation, we need a measure for the average flux $\vec{v}$. We therefore look at Darcy's law, which relates the average flux with the pressure gradient.

3.3 Darcy's law

Through empirical results, the average flow $\vec{v}$ through a cross-section of a porous medium is found to be linearly proportional to the applied pressure difference. This linear relation is called Darcy's law and is valid when the fluid flows slowly without creating turbulence. In three dimensions it is expressed as

$$\vec{v} = \frac{-\mathbf{K}(\vec{\nabla}p + \rho g\vec{\nabla}z)}{\mu}. \tag{3.2}$$

Here, μ is the fluid's viscosity, g is the gravitational constant, and z is the upward vertical direction. Both ρ and g are constants. Thus, the expression within the parenthesis $\vec{\nabla}u := (\vec{\nabla}p + \rho g\vec{\nabla}z)$ gives the driving forces for flow, and points in the flow-direction when there are no obstacles. Obstacles and special geometries in the medium is described by the *permeability* $\mathbf{K}$ which is a 3×3 tensor field. It has dimension $(\text{length})^2$ and depends only on the geometry of the porous medium. The porosity describes partially the properties of the medium, but the geometry of the porous medium is important to describe how fluid will propagate through it, these geometric factors are included in the permeability $\mathbf{K}$.

Having good estimates for $\mathbf{K}$ gives a better characterization of the flow-problem. On small scales, the permeability is a diagonal tensor field, and is therefore represented as a diagonal matrix in most models. Commercial reservoir simulators can seldom run simulations directly on the models when they become large, and up-scaling has to be performed. The permeability is then designed to represent the effects of small-scale structures on a coarser grid. This effective permeability tensor field tries to model the principal flow directions, which may not be aligned with the Cartesian coordinate

directions. We assume therefore only that the permeability $\mathbf{K}$ is symmetric positive semi-definite. This means that if $\hat{d}$ is a directional unit vector, then $\hat{d} \cdot \mathbf{K}_i \hat{d}$ gives a positive number describing how easily a fluid flows in $\hat{d}$-direction. The value zero means no flow at all. Parts of the model where fluid is unable to flow, can therefore be represented by a cell with zero permeability in all directions. Although finding the permeability is an important step when simulating flow in porous media, we assume in this thesis that the geometric structure and permeability is given.

3.4 Assembling the elliptic PDE

The first integral of equation (3.1) vanishes if we assume that the density and porosity are constant in time. This is the same as assuming incompressible fluid and that rock porosity ϕ does not change with the pressure. Then, simplifying notation by writing $f = \rho^{-1} q$ results in

$$\int_{\partial\Omega} -\mu^{-1}(\mathbf{K}\vec{\nabla}u) \cdot \hat{n}\, dA = \int_{\Omega} f\, dV, \tag{3.3}$$

which is the origin of most finite volume discretizations. A finite volume discretization is the technique of dividing the porous medium into grid-cells, or control-volumes, and approximate this integral for every cell to get a linear system.

Another way to represent the two equations (3.2) and (3.1) is to write them separately as partial differential equations (PDEs) on the form

$$\begin{aligned} \vec{v} &= -\frac{\mathbf{K}\vec{\nabla}u}{\mu} \\ \vec{\nabla} \cdot \vec{v} &= f. \end{aligned} \tag{3.4}$$

This system is called the mixed formulation or the first order formulation.

To have a complete model, we also need boundary conditions. These can be either Dirichlet conditions if the pressure u is given at the boundary, or Neumann boundary conditions which is when we know the flux $\vec{v} \cdot \hat{n}$ through the boundary. In this thesis, we assume that the models are surrounded by an impermeable matter if we do not specify boundary conditions. This means that there are no flux in or out of the boundaries.

Chapter 4

Numerical methods

We assume that the geometry of the physical models we solve are partitioned into grids with given permeabilities for each cell. Our task is therefore to discretize such a model into a linear system of equations, solve the system, and read back the results for analysis. In this chapter, we describe the first task of reducing the elliptic PDE describing flow in porous media into a set of linear equations.

The methods we present are two-point flux-approximation (TPFA) scheme, the O-method, and a mixed finite-element method. TPFA finds the flux through a cell-wall from the difference in pressure between the centers of two cells sharing the cell-wall. In the O-method, we use several pressure-values at neighboring cell-centers together with the geometry of the cells to get a more accurate estimate for the flux.

Discretization of the mixed formulation is found by using a finite-element approach on equation (3.4). It leads to an indefinite linear system which can not be solved by the conjugate gradient method. Therefore, we introduce a hybrid formulation to get a symmetric positive definite (SPD) system. When we analyze geometrically one of the bilinear forms that makes up the hybrid system we find an inner product that mimics its behavior. Exchanging the bilinear form with this inner product yields the mimetic finite element method, and we explain how to discretize general grids by this numerical scheme.

4.1 Two point flux approximation scheme

The main idea in the first finite volume method we present is to approximate the flux over an interface by the difference of the driving forces for flow u

at the cell centers of the two neighboring cells sharing the interface. If we look at an interface γ_{ij} shared by cell i and cell j we approximate the driving force

$$\vec{\nabla} u|_{\gamma_{ij}} \approx \delta u_{ij} \cdot \hat{d}_{ij} \text{ , where } \delta u_{ij} = \frac{u_j - u_i}{L_{ij}}. \tag{4.1}$$

Here, $\hat{d}_{ij}$ is the direction from the midpoint of cell i to the midpoint of cell j and $L_{ij} = L_i + L_j$ is the distance between them. Finding the change in u in this fashion is called a two point flux approximation (TPFA).

To assure that the flux can be approximated from only two points, we need $\mathbf{K}\hat{n}_{ij}$ to be parallel to $\hat{d}_{ij}$. More precisely, the grid has to consist of parallelepipeds satisfying $\hat{n}_{ik} \cdot \mathbf{K}\hat{n}_{ij} = 0$, where $\hat{n}_{ik}$ and $\hat{n}_{ij}$ are any two non-parallel normal vectors of the cell. We call this a $\mathbf{K}$-*orthogonal* grid, and note that the contribution of $\vec{\nabla} u$ in the first integral of (3.3) comes in this case only from changes in u along the direction $\hat{d}_{ij}$, and we get

$$\int_{\gamma_{ij}} (-\mu^{-1}\mathbf{K}\vec{\nabla} u) \cdot \hat{n}\, dA \approx -\mu^{-1}\delta u_{ij} \int_{\gamma_{ij}} \hat{n}_{ij}^T \mathbf{K}\hat{d}_{ij}\, dA = -\mu^{-1}\delta u_{ij} |\gamma_{ij}| \hat{n}_{ij}^T \mathbf{K}\hat{d}_{ij}. \tag{4.2}$$

A Cartesian grid is $\mathbf{K}$-orthogonal if the permeability of each cell i is diagonal $\mathbf{K}_i = \text{diag}(k_{ix}, k_{iy}, k_{iz})$ in each cell. This is obvious since both the normal vectors $\hat{n}_{ij}$ and $\hat{d}_{ij}$ is equal to the elementary direction-vectors, and $\mathbf{K}_i$ does not change the direction of these when it is diagonal.

The permeability is usually defined as a constant matrix for each cell, and is therefore undefined at the interfaces. So in order to use equation (3.3) we need to approximate $\mathbf{K}$. Since the volumetric flux $\vec{v} = -\mu^{-1}\mathbf{K}_i\vec{\nabla} u$ is assumed to be a constant velocity in each cell, we can find the average velocity by

$$\bar{v} = \frac{\int_i^j \vec{v}\, dt}{\int_i^j dt} = \frac{\int_i^j d\vec{x}}{\int_i^j 1/\vec{v}\, d\vec{x}} = \frac{L_i + L_j}{\frac{L_i}{\vec{v}_i} + \frac{L_j}{\vec{v}_j}} = \frac{-\mu^{-1}L_{ij}}{\frac{L_i}{\mathbf{K}_i\vec{\nabla} u} + \frac{L_j}{\mathbf{K}_j\vec{\nabla} u}} = \left(\frac{-\mu^{-1}L_{ij}}{\frac{L_i}{k_{ir}} + \frac{L_j}{k_{jr}}}\right) \delta u_{ij}\hat{e}_k,$$

where $\hat{e}_k$ is the Cartesian unit vector in the direction from i to j. Hence, we use the distance-weighted harmonic average $\mathbf{K}_{ij}\hat{e}_k \approx L_{ij}\left(\frac{L_i}{k_{ir}} + \frac{L_j}{k_{jr}}\right)^{-1}$ to find the mean behavior of two neighboring cells in $\hat{e}_k$-direction.

The first integrand of (3.3) is in this case a constant and the integral over cell Ω_i is reduced to

$$\int_{\partial\Omega_i} (-\mu^{-1}\mathbf{K}_i\vec{\nabla} u) \cdot \hat{n}\, dA \approx \sum_j t_{ij}(u_i - u_j) \text{ , where } t_{ij} = \mu^{-1}|\gamma_{ij}|(\frac{L_i}{k_{ir}} + \frac{L_j}{k_{jr}})^{-1}. \tag{4.3}$$

From (3.3) combined with (4.3) we see that the TPFA method seeks a cell-wise constant u, such that

$$\sum_j t_{ij}(u_i - u_j) = \int_{\partial\Omega_i} f \, dA \,, \text{ for all cells } i. \tag{4.4}$$

If we have a grid with n cells and Neumann boundary conditions, we get $n-1$ linearly independent equations with n unknowns and one degree of freedom. Letting the first element of u be a reference with value zero adds a necessary constraint to have a unique solution. In this case, we assemble a linear equation $\mathbf{A}\vec{u} = \vec{b}$, where element i of $\vec{b}$ is the right-hand side of equation (4.4). Except for the first diagonal element (which is adjusted for the reference), the matrix $\mathbf{A} = [a_{ij}]$ has elements

$$a_{ik} = \begin{cases} \sum_j t_{ij} & \text{if } k = i, \\ -t_{ik} & \text{if } k \neq i \text{ and cell } k \text{ is adjacent to } i, \\ 0 & \text{otherwise.} \end{cases} \tag{4.5}$$

This system is clearly symmetric, since $t_{ij} = t_{ji}$. The permeability cannot be negative, so all t_{ij} are non-negative, hence $\mathbf{A}$ is positive definite. Adding the positive constant to the first diagonal element $a_{1,1}$ to account for the reference pressure still makes the system symmetric positive definite.

The matrix $\mathbf{A}$ has a band-structure, with only 7 non-zero diagonals (5 in 2D). Optimizations using both the sparsity and the symmetry of $\mathbf{A}$ significantly reduce the storage space needed. Such a compact form of $\mathbf{A}$ improves runtime as well, since fewer elements of the matrix need to be evaluated.

Unfortunately, the structures of porous media are often better described by grids that are not $\mathbf{K}$-orthogonal, and TPFA is not guaranteed to converge to the correct solution on these grids. In spite of this major shortcoming, the TPFA is still the predominant method in industry simulators also on non-$\mathbf{K}$-orthogonal grids. The main reasons for this is its simplicity and speed. Multi-point flux approximation (MPFA) schemes are generalizations of TPFA that are constructed to amend the shortcomings of TPFA. The method we present next is one example of a MPFA scheme.

4.2 The O-method

If we approximate the driving forces of flow $\vec{\nabla}u$ from more than two points, we are able to handle more complex geometries and permeability-tensor-fields. We assume that every grid cell have six quadrilateral cell-walls, and create a control volume between eight neighboring cells like illustrated in Figure 4.1.

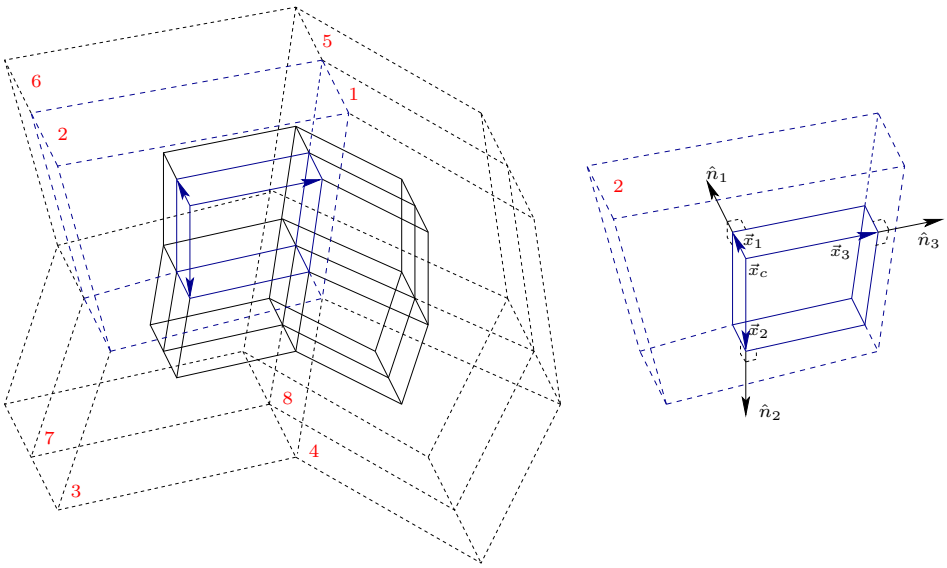


Figure 4.1: Illustration of the control volume used in the O-method for eight cells sharing a point. The dotted line outlines the cells and the solid lines represents eight of the control-volumes. To the right we show the center position $\vec{x}_c$ and the three vectors $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$ of cell number 2. These vectors are used to find an expression for $\vec{\nabla}U$, and we have enough information to build a linear system of equations if we also calculate the average normal vectors $\hat{n}_j$ of each interface j of the cell.

To study (3.3) we find the mid-points of the cell-walls $\vec{x}_1, \cdots, \vec{x}_6$ and the center-point $\vec{x}_c$ of a cell. Then we seek an approximation of (3.3) that involves values at these points. We assume u to be linear in each cell, by the relation $u = u_0 + \vec{\nabla}U \cdot (\vec{x} - \vec{x}_c)$ where $\vec{\nabla}U$ is a constant vector approximating $\vec{\nabla}u$ for the cell. We then get three relations defining $\vec{\nabla}U$, that can be expressed as matrices:

$$\begin{aligned} u_1 - u_c &= \vec{\nabla}U \cdot \vec{x}_1 \\ u_2 - u_c &= \vec{\nabla}U \cdot \vec{x}_2 \\ u_3 - u_c &= \vec{\nabla}U \cdot \vec{x}_3 \end{aligned} \implies \underbrace{\begin{bmatrix} u_1 - u_c \\ u_2 - u_c \\ u_3 - u_c \end{bmatrix}}_{\vec{u}} = \underbrace{\begin{bmatrix} \vec{x}_1^T \\ \vec{x}_2^T \\ \vec{x}_3^T \end{bmatrix}}_{\mathbf{X}} \vec{\nabla}U \tag{4.6}$$

We want to solve this for $\vec{\nabla}U$, and seek an inverse of the matrix $\mathbf{X}$. From the definition of cross-product and dot-product we know that if we cross two vectors and dot the result with a third vector following the right hand rule, we get the volume of the parallelepiped spanned by the three vectors. Together with the fact that the cross-product of two vectors is orthogonal to

both of them, we find

$$\mathbf{X}^{-1} = \underbrace{\frac{1}{(\vec{x}_1 \times \vec{x}_2) \cdot \vec{x}_3}}_{\text{volume of parallelepiped}} \left[\vec{y}_1 \middle| \vec{y}_2 \middle| \vec{y}_3\right], \text{ where } \begin{cases} \vec{y}_1 = (\vec{x}_2 \times \vec{x}_3) \\ \vec{y}_2 = (\vec{x}_3 \times \vec{x}_1) \\ \vec{y}_3 = (\vec{x}_1 \times \vec{x}_2) \end{cases}.$$

As a check, we see that

$$\mathbf{X}\mathbf{X}^{-1} = \frac{1}{(\vec{x}_1 \times \vec{x}_2) \cdot \vec{x}_3} \begin{bmatrix} (\vec{x}_2 \times \vec{x}_3) \cdot \vec{x}_1 & 0 & 0 \\ 0 & (\vec{x}_3 \times \vec{x}_1) \cdot \vec{x}_2 & 0 \\ 0 & 0 & (\vec{x}_1 \times \vec{x}_2) \cdot \vec{x}_3 \end{bmatrix} = \mathbf{I}_3.$$

We now have the expression $\vec{\nabla} U = \mathbf{X}^{-1}\vec{u}$, and inserting this in our approximation of $\vec{\nabla} u$ in (3.3) for each cell Ω_i gives

$$\int_{\Omega_i} (-\mathbf{K}_i \vec{\nabla} U) \cdot \hat{n} \, dA = \sum_{j=1}^{3} -\mathbf{K}_i \vec{\nabla} U \cdot \int_{\gamma_{ij}} \hat{n}_j \, dA = \frac{1}{V} \sum_{k=1}^{3} \sum_{j=1}^{3} |\gamma_{ij}| \vec{y}_k^T \mathbf{K}_i \hat{n}_j (u_c - u_k).$$

Here, $\hat{n}_j$ is the average normal vector $\frac{1}{|\gamma_{ij}|} \int_{\gamma_{ij}} \hat{n} \, dA$ and V is the volume of the parallelepiped spanned by $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$ (see Figure 4.1). Parameterizing the surface γ_{ij} and mapping to a reference element, makes us able to find $\hat{n}_j$. It is described in detail for corner-point grids in [1]. The result is a linear system $\mathbf{A}\vec{u} = \vec{b}$, where each row of $\mathbf{A}$ has six non-zero elements (four in 2D).

Both the O-method and the TPFA discretization are finite volume schemes that approximate the integrals in equation (3.3). In the rest of this chapter, we present methods using finite elements to discretize the mixed formulation in equation (3.4).

4.3 Mixed formulation

Equation (3.4) together with the no-flow boundary condition on a domain Ω gives us the PDEs defining our problem

$$\begin{array}{lrll} \text{(I)} & \vec{v} & = \frac{-\mathbf{K}\vec{\nabla} u}{\mu} & \text{on } \Omega \\ \text{(II)} & \vec{\nabla} \cdot \vec{v} & = f & \text{on } \Omega \\ \text{(III)} & \vec{v} \cdot \hat{n} & = 0 & \text{on } \partial\Omega. \end{array} \tag{4.7}$$

A solution $\vec{v}$ must satisfy the boundary condition and is therefore an element of the space $H_0^{\text{div}}(\Omega) = \{\vec{v} \in L^2(\Omega)^3 \mid \vec{\nabla} \cdot \vec{v} \in L^2(\Omega), \text{ and } \vec{v} \cdot \hat{n} = 0 \text{ on } \partial\Omega\}$.

We find the weak form of equation (I) by multiplying it with a vector test-function $\vec{q}_1$ from $H_0^{\text{div}}(\Omega)$ and integrating over the domain using partial integration. Similarly, equation (II) is multiplied with a scalar test-function q_2. We obtain the mixed formulation for (I) and (II):

$$\begin{array}{rcll} \int_\Omega \vec{q}_1 \cdot \mu \mathbf{K}^{-1} \vec{v}\, dV & - & \int_\Omega u \vec{\nabla} \cdot \vec{q}_1\, dV = 0 & \forall \vec{q}_1 \in H_0^{\text{div}}(\Omega) \\ \int_\Omega q_2 \vec{\nabla} \cdot \vec{v}\, dV & & = \int_\Omega q_2 f\, dV & \forall q_2 \in L^2(\Omega) \end{array} \tag{4.8}$$

We introduce three bilinear forms $(\cdot,\cdot) : L^2(\Omega) \times L^2(\Omega) \mapsto \mathbb{R}$, $b(\cdot,\cdot) : H_0^{\text{div}}(\Omega) \times H_0^{\text{div}}(\Omega) \mapsto \mathbb{R}$, and $c(\cdot,\cdot) : H_0^{\text{div}}(\Omega) \times L^2(\Omega) \mapsto \mathbb{R}$ representing the integrals such that (4.8) can be written as

$$\begin{array}{rcll} b(\vec{q}_1, \vec{v}) & - & c(\vec{q}_1, u) = 0 & \forall \vec{q}_1 \in H_0^{\text{div}}(\Omega) \\ c(\vec{v}, q_2) & & = (f, q_2) & \forall q_2 \in L^2(\Omega). \end{array} \tag{4.9}$$

A solution to the weak form is a pair $(\vec{v}, u) \in H_0^{\text{div}}(\Omega) \times L^2(\Omega)$ solving this equation.

The solution is also a stationary point of the Lagrange functional:

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

This is apparent from variational analysis setting $dL = \frac{\partial L}{\partial \vec{v}} d\vec{v} + \frac{\partial L}{\partial u} du = 0$, and then letting the variation $\vec{q}_1 = d\vec{v}$ and $q_2 = du$ be arbitrary. Looking at the determinant of the Hessian matrix $\Delta L = \frac{\partial^2 L}{\partial \vec{v}^2} \frac{\partial^2 L}{\partial u^2} - \left(\frac{\partial^2 L}{\partial \vec{v} \partial u} \right)^2 = -c(\vec{1}, 1)^2$ we see that the stationary point of L is a saddle-point. This gives a linear system with both positive and negative eigenvalues. Such indefinite systems require special linear solvers and are often considered hard to solve. Next, we present an alternate formulation that gives a positive-definite linear system.

4.4 Hybrid formulation

Positive-definite systems are easier to solve, and we modify equation (4.10) by including continuity of flux as a Lagrange multiplier. This result is a hybrid formulation which gives a positive definite system. The following way of improving (4.10) is sometimes called the Lagrange multiplier technique.

Assume that we have a partitioning $\mathcal{T} = \{T\}$ of the domain Ω into cells. Fluxes are calculated on each interface γ on the boundary ∂T of a cell T, and we introduce the space $H^{\frac{1}{2}}(\mathcal{T})$ consisting of the traces on $\partial \mathcal{T}$ of functions in

$H^1(\mathcal{T})$. Setting $\partial\mathcal{T} = \cup_{T\in\mathcal{T}}\partial T$ gives us the integral representing flux as a bilinear form $d(\cdot,\cdot) : H_0^{div}(\mathcal{T}) \times H^{\frac{1}{2}}(\partial\mathcal{T}) \to \mathbb{R}$ where

$$d(\vec{v},\pi) = \int_{\partial\Omega} \pi\vec{v}\cdot\hat{n}\, dA. \tag{4.11}$$

We include the constraint of flux-continuity, by adding this term to (4.10) and get a new Lagrange functional

$$L_M(\vec{v},u) = \frac{1}{2}b(\vec{v},\vec{v}) - c(\vec{v},u) + (f,u) + d(\vec{v},\pi). \tag{4.12}$$

Finding the stationary point of the Lagrange functional L_M corresponds to finding $(\vec{v},u,\pi) \in H_0^{\text{div}}(\Omega) \times L^2(\Omega) \times H^{\frac{1}{2}}(\partial\mathcal{T}\backslash\partial\Omega)$ which satisfy

$$\begin{array}{lcll} b(\vec{q}_1,\vec{v}) - c(\vec{q}_1,u) + d(\vec{q}_1,\pi) & = & 0 & \forall\vec{q}_1 \in H_0^{\text{div}}(\mathcal{T}) := \cup_{T\in\mathcal{T}} H_0^{\text{div}}(T) \\ c(\vec{v},q_2) & = & (f,q_2) & \forall q_2 \in L^2(\Omega) \\ d(\vec{v},q_3) & = & 0 & \forall q_3 \in H^{\frac{1}{2}}(\partial\mathcal{T}\backslash\partial\Omega). \end{array} \tag{4.13}$$

In other words, we have included the flux in the equations and changed the space for $\vec{q}_1$ to remove the constraint of continuous velocity across cell-interfaces. Instead, we allow velocity-jumps at the cell-boundaries but ensure continuity in the normal-component of the flux with the third equation of (4.13).

The Hessian ΔL_M of L_M is here a 3×3 matrix, and we investigate its eigenvalues to assert that the stationary point is a minima. We write b, c, and d for $b(\vec{1},\vec{1})$, $c(\vec{1},1)$, and $d(\vec{1},1)$ respectively ($\vec{1}$ is the vector with all components set to one). Then the Hessian becomes

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

which implies that λ is zero or

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

Since $b(\cdot,\cdot)$ is positive definite and $c = 0$, we have that the eigenvalues λ are bigger than or equal to zero. Hence, the system is positive semi-definite.

The reason why we have an eigenvalue equal to zero is the pure Neumann boundary. A solution is only unique after we add an extra constraint like we did for TPFA by having a reference value for u. Such a constraint makes the system symmetric positive definite.

4.4.1 Building the linear system

A straight-forward discretization of the hybrid formulation presented above is to introduce finite dimensional subspaces $V \subset H_0^{\text{div}}(\mathcal{T})$, $U \subset L^2(\Omega)$, and $\Pi \subset H^{\frac{1}{2}}(\Omega)$, and find a solution $(\vec{v}, u, \pi) \in V \oplus U \oplus \Pi$ of (4.13). One such choice of subspace V, suitable for grids consisting of tetrahedrons, is the lowest order Raviart-Thomas elements [33] given by

$$V = \{\vec{v} \in L^\infty(\Omega) | \quad \vec{v} \text{ is linear on every } T \in \mathcal{T}, \\ \vec{v} \cdot \hat{n}|_\gamma \text{ is constant for every } \gamma \text{ in } \partial\mathcal{T} \text{ and continuous on } \Omega\}.$$

One half of a typical basis-functions for the Raviart-Thomas elements is showed for two-dimensions in Figure 4.2. In three dimensions, the basis function has the same property of being parallel to all the interfaces except one. Joining this half with one in the neighboring cell, also being non-parallel to the same interface γ, gives a basis function $\vec{\psi}_\gamma$ with support on the two cells.

Subspaces U and Π can for example be chosen as the piecewise linear functions on the cells T in $\mathcal{T}$ and the interfaces γ in $\partial\mathcal{T}$ respectively. The solution vectors can be expressed as linear combinations of the basis-functions on the form

$$\vec{v} = \sum_{\gamma \in \partial\mathcal{T}} v_\gamma \vec{\psi}_\gamma, \quad u = \sum_{T \in \mathcal{T}} u_T \xi_T, \quad \pi = \sum_{\gamma \in \partial\mathcal{T}} \pi_\gamma \eta_\gamma, \tag{4.14}$$

where ξ_T gives a basis function for U with support on the cell T, and η_γ is the basis function of Π with support on γ. Let the cells T and the interfaces

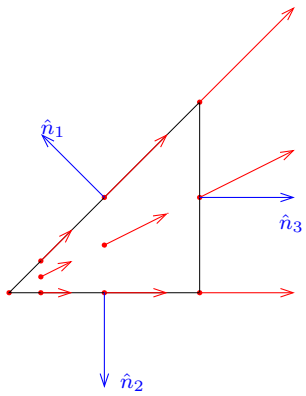


Figure 4.2: Illustration of the half of a basis function (in red) for the lowest order Raviart-Thomas elements in two dimensions on a triangular grid-cell. It is a linear vector field which at two of the three edges is parallel to them. The dot product between the unit normal vector and the basis function is one at the third edge.

γ be numbered globally, with l being the total number of cells and m the number of interfaces in Ω. The system (4.13) can now be written as a linear system in the unknowns $\vec{v}_h = [v_1, v_2, \cdots, v_m]^T$, $\vec{u}_h = [u_1, u_2, \cdots, u_l]^T$, and $\vec{\pi}_h = [\pi_1, \pi_2, \cdots, \pi_m]^T$ on the form

$$\begin{bmatrix} \mathbf{B} & \mathbf{C}^T & \mathbf{D}^T \\ \mathbf{C} & 0 & 0 \\ \mathbf{D} & 0 & 0 \end{bmatrix} \begin{bmatrix} \vec{v}_h \\ \vec{u}_h \\ \vec{\pi}_h \end{bmatrix} = \begin{bmatrix} \vec{0} \\ \vec{f}_h \\ \vec{0} \end{bmatrix}. \tag{4.15}$$

Here, $\vec{f}_h = [f_1, f_2, \cdots, f_l]$ where $f_T = (f, \xi_T)$. The matrix $\mathbf{B} = [b_{ij}]$ has elements $b_{ij} = b(\vec{\psi}_i, \vec{\psi}_j)$ and is block-diagonal if the basis functions are ordered cell-wise.

4.4.2 Schur-complement reduction

The matrix $\mathbf{B}$ is also invertible since the system is positive definite. Multiplying the first row of the matrix-system in (4.15) with $\mathbf{B}^{-1}$ gives us $\vec{v}_h = \mathbf{B}^{-1}(-\mathbf{C}^T\vec{u}_h - \mathbf{D}^T\vec{\pi}_h)$ which can be substituted into the second and third line. Note that inverting $\mathbf{B}$ can be performed for each cell by itself since it is block diagonal, and it is therefore not a time consuming operation to invert $\mathbf{B}$. This reduction is called Schur-complement and leads to a smaller linear system

$$\begin{bmatrix} \mathbf{E} & -\mathbf{F}^T \\ \mathbf{F} & -\mathbf{D}\mathbf{B}^{-1}\mathbf{D}^T \end{bmatrix} \begin{bmatrix} -\vec{u}_h \\ \vec{\pi}_h \end{bmatrix} = \begin{bmatrix} \vec{f}_h \\ \vec{0} \end{bmatrix}, \tag{4.16}$$

where $\mathbf{E} = \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^T$ and $\mathbf{F} = \mathbf{D}\mathbf{B}^{-1}\mathbf{C}^T$. The basis functions ξ_T of U are usually chosen to be piecewise constants like the one we suggested above. If ξ_T is set to be one on the cell T and zero everywhere else, then we get a matrix $\mathbf{C}$ with the same form as the mimetic discretization in Figure 4.4. Thus, $\mathbf{E}$ is a diagonal matrix. Another Schur-complement reduction can therefore be performed, where $\mathbf{E}$ is computationally cheap to invert. The final linear system becomes $\mathbf{A}\vec{\pi}_h = \vec{b}$ where $\mathbf{A} = \mathbf{D}\mathbf{B}^{-1}\mathbf{D}^T - \mathbf{F}\mathbf{E}^{-1}\mathbf{F}^T$ and $\vec{b} = \mathbf{F}\mathbf{E}^{-1}\vec{f}_h$.

4.5 Mimetic finite element method

It is not trivial to find good subspaces V of $H_0^{\text{div}}(\Omega)$ that make the bilinear form $b(\cdot,\cdot)$ easy to compute for irregular grids. One solution to this is to partition the cells of the grid into tetrahedrons, but this increases the number of unknowns in the linear system and thus, increase the time needed to

solve it. Mimetic method finds instead a replacement $m(\cdot,\cdot)$ that mimics the behavior of $b(\cdot,\cdot)$. Similar to the hybrid formulation, the mimetic formulation uses the bilinear forms in (4.13) in order to have a symmetric positive definite system.

We present a geometric approach to find this inner product $m(\cdot,\cdot)$. This technique is a type of finite difference method that uses a certain set of discrete points within each cell, to get a system of equations in the same form as equation (4.15). This system gives exact solution of the hybrid formulation if u is linear and $\mathbf{K}$ is constant in each cell.

Consider the two-dimensional illustration of a grid-cell in Figure 4.3. The flux is only used at the faces of the cell (edges in 2D). We can write the flux $\vec{v}|_i$ that goes through face i as a scalar v_i times an average unit normal vector $\hat{n}_i$

$$\vec{v}|_i = v_i\hat{n}_i = \int_{\gamma_i} \vec{v}\cdot\hat{n}\,dA.$$

From (3.2) we know that the flux $\vec{v}$ is a linear transformation of $\vec{\nabla}u$. Assume therefore that the fluxes through each interface i of the cell can be written as a linear transformation of the drops in driving force u at the face compared to the mass-center of the cell. In other words, if we create a vector containing the values v_i at the n faces of a cell and call it $\vec{v}_f = [v_1, v_2, \cdots, v_n]^T$, then there is a matrix $\mathbf{T}$ such that

$$\vec{v}_f = \mathbf{T}(\vec{u}_f - \vec{u}_c), \tag{4.17}$$

where the right hand side gives the drops in driving force from the mass-center $\vec{u}_c = u_c\vec{1} = [u_c, \cdots, u_c]^T$ of the cell to the centroid $\vec{u}_f = [u_1, u_2, \cdots, u_n]^T$ of each face.

In this way we consider only values for u and v at the places illustrated in Figure 4.3, that is, the centroid at each interface i, and the mass-center c of each cell. Although mimetic finite difference methods are able to handle curved surfaces, we will henceforth consider only planar interfaces.

If we assume that u is linear and can be written on the form $u = \vec{x}^T\vec{a} + c$, for a position-vector $\vec{x}$, we get

$$v_i = -\mu^{-1}A_i\hat{n}_i^T\mathbf{K}\vec{\nabla}u = -\mu^{-1}A_i\hat{n}_i^T\mathbf{K}\vec{a}, \tag{4.18}$$

where A_i is the area of interface i. Note that $u_i - u_c = (\vec{x}_i - \vec{x}_c)\vec{a}$, where $\vec{x}_i$ and $\vec{x}_c$ are the position-vectors to the centroid and the mass-center of face i respectively. This gives us two equations for $\vec{v}_f$ in each cell and we are able

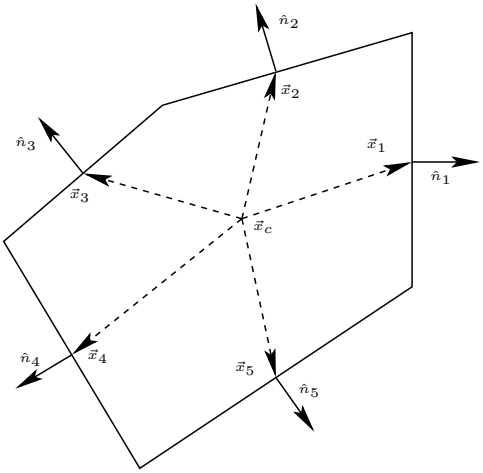


Figure 4.3: Two-dimensional illustration of the vectors used to establish the mimetic formulation. The potential for flow u is discretized into u_c at the mass-center $\vec{x}_c$ (cell centroid in 2D) of the cell and u_i at the centroid of face i (edge midpoint in 2D.) Each vector $\vec{x}_i$ starts at the mass center and ends at the centroid of interface i.

to find $\mathbf{T}$ from them since

$$\vec{v}_f = \mathbf{T} \overbrace{\begin{bmatrix} (\vec{x}_1 - \vec{x}_c)^T \\ (\vec{x}_2 - \vec{x}_c)^T \\ \vdots \\ (\vec{x}_n - \vec{x}_c)^T \end{bmatrix}}^{\mathbf{X}} \vec{a} = \mu^{-1} \overbrace{\begin{bmatrix} A_1 \hat{n}_1^T \\ A_2 \hat{n}_2^T \\ \vdots \\ A_n \hat{n}_n^T \end{bmatrix}}^{\mathbf{N}} \mathbf{K}\vec{a} \implies \mathbf{TX} = \mu^{-1}\mathbf{NK}. \tag{4.19}$$

Lemma 1. *We find a valid solution of* (4.19) *by setting* $\mathbf{T} = \frac{\mu^{-1}}{V}\mathbf{NKN}^T + \mathbf{T}_2$, *where* $\mathbf{T}_2\mathbf{X} = 0$ *and* V *is the volume of the cell.*

Proof. This is clearly the case if $\mathbf{N}^T\mathbf{X} = V\mathbf{I}_3$ for the three-dimensional identity-matrix $\mathbf{I}_3$. We therefore check index z_{ij} in the matrix $\mathbf{N}^T\mathbf{X} = [z_{ij}]$. Writing $\mathbf{N} = [\vec{n}_1|\vec{n}_2|\vec{n}_3]$ and $\mathbf{X} = [\vec{x}_1|\vec{x}_2|\vec{x}_3]$ and using superscript $\hat{n}_k^{(i)}$ to represent the i-th Cartesian coordinate of $\hat{n}_k$ gives

$$z_{ij} = \vec{n}_i^T x_j = \sum_{k=1}^{n} A_k \hat{n}_k^{(i)} (\vec{x}_k - \vec{x}_c)^{(j)} \tag{4.20}$$

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

the center to a cell-wall, we can use the divergence theorem to conclude

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

and the proof is complete. The outline of this proof was given to the author through private communications with Stein Krogstad at SINTEF. □

Having an expression for $\mathbf{T}$ makes it possible to build a system of equations, while ensuring continuity in the driving force of flow u and in the flux $\vec{v}_f$ at the centroids of the cell-walls. Equation (4.17) holds for this $\mathbf{T}$, and we get the following expression for each cell in the grid.

$$\mathbf{T}^{-1}\vec{v}_f - \vec{u}_f + \vec{u}_c = 0 \tag{4.21}$$

From this, we can build the system of equations in (4.15) as long as $\mathbf{T}$ is invertible. However, $\mathbf{T}$ is positive semidefinite and not strictly positive definite. We are therefore not guaranteed that $\mathbf{T}$ is invertible for any $\mathbf{T}_2$ satisfying Lemma 1. The following Theorem is presented by F. Brezzi in [7] and gives a recipe for choosing $\mathbf{T}_2$ in a way to ensure positive definiteness.

Theorem 1. *Let* $\mathbf{F}$ *be a* $n \times (n-d)$ *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*

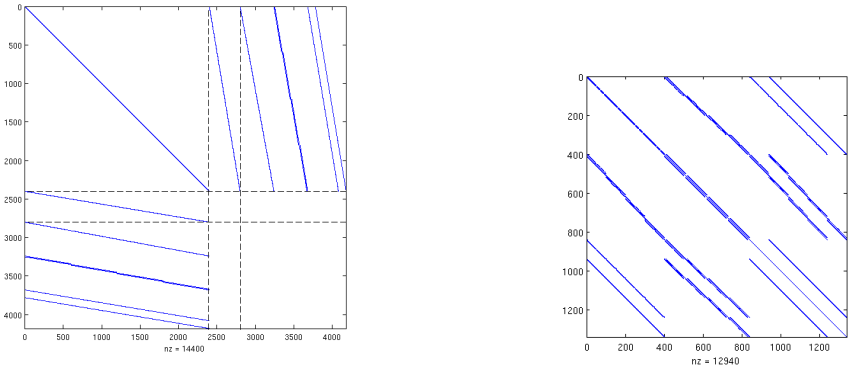
$$\mathbf{T} = \frac{\mu^{-1}}{V}\mathbf{N}\mathbf{K}\mathbf{N}^T + \mathbf{F}\mathbf{U}\mathbf{F}^T$$

which makes $\mathbf{T}$ *symmetric positive definite (SPD), satisfying* $\mathbf{T}\mathbf{X} = \mu^{-1}\mathbf{N}\mathbf{K}$*.*

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

If we assume that there exists a non-zero vector $\vec{v} \neq \vec{0}$ such that $\mathbf{T}\vec{v} = 0$, then we have

$$||\frac{\mu^{-1/2}}{V^{1/2}}\mathbf{K}^{1/2}\mathbf{N}^T\vec{v}||_2^2 + ||\mathbf{U}^{1/2}\mathbf{F}^T\vec{v}||_2^2 = 0.$$



(a) Full linear system **(b)** Reduced system

Figure 4.4: Sparsity pattern of the linear systems coming from a $10 \times 10 \times 4$ Cartesian grid, which is discretized by the mimetic finite element method. To the left, the matrices $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ are written together, like in equation (4.15), and are separated by dotted lines. At the right, we have the sparsity pattern of $\mathbf{A}$ coming from the reduced linear system after Schur decomposition.

This implies that $\mathbf{N}^T\vec{v} = \vec{0}$ and $\mathbf{F}^T\vec{v} = \vec{0}$, since both $\mathbf{K}^{1/2}$ and $\mathbf{U}^{1/2}$ are positive definite. This means that $\vec{v} \in \ker(\mathbf{F}^T) = \{\text{im}(\mathbf{F})\}^T = \{\ker(\mathbf{X})\}^T = \text{im}(\mathbf{X})$, and we can write $\vec{v}$ as a linear combination of the columns of $\mathbf{X}$ on the form $\vec{v} = \mathbf{X}\vec{w}$ for a vector $\vec{w}$. Hence, $\mathbf{N}^T\vec{v} = \mathbf{N}^T\mathbf{X}\vec{w}$. From Lemma 1 $\mathbf{N}^T\mathbf{X} = V\mathbf{I}_d$, which implies that $\vec{w}$ has to be zero since $\mathbf{N}^T\vec{v} = \vec{0}$ Thus, $\vec{v}$ is also zero, which is a contradiction to our assumption of $\vec{v}$. Hence, the matrix $\mathbf{T}$ presented in this theorem is SPD. □

Remember that (4.21) corresponds to a single cell, while equation (4.15) is including the whole grid. Thus, to build the complete system, we have to calculate $\mathbf{T}_i^{-1}$ for each cell i. Then, $\mathbf{B}$ is block-diagonal and invertible, $\mathbf{B} = \text{diag}(T_1^{-1}, T_2^{-1}, \cdots, T_l^{-1})$. Building $\mathbf{C}$ and $\mathbf{D}$ of equation (4.15) is fairly simple. They have value one on the places corresponding to the values of u at the cell-centers and face-centroids, and value zero elsewhere. The sparsity patterns of $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ are illustrated for a Cartesian grid in Figure 4.4a. The resulting system can be compressed by Schur-complement reduction, just like we did in Section 4.4.2 for the hybrid system. This gives a system $\mathbf{A}\vec{x} = \vec{b}$, where $\mathbf{A}$ have the structure of Figure 4.4b.

If we choose a SPD matrix $\mathbf{U}$, we have all we need to build and solve the linear system. When the system is solved by the CG-algorithm we present in Chapter 5, or by other iterative solvers, the rate of convergence is typically decided by the condition number κ. It is the ratio between the largest and the smallest eigenvalue of the matrix $\mathbf{A}$. The eigenvalues of A is strongly related

to the eigenvalues of $\mathbf{T}$, and we should choose $\mathbf{U}$ such that the eigenvalues lie closely together. One common choice is to construct $\mathbf{U}$ from the normal-vectors of the cell, in a way such that the mimetic discretization is equal to the hybrid discretization on tetrahedral cells with Raviart-Thomas elements. This approach is used in the toolkit we mention in Section 6.5.

The most apparent advantage of mimetic formulation is the ability to handle complex polyhedral grid cells in a straight-forward manner. Stability and convergence of mimetic finite difference methods are established by F. Brezzi for very general grids in [6]. In this thesis, we discretize grids where the cells not necessarily are aligned. The mimetic method is therefore used to discretize most of our models.

Chapter 5

Linear algebra solvers

All the discretization techniques we described in the previous chapter construct linear systems $\mathbf{A}\vec{x} = \vec{b}$ and seek the solution $\vec{x}$, when $\mathbf{A}$ is symmetric positive definite (SPD). So our next task is to build these systems on a computer, and find $\vec{x}$ numerically.

Constructing the linear system on a computer from a grid with permeability values varies in complexity between the different discretizations. For example, the TPFA method is only required to calculate the harmonic average of $\mathbf{K}$, the length between the cell-centers, and the area of each interface to estimate the flux. Building the resulting linear system is therefore simple and calculated quickly. Irregular grids demand more calculations than the $\mathbf{K}$-orthogonal ones. The methods designed to handle irregular grids have to consider more values when the system i built. In any case, the behavior in one cell of the grid is usually fully described by the values of its neighboring cells. Building the system has therefore in most cases a running time proportional to the number of cells in the grid. We show this for our discretization methods in Section 7.5.

Solving the linear system of equations is, however, asymptotically slower. It is usually the most time-consuming part of analyzing a physical flow-problem. We therefore look closer at techniques for solving linear systems which are SPD.

5.1 Choosing an appropriate solver

Finding $\vec{x} = \mathbf{A}^{-1}\vec{b}$ is performed by either a direct method solving the system exactly using matrix-calculus, or by iterative techniques. The most common direct solver is Gaussian elimination. It carries out an Cholesky-factorization

$\mathbf{L}\mathbf{L}^T$ of $\mathbf{A}$, where $\mathbf{L}$ is a lower-triangular matrix. Then, it solves $\vec{z} = \mathbf{L}^{-1}\vec{b}$ and $\vec{x} = \mathbf{L}^{-T}\vec{z}$ by back substitution.

Gaussian elimination and other direct methods are not very effective on sparse matrices. This is because they alter the matrices themselves when solving the system. The matrices loose some of their sparsity and this effect is called *fill-in*. Iterative techniques however, rely on repeatedly applying the matrix (or parts of it) on vectors, and can therefore fully exploit the sparsity of a linear system.

The fill-in of Gaussian elimination comes during the Cholesky-factorization. If we have the matrix resulting from the TPFA discretization described in Section 4.1, then all the values between the outer-most diagonals turn to non-zero values (look at the top left corner of Figure 7.5 for an illustration of the matrix). The complexity of Gaussian elimination on banded matrices with bandwidth B is therefore $\mathrm{O}(B^2 n)$.

In spite of the fill-in effect, Gaussian elimination is faster than most iterative techniques for two-dimensional problems where the bandwidth is typically of order $B \approx \sqrt{n}$. On three-dimensional problems however, several iterative methods surpass Gaussian eliminations in both speed and storage-requirements. The following method is an example of such an effective iterative solver.

5.2 Overview of the CG method

The conjugate gradient (CG) method is an iterative technique, which has become the most prominent method for solving sparse systems of linear equations. It was originally intended as an exact solver, which step by step found the distance of the correct solution along $\mathbf{A}$-orthogonal search directions, for a SPD matrix $\mathbf{A}$. Later, it was reinvented as an iterative method due to a quick convergence rate. An excellent explanation of the CG method is written in [38].

Two criteria make up the CG algorithm. The first is that the search directions $\vec{p}_k$ are kept $\mathbf{A}$-orthogonal such that $\vec{p}_i^T \mathbf{A}\vec{p}_j = 0$ for $i \neq j$. The second is that the residuals $\vec{r}_k = \vec{b} - \mathbf{A}\vec{x}_k$ are orthogonal to each other. These criteria are enough to pick the correct length α_k to move along each direction $\vec{p}_k$, and in this manner we cut down the error one component at the time.

We start with an initial guess of $\vec{x}_0$ with search direction $\vec{p}_0$ parallel to $\vec{r}_0$. Then $\vec{x}$ should be updated along the search direction $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$ by the correct scalar α_k. The residual is therefore updated as well

$$\vec{r}_{k+1} = \vec{b} - \mathbf{A}(\vec{x}_k + \alpha_k \vec{p}_k) = \vec{r}_k - \alpha_k \mathbf{A}\vec{p}_k. \tag{5.1}$$

Since the residual is set to be orthogonal, we find α_k from

$$\vec{r}_{k+1}^T\vec{r}_k = (\vec{r}_k - \alpha_k\mathbf{A}\vec{p}_k)^T\vec{r}_k = 0 \implies \alpha_k = \frac{\vec{r}_k^T\vec{r}_k}{\vec{r}_k^T\mathbf{A}\vec{p}_k}. \tag{5.2}$$

Next, we update the search direction as a linear combination of the following residual and the current search direction $\vec{p}_{k+1} = \vec{r}_{k+1} + \beta_k\vec{p}_k$, where we need to find β_k. This expression can be used to simplify the denominator in (5.2) into

$$\vec{r}_k^T\mathbf{A}\vec{p}_k = (\vec{p}_k - \beta_{k-1}\vec{p}_{k-1})^T\mathbf{A}\vec{p}_k = \vec{p}_k^T\mathbf{A}\vec{p}_k \tag{5.3}$$

since $\vec{p}_{k-1}$ is $\mathbf{A}$-orthogonal to $\vec{p}_k$.

We can calculate β_k from the $\mathbf{A}$-orthogonality of the search directions, and use (5.1) to insert for one $\mathbf{A}\vec{p}_k$

$$\vec{p}_{k+1}^T\mathbf{A}\vec{p}_k = (\vec{r}_{k+1} + \beta_k\vec{p}_k)^T\mathbf{A}\vec{p}_k = \vec{r}_{k+1}^T\frac{1}{\alpha_k}(\vec{r}_k - \vec{r}_{k+1}) + \beta_k\vec{p}_k^T\mathbf{A}\vec{p}_k = 0. \tag{5.4}$$

The residuals are orthogonal, and we get β_k by using the expression of α_k from (5.2)

$$\beta_k = \frac{\vec{r}_{k+1}^T\vec{r}_{k+1}}{\alpha_k\vec{p}_k^T\mathbf{A}\vec{p}_k} = \frac{\vec{r}_{k+1}^T\vec{r}_{k+1}}{\vec{r}_k^T\vec{r}_k}. \tag{5.5}$$

As we mentioned earlier, the initial error $\vec{e}_0$ can be expressed as a linear combination of the $\mathbf{A}$-orthogonal search directions $\vec{e}_0 = \sum_{k=0}^{n} -\alpha_k\vec{p}_k$ and is reduced one component at the time [38]. The error is therefore minimized in the energy norm $||\vec{e}_k||_\mathbf{A}^2 = \vec{e}_k^T\mathbf{A}\vec{e}_k = \sum_{j=k}^{n} -\alpha_j\vec{p}_j^T\mathbf{A}\vec{p}_j$ along the search direction $\vec{p}_k$ at each iteration.

The Euclidean norm is, however, not necessarily minimized. Small components of the error may actually increase during the CG algorithm (though never for a long time). For this reason, the CG is called a *rougher* [38]. Roughers stand in contrast to *smoothers* which reduce every component of the error in each iteration. Jacobi and Gauss-Seidel iterations are examples of smoothers, and we discuss them briefly in Section 5.3.

The condition number $\kappa(\mathbf{A}) = \frac{\lambda_{\max}}{\lambda_{\min}}$ is descriptive when looking at convergence of iterative methods. It is defined as the ratio between the largest and smallest eigenvalues $\lambda_{\max}$ and $\lambda_{\min}$ of $\mathbf{A}$. From [38] we have that $||\vec{e}_i||_\mathbf{A} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i ||\vec{e}_0||_\mathbf{A}$, where e_0 is the error in the initial guess of the solution. We stop the iteration when the error is small compared to the initial error, that is when $\frac{||e_i||_\mathbf{A}}{||e_0||_\mathbf{A}} \leq t$ for a chosen relative tolerance t. This gives a convergence order of $\mathrm{O}(\sqrt{\kappa}n)$, where n is the number of non-zeros in $\mathbf{A}$. The CG method is therefore a good choice as solver for sparse systems.

The condition number κ usually increases with problem size, and high values of κ substantially slow down the CG algorithm. It is therefore common to use a *preconditioner* to lower the condition number. A preconditioner is an operation that can be represented by an invertible matrix $\mathbf{M}$, which mimics $\mathbf{A}^{-1}$ and we solve $\mathbf{MA}\vec{x} = \mathbf{M}\vec{b}$ instead of $\mathbf{A}\vec{x} = \vec{b}$. The number of iterations in the CG algorithm is then dependent on the condition number of $\mathbf{MA}$.

5.3 Iterative smoothers and multi-grid

Weighted Jacobi iteration is a simple iteration technique. It splits the diagonal $\mathbf{D}$ from the matrix $\mathbf{A} = \frac{1}{\omega}\mathbf{D} - (\frac{1}{\omega}\mathbf{D} - \mathbf{A})$, for a weight ω between zero and one. Utilizing this splitting, the system $\mathbf{A}\vec{x} = \vec{b}$ is solved by the iteration $\vec{x}_{i+1} = \mathbf{D}^{-1}((\omega\mathbf{A} - \mathbf{D})\vec{x}_i + \vec{b})$. The initial error $\vec{e}_0$ of a guessed solution vector can be written as a linear combination of the eigenvectors $\vec{w}_k$ of $\mathbf{A}$ on the form $\vec{e}_0 = \sum_k e_0^{(k)}\vec{w}_k$. Dependent on the value of ω, Jacobi iteration reduces some components of this error quickly while the others decrease slowly, and might not even converge.

Jacobi iteration decreases all the components of the error in every step. It is therefore a smoother. It is an especially useful when we want to decrease certain components of the error. Other well-known smoothers are Gauss-Seidel iteration and successive over-relaxation. Gauss-Seidel iteration is similar to Jacobi iteration, but it is in addition splitting out the lower triangular part $\mathbf{L}$ of $\mathbf{A}$ such that $\mathbf{A} = (\mathbf{L}+\mathbf{D}) - (\mathbf{L}+\mathbf{D}-\mathbf{A})$. The iteration then becomes $\vec{x}_{i+1} = (\mathbf{D}+\mathbf{L})^{-1}((\mathbf{A}-(\mathbf{D}+\mathbf{L}))\vec{x}_i + \vec{b})$. Successive over-relaxation is a weighted version of Gauss-Seidel iteration. All these smoothers are described in detail by J. Saad in [34].

Multi-grid is an iterative solver that takes advantage of the ability a smoother has to dampen specific components of the error. What a multi-grid solver does, is to change the resolution of the grid that $\mathbf{A}$ is built from and interpolate both the linear system and the partial solution between these grid-resolutions. The eigenvectors of $\mathbf{A}$ change, and if we apply a smoother at different resolutions, then it will reduce other components of the error. This gives a faster convergence rate than what the smoother has by itself, and is not very affected by high condition numbers $\kappa(\mathbf{A})$.

A multi-grid solver may also be created without knowledge of the underlying grid. Downscaling (*contraction*) and up-scaling (*relaxation*) can be defined from the structure of $\mathbf{A}$. This approach is called *algebraic multi-grid* [41], and is well suited as a preconditioner to the CG algorithm for general SPD matrices.

The CG algorithm performs substantially better with a preconditioner

like multi-grid. It is seldom used in industry without a good preconditioner. In spite of this, we do not implement a preconditioner in this thesis, but leave this as further work.

Chapter 6

Implementations

The conjugate gradient (CG) algorithm in the previous chapter finds α_k and β_k from (5.2) and (5.5) and use them to update the residual, the search direction, and the solution. To find these, it has to calculate a matrix-vector product, a vector-vector product, and the *axpy* routine of finding $\vec{y} = a\vec{x} + \vec{y}$ for a scalar a and vectors $\vec{x}$ and $\vec{y}$. The axpy calculation is defined for single precision (saxpy) and double precision (daxpy) as part of the library for basic linear algebra subroutines (BLAS) [8].

Solving the linear system $\mathbf{A}\vec{x} = \vec{b}$ is the most computationally costly operation we do when we analyze the flow in porous media. We therefore focus on optimizing the speed of the CG solver. To efficiently implement the CG method, we need to locate and classify its kernels, or the expensive calculations, of the algorithm.

6.1 Finding the kernels of the program

Vector-vector product and axpy calculation have sequentially a complexity of $\mathrm{O}(n)$, where n is the number of cells in the grid and is proportional to the number of elements in $\vec{x}$. For a full matrix $\mathbf{A}$, matrix-vector product has complexity of $\mathrm{O}(n^2)$, but for sparse matrices with N_{nz} non-zeros we can reduce the complexity to $\mathrm{O}(N_{nz})$.

For the linear systems we found in Chapter 4, the number of non-zeros in $\mathbf{A}$ scales linearly with n. The matrix-vector product has therefore in this case asymptotically the same complexity as the vector-vector product and the axpy calculation. Thus, the kernels of our program are these three operations.

The axpy operation is highly parallel, and a straight-forward implementa-

Listing 6.1: CG-system interface defining the functions that are needed to solve a system with the conjugate gradient method.

```
1  #pragma once
2  #include "commonHeader.h"
3
4  template <class ValueType>
5  class CGsystem{
6  public:
7    CGsystem(){}
8    ~CGsystem() {};
9    virtual void init(ValueType **x, ValueType **r, ValueType **p, ValueType **Ap) = 0;
10   virtual ValueType vectorDotProduct(const ValueType*v1, const ValueType *v2) const = 0;
11   virtual void matrixVectorProduct(const ValueType *v, ValueType *result) const = 0;
12   virtual void axpy(const ValueType a, const ValueType *x, ValueType *y) const = 0;
13   virtual void scale(const ValueType scalar, ValueType *v) const =0;
14   virtual void getSolution(ValueType **solution) const = 0;
15   virtual size_type getN() const = 0;
16   virtual void printVector(const ValueType *v) const = 0;
17   virtual void printMatrix() const = 0;
18 };
```

tion letting each thread calculate one index of the solution each is an optimal implementation. Matrix-vector and vector-vector product have more potential for optimalization. We therefore mention, in Section 6.8, how the vector-vector kernel is made optimal. In Section 6.3 we present different sparse formats that we use to get optimal matrix-vector kernels, but before that, we introduce the CG algorithm and how the different kernel-implementations should be written to fit into a common framework.

6.2 The CG algorithm

The CG method is simple to implement if we have the functionality the three kernels available. We therefore create an C++ interface defining the necessary functions for a CG routine. We name the interface *CG-system* and the code is shown in Listing 6.1. Note that we have included a function scale, which scales a vector with a scalar. This function comes in handy when we set vectors to zero and to ease the update of the search-direction.

An object implementing the CG interface holds the matrix $\mathbf{A}$, the vector $\vec{b}$, and functions for the operations that will be performed on them. In addition, a CG-system holds the temporary variables used in the CG algorithm, and these are linked to local pointers by the call `init()`. This allows the objects implementing the CG-system to store the vectors and matrices at different memory spaces and in different formats.

For example, we have the possibility of letting one object solve with matrix-vector multiplication for full matrices on the CPU, while another is working on a sparse matrix with an optimized GPU-implementation of matrix-vector product. The CPU implementation should store the matrix

Listing 6.2: The conjugate gradient algorithm finding $\vec{x}$ from the equation $\mathbf{A}\vec{x} = \vec{b}$ implemented in C++. The input to the algorithm is an object implementing the CG-system of Listing 6.1. The input object has to contain a representation of the matrix $\mathbf{A}$ and the vector $\vec{b}$, stored in any format.

```
#pragma once
#include "CGsystem.h"
#include "math.h"

template <class ValueType> ValueType cg(
    CGsystem<ValueType> *A, ValueType tolerance, ValueType threshold
){
  ValueType alpha, beta, rnorm, oldnorm, pTAp, endNorm; // scalars
  ValueType *x, *r, *p, *Ap;  // vectors
  A->init(&x, &r, &p, &Ap); // x = 0, r = b, p = b
  rnorm = A->vectorDotProduct(r,r);
  endNorm = tolerance*tolerance*rnorm*threshold;
  for (unsigned int k=0; (k < A->getN()) && (rnorm > endNorm); ++k){
    A->matrixVectorProduct(p, Ap);  // Ap = A*p
    pTAp = A->vectorDotProduct(p, Ap);
    // If pTAp == 0, we have a lucky breakdown.
    if (pTAp == (ValueType)0.0) break;
    alpha = rnorm / pTAp;
    // The iteration stagnates if alpha becomes zero
    if (alpha == (ValueType)0.0) break;
    // axpy calculating x += alpha*p and r -= alpha*Ap:
    A->axpy(alpha, p, x);
    A->axpy(-alpha, Ap, r);
    oldnorm = rnorm;
    rnorm = A->vectorDotProduct(r,r);
    beta = rnorm / oldnorm;
    // Calculate the update p = beta * p + r.
    A->scale(beta, p);
    A->axpy(1.0, r, p);
  }
  return sqrt(rnorm);
}
```

and temporary variables on the motherboard RAM, while the GPU profits from having temporary variables stored on the device memory which is closer to the GPU. Both these objects can implement the CG-system interface and their linear systems can be solved by the same CG algorithm. We use this interface to write one object for every matrix format we implement.

Listing 6.2 shows the CG algorithm written for a CG-system. It starts by a solution guess $\vec{x}_0 = \vec{0}$ and sets $\vec{r}_0 = \vec{b}$ and $\vec{p}_0 = \vec{b}$ accordingly. Then, it iterates until the norm has decreased with more than a given tolerance factor, and accounts for *lucky breakdown*. Lucky breakdown is when the search direction p_k has $\mathbf{A}$-norm zero, which means that we have an exact solution if we do not account for round-off errors [34].

6.2.1 Additions to the CG code

As mentioned in Section 5.2, the conjugate gradient method exploits a smart way of updating the search direction $\vec{p}_k$ so that it is $\mathbf{A}$-orthogonal to all the previous search directions. Round-off errors may alter this behavior such that the search-directions are only close to $\mathbf{A}$-orthogonal, and this defect may lead

to larger errors in following steps. The CG algorithm presented here may therefore not converge to the solution with full floating point accuracy at the first try.

A way to fix the errors coming from limited floating point accuracy is to restart the algorithm. This updates the first residual $\vec{r}_0$ and the search direction $\vec{p}_0$ to the correct values, and starts the algorithm without round-off errors, but closer to the solution.

We modify the code to recompute the norm when the main loop is finished, and assure that the real norm is still within the criteria of the tolerance. Restarting the algorithm means that we loose the information ensuring **A**-orthogonality of the search directions. Because of this, we do not enjoy as fast convergence towards the correct solution, in the first iterations after a restart, as we do after several iterations. To make sure that we do not restart more often than what is necessary, we decrease the `endNorm` by a given threshold. For double precision decreasing the relative tolerance by a factor of $\sqrt{0.8}$ gave good results. If restarting does not give a better result, we let the algorithm terminate with the best solution found so far.

CG is a rougher and the error is not necessarily decreasing in every iteration. The calculated solution starts moving away from the correct solution when round-off errors builds up. We therefore modify the code to store a backup of the solution with the lowest residual-norm so far. To avoid taking backup too often, we only store the solution if the residual-norm $||r_{k+1}||_2$ rises from a previous norm which was the smallest so far. Line 22 in Listing 6.2 is therefore moved to after line 25, so that the next norm is calculated before the solution vector is updated.

Higher floating point accuracy also helps to decrease the effect of round-off error. We therefore use a template `ValueType` when creating the CG algorithm and objects implementing the CG-system interface. In this way, we can use the same object and CG code for both single and double precision floating point accuracy. This is handy since most GPUs support single precision, while only some newer cards support double precision. Single precision is also faster to compute in some cases.

6.2.2 Solving in mixed precision

Solving the system $\mathbf{A}\vec{x} = \vec{b}$ directly in the CG algorithm of Listing 6.2 does not give a high accuracy with single precision floating point accuracy. When the residual $\vec{r}$ becomes small compared to $\vec{x}$, round-off errors become more dominant. To compensate for this, we may solve for the increment $\Delta\vec{x}$ of the current solution guess $\vec{x}$, instead of finding $\vec{x}$ directly [12]. We call this approach *defection-correction*.

The idea of defection-correction is to solve the system $\mathbf{A}(\vec{x} + \Delta\vec{x}) = \vec{b}$ for $\Delta\vec{x}$. This gives the linear system $\mathbf{A}(\Delta\vec{x}) = \vec{z}$ where $\vec{z}$ is the residual of the current solution candidate $\vec{x}$. Two advantages of this method decrease the round-off error. One is that the magnitude of $\vec{x}$ no longer affects CG algorithm, the other is that we can scale the system so that the values of $\vec{\Delta}x$ are close to one. The algorithm becomes as follows:

1. Make a guess of the solution $\vec{x}_{\text{out}}$.
2. Find the residual $\vec{z}_{\text{out}} = \vec{b} - \mathbf{A}\vec{x}_{\text{out}}$.
3. Scale the residual by its norm: $\vec{z}_{\text{in}} = \vec{z}_{\text{out}}/||\vec{z}_{\text{out}}||_2$.
4. Solve $\mathbf{A}(\Delta\vec{x}_{\text{in}}) = \vec{z}_{\text{in}}$ with respect to $\Delta\vec{x}_{\text{in}}$ for example by the CG algorithm.
5. Update $\vec{x}_{\text{out}} = \vec{x}_{\text{out}} + ||\vec{z}_{\text{out}}||\Delta\vec{x}_{\text{in}}$.
6. If the new residual $\vec{z}_{\text{out}} = \vec{b} - \mathbf{A}\vec{x}_{\text{out}}$ is smaller than the given tolerance factor, then the solution is $\vec{x}_{\text{out}}$ and we are finished. Otherwise, continue from step 3.

This technique for solving for the residual is presented by Göddeke and Strzodka in what they call a *mixed precision solver* [12]. A mixed precision solver solves the inner system $\mathbf{A}(\Delta\vec{x}) = \vec{z}$ of the algorithm in a lower precision, while the other steps are calculated in a higher precision. The algorithm is designed to use a co-processor, which performs better in lower precision, to achieve as high accuracy as one would with a full implementation in the higher precision.

This suits the GPU since many GPUs does not support double precision. However, it can also offer speed-up to GPUs which do support double precision. This is mainly because the amount of memory that is transferred to the GPU is halved in single precision compared to double precision, but also because single precision operations may be computed faster on some GPUs.

The relative tolerances for the inner and outer iteration need to be provided when this algorithm is started. In the mixed precision (MP) implementation we let two CG-systems specify the matrix-formats. Using the CSR format on the CPU for the outer system and single precision HYB format for the GPU as the inner is typical for a GPU that does not support double precision However, the best performance is achieved by using two HYB implementations on the GPU, where the outer system uses double precision while the inner uses single precision. This is used in Chapter 7 to represent the MP solver.

The number of solvers we have implemented for the GPU is three:

- The straight forward CG algorithm in Listing 6.2 with the improvements of Section 6.2.1
- The algorithm using defection-correction (DC) as explained above
- The mixed precision (MP) solver.

6.3 Sparse representations

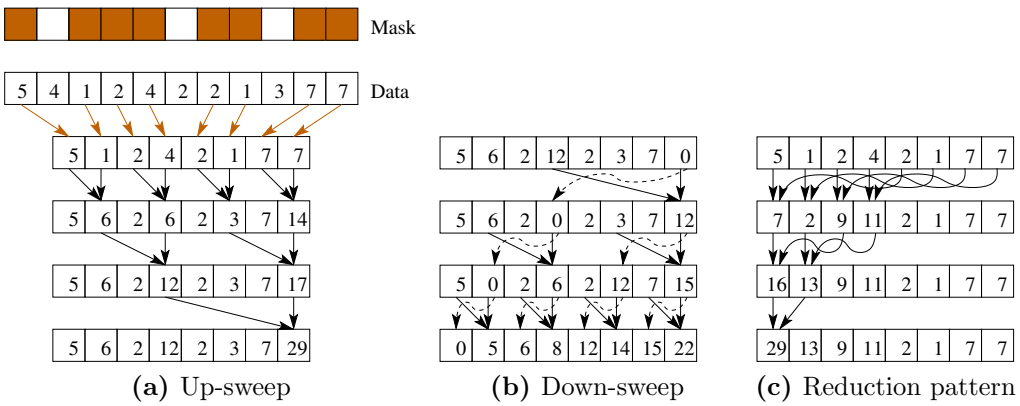
All the discretization methods we have investigated in Chapter 4 builds the matrix **A** from local dependencies between neighboring cells in a grid. A discrete value in one cell is fully determined from discrete values in the same cell or in neighboring cells. In other words, a row in **A** which corresponds to one discrete value has a limited number of non-zero entries.

For the mimetic finite element, the number of non-zeros of a row is at most twice the number of neighboring cells (actually one less). Thus, **A** is sparse with $O(n)$ non-zero values, where n is the number of cells in the grid. The other discretizations presented here have also this property, containing $O(n)$ non-zero values. Both storage space and computation time is saved if we store the matrices in a sparse format.

6.3.1 Sparse matrix libraries for CUDA

The current release of CUDA and its source development kit (SDK) version 2.1 have support for matrices and matrix operations in the basic linear algebra subroutines (BLAS) called the CUBLAS library [28]. CUBLAS has mainly support for full matrices, but may also be used to represent symmetric and band matrices in storage-efficient ways. Symmetric matrices can be stored as just their upper or lower triangular part in a compact array. Band matrices have the possibility to store only the values between the two outer-most non-zero diagonals. CUBLAS has also support for a combination of symmetric- and band-matrices.

There are two drawbacks of these formats: One is that they are currently only supported in single precision. The other is that when used with the diagonal matrices we get from TPFA and the O-method in Chapter 4, the band structure stores many zero-valued diagonals between the non-zero ones. These matrices are more efficiently stored in the sparse diagonals (DIA) format we present in Section 6.3.3. We therefore restrict us to implement the



(a) Up-sweep **(b)** Down-sweep **(c)** Reduction pattern

Figure 6.1: Illustration of the segmented scan with addition as its binary operation. A mask tells which elements to consider and the cumulative partial sum is found by first doing a up-sweep of calculating a segmented reduction (a) and finish with setting the last element to zero and do a down-sweep (b) to get the desired result. For matrix-vector multiplication we only need the segmented reduction, and it is better implemented for a GPU if we use the index-pattern illustrated in (c).

format for full matrices in CUBLAS, and use it as a reference for speed and correctness.

A library for sparse matrices is unfortunately not yet included in the current CUDA SDK. However, employers at Nvidia have written a paper on sparse matrices [4] which, according to the forum where it was released, will be included (probably a modified version) in a future SDK release.

A sparse matrix-vector multiplication is also presented in a CUDA data parallel primitives (CUDPP) library [37]. This library includes a parallel reduction method called *segmented scan* or prefix-sum. The sparse matrix-vector multiplication included in CUDPP is implemented as a demonstration of how this segmented scan may be utilized. Segmented scan calculates the cumulative sum of elements prior to each element in an array, and is described in detail by M. Harris in [16]. The algorithm uses a bit-mask to tell which elements to include in the computation (see Figure 6.1), and can exchange the sum operation with any other binary operation.

The operation of sparse matrix-vector multiplication is better performed by *segmented reduction* which are simply to find the sum of an array. Segmented reduction can be implemented more efficiently since we only need to find the final sum and not every partial sum along the way. Optimizing the segmented reduction for the GPU is described by A. Torp in [42], and results in a reduction illustrated in Figure 6.1c. This segmented reduc-

tion is designed as a vector-vector product, and we use that code as our kernel-implementation. For matrix-vector product, we implement the sparse formats which are presented in [4]. Matrix-vector product can be viewed as a calculation of many vector-vector products. Segmented reduction is an efficient way of calculating the sparse vector-vector products if we have many non-zeros per row. It performs best when the number of non-zeros is larger than the warp-size, and we comment further on this in Section 6.9.

6.3.2 General sparse formats

There are a multitude of a sparse matrix representations. Some formats are created for matrices with special patterns or regularities while others focus on handling general matrices of any form. The formats have different storage requirements, computational characteristics and methods of accessing and manipulating entries of the matrix.

In this thesis, we assume that building the matrix is quick compared to the amount of time used on matrix-vector multiplication, and we use matrix formats with little optimization for insertions of values. We present different sparse formats and discuss how well suited they are for solving our system.

The simplest sparse format for $\mathbf{A} = [a_{ij}]$ is the *coordinate* (COO) format. Here, three arrays `row`, `col`, and `data` make up the i-coordinate, j-coordinate and the value a_{ij} respectively. This is a general sparse representation since any matrix can be stored in this format with storage requirements proportional to the number of nonzero elements in the matrix. In the implementation we present, the `row`-array is sorted to ensure that entries of the same row are stored contiguously in memory. This is to increase the number of coalesced reads when matrix-vector multiplication is performed on the GPU.

The COO format can easily be compressed when the `row`-array is sorted, and when there are (in average) more than one non-zero element in each row. This is achieved by storing the row-wise cumulative number of elements in `row` instead of storing the index. We choose to start the cumulative `row` array on zero. In this way, `row` has $(n + 1)$ elements, where the last element holds the number of non-zero elements in $\mathbf{A}$. This compression is called *compressed sparse row* (CSR) and is a widely used format.

Matlab uses the *compressed sparse column* (CSC) format, which is the same as CSR, but compressing `col` instead of `rows`. For a symmetric matrices, swapping `row` with `col` changes format from CSR to CSC. This comes in handy when we import symmetric matrices from Matlab into the CSR format for C++. An example of a matrix stored in COO, CSR, and CSC is showed in Figure 6.2.

$$\mathbf{A} = \begin{bmatrix} 5 & -1 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & 0 & 2 \end{bmatrix}$$

COO	`row`	0	0	1	2	2	2	3	3
	`col`	0	1	1	0	2	3	1	3
	`data`	5	-1	-1	2	-1	2	-1	2
CSR	`row`	0	2	3	6	8			
	`col`	0	1	1	0	2	3	1	3
	`data`	5	-1	2	-1	2	-1	-1	2
CSC	`row`	0	2	0	1	3	2	2	3
	`col`	0	2	5	6	8			
	`data`	5	-1	-1	2	-1	2	-1	2

Figure 6.2: A symmetric matrix **A** and it's sparse representations in coordinate (COO), compressed sparse row (CSR), and compressed sparse column (CSC) format.

$$\mathbf{A} = \begin{bmatrix} 5 & -1 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & 0 & 2 \end{bmatrix} \texttt{data} = \begin{bmatrix} * & 5 & -1 \\ * & 2 & 0 \\ -1 & 2 & -1 \\ -1 & 2 & * \end{bmatrix} \texttt{offset} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

Figure 6.3: An example of the sparse diagonal (DIA) format. The matrix `data` contains the diagonals of **A**, and `offset` holds their index relative to the main diagonal. The values marked $*$ can be any value. They are allocated in memory but never read or modified.

6.3.3 Problem-specific sparse formats

The TPFA method presented in Section 4.1 is aimed on structured grids. For Cartesian grids or **K**-orthogonal grids, we have that each cell has exactly six neighbors (except at the boundary). Each row of **A** in TPFA has in this case seven or less non-zero entries. The matrix gets a band structure, like the top left part of Figure 7.5, if we number the cells in the standard way of Cartesian grids. The *sparse diagonals* (DIA) format is intended for these types of matrices where the non-zero entries lay in a confined number of sub- or super-diagonals. This format stores the vectors representing the diagonals in a two-dimensional matrix `data` and the corresponding location relative to the main diagonal in an `offset` vector. An example of the DIA format is shown in Figure 6.3.

The mimetic discretization results in the three matrices **B**, **C** and **D**,

$$\mathbf{A} = \begin{bmatrix} 5 & -1 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & 0 & 2 \end{bmatrix} \texttt{data} = \begin{bmatrix} 5 & -1 & * \\ 2 & * & * \\ -1 & 2 & -1 \\ -1 & 2 & * \end{bmatrix} \texttt{col} = \begin{bmatrix} 0 & 1 & z \\ 1 & z & z \\ 0 & 2 & 3 \\ 1 & 3 & z \end{bmatrix}$$

Figure 6.4: An example of the ELLPACK/ITPACK (ELL) format. The matrix `data` contains the non-zeros of $\mathbf{A}$, and `col` holds their column indices. The values marked $*$ can be any value, and the value z represent a number which tells that the corresponding value is zero, and should not be read.

which after Schur decomposition is reduced to one matrix $\mathbf{A}$. This $\mathbf{A}$ does not have a diagonal structure, even with a structured grid. The DIA format will therefore in this case store more zero-values than non-zero values, and becomes ineffective. The pattern of such a reduced matrix from a Cartesian grid is showed in Figure 4.4b.

Even though its not diagonal, the matrix still has a limited number of non-zeros per line. The ELLPACK/ITPACK (ELL) format exploits this [13]. It stores the non-zeros of an M-by-N matrix in an M-by-K dense matrix `data`, where K is the maximum number of non-zeros per line. The rows with less than K non-zeros are zero-padded. The corresponding column indices are stored in a matrix `col` which is also padded with a sentinel value. An example of the ELL format is showed in Figure 6.4.

The ELL format is an efficient sparse matrix representation when the maximum number of non-zeros does not substantially differ from the average. This is the case for the discretizations presented herein as long as the grid is structured such that each cell has approximately the same number of neighboring cells. For this situation, the ELL format performs very well.

6.3.4 Hybrid representation

From the geometry of a three-dimensional grid, we have that the average number of neighboring cells is six, independent of the grid-structure. Grids which are badly suited for ELL format are those with a few cells having substantially more than six neighbors. These cause rows in the matrix $\mathbf{A}$ to contain several more non-zero values than the average. A remedy is to store the excess non-zero values from these rows in a different format.

The COO format has performance and storage requirement which is proportional to the number of non-zero elements, and is invariant of the number of non-zeros per row. N. Bell presents in [4] a hybrid (HYB) format as a mixture between ELL and COO. The HYB format relies on an estimate for

$$\mathbf{A} = \begin{bmatrix} 5 & -1 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & 0 & 2 \end{bmatrix} \texttt{Edata} = \begin{bmatrix} 5 & -1 \\ 2 & * \\ -1 & 2 \\ -1 & 2 \end{bmatrix} \texttt{Ecol} = \begin{bmatrix} 0 & 1 \\ 1 & z \\ 0 & 2 \\ 1 & 3 \end{bmatrix} \begin{matrix} \texttt{Crow} = & 2 \\ \texttt{Ccol} = & 3 \\ \texttt{Cdata} = & -1 \end{matrix}$$

Figure 6.5: An example of the hybrid (HYB) format. The matrix `Edata` and `Ecol` contains the ELL part, and `Crow`, `Ccol`, and `Cdata` holds the COO part.

how well ELL performs compared to COO, and uses this ratio to decide the ideal number of non-zeros per row in the ELL format. Assuming that the ELL format is three times faster than the COO format gave good results for the implementations in this thesis. The elements which do not fit into this ELL format are then stored in the COO format instead. An example of a matrix in the HYB format is in Figure 6.5.

The standard matrix-vector product $\vec{y} = \mathbf{A}\vec{x} + y$ is computed two times in the HYB format, once for the ELL representation of $\mathbf{A}$ and once for the COO representation. The CG algorithm we presented in Listing 6.2 asks for the computation of $\vec{y} = \mathbf{A}\vec{x}$ (without the update of $\vec{y}$). In this case, the matrix-vector product of HYB is calculated by first setting $\vec{y} = \mathbf{A}_{\text{ELL}}\vec{x}$ for the ELL part, and then update it by setting $\vec{y} = \mathbf{A}_{\text{COO}}\vec{x} + \vec{y}$ for the COO part.

6.4 Implementation of the matrix-formats

To analyze the different matrix-formats, we create one C++ object for each that we implement for the GPU. This gives a total of six systems for the GPU and three for the CPU. The first we implement is the CUBLAS system which uses the native support CUDA has for full matrices [28]. This class reads matrices coming from Matlab, and stores them directly into GPU memory in the CUBLAS format. The implementation of the DIA format is taken from the specialization project [42].

The sparse formats DIA, COO, ELL, HYB, and CSR also get one object each that implements the CG-system interface. They store the matrix and all other variables in the device memory close to the GPU. Sparse matrices coming from Matlab can be stored in either of the sparse formats, as we illustrate in Figure 6.6. Matlab is therefore required to specify the format when calling the C++ program. We also implement the CSR format and the HYB format on the CPU for comparison.

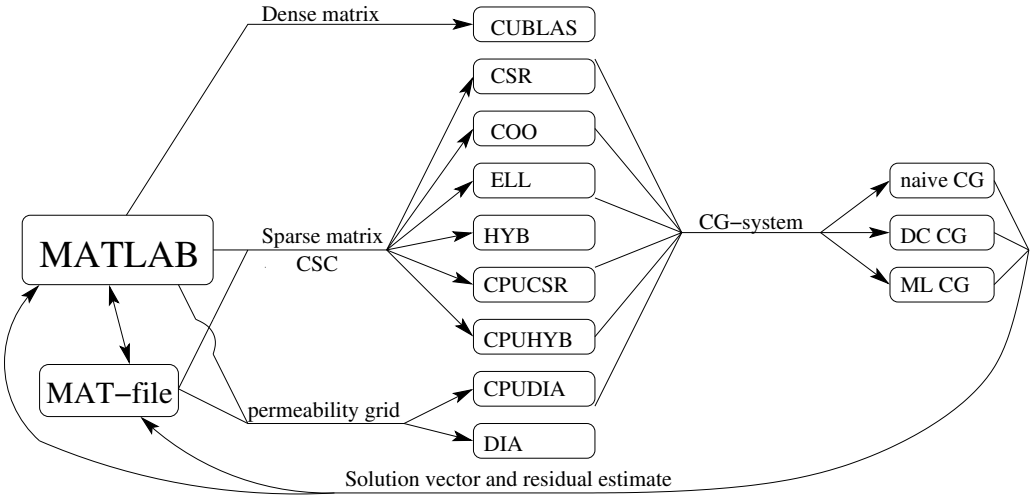


Figure 6.6: Schematic showing how the data flows through the program. Matlab sends either the permeability grid or a matrix which is either full or sparse to the C++ program. The program converts it to one of the objects listed. This object is passed to one of the CG algorithms which finds the solution and sends it back to Matlab for visualization.

When the linear system is built from a grid by the C++ program, Matlab has to specify if the grid is **K**-orthogonal or not. A **K**-orthogonal grid will be discretized by the TPFA method and stored in the DIA format. Otherwise, the linear system is found by the mimetic formulation, and is stored in the HYB format. Converting the **K**-orthogonal grid is performed by the code attached to the specialization project [42], and the mimetic discretization is carried out by the Matlab reservoir simulation toolkit.

6.5 The Matlab reservoir simulation toolbox

The Matlab reservoir simulation toolbox (MRST) consists of a set of routines and data types used to process and represent unstructured grids. It is especially designed for the corner-point grid-format which is commonly used in the petroleum industry.

The corner-point grid-structure uses the idea that rock-formations consists of sediment layers that have approximately the same height everywhere. When these formations of rock were twisted and shaped by the forces of nature, the formation changed from an initial position that would be well described by a Cartesian grid. The corner-point grid uses this initial grid to describe the current formation. Vertical pillars are kept straight, and di-

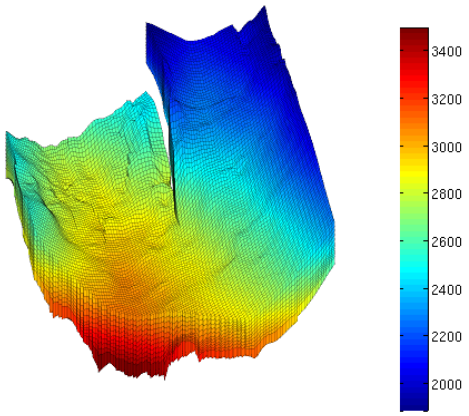


Figure 6.7: Height-map of the $100 \times 100 \times 11$ part of the Johansen formation. The formation is partitioned into a corner-point-grid, where vertical pillars separate the cells of the model. It is evident that there is what we call a fault in the model.

vides up the model. All cells have edges that are located along these vertical pillars. Each cell has therefore six or less interfaces.

When the rock-formation has a vertical crack, we call it a *fault*. Such a fault cause cells which belong to the same sediment layer to be misaligned like in Figure 7.9b. In Figure 6.7 we see a height-map of a representation of a rock-formation in the corner-point grid. We can see a crack formed in the center of the formation, which is represented as a fault in the grid-model.

The Matlab reservoir simulation toolbox contains functions for creating grids with permeability values, but is also able to process corner-point grids stored in the Eclipse format. Discretizing the grids are performed by either the mimetic method, the TPFA discretization, or by the hybrid method using Raviart-Thomas elements. The toolbox contains also functions performing up-scaling to reduce the size of the linear system. However, we will not use this technique here. Nevertheless, our solver is designed to handle all types of SPD linear systems created in MRST. In order to be able to solve these systems, the C++ program has to communicate with Matlab.

6.6 Data transfer between Matlab and C++

Sending information between Matlab and C++ can be done in two ways. One way is to keep the Matlab program and the C++ program separate and let Matlab store data in a file which is later read by the C++ program.

Allowing Matlab to execute the C++ program is the second option. This is possible if we compile the C++ program into a Matlab executable (MEX) file. Matlab is then able to send pointers to data stored in memory when the MEX file is started.

One advantage of creating a MEX file is that we avoid wasting time on storing the data before reading it back. We also get a user-friendly interface, since users are able to utilize our program in the same way as they use any other custom-made function in Matlab.

The main disadvantages of compiling to a MEX file is that debugging and analysis of the program becomes harder. Using a debugger on a normal program is quite easy. If the program is compiled with the debug-flag set, then we can simply let the debugger execute the program and start stepping through the code. For a MEX file, however, we have to start Matlab and attach the debugger to the Matlab application before executing the program from Matlab. Matlab is not compiled for debugging and will usually crash with this approach.

A better way of debugging the MEX file is therefore through writing variables with the `mexPrintf()` function and then look for when the program starts to produce faulty output. This method can be time-consuming when the bug we are looking for is hard to localize.

Another problem is that we do not have the possibility to run the CUDA Visual Profiler that is provided by Nvidia (see Section 6.9). This program analyzes a CUDA program and give reports about the used memory bandwidth, how threads diverge, how memory is coalesced, and other useful benchmarks.

It is therefore nice to create support for both approaches: The read and write to file method for debugging and analysis of the code, and the MEX file for user-friendliness, for speed, and to be able to compare the program with similar Matlab programs.

6.7 Compiling the program

To easily let the program be compiled as both Matlab executable (MEX) file and a normal program, we create a make-file that sets up the environment. The make-file is based on Nvidia's make-file for the CUDA SDK, and accepts a few extra parameters. One parameter is telling if we are creating a MEX file or not. If a MEX file is created, the libraries for Matlab executables are included, and the resulting program is put into the correct folder ready to be used by MRST. The other compile-flags we create is one which enables double-precision support for newer graphic cards, one specifying if we are manipulating files stored by Matlab, and one including the CUBLAS library.

Except from the compile options, creating a MEX file is not very different from a normal C++ program. The main difference is that we use another entry point in the code. Normally a C++ program starts by calling the function `main()`. Matlab is instead starting the `mexFunction()` when a MEX-file is started.

The entry point `mexFunction()` is called like any other function in Matlab. However, to ensure that the function is called correctly, and to make it easier to use, it is wise to create wrapper-functions. In the implementation presented here, we have one MEX-file which is called `mexMaster` and has two functionalities. One is to solve a linear system, the other is to build and solve a linear system from a Cartesian permeability grid. The main purpose of the wrapper-function is to make sure that all the input and output variables are provided before the MEX file called.

The mexFunction is started with four parameters when the following command is executed from Matlab.

```
[x, nrm] = mexMaster(A, b, tolerance, sparseType, maxIterations);
```

The left hand side `[x, nrm]` is parsed to the function in the form of an integer `nlhs` giving the number of left hand side elements, in this case two. Matlab's data-type is called an mxArray, and the data of the left hand side is passed in an array `plhs` of length `nlhs` containing mxArrays. The right hand side is passed in the same manner, with `nrhs` being the number of parameters (in this case five) and `prhs` the array containing them.

In the MEX file, we use functions from Matlab's C++ library to assert that the input has the correct format. The library is well documented in Matlab's help-files for external interfaces. Matlab also offers a C++ library for manipulating MAT files. This is the format Matlab uses to store variables to a file, and we utilize this when the C++ program and the Matlab program run separately.

6.8 Implementing a kernel for CUDA

To describe some of the syntax in CUDA programming language, we present in Listing 6.3 the kernel calculating vector dot-product from the specialization project [42]. The code calculates the sum $s = \sum_{i=0}^{N-1} a_i b_i$, where $a = [a_0, \cdots, a_{N-1}]^T$ and $b = [b_0, \cdots, b_{N-1}]^T$ are the two input vectors.

On line eleven, we do a loop such that each thread is calculating several products and summing them together. This is done both to hide memory latency, and to minimize the number of idle threads during execution. After this, a segmented reduction is performed on the result. To get a resource usage complexity of $O(n)$, the number of loads for each thread should be of

Listing 6.3: The kernel for vector-vector product written for CUDA.

```
1  template <unsigned int blockSize, class ValueType, bool useCache>
2  __global__ void vectorVector_kernel(
3    const ValueType *a, const ValueType *b, const size_t N, ValueType *s
4  )
5  {
6    extern __shared__ ValueType sdata[];
7    int tid = threadIdx.x;
8    int i = (blockIdx.x * blockSize) + threadIdx.x;
9    int gridSize = gridDim.x * blockSize;
10   //load the data from global to shared memory a number of times to hide latency.
11   sdata[tid] = 0.0f;
12   do{
13     sdata[tid] += fetch_x<useCache>(i, a) * fetch_y<useCache>(i, b);
14     i += gridSize;
15   }while(i < N);
16   __syncthreads();
17   //maximum block-size is 512
18   if (blockSize >= 512){
19     if (tid < 256) sdata[tid] += sdata[tid + 256]; __syncthreads();
20   }
21   if (blockSize >= 256){
22     if (tid < 128) sdata[tid] += sdata[tid + 128]; __syncthreads();
23   }
24   if (blockSize >= 128){
25     if (tid <  64) sdata[tid] += sdata[tid +  64]; __syncthreads();
26   }
27   //all operations within a warp is executed simultaniously (SIMD),
28   //and do not require syncronization.
29   if (tid <  32){
30     if (blockSize >= 64){ sdata[tid] += sdata[tid +  32];}
31     if (blockSize >= 32){ sdata[tid] += sdata[tid +  16];}
32     if (blockSize >= 16){ sdata[tid] += sdata[tid +   8];}
33     if (blockSize >=  8){ sdata[tid] += sdata[tid +   4];}
34     if (blockSize >=  4){ sdata[tid] += sdata[tid +   2];}
35     if (blockSize >=  2){ sdata[tid] += sdata[tid +   1];}
36   }
37   //first thread stores the result
38   if (tid == 0) s[blockIdx.x] = sdata[0];
39 }
```

order $\log(n)$ [42]. In order to hide memory we would typically assign little more to each thread.

Most of the syntax that are special for CUDA are written on the first six lines of the code. The prefix `__global__` to the kernel-function tells the compiler that the code should be executed on the GPU, and can be called by the CPU. Shared memory is used by the prefix `extern __shared__` on line three, which says that `sdata` is a pointer to the start of the shared memory for the block. The values `threadIdx` and `blockIdx` hold the x, y, and z-coordinates in the thread-block and in the grid of blocks respectively. The dimensions of the grid is stored in `gridDim`.

Both of the input-vectors are held constant throughout a kernel-execution. We therefore use a template-parameter `useCache` to tell if these should be fetched through the texture-memory. If this is the case, then we have to bind the variables to a texture before the kernel launch. We create in-line functions for binding, fetching and unbinding the textures to increase the readability and since the procedure is different between single and double precision.

The block-size of the kernel is also parsed as a template parameter to

make the segmented reduction as quick as possible. Since `blockSize` is a template parameter, the if-tests of the `blockSize` is performed on compile-time and not when the kernel is executed on the GPU. The segmented reduction exploits the threads best when the block-size is a power of two. For large vectors, an efficient implementation has grid-size larger than one. We then execute another kernel, doing another segmented reduction. When a CG-system is created, we make sure that the grid-size and block-size of vector-vector product are powers of two, as well as ensuring that each thread is performing more than $\log(n)$ loads into the shared memory.

The kernel can be started by writing the following command in a program which is compiled by nvcc.

```
vectorVector_kernel<bs , ValueType, uc> <<<gs, bs, sharedMemory>>> (v1, v2, n, solution)
```

The parameters within the first ¡¿-brackets are the template parameters. Parameters special for CUDA are the ones inside the triple brackets. Here, the block-size is given by `bs`, the grid-size is set by `gs`, and the size of the shared memory needed for each block is given by `sharedMemory`. The normal function parameters are given between the parenthesis as usual. We may also give the dimensions of the grid-size and the block-size in three dimensions, but except for this, all CUDA programs are called in this manner with the triple brackets.

6.9 Using the CUDA profiler

Getting the most out of the graphics card, requires careful handling of memory. How memory is organized and in which order it is read determine an important part of a kernel's running time.

To describe the tools we have for optimizing the running time of our kernels, we look at two different implementations of the CSR format. The first is assigning one thread to each row of the matrix and we call it the *scalar* kernel. The second is distributing the workload further by assigning one warp (32 threads) to each row, and we call it a *vector* kernel. Simplified algorithms for the two are presented in Listing 6.4 and Listing 6.5. Note that the vector implementation uses segmented reduction on line twenty to twenty-two. This is the same procedure that we use on the vector-vector implementation, with the difference that we sum over the whole vector-size instead of only one warp-size.

For our problem, the number of non-zeros are below the warp-size, and the vector implementation will give idle threads. It is therefore less efficient than the scalar implementation. We use this to show how the CUDA profiler is used to pick up such inefficiencies.

Listing 6.4: The CSR scalar approach

```
__global__ void CSRscalar_kernel(
  const int n, const int *row, const int *col, const float *data, const float *x,
  float *y
)
{
  int rowForThisThread = blockDim.x * blockIdx.x + threadIdx.x;
  if (rowForThisThread < n){
    float dot = 0;
    int row_start = row[rowForThisThread];
    int row_end = row[rowForThisThread + 1];
    for ( int i = row_start ; i < row_end ; i++)
      dot += data[i] * x[col[i]];
    y[rowForThisThread] += dot;
  }
}
```

Listing 6.5: The CSR vector method

```
__global__ void CSRvector_kernel(
  const int n, const int *row, const int *col, const float *data, const float *x,
  float *y
)
{
  __shared__ float sdata[];
  int thread_id = blockDim.x * blockIdx.x + threadIdx.x;  // global thread index
  int warp_id = thread_id / 32;                           // global warp index
  int lane  = thread_id & (32 - 1);                       // index within the warp
  // one warp per row
  int rowForThisWarp = warp_id;
  if (rowForThisWarp < n){
    int row_start = row[rowForThisWarp];
    int row_end = row[rowForThisWarp + 1];
    // compute running sum per thread
    sdata[threadIdx.x] = 0;
    for (int i = row_start + lane ; i < row_end ; i += 32)
      sdata[threadIdx.x] += data[i] * x[col[i]];
    // segmented reduction in shared memory
    if (lane < 16) sdata[threadIdx.x] += sdata[threadIdx.x + 16];
    if (lane <  8) sdata[threadIdx.x] += sdata[threadIdx.x +  8];
    if (lane <  4) sdata[threadIdx.x] += sdata[threadIdx.x +  4];
    if (lane <  2) sdata[threadIdx.x] += sdata[threadIdx.x +  2];
    if (lane <  1) sdata[threadIdx.x] += sdata[threadIdx.x +  1];
    // first thread writes the result
    if (lane == 0)
      y[rowForThisWarp] += sdata[threadIdx.x];
  }
}
```

If we profile a CUDA program, we get a status report from the hardware. It tells us the number of uncoalesced reads from memory, how often threads diverge, number of bank conflicts and other informative benchmarks. To enable profiling we need to set a few environment variables in the operation system before a CUDA program is executed. These variables tells where a log will be created, and which counters we are interested in recording. Enabling profiling may slow down the program, and should not be enabled except for when we are profiling a program. An alternative way of profiling is to use the CUDA Visual Profiler provided by Nvidia. This program works only for independent program, and not on MEX-files. It offers additional tools for visualizing, and calculating averages of the results coming from a profile run.

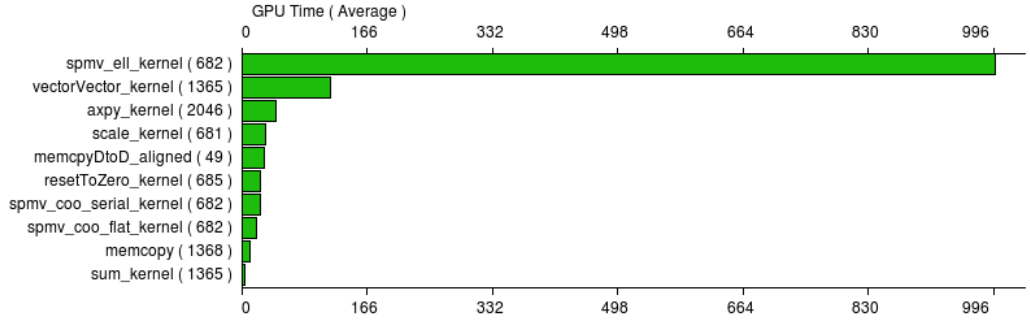
We execute the kernels on a linear system, and note some of the important

Table 6.1: Some counters from the CUDA profiler, found when we solve the linear system coming from the $100 \times 100 \times 11$ cells version of the Johansen formation explained in Section 7.3. We use two implementations of CSR as examples, and compare with the optimized ELL format.

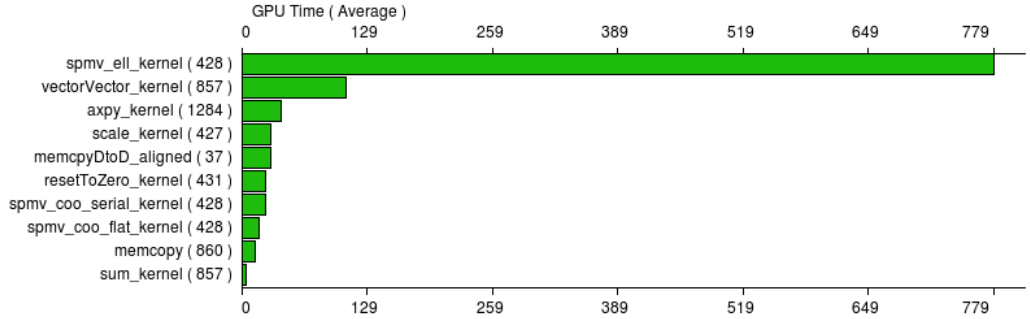
Format	Time	Occupancy	Branches	Divergent branches	Warps serialized
CSR scalar	2018	1	3906	235	0
CSR vector	3448	0.75	103304	22128	56376
ELL	776	1	3680	0	0

profile measures in Table 6.1. The occupancy column represents the ratio between used threads and the available threads. We see that the vector implementation leaves 25% of the treads idle. The same behavior is reflected by the divergent branches, which is 6% for the scalar kernel and 21% for the vector kernel. Divergent branches is when two or more threads of the same warp execute different instructions and have to wait for each other. The warp serialize counter is recording the number of reads from shared memory that cause bank conflicts. This counter does not apply to the scalar kernel, since it is not using shared memory.

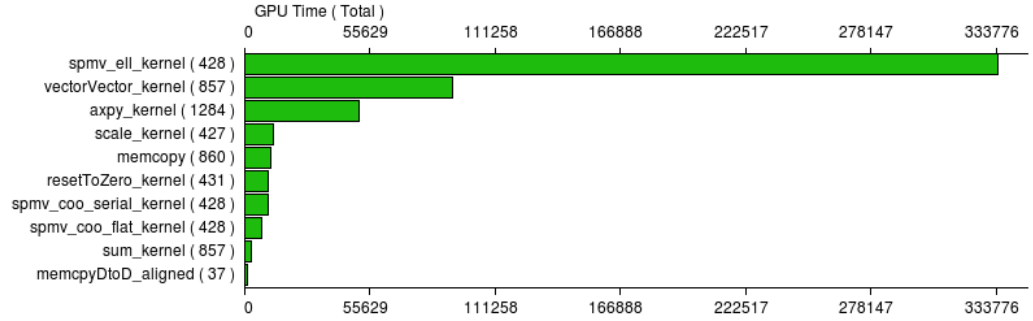
We also modify our kernels to be able to fetch data through the texture memory. This is possible for constant data, so we improve all our kernels with a template parameter specifying if data, that remains unchanged throughout the kernel execution, should be fetched through the texture memory. As an example, we solve a linear system with the HYB format with and without texture fetching. The average running time of the kernels is recorded by CUDA Visual Profiler, and is plotted in Figure 6.8. Using texture fetching decreases the running time of the ELL kernel by around 20% and the other kernels with approximately 10%. The timing result we perform from here on is therefor using texture cache, to speed up the calculations. The graph with the total time used confirms that matrix-vector product, vector-vector product, and the axpy kernel are the functions that contributes the most to the running time.



(a) Average time used without fetching data through texture memory



(b) Average time used when fetching data through texture memory



(c) Total time used by the kernels on a seven minutes test-run

Figure 6.8: Comparison done by CUDA Visual Profiler between the kernels which do not use texture memory (at the top) and the kernels using texture fetching (in the center). The graph at the bottom shows how big portion of the total running time each kernel use when texture fetching is turned on. The time is given in milliseconds.

Chapter 7

Numerical Experiments

Analyzing the correctness of a numerical simulator is usually performed in three steps. First, we validate the discretization and the solver with a well-known example. It should have a simple structure where errors are easy to locate. Secondly, we make sure that the simulator is able to handle more complex models to find its potential and its limitations. After this, a validation of the numerical simulator is performed on a real physical problem. The simulator is correct if it reproduces the same results as we observe in real life.

In Section 7.1, we perform the verification with a test-problem called *the quarter-five spot problem*. Then, the performance and the limitations of the simulator are tested on two example models that we describe in Section 7.2. We call the first example for *the narrow passage*. It has a simple geometry with one fault, creating a narrowing of the permeable area. The second example have two faults and a more irregular geometry. We call it *the fault-crossing*. The fault-crossing example give high condition numbers of the linear system, which makes the model harder to solve by the CG algorithm.

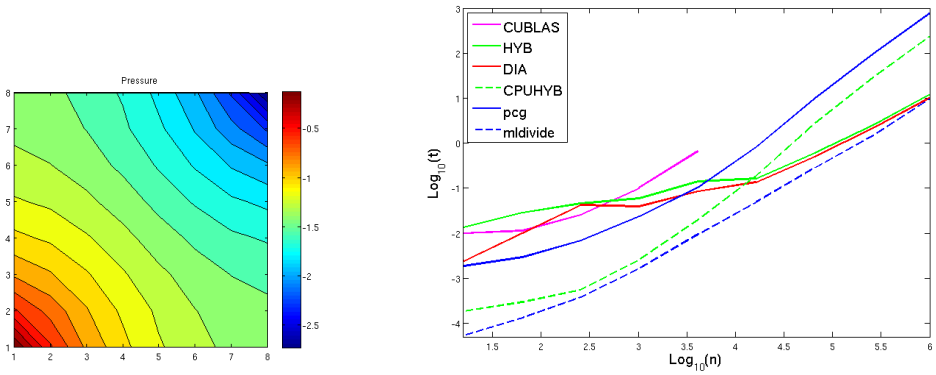
Validation of our simulator is not performed in this thesis, since we do not have output-data from a real physical problem. However, we have a field model of a CO_2 deposit-cite called *the Johansen formation*. This is a large model, which is time-consuming to solve without up-scaling. We introduce it in Section 7.3 and use it to describe what the GPU has to offer in terms of speed on realistic problems. Correctness of the solver is only verified on the linear system by checking that the CG algorithm is producing results within the relative tolerance given. We sum up the results in Section 7.9 to get an overview of the running time the implementations use to solve the models at different grid-sizes.

7.1 Two-dimensional verification model

First, we would like to test for correctness. We do this by considering a test-case in $\mathbb{R}^2$ with homogeneous and isotropic permeability such that $\mathbf{K}$ is the identity matrix everywhere. We place an injection well at the origin and production wells at the positions $(\pm 1, \pm 1)$ making a five-spot pattern. Repeating this pattern to infinity produces the same results as imposing no-flow boundary conditions on the unit box in the first quadrant $\Omega = [0, 1] \times [0, 1]$, and a reference solution here gives a pattern for the global solution. This problem is called the *quarter-five spot* problem, and is a standard test-case for simulators of flow in porous media.

We build a TPFA discretization of this test-case by using a Cartesian grid, which is $\mathbf{K}$-orthogonal since $\mathbf{K}$ is diagonal. The linear system is constructed by $\mathbf{A}$ from equation (4.5) and $\vec{b}$ becomes zero everywhere except at the two well-positions. A contour-plot of the solution to the quarter-five spot problem is in Figure 7.1a.

Using the MEX-file explained in Section 6.4 we can build the system directly from the permeability grid into the DIA format in C++. However, we create a Matlab-script, in addition, to be able to use this verification model for all the other formats we have implemented (see Figure 6.6). This test-case is simple, and therefore well-suited when we want to debug the various formats and compare them with the built-in solvers in Matlab.



(a) quarter-five spot solution

(b) Computation time

Figure 7.1: Running times for a verification model called the quarter-five spot problem. To the left is a surface plot of the pressure solution with a 8×8 grid. This is a nice and simple example-size for debugging. To the right is a log-log-plot of timing data compared with the grid-size.

We run the test-case with different grid-sizes to get a preview of how well the DIA format works for a TPFA discretization. Matlab's preconditioned conjugate gradient method, `pcg()`, is running without a preconditioner as a CG solver for Matlab. It is compared with the CG algorithm in C++ which can run the sparse formats on the CPU or the GPU. Gaussian elimination is represented by Matlab's matrix left divide, `mldivide()`.

We can see indications that the GPU implementation is faster than `pcg` when the problem size gets large, but it is not able to surpass the direct solver. This supports the assumption that Gaussian elimination is quicker than CG on structured two-dimensional grids, where matrices have a small bandwidth. We investigate the performance of the solvers and the sparse formats further in Section 7.6, where we analyze the three dimensional examples.

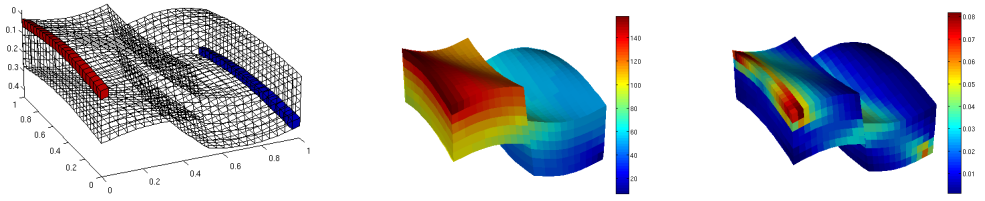
7.2 Models from the Matlab Reservoir Simulation Toolbox

Three-dimensional problems make the advantage of CG more visible. The Matlab Reservoir Simulation Toolkit (MRST) we mentioned in Section 6.5 is a toolbox for reading, processing, and visualizing grids in the eclipse format. It also includes some routines for creating example problems. We use two of these example-models at different resolutions. Increasing the resolution of a grid increases the size of the resulting linear system. Changes in the permeability changes the condition number. This gives us several linear systems that we use to analyze our CG solver.

7.2.1 The narrow passage example

The first model we look at is created from the unit cube in a Cartesian domain, where we give the vertical pillars wave-formed perturbations and create a fault at a cross-section to get the geometry in Figure 7.2. One horizontal well is added on each side of the model. The red one is injecting fluid at a fixed rate of 1 m^2/day at each cell it passes through, the flux out of the blue well is controlled by the bottom pressure, which we set to be fixed at 100 kPa. The values here are not important, they are just chosen to get a model we can use to test the speed of our solvers.

When we add wells cell-wise, like we do in this example, the wells are added by modifying the matrix $\mathbf{A}$. The amount of elements in $\mathbf{A}$ that turns non-zero is five times the number of cells with sources or sinks. These values are inserted into the last column and the last row of $\mathbf{A}$. The ELL matrix-



(a) Geometric structure **(b)** Pressure profile **(c)** Flux intensity

Figure 7.2: Illustration of the narrow passage example. The left picture shows the geometric structure of an $24 \times 24 \times 6$ grid created from the example function `simpleGrdecl()` in MRST. Cells are colored red where fluid is injected. At the blue cells, the pressure is set to be 100 kPa and the out-flow is adjusted accordingly. The calculated pressure profile is shown in the center, and the square root of the flux intensity through the cell walls is illustrated to the right.

format is therefore not well suited for solving this system, and runs quickly out of memory.

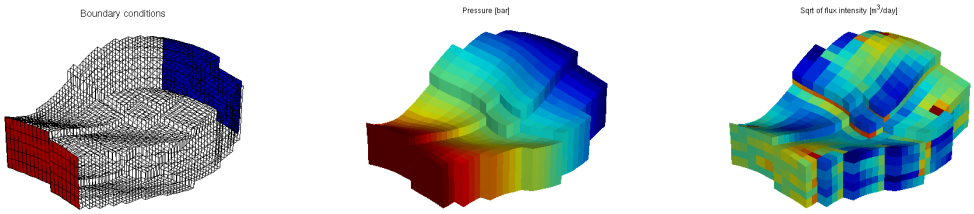
In order to use the ELL format on some of the example models, we also create models without sources and sinks. Instead we use Neumann boundaries to create a constant flow through the model, entering from the west boundary and exiting out the east boundary. Illustrations of the Neumann boundaries are showed in Figure 7.3a and in Figure 7.9a.

Initially, we are interested in creating a simple model which does not change much when we increase grid-size. Having the same permeability in each cell gives this effect, so we choose the permeability to be diagonal with value $\mathbf{K} = \text{diag}(10, 20, 1)$ in milliDarcy for every cell. The pressure profile and flux intensity that we find for this model is illustrated in Figure 7.2.

7.2.2 The fault-crossing example

The next model we consider has a more irregular geometry. It contains two faults that cross each other twice, like we see in Figure 7.3, and we therefore call it the fault-crossing example. The boundary of the model is flat at the east side, but has a jagged oval-shaped boundary in the north and south.

The geometric complexity of the model increase the condition number of the resulting linear system of equations. The condition-numbers are high compared to the narrow passage at the same grid-size. Hence, the linear system from this model is harder to solve with the CG algorithm.



(a) Boundary condition **(b)** Pressure profile **(c)** Flux intensity

Figure 7.3: Illustration of the fault-crossing model. The boundaries with in-flow (red) and out-flow (blue) are colored in the left picture. Pressure profile of the solution is in the center, and the square root of the flux intensity is illustrated to the right.

7.3 The Johansen data set

The Johansen formation is a candidate site for sub-sea CO_2 storage, located offshore the south-west of Norway close to the Troll oil-platform. A number of seismic measurements has been performed of the formation, and the model has been analyzed by several companies.

A strategic research program has been founded by the Climate program at the Research Council of Norway together with the companies Norsk Hydro, Statoil, and Shell. It aims so improve the knowledge of how CO_2 flows in sub-sea deposits to better analyze the success or failure of storage operations. The Johansen formation has been analyzed as part of this research program.

A geological model of the Johansen formation is publicly available in the eclipse format. The permeable part of the Johansen model, where CO_2 can flow, is discretized by a $149 \times 189 \times 16$ grid. This is a quite large data set representing the approximately 75km by 100km formation. To limit the number of active cells in simulation, grids representing a 100×100 section of the cells are also available for download. The section is situated around a suitable position for an injection-well, and is available with three different resolutions in the vertical direction. This gives us four models with realistic geometries at different resolutions, which we can use to challenge the solvers on the GPU. The $100 \times 100 \times 11$ section is illustrated in Figure 6.7 and in Figure 7.4.

We use a simplified simulation model that is motivated by Eigestad et al [9]. The Johansen formation is proposed as a deposit site for the CO_2 waste from two gas power plants, with a total production of 3.3 Mt CO_2 per year. A simulation is carried out where this injection is carried out over a period

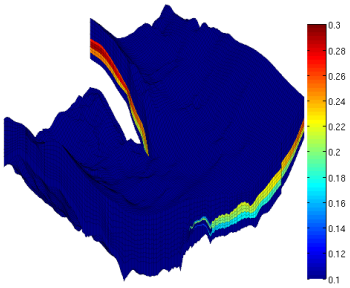


Figure 7.4: Permeability plot of the $100 \times 100 \times 11$ section of the Johansen formation. We see that there are permeable layers surrounded by impermeable rock.

of 110 years, and followed by 440 years after injection-stop to simulate the effects of buoyancy-driven flow. This simulation is carried out in the Eclipse 100 simulator, and we recreate the simulation using MRST with the help of a code provided by Knut-Andreas Lie at SINTEF.

7.4 Analyzing the matrix-structures

To get more insight in how MRST builds the linear system, we analyze the matrices created from our models. How the boundary conditions are included is important to the size as well as the structure of the system. Looking at the sparsity pattern of the matrices gives also an indication on which matrix-format is best suited when the matrix is stored on the GPU.

7.4.1 TPFA discretization

The discretization of a Cartesian grid mentioned in Section 7.1 incorporate boundary conditions into the vector $\vec{b}$ of the linear system. The MRST, however, adds these to the matrix $\mathbf{A}$, and increases the size of the system. If for example we discretize the quarter-five spot example, Then $\mathbf{A}$ gets the the sparsity pattern in Figure 7.5. In this figure, the original system is the upper-left 256×256 corner, where $\mathbf{A}$ consists of seven diagonals. The rest of the matrix comes from the boundary-conditions. Some of the reason why the boundary conditions take up so much space is that the model is created in three dimensions, and we get no-flow boundary conditions on the top and bottom of each cell in the grid. Still, the boundary conditions is increasing the problem size, and therefore the time it takes to solve it.

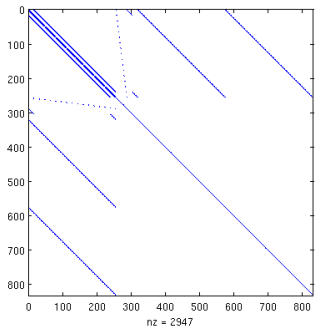


Figure 7.5: The sparsity pattern of a matrix **A** coming from a TPFA discretization by MRST of the quarter-five spot problem.

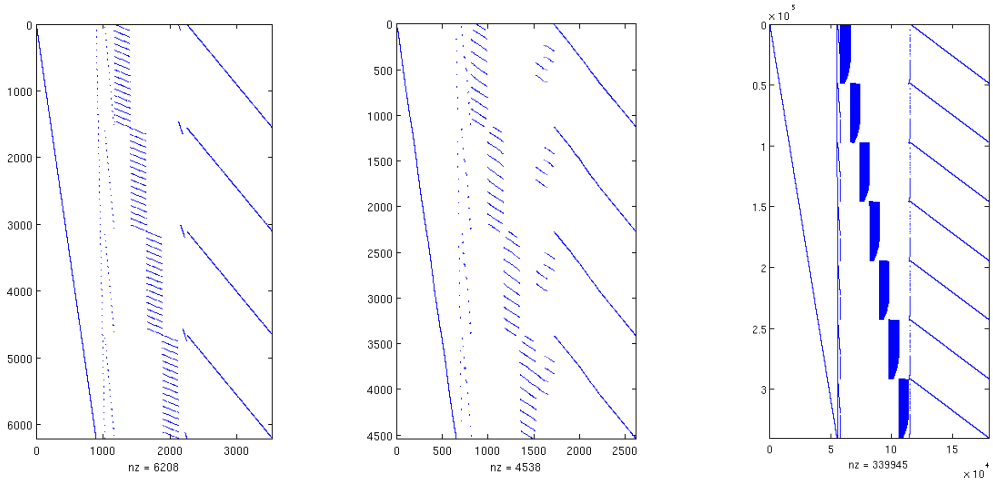
The implementation we use for the quarter-five spot problem stores only this upper-left corner as **A**. It is therefore smaller, and faster to solve. It is also well-suited for the DIA format, while the MRST discretization is less suited due to several diagonals with only one non-zero value.

7.4.2 Mimetic discretization

When we discretize the narrow passage and the fault-crossing with the mimetic discretization, we get a different sparsity pattern than the Cartesian grid we presented in Figure 4.4. Most of the cells in our example models are quadrilateral, and **C** has therefore similar structure to the Cartesian example. The matrix **B** will have the same structure of being block-diagonal with quadratic blocks of the same size as the number of faces of the cell it corresponds to. The difference in pattern is coming mainly from the matrix **D** which holds the connections at the cell-interfaces.

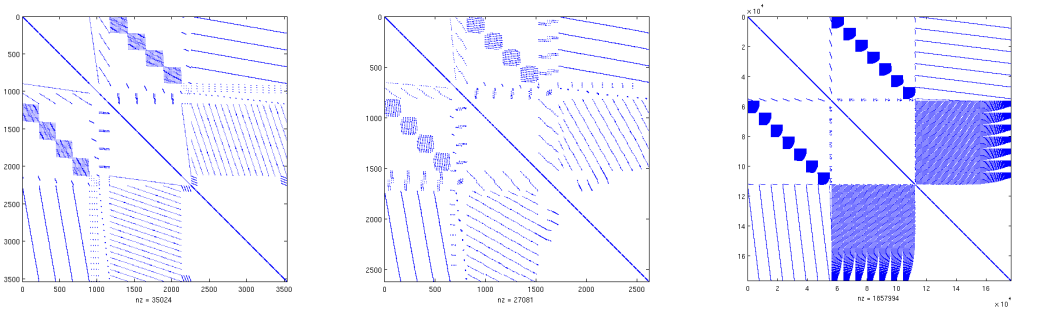
The different patterns of **D** are illustrated in Figure 7.6, and after Schur-decomposition we obtain the matrices in Figure 7.7. Production- or injection-wells are added to the model by treating them in the same way as interfaces between cells. They are therefore included in the matrices before the Schur-decomposition. The number of extra equations in this system is the same as the number of cells with sources or sinks. After the Schur-decomposition, the number of non-zeros per row is at most twice the number of neighboring cells, except for the rows in **A** which depends on the sources and sinks.

Adding boundary conditions in the form of reference-pressure or flow through the boundaries does not change the number of non-zeros per row in **A**. The exception is when we add wells, like we do for the narrow passage example. This creates one row and one column in **A**, with several non-zero



(a) the narrow passage **(b)** the fault-crossing **(c)** the Johansen formation

Figure 7.6: The sparsity pattern of the **D** matrix found before Schur-decomposition, when a mimetic discretization is performed.



(a) the narrow passage **(b)** the fault-crossing **(c)** the Johansen formation

Figure 7.7: The sparsity pattern of **A** in the linear systems after the Schur-complement reduction has been performed.

values. On the problem-sizes we operate with here, this typically increases the maximum number of non-zeros per row from around 20 to around 300. Thus, the efficiency of the ELL format is decreased.

7.5 Assembling the linear systems

Building the systems $\mathbf{A}\vec{x} = \vec{b}$ in MRST is quite fast. Time needed to build the system is proportional to the number of cells in the grid. We double-check this by noting the time used to assemble the linear system from grids with permeabilities belonging to the different-sized models of the Johansen formation. The time used is listed in Table 7.1, and we see that it is increasing close to linearly with the number of cells. At the most, we use 53 seconds to build the mimetic discretization. this is short compared to the time spent on solving the linear system, and moving this operation to the GPU gives little improvements in the running time.

Building the system on the GPU may also have another purpose. There are two ways to create a multi-grid preconditioner for the CG algorithm. One is to use the grid-structure to create different levels of coarser grids. The other is to use the matrix-structure, to create the different levels. When implementing multi-grid using the grid-structure, the coarsening of the grid relies on the discretization. In this case, to avoid too much memory communication between the graphics card and the motherboard RAM, the discretization should be implemented on the GPU. However, the algebraic multi-grid approach can handle more general discretizations, and makes the solver more independent from the rest of the program. Since the solver is designed to handle different discretizations, the algebraic multi-grid is the recommended preconditioner for the CG solver presented herein.

Table 7.1: Time used to assemble the linear system for the different sized models of the Johansen formation.

Number of cells	Mimetic discretization	TPFA discretization
$100 \times 100 \times 11$	11.9s	8.5s
$100 \times 100 \times 16$	20.1s	14.3s
$100 \times 100 \times 21$	28.3s	20.3s
$149 \times 189 \times 16$	53.3s	39.3s

7.6 Speed of the sparse formats

When simulating flow in models of sub-sea rock-formations, the relative tolerance is typically set to 10^{-6}. Picking this number is a trick-of-the-trade, and give reasonable results. We analyze how fast the different sparse formats are by solving the models with this precision.

The narrow passage

Solving the fluid flow for the narrow passage with the different sparse formats gives the time plotted in Figure 7.8a. As expected, we see that the HYB format gives the best overall performance, followed by COO and CSR. The ELL format is unsuitable for this example, because it runs out of memory already on a grid-size of $40 \times 40 \times 10$ cells.

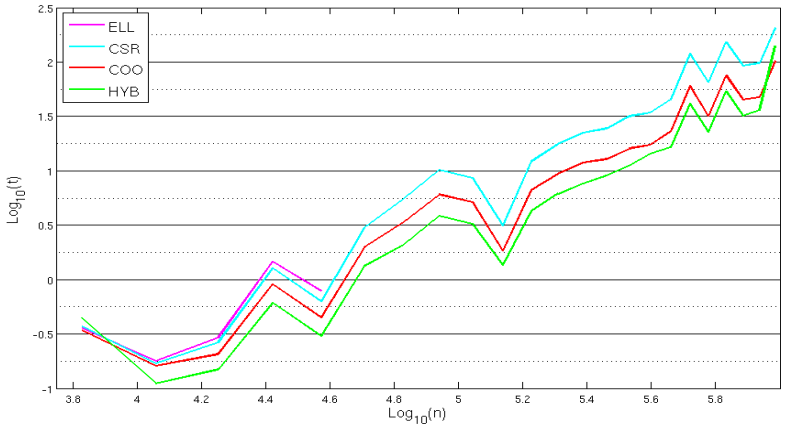
Jumps appear in the log-log plot and we assume that these are due to variations in the condition number $\kappa(\mathbf{A})$ of the linear system. To assert this, we plot in Figure 7.8b estimates for the condition number at some of the small problem-sizes. We see that the condition number follows the same pattern as the jumps in time used.

In Figure 7.8c, we compare the CG solvers with the direct method of Gaussian elimination. The difference between the solvers is not substantial until the matrix $\mathbf{A}$ has around three hundred thousand rows an columns. Storing a full matrix of the same size is then demanding many times the amount of available memory, which in this test is 4 GB. The program is therefore using swap-memory on the hard-drive to store the Schur-decomposition. Most of the running time is therefore used on memory-communication, and `mldivide` is unable to solve the systems when they become too large.

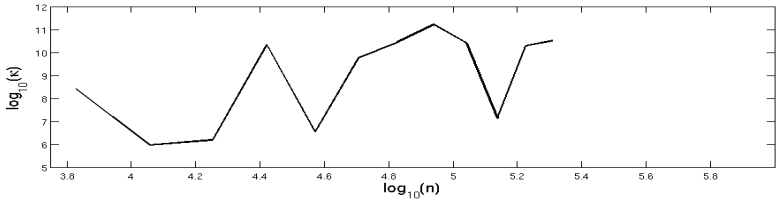
We can also see that the Gaussian elimination is following a straight curve before the memory-communication starts to slow it down. The steepness of this curve reflects the asymptotic running time of $\mathrm{O}(n^3)$. We may also see that the CG algorithms follows a less steep tendency, and would out-perform Gaussian elimination on large problems even if we had an infinite amount of low-latency memory available.

We have two matrix-formats implemented in C++ for the CPU. These are the CSR format and the HYB format. Here, we see that the CSR format runs better on the CPU than the HYB format, and this is also the case for the other models. We therefore use only the fastest, which is CSR, to represent the CPU on the other tests we perform.

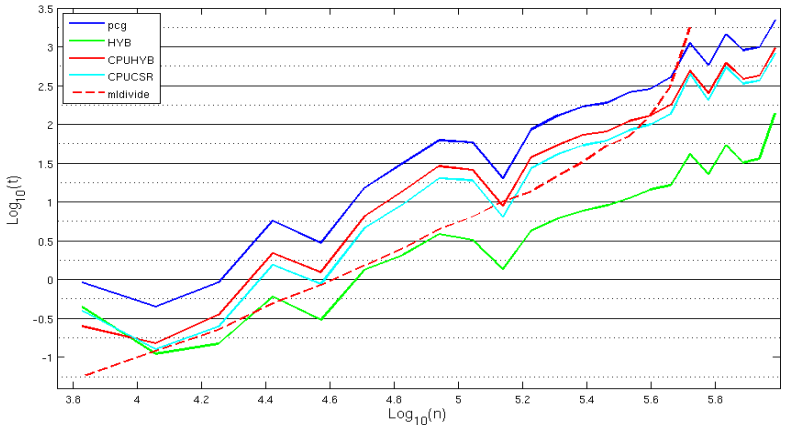
The speedup we get compared to Matlab's `pcg` can be explained in two steps. First, the speedup between `pcg` and the CSR format on the CPU, which gives the speedup we gain from using C++ instead of Matlab. Then,



(a) Runtime of the CG solvers on the GPU

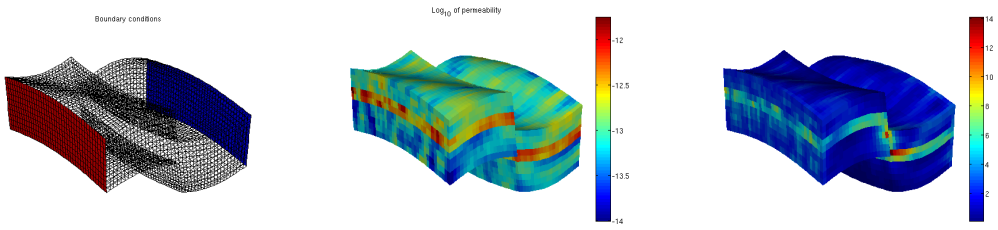


(b) Condition numbers



(c) Comparing HYB with the solvers for the CPU

Figure 7.8: Time used solving the narrow passage example illustrated in Figure 7.2 on different grid-sizes. At the top, we plot the different sparse formats, and see that the HYB format performs best. At the bottom, we compare it with the implementations for the CPU and the Matlab functions `mldivide` and `pcg`. The condition numbers at some of the problem-sizes is shown in between.



(a) Neumann boundary **(b)** Permeability **(c)** Flux intensity

Figure 7.9: The narrow passage example with Neumann boundaries and three layers of a log-normal distributed permeability. The boundary condition is illustrated to the left, fluid enters at the red boundary and exits at the blue. The permeability values are illustrated in the center, and the calculated flux intensity to the right. The permeability has here three layers representing sediment layers of a rock-formation, and we see that a fault has caused them to no longer be aligned.

the extra increase between the CPU and the GPU explains the speedup due to hardware differences.

Log-normal distributed permeability layers

In Section 3.1 we presented four examples of problems which are solved by simulation of flow in porous media. Three of these are problems where fluid propagates through layers of soil, earth, or rock. The corner-point-grid structure aim to reflect the structure of such formations, where each layer in the vertical direction typically represent a sediment layer where the materials have similar properties. To get a model that mimics this, we pick permeability values from a log-normal probability-distribution and let the mean of the distribution vary between the layers. In this way, we pick values from the same distribution for cells having the same vertical cell-coordinate in the corner-point grid. An example of such a permeability having three distinct layers is illustrated in Figure 7.9b.

Variations in the permeability of this kind increases the condition number of the linear system slightly. Time needed to solve the system is therefore increased, as we can see from Figure 7.10. The grid-size needs to be larger than before in order for the CG algorithm to become faster that Gaussian elimination.

Here, we have used Neumann boundary conditions, like shown in Figure 7.9a, instead of adding wells to the model. The ELL format is therefore solving this system well. The HYB format will in this case not use it's COO part at all, and has therefore approximately the same running time as the

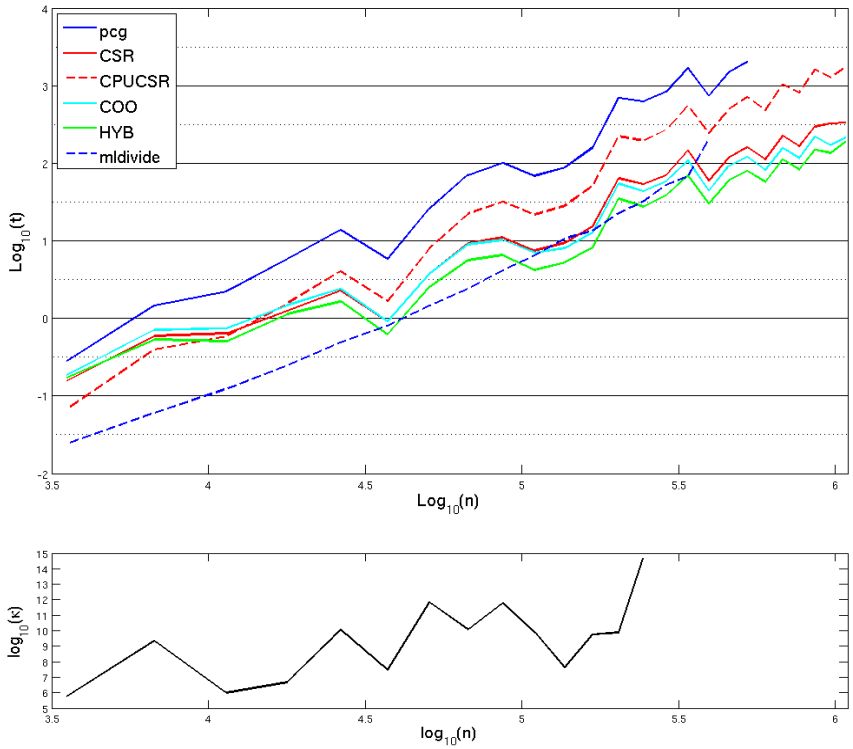


Figure 7.10: Running time for the narrow passage example with layered log-normal distributed permeability values. The condition numbers are plotted below for comparison.

ELL format. Running time of the COO format and the CSR format are also approximately the same. We therefore only plow the ELL and the CSR format, since they represent the fastest and slowest of the formats running on the GPU.

The fault-crossing

Limitations of our linear solver becomes most visible when we solve the fault-crossing model, even if we use a homogeneous permeability fore each cell. The geometry of the model is creating irregularity in the boundary conditions and in the linear system. This cause higher variation in the condition number, and the problem-size does not have to be large before the solver is getting slow. Figure 7.11 illustrates these running times. We even experience that when we solve with defection-correction, the solver is unable to converge to the relative tolerance of 10^{-6}. This happens already on a problem-size of

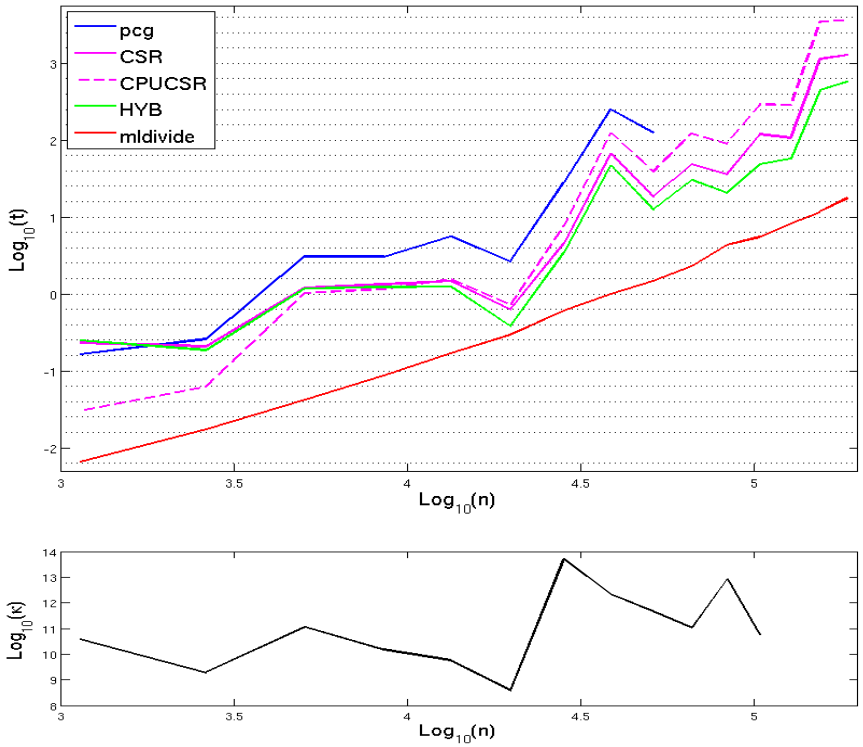
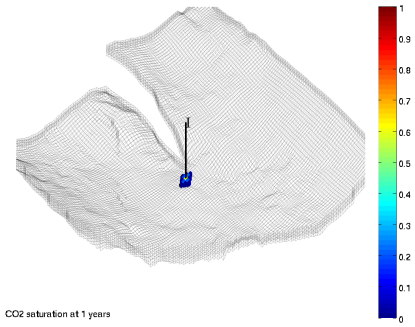


Figure 7.11: Running time for the fault-crossing example, with plot of the condition numbers.

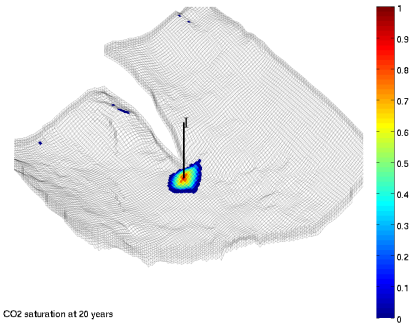
87500 cells, and we stop the test-runs at this point.

7.7 Solving the Johansen formation

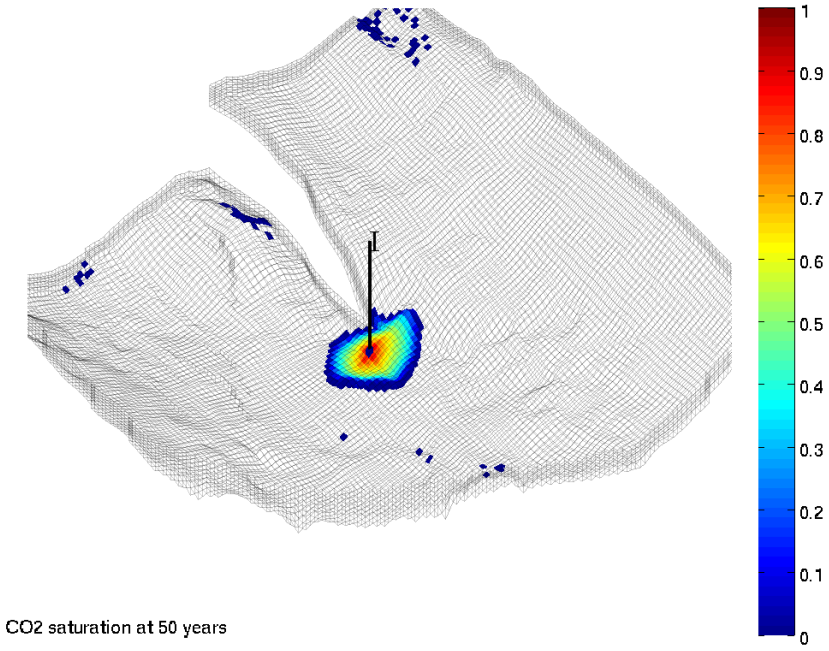
Finding the fluid flow in the Johansen formation gives more insight into how the solver performs on a realistic model. We start by investigating the smallest model of the Johansen formation, having a height of eleven cells. Reference pressure is used as boundary condition around the permeable grid, and we look at the saturation of CO_2 in the formation, assuming that it initially is filled with water. Flow of CO_2 is calculated by first finding the fluxes through the cell-walls by solving the linear system resulting from a mimetic discretization. A new saturation representing one year later is calculated based on these fluxes, and the pressure at the cells are updated to be ready for the next iterate. Figure 7.12 shows a few of the calculated saturations, when the linear system is solved with the HYB format on the



(a) CO_2 saturation after 1 year



(b) CO_2 saturation after 20 years



(c) CO_2 saturation after 50 years

Figure 7.12: Saturation distribution in the Johanson formation for a injection well injecting 10 000 m^2 CO_2/day. The flux is calculated with CG algorithm with a precision of 10^{-6}, and we can see small, but visible errors appearing. These errors comes when the flux is calculated, and accumulates in the simulator at succeeding timesteps.

GPU with a relative tolerance of 10^{-6} at each time step.

We notice that there are some small errors that become visible after a few time-steps. The initial guess of solution is zero flux throughout the whole grid, which is the correct value for the cells outside the area with CO_2. However, the CG algorithm may alter the values which already are correct. This illustrates the side-effects of a rougher that we mentioned in Section 5.2, and we also see that error accumulates when the solver is used for succeeding time steps.

Even though the error is noticeable in the plot, these errors are small. We can decrease them further by demanding a higher relative tolerance of the CG algorithm, or by running a few iterates with a smoother that damp down errors of high frequency.

Decreasing the relative tolerance

Demanding a higher relative accuracy increases the running time of the solver. The straight-forward CG algorithm uses the times plotted in Table 7.2 to achieve the different relative tolerances without restart. This CG algorithm is using close to the same number of iterations for all the sparse matrix formats. Matlab's `pcg` is also using close to the same number of iterations, but we see that the speed is different. The GPU implementation achieves here a speedup of between seven and fourteen compared to the CPU implementation, and is around twenty times as fast as `pcg`.

Such a CG algorithm without restart can only reach a certain tolerance.

Table 7.2: Number of iterations k and time t used to achieve different levels of relative tolerance on the $100 \times 100 \times 11$ cells large Johansen model. Here, we use the straight-forward implementation of the CG algorithm. The speedup column gives the CPU implementation compared with the GPU implementation. We show the speedup compared to Matlab's `pcg` in parenthesis.

Relative	`pcg()` on CPU		CSR on CPU		HYB on GPU		
tolerance	k	t	k	t	k	t	Speedup
10^{-1}	235	8.51s	277	2.84s	280	1.27s	2.2x (6.7x)
10^{-2}	2414	70.9s	2867	28.2s	2986	3.91s	7.2x (18x)
10^{-3}	9906	291s	10164	101s	10587	13.6s	7.4x (21x)
10^{-4}	46800	1390s	49119	630s	46794	62.7s	10x (22x)
10^{-5}	95324	2830s	99548	934s	96012	128s	7.3x (22x)
10^{-6}	-	-	143934	1340s	137518	184s	7.3x
10^{-7}	-	-	349573	3240s	177036	237s	14x

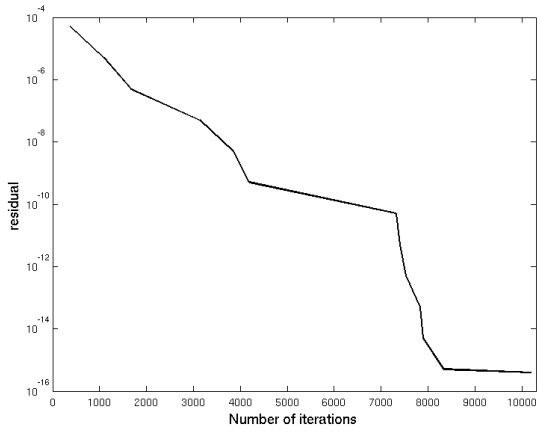


Figure 7.13: Iterations used to achieve the different relative tolerances. It is measured on a 64000 cells large version of the narrow passage example, with mimetic discretization.

When the relative tolerance is set to 10^{-8}, we iterate as many times as the number of rows in $\mathbf{A}$. In order to achieve higher precision, and faster convergence, we have to use defection-correction with restart. However, standard relative tolerance for these type of simulations is 10^{-6}, so we manage to stay within this accuracy.

An example of the number of iterations used to achieve different levels of accuracy is shown in Figure 7.13. Here, we plot the residual norm compared to the number of iterations when we solve a medium sized model of the narrow passage. In such an example we achieve a residual which is close to the precision of double floating points, which is 10^{-16}. The algorithm reports on the relative tolerance 10^{-13} that trying to correct the defect further leads to worse results. It has at this point achieved a residual of $3.4 \cdot 10^{-16}$.

Solving the full model of the Johansen formation in the same manner, gives the iterations listed in Table 7.3. The algorithm restarts once to achieve the relative tolerance of 10^{-6}. When the algorithm restarts, we have iterated through as many iterations as there are rows in $\mathbf{A}$. If we did not have round-off errors, we would have the exact solution at this point. Reducing the residual further after this is demanding substantially more iterates per level of precision achieved than before the restart. We see this behavior for all the linear systems solved with restarts. Most probably, the round-off errors are becoming more dominant after this point.

Table 7.3: Time used to solve the full Johansen formation model of grid-dimension $149 \times 189 \times 16$. The CPU performance is calculated from the CSR implementation in C++, and we use the HYB format for the timing results on the GPU. The speedup is here the ratio between the CPU and the GPU.

Tolerance	CPU	GPU	Speedup
10^{-1}	71.0s	5.67s	13x
10^{-2}	941s	85.7s	11x
10^{-3}	3430s	354s	10x
10^{-4}	7960s	715s	11x
10^{-5}	18800s	1720s	11x
10^{-6}	-	5620s	-

7.8 Using the mixed precision solver

The main idea of the mixed precision (MP) solver is to use single precision for the inner loop of the algorithm is Section 6.2.2 to gain speed. This modifications makes the algorithm loose accuracy and it has to perform restarts more frequent than its double-precision counter-part. However, it performs well since the gained speed often surpass the time lost due to the extra iterations that are needed.

Round-off errors becomes a problem in the MP solver even when we normalize the residual vector. The inner loop of the algorithm is stopped by a relative tolerance. This stopping criterion will make the loop iterate through as many iterates as there are rows in the matrix when the errors dominate. An improvement to the MP algorithm could therefore be to have control-points after a fixed number of iterations. At these control-points we can check that round-off errors are not too large and perform a restart if they are. Fine-tuning of the number of iterations between each such check-point may then improve the speed further, but this is not performed in this thesis. We have only adjusted the relative tolerance of the inner loop. In single precision, the inner tolerance of 10^{-4} gave good results.

The MP solver used herein is having trouble on the more complex models. It is slower than the double-precision on the fault-crossing examples, and is unable to solve the full version of the Johansen model because the iterations stagnate. This shows that the MP solver is more limited than than the defection-correction. A suggestion for further work is to implement a solver using single precision in the inner loop as long as it is possible, and then increase the precision of the inner loop to double. Still, the MP solver

performs quite well without improvements, at least on simple models. The next section summarize the running times and include a MP solver using the HYB format.

7.9 Summary of the results

To get an overview on the speedup the GPU has compared to the CPU, we list our test-models and their running time for the different sparse formats in Table 7.4. We present a speedup multiplier for the HYB format on the GPU compared with the solvers for the CPU. The CPU-solvers are `pcg` representing Matlab, and CSR representing C++. We see that the HYB format on the GPU is between six and eleven times faster than the CPU implementation. Compared to Matlab it achieves a speedup multiplier close to twenty.

The mixed precision (MP) solver is faster on most of the models where it can be used. It achieves speedups of around eleven times the CPU-speed on the large models. However, the MP solver is unable to solve the full Johansen model. Therefore, the defection-correction version of the HYB format is used to represent the GPU. This model is more robust, solving all the models relatively fast.

Table 7.4: Summary of the different models and their running time with relative tolerance of 10^{-6}. Here, NP and NPP represent the narrow passage examples with homogeneous and log-normal permeability respectively. The fault-crossing is abbreviated as FC, and J stands for the Johansen formation. The speedup-multipliers give the fraction between the running time of `pcg` or the CSR format on the CPU compared with the HYB format on the GPU.

	NP_{small}	NPP_{small}	FC	NP_{big}	NPP_{big}	J_{small}	J_{big}	J_{full}
n	243993	243992	103938	965328	965327	177036	418924	757146
κ	10^6	10^{14}	10^{11}	-	-	10^{17}	-	-
CPU:								
`pcg`	169s	348s	952s	2420s	6440s	4300s	-	-
CSR	54.0s	112s	291s	820s	2490s	2300s	8540s	59100s
GPU:								
CSR	24.1s	31.5s	120s	216s	515s	377s	1930s	-
COO	11.9s	24.2s	94.0s	105s	324s	319s	1450s	-
ELL	-	16.5s	69.1s	-	292s	227s	1610s	-
MPHYB	6.83s	13.6s	68.1s	97.3s	219s	157s	779s	-
HYB	8.83s	16.3s	48.7s	102s	277s	196s	905s	5620s
Speed-multiplier:								
Matlab	19x	21x	20x	24x	23x	22x	-	-
C++	6.1x	6.9x	6.0x	8.0x	9.0x	12x	9.4x	11x

Chapter 8

Conclusions

In this thesis, we have created a linear algebra solver using the conjugate gradient method. It was implemented for graphic processing units (GPUs) that support the compute unified device architecture (CUDA) designed by Nvidia. Todays graphics cards consist of parallel processors capable of calculating close to a trillion floating point operations per second (Tflops).

Programming for these graphics cards is becoming easier. The programming language of CUDA is similar to C language, and there are available tools for debugging and analyzing how the program performs on the specific hardware. General compilers for parallel processors are under development, but knowing the layout of the hardware is crucial when optimizing a program for speed. Transferring memory to and from the graphics card has a high latency, and should be avoided as much as possible. Data is transferred faster when special access patterns are followed, and when the workload can be divided among many parallel threads.

Trying to follow these guidelines, we created a Matlab executable (MEX) file, solving a linear systems on the GPU. Numerical experiments show that it is around ten times faster than the corresponding implementation for the central processing unit (CPU). Compared with Matlab's conjugate gradient solver, which does not use a preconditioner, it is between twenty and thirty times faster on large systems. The comparison is carried out on a single core of an Intel Core Quad 6600 running on 3,53 GHz and the GeForce 260 graphics card running at 650 MHz.

8.1 Conclusions in more detail

The solver is used to simulate flow in a porous medium. We created such a simulator in three steps: Discretization, solving the discretization, and optimize the solver for the GPU.

Discretization

Discretizing physical problems into linear systems can be performed by different methods. As examples, we considered the two-point flux-approximation (TPFA) scheme, the O-method, and finite element discretizations. The mimetic finite element scheme is a finite element discretization, which is able to handle grids with very complex geometries. The Matlab reservoir simulation toolkit (MRST), which is created by SINTEF, can be used to discretize models into linear systems by the mimetic discretization. It can be used to simulate CO_2-propagation in a sub-sea formation.

Linear algebra solver

The CG method is an iterative algorithm that performs well on three-dimensional problems. It has low storage requirements, and is well suited for parallel implementation. The CG algorithm should be used with a preconditioner, but the CG algorithm in this thesis does not.

High condition numbers of the linear system increases the number of iterations used by CG, and therefore the running time. The condition number is mainly dependent on the geometry and permeability of the model, but also on the problem size, and the discretization technique.

Restarting the algorithm and solving for defection-correction increase the precision we were able to obtain. We also gained speed, since round-off errors became less dominant. Additional speed was also achieved when the problems could be solved in a mixed precision between single and double. However, the solver used here was not able to use mixed precision on all the models considered in this thesis.

Small and high-frequent errors appeared because of the rougher-behavior of the CG method. A smoother may be applied to dampen these errors, for example, as a preconditioner.

Optimizing for the GPU

The most important calculations of the CG algorithm are the axpy calculation, vector-vector product and matrix-vector product. The matrix-vector

product was improved by choosing appropriate matrix formats. The compressed sparse row (CSR) format was the best suited format for the CPU. A hybrid (HYB) matrix-format between the ELLPACK (ELL) format and the sparse coordinates (COO) format gave the best best overall performance on the GPU.

The essence of optimizing a format for the GPU is to make sure that data that are read consecutively are stored in that same order in memory. Further run-time improvements are achieved by avoiding bank conflicts or idle threads, and use texture memory when it is possible.

Creating a link between Matlab and C++ is done by creating a MEX file. The technique is well documented, but such a program is not easy to debug. It is therefore smart to also create a normal program which reads from a Matlab (MAT) save-file for debugging purposes.

8.2 Further work

The mixed precision solver gave good results, but the implementation was not able to handle complex models, were the round-off errors became too large. Fine-tuning of this algorithm may fix this. One idea that was not tested is to calculate the true residual at some checkpoints in the inner iteration. In this way, we may register when the round-off errors are becoming dominant.

Creating visualization for C++ using for example OpenGL would make the program less dependent on Matlab. A Matlab license is not free, and it would be nice to have a program which is more independent. For this cause, the discretization process should be written in C++ and use the GPU to speed up the linear algebra.

Adding extra boundary conditions or sources and sinks to a model created in MRST may increase the size of the linear system. The system would be solved faster if these additions were included into the right hand side of the linear system $\mathbf{A}\vec{x} = \vec{b}$ instead.

A preconditioner should be implemented for the GPU. The algebraic multi-grid would be a nice choice for this job, using a smoother to dampen error-components at different resolutions. Having such a preconditioner would decrease the number of iterations, improve the accuracy, and thus increase the speed further.

Bibliography

[1] I. Aavatsmark. An introduction to multipoint flux approximations for quadrilateral grids. *Computational Geosciences*, 6(3):405–432, 2002.

[2] Advanced Micro Devices Inc. GPU technology for accelerated computing. http://ati.amd.com/technology/streamcomputing/index.html, 2008.

[3] K. Asanovic, R. Bodik, et al. The Landscape of Parallel Computing Research: A View from Berkeley. *Electrical Engineering and Computer Sciences*, 18(2006-183):19, 2006.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Nvidia technical report NVR-2008-004, Dec. 2008.

[5] D. Blythe. The Direct3D 10 system. In *ACM SIGGRAPH 2006 Papers*, pages 724–734.

[6] F. Brezzi, K. Lipnikov, and M. Shashkov. Convergence of mimetic finite difference method for diffusion problems on polyhedral meshes. *Convergence*, 43(5):1872–1896.

[7] F. Brezzi, K. Lipnikov, and V. Simoncini. A family of mimetic finite difference methods on polygonal and polyhedral meshes. *Mathematical Models and Methods in Applied Sciences*, 15(10):1533–1552, 2005.

[8] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002.

[9] G. T. Eigestad, H. K. Dahle, B. Hellevang, F. Riis, W. T. Johansen, and E. Øian. Geological modeling and simulation of CO2 injection in the Johansen formation. draft, 2009.

[10] W. Feng and K. Cameron. The Green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, 2007.

[11] R. Fernando and M. Kilgard. *The Cg tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Longman Publishing Co., 2003.

[12] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, 2007.

[13] R. Grimes, D. Kincaid, and D. Young. ITPACK 2.0 User's Guide. *Center for Numerical Analysis, University of Texas at Austin*, 1979.

[14] M. Harris. Optimizing CUDA. In *Supercomputing*, 2007.

[15] M. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware*, pages 109–118. Eurographics Association, 2002.

[16] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3, 2007.

[17] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[18] J. Hensley. AMD CTM overview. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*.

[19] A. Inc. Brook+ SC07 BOF session. In *Supercomputing 2007 Conference*, 2007.

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

[21] H. Kalish and J. Isaac. The AMD-K6 3D Processor. *Abacus Software*, 1998.

[22] K. Kettler. Technology trends in computer architecture and their impact on power subsystems. *Applied Power Electronics Conference and Exposition*, pages 7–10 Vol. 1, March 2005.

[23] J. Larus and J. Larus. Spending Moores Dividend. Technical report, Microsoft Research, 2008.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[25] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907.

[26] A. Munshi. OpenCL - Parallel Computing on the GPU and CPU. In *SIGGRAPH 2008*.

[27] Nvidia Corporation. 200 GPU architectural overview, second-generation unified GPU architecture for visual computing. Technical report, 2008.

[28] Nvidia Corporation. CUBLAS Library version 2.0. 2008.

[29] Nvidia Corporation. Nvidia CUDA Programming Guide version 2.2. *NVIDIA, October*, 2009.

[30] Nvidia Corporation. OpenCL for Nvidia. http://www.nvidia.com/object/cuda_opencl.html, 2009.

[31] J. Owens, D. Luebke, et al. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. Blackwell Publishing Ltd, 2007.

[32] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Commun. ACM*, 40(1):24–38, 1997.

[33] P. Raviart and J. Thomas. A mixed finite element method for 2nd order elliptic problems. *Lecture notes in mathematics*, 606:292–315, 1977.

[34] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, 2003.

[35] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 2.0). *Jon Leech and Pat Brown*, 2004.

[36] L. Seiler, D. Carmean, et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[37] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM.

[38] J. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. *Computer Science Tech. Report*, pages 94–125.

[39] A. Shilov, A. Stepin, and Y. Lyssenko. The grand clash for watts: power consumption of modern graphics cards. http://www.xbitlabs.com/articles/video/display/gpu-consumption2006.html.

[40] P. Shirley and R. Morley. *Realistic ray tracing*. AK Peters, Ltd., 2003.

[41] J. Sun and P. Monk. An adaptive algebraic multigrid algorithm for micromagnetism. *Magnetics, IEEE Transactions on*, 42(6):1643–1647, 2006.

[42] A. Torp. Reservoir simulation on a gpu. Specialization project at Norwegian University of science and engineering, 2008.

[43] D. Triolet. Product review: The Nvidia GeForce GTX 280 and 260, 2008.

[44] VG charts. Americas sales for the week ending december 27, 2008. http://news.vgchartz.com/news.php?id=2730.