UNIVERSITY OF AGDER

GPU-accelerated simulation of flow through a porous medium

by

Asbjørn Bydal

Thesis submitted in Partial Fulfillment of the Requirements for the Degree Master of Technology in Information and Communication Technology

Faculty of Engineering and Science
University of Agder

Grimstad
May 25, 2009

Abstract

In this thesis we explore the area of general purpose computing on GPU, by looking at how the GPU can be utilized to accelerate transport in reservoir simulations. We implement a general finite volume scheme and a high resolution scheme with bilinear reconstruction. We look into how the result is affected by using single precision instead of double as the GPUs today are best suited for single precision. We propose a model for a corner-point grid solver for GPU. Last we look at a heterogeneous model where we run a pressure solver in parallel on the CPU.

Preface

This master thesis is submitted in partial fulfillment of the requirements for the degree Master of Science in Information and Communication Technology at the University of Agder, Faculty of Engineering and Science. The work was carried out under the supervision of Trond Runar Hagen and Knut-Andreas Lie at SINTEF ICT, department of applied mathematics and co-supervisor Ole-Christoffer Granmo at the University of Agder, Norway.

First I would like to thank my supervisors at SINTEF ICT, they have provided me with strong guidance and advice through the entire process. I would also like to thank Halvor Møll Nilsen, Jostein Natvig, and everybody at SINTEF who always helped me whenever I asked. Second I would like to thank Ole-Christoffer who supported me and guided me in my decision to write my thesis at SINTEF. Last but not least I would like to thank my family and friends who have supported me in my decision to move to Oslo to write my Master thesis.

Oslo, May 2009.
Asbjørn Bydal

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In this work we study the use of GPUs to accelerate simulation of flow through porous medium. This kind of simulation is most extensively used in reservoir simulations to predict and estimate production. We look into how the flow transport simulation can be adapted to a GPU. We start with a general finite volume scheme and then move over to a more advanced high resolution scheme. Further we demonstrate a heterogeneous computing model where the CPU computes pressure in parallel with the GPUs transport computations. Last we propose a GPU implementation for the corner-point grid, which is widely used for geologic modelling of rock formations. We do a performance comparison between the GPU implementations and CPU implementations of equal design. Last, we look into how the GPUs lack of double precision can affect the final results.

1.2 Motivation

In this section we will establish our argument to why GPU computing is important, and how this research fits into a larger area of high performance computing. The arguments for GPU computing presented in this section can be found in a state of the art report published by Brodtkorb et. al[5].

1.2.1 Today's challenges of high performance computing

For the last twenty to thirty years the main contributing factor in computational speed has been the continued increase in CPU clock frequencies. This has been possible due to the ever shrinking transistor. However, today we have reached limiting factors that can not be easily circumvented. Power densities on the chip are becoming too high, and the bandwidth between the chip and main memory is not able to keep up with the operational rate of CPUs.

The power wall

From the year 1983 to 2003 the typical CPU frequency have increased from a mere 5MHz to 3GHz [19]. Most of the microprocessor performance increase has come from increases in frequency. However, from the beginning of the 21st century, increases in heat and power densities started to limit the frequency speed for silicon chips, and today we are at a standstill around 3 to 4 GHz.

With increased transistor densities and increased frequencies, the power densities on today's chip are well above that of an ordinary kitchen stove. To continue the frequency climb, new materials or better cooling techniques are needed. Both these options are currently not feasible. Liquid nitrogen cooling is expensive and impractical. Carbon nano tubes will probably replace the silicon in the future, but this paradigm shift is still at least three years away and will be costly before good

production techniques are established.

The benchmark for a computing system has for the past thirty years been flops, or floating-point operations per second. However, flops per watt have increasingly started to become a more valued measure. The power consumption of high performance computers are today often a bigger cost than the actual hardware. And much of that power is transferred into heat which again requires more power to remove. Processing units which can operate well below their frequency limit and still perform well in terms of flops are therefore becoming more important.

The memory bandwidth limit

Memory bandwidth between processor and main memory have long been known by scientists and engineers to be a potential bottleneck in a computing system. In the *Von Neumann architecture*, main memory is separated from the processor, and thus resulting in a fetch-execute cycle. With the CPUs steadily increase in number of operations executed per second, the speed at which data needs to be delivered has increased proportionally. However, this has not happened. The number of floating-point operations per memory read has therefore steadily increased over the years, and today it is several orders of magnitude faster to perform an arithmetic operation compared to a memory read or write. This effect is called *red shift* and it leads to processors not able to utilize their full arithmetic potential.

1.2.2 Parallel computing

With all the roadblocks that are now surfacing for the development of the CPU, engineers are looking for new directions to continue the climb in computing power. One of the methodologies that are being adopted is parallel computing. The idea behind parallel computing is to split large tasks up into smaller tasks and let multicore processors process these small tasks in parallel. However, even though con-

sumer CPUs are starting to embrace this methodology, they are still easily outperformed by super parallel processors made up of hundreds of simpler floating-point arithmetic cores.

Tasks that are highly suited for parallelization can often perform many times better on a processor designed with a parallel computing model in mind. There are today several highly capable parallel processors. The most well known are the Cell BE and commercial GPUs. The GPUs are the most recent addition for scientific computing with their recent launch of general purpose computing APIs.

With the introduction of parallel accelerator processors working together with the CPU, came the term *heterogeneous computing*. This term refers to a computing architecture where processing units of different architectures are combined and tasks are assigned according to which processor is the most capable. In this sense the CPU/GPU computing architecture is a heterogeneous computing system which is very capable in most computing tasks.

GPU vs. CPU

The GPU manufacturers today do not face the same problems as the CPU manufacturers do. The core clock frequencies of the latest GPUs only barely pass the 1 Ghz frequency range. This is because an increase in frequency on the GPU will result in a significant increase in power consumption. In addition the performance increase would be minimal due to the bandwidth and memory frequency limitations. However, the GPU still has a huge potential for increased performance since its performance mainly lies in its super parallel architecture, which is easy to expand. Bandwidth is also easier to increase because the memory lies closer to the cores and an increase in performance is added by adding more cores, not more speed. This means that the memory bandwidth does not need to increase in speed. Each core instead gets a separate channel to global memory.

From the two graphs in Figure 1.1, we see how the computational power and

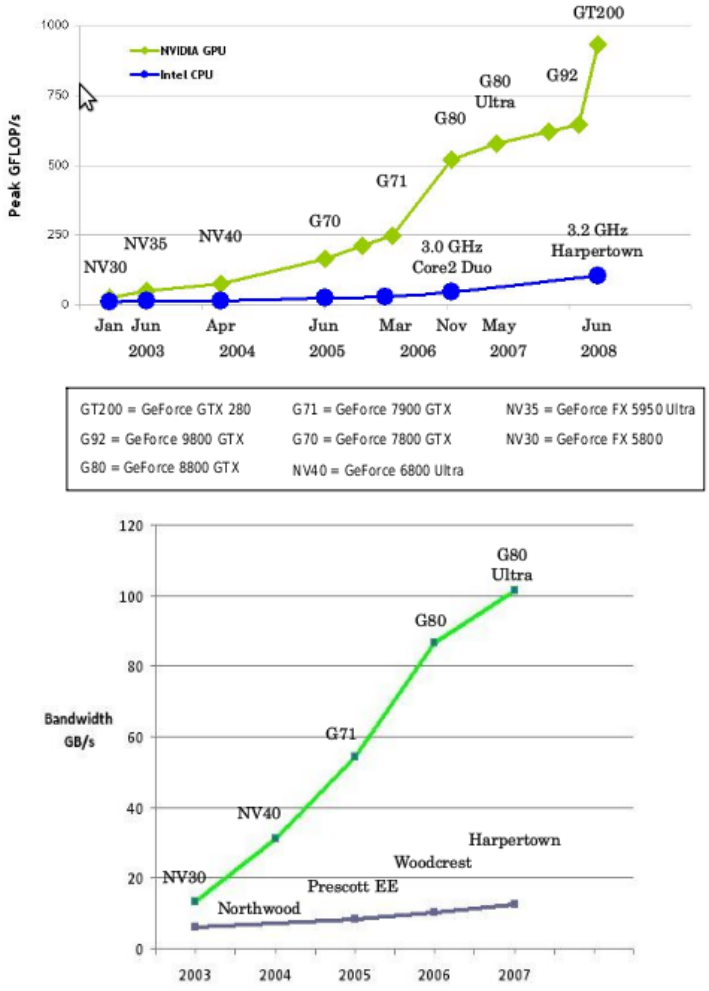

Figure 1.1: The two charts show the difference in memory bandwidth and flops between CPUs and GPUs over the past years. Taken from the CUDA Manual[18]

the bandwidth rate increase significantly faster for GPUs than for CPUs. The reason is that GPUs can dedicate a much larger part of their chip to arithmetic logic units than the CPU, which uses a lot of the chip for caching and advanced prefetching techniques. Therefore the CPU is much more effective for serial execution with advanced branch prediction techniques, which the GPU does not have.

The rate at which new generations of GPUs are released is almost twice as fast compared to CPUs. We can expect a new GPU chip generation approximately once every ten to twelve months. While CPU generations have a span of almost two years.

The role of the GPU

The GPU has until very recently served as the graphical processor of the desktop PC. This was the only real area of application it was designed for. In its earliest stage the GPU was produced with non-programmable hardware that could perform various arithmetic operations that was useful for image processing in games. These different operations ranged from applying fog or blur effects, to smooth surface effects for 3D meshes. These individual effects were called shaders, and over time the GPU developed from consisting of pure hardware specific shaders to becoming a programmable processor where developers could design their own software shaders which ran on the GPU as small programs. Eventually the GPU transferred completely from using hardware shaders to the unified shader pipeline, where all the processing power of the GPU can be utilized on any type of shader the programmer wants. This inspired the use of GPUs as a general purpose computing engine.

Some minor research and hobby programming has gone into general purpose computing on GPUs, but before the arrival of programmable shaders and unified shader architecture, general purpose computing with GPUs was limited.

Today however the GPU manufacturers have acknowledged the interest in using their hardware for more than just graphics. With the arrival of tools and libraries such as CUDA[18] and OpenCL[13] that allows for easier development of general purpose computing software, we argue that over the coming years the GPU will go from being a dedicated graphics processor to becoming a parallel processing unit dedicated for all parallel computing tasks.

1.3 Goal

The goal of this thesis is to study how a GPU can be utilized to accelerate a simulation of advective transport through porous medium. The area of general purpose GPU computing is a relatively young research topic and therefore the exploration of it is limited, but growing fast. We will explore this area in several directions to demonstrate the feasibility of using GPUs in reservoir flow simulations.

- First we will present a very simple solver for the transport equation Buckley-Leverett running on a GPU.
  - We show its numerical accuracy by comparing it to an equal solver written for the CPU where double precision is used.
  - We show what increase in computational speed we get compared to a serial CPU solver and a multi-threaded CPU solver.
- We present a heterogeneous solution where the CPU will solve pressure while the GPU will solve transport.
  - We show that despite a loss in numerical accuracy this model provides a result that is well within acceptable ranges.
  - We show how this computing model benefits greatly in terms of computing time compared to a homogeneous computing model where only one standard CPU is used.
  - We show that the small sacrifice in numerical accuracy is acceptable in this kind of simulation compared to a stricter model where pressure is solved for each timestep.
- We propose a prototype GPU solver for corner-point grids (see Section 2.4.6 for further description).
  - We show how this is possible despite the strict hardware restrictions on the GPU.

- We present a prototype using a high resolution numerical scheme running on a GPU.
  - We show how this scheme benefits greatly in speed by using a GPU instead of a CPU.

1.4 Thesis definition

"We will investigate the use of GPU as a computational engine in reservoir flow simulations and prove that this approach will provide faster runtimes compared to a serial computation on a CPU. We will present GPU models for a high resolution scheme, corner-point grids, and a heterogeneous model where the CPU computes pressure while the GPU computes transport. Further, we will show that these models perform better than serial CPU solvers and we will evaluate their numerical accuracy compared to double floating point precision."

1.5 Contributions

In this thesis we demonstrate an implementation of a reservoir transport solver on GPU. We investigate into how corner-point grids, which are vital in reservoir modeling, can fit on a GPU with its restricting hardware architecture. Further we look at a high resolution scheme, which is more computationally demanding than the regular models used. And last we demonstrate a heterogeneous model where we solve pressure and transport in parallel by utilizing both the CPU and the GPU. We provide a benchmark for the GPU prototypes compared to regular serial execution on a CPU and a thread based execution. Finally we do numerical testing to find inaccuracies that may arise when we move from double precision on the CPU to single precision on the GPU.

We believe that today the most important research into parallel heterogeneous computing is providing algorithms and techniques for various computing tasks. This field of computing has only been active for the past two years and there is still many areas that are totally unexplored. In this thesis we contribute with a better understanding of benefits and limitations for running reservoir flow simulation on a GPU. In addition we uncover some new questions that will need answering.

1.6 Target audience

This thesis is a study into the use of GPU as a computational engine for simulating flow through a porous medium. The reader will benefit if they have knowledge in GPGPU computing, reservoir modeling or numerical modeling. However the paper is written in way so that any reader familiar with parallel programming and partial differential equations should be able to interpret and understand the concepts we have used.

1.7 Report outline

In Chapter 2, we start by introducing the reader to the various fields this thesis spans. We begin by introducing the GPU and its role in general purpose computations. We then go into floating-point numbers and the importance of accuracy in physical simulations. Last in this chapter, we introduce the reader to reservoir simulations and the numerical methods used in this thesis. In Chapter 3, we present the GPU computing prototypes to the reader. We start with the most general and iterate through more advanced models using a high resolution scheme and dealing with corner-point grids. We then present the various tests performed on the different prototypes. Chapter 4 presents the results gained from testing, both performance and numerical accuracies. In Chapter 5, we discuss the various

findings and how they relate to our claims

Chapter 2

Background

2.1 The Graphics Processing Unit

The Graphics Processing Unit or GPU has undergone a rapid development since its mainstream emergence in the mid-nineties. The adoption of GPUs for real time rendering has increased steadily since then, and today no 3D game renders its frames without taking advantage of the GPU.

The gaming industry has had a strong influence on the development of the GPU. Ever more computational demanding games have lead GPU manufacturers to constantly strive for higher arithmetic throughput. The result is an extreme parallel computing engine, which dwarfs the computational throughput of a standard CPU by at least an order of magnitude. And despite the extreme parallelism it is easy to write programs or shaders for the GPU.

2.1.1 Hardware

The GPU hardware architecture forms what is known as the graphics pipeline. The pipeline contains a number of steps that should be done in a specific order to render a frame correctly.

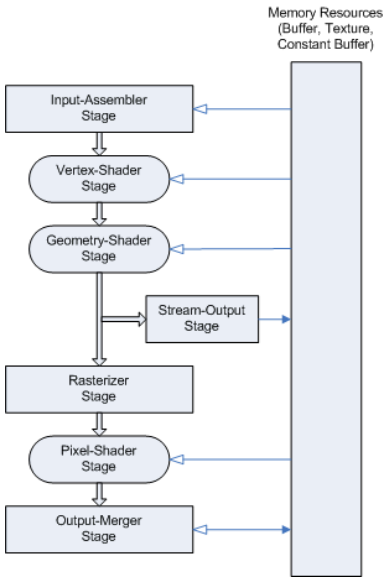

Figure 2.1: The graphics pipeline. From Direct3D section on MSDN[16]

Figure 2.1 shows how the pipeline is structured. All the shader steps are programmable and can take input from the internal memory on the GPU. These shaders are programmed in special C-like shader languages and then compiled to run on the GPU hardware. Shaders are today a very important component in the process of generating each frame in an interactive game. The latest GPU's use what is called a unified shader pipeline, which means that shaders of all types can run on any of the GPUs cores. This gives programmers more flexibility in how they want to utilize the GPU's processing power.

2.1.2 General purpose GPU computing

Using the GPU for non-graphical computations has become known as general purpose computing on GPUs or just GPGPU[10]. Researchers quickly recognized the power in the extreme parallel nature of the GPU. When the transition from a fixed to a programmable pipeline was made, many started to experiment with the GPU to speed up computationally demanding applications, and when the fragment and pixel pipelines were made into the unified shader pipeline, the potential for GPU computing grew even more. The early test results showed great promise[11] and in 2007 the GPU manufacturers released APIs that greatly simplified the process of writing GPGPU programs. Up until then, the only way of accessing the GPU was by following the graphics pipeline through a graphics API like OpenGL[14] or Direct3D[15]. Now with the new APIs like Nvidias CUDA[18] or ATIs Stream[3], programmers get direct access to the processor cores on the GPU.

2.2 CUDA

CUDA or Compute Unified Device Architecture[18] is Nvidia's API for general purpose computing on their GPUs. The API consists of a compiler and libraries that provide extensions to the C language in order for programmers to easily map their problem onto the GPU architecture.

2.2.1 Programming model

CUDA provides developers with a parallel programming model where the CPU initializes data and execution configurations of functions that are then executed on the GPU. The data can be moved between the CPU and the GPU but this is an expensive operations and it is recommended to keep the number of copy

operations between the GPU and the CPU as low as possible, or by asymmetric data transfers so that the GPU is not forced to wait for data.

Host and device

In the CUDA programming model the main memory of the computer and the CPU is referred to as the host. The GPU is referred to as the device. A host can have several devices, but there can only be one host.

Kernels

Functions that are executed on the GPU are referred to as kernels. Kernels have only access to GPU memory areas and data need to be copied from the host memory to the device memory in order for kernels to process it.

When a kernel finishes, the results are stored in the device global memory and is available for further processing or copying back to host main memory.

2.2.2 Thread batching

CUDA uses an intuitive programming model in order for programmers to easily split their problem up into parallel computations. Three main constructs are used: threads, blocks and grids. A block contains a predefined number of threads. This number is defined at the entry-point for the kernel. When writing a kernel, the programmer has access to variables that identify each thread by a thread-id, a block dimension, and a block-id. By using these three variables, the programmer can index different data in different parts of memory for each thread and so write programmes that are SIMD (single instruction multiple data). CUDA refers to its architectural model as SIMT (single instruction multiple thread), because of its multi-threaded pipeline architecture. Figure 2.2 shows how a grid is split into

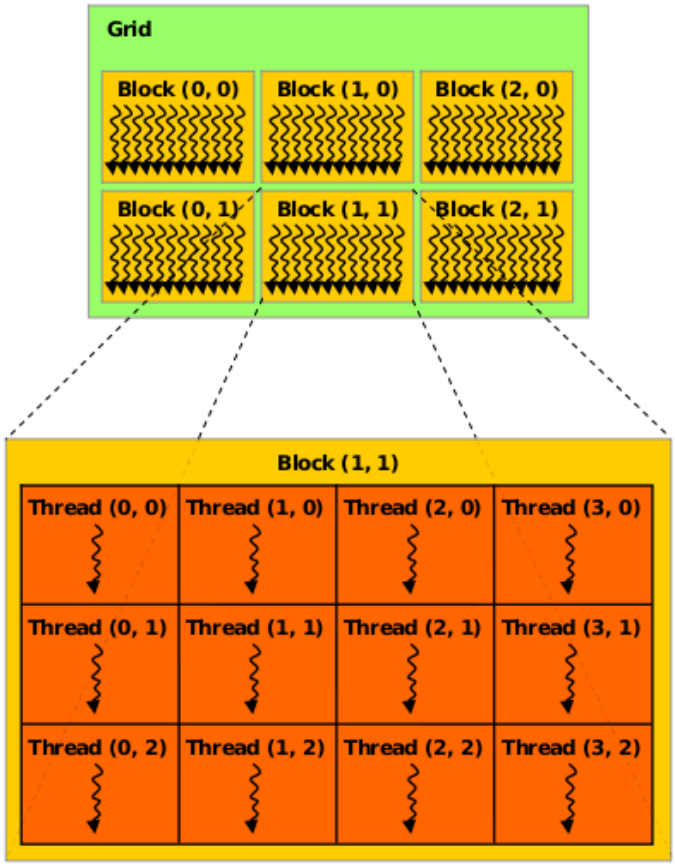

Figure 2.2: CUDAs thread block grid model. A grid is split into block which again is split into threads. From the CUDA Manual[18]

blocks, which again is split into threads.

```
float* deviceArray; //pointer to array on device

// Kernel that increments an array on the device by 1.
__global__ void incrementArrayKernel(float* array)
{
        //variable that defines the dataIndex for each thread.
        int dataIndex = blockIdx.x*blockDim.x+threadIdx.x;

        //Increment operation. Notice no loop is needed.
        array[dataIndex] =  array[dataIndex] + 1.0f;
}

int main()
{
        // Function that initializes array on device.
        // This function must do initialization of data on host,
        // allocation of memory on device and copy operations
        // of data from host to device.
        initalizeArrayOnDevice(devicearray);

        // Execute kernel with 10 threads in each block
        // and 2 blocks in the grid.
        incrementArrayKernel<<<2, 10>>>(deviceArray);

        return 0;
}
```

Figure 2.3: Code that demonstrates the SIMT programming model

In the code example in Figure 2.3 we demonstrate how a kernel function in CUDA increments twenty elements by executing twenty threads, one for each data element. By specifying how many threads and blocks to execute in a kernel, the programmer can perform operations on varied data structure sizes.

2.2.3 Performance optimization

CUDA provides a host of techniques and tricks that programmers can utilize to speed up parts of their kernels. Most of these techniques deal with minimizing memory latency between the GPU cores and GPU global memory. Data that are shared within each block and accessed often, should be copied to shared memory, which can be described as manually managed cache. Access to shared memory is instant. Shared memory is only shared within each block, and so if all threads in a kernel need access to a particular data item, this has to be kept in global memory.

Access to global memory can be optimized by doing texture referencing. This means that you define a memory area in the internal memory of the GPU and provide caching for this area. This will reduce access times greatly, but it becomes a read-only area. Other techniques that can be utilized are coalesced access. This is an access method in which the programmer makes sure that each thread accesses consecutive memory areas in their corresponding access operations (like in Figure 2.3). The GPU can then utilize the fact that it can fetch a 64-bit or even 128-bit word from memory and so get two or four memory-fetches at the latency cost of one.

One of the more important things a programmer should pay attention to is how much of the core register a block occupies. If the blocks in a grid are small enough to fit more than one block in each core register, the GPU can mask memory latency by switching between blocks. It is therefore recommended to have room for at least two active blocks per processor core.

2.2.4 Kernel execution

When a kernel is executed, all the cores on the multiprocessors of the GPU are loaded up with as many blocks as possible. The blocks loaded onto a GPU core are active blocks and context switching between these blocks is extremely efficient.

Each core executes a warp, which is 32 threads from one of the active blocks. A thread warp is executed through a SIMD pipeline. When a warp has to wait for memory fetches, the core can switch to another warp to hide much of the latency related to memory fetching.

2.3 Precision and accuracy

The words *precision* and *accuracy* will be used throughout this report in order to describe and discuss the results of my research. We will therefore state the formal definition of the words here.

In the domain of computing, Websters dictionary defines *accuracy* as:

> *"How close to the real value a measurement is"*

precision is defined as

> *"The number of decimal places to which a number is computed"*

Both the precision and accuracy of a computation are very important when it comes to the field of physics simulations. One should especially notice that these two attributes of a computation are highly connected. If we increase the amount of decimal digits computed (higher precision), we will also increase how much our computed number deviates from the actual answer (lower accuracy). This is because computers can only have a finite set of bits to represent an infinite amount of real numbers. When representing very low values or very high values, the computer often has to round off to the nearest number it knows how to represent. When the computation requires higher precision, the computed value will deviate more from the actual real answer. The lower digits will only add more to the difference between the computed value and the real value.

2.3.1 Floating point numbers

The floating point number is described in the IEEE Standard for Floating-Point Arithmetic [4]. This standard describes how floating point operations should be implemented and especially how all possible exceptions should be handled. These exceptions include overflow, division by zero, and other special cases. The standard was revised in 2008 to limit some of the more loose specifications from the 1985 version.

Floating point numbers are represented with three main components. In the 32-bit (single precision) floating point number, the first bit signifies the sign of the number. Then there are eight bits representing the exponent and last 23 bits for the significand. The exponent is biased so that it ranges from 127 to -126. The double precision floating point number has 11 bits for the exponent and 52 bits for the significand.

Units in the last place (ULP) is the unit of measure used to describe the accuracy of floating point operations. ULP(x) is the difference between the two floating point numbers that are closest to the real value of x (x is in the range between the two floating point numbers) [17].

Most of the functions in the CUDA mathematical standard library do not conform to the IEEE-754 standard when using single precision. An average of 2-4 ulps are present in most of the functions, while some go even higher. Addition and multiplication are IEEE-compliant, though (0 ulps)[18]. These small errors will in most cases be acceptable, but one should be aware that they are present. Double precision will give greater accuracy (average 2-3 ulps and, in addition to add and multiply, sqrt and division will also be fully IEEE-compliant). However, the GPU hardware conforms closer to the standard than the CUDA API. Therefore one could implement these arithmetic operations to achieve better compliance with the standard, if this is needed.

2.3.2 GPU single and double precision

The GPU have originally not supported double precision operations. However when GPGPU computing started to gain interest, manufacturers saw the need to add this feature. The newest generation of GPUs do now support double precision, but the computing speed is significantly slower than single precision. The ratio of single-precision arithmetic units to double-precision arithmetic units varies between the two main GPU manufacturers Nvidia and ATI.

2.3.3 Hybrid approach

To avoid both the error of single precision and the slow-down of using double precision on the GPU, one can use what Göddeke[9][8] describe as a hybrid approach. Their technique is to use the single precision GPU for doing fast calculations and then doing an error correction with the CPU. This approach proves to give a highly accurate result and still outperform a fine tuned CPU implementation of the same solver.

2.4 Reservoir simulation

In this section we will explain the various aspects concerning reservoir simulations, as described by Aarnes et. al[1].

2.4.1 Applications

Reservoir simulations are fundamental when modeling oil-reservoirs. In order for drilling to be the most effective, it is vital to do simulations of the reservoir both before and during production. This gives important information that helps when deciding how wells are placed and managed during production.

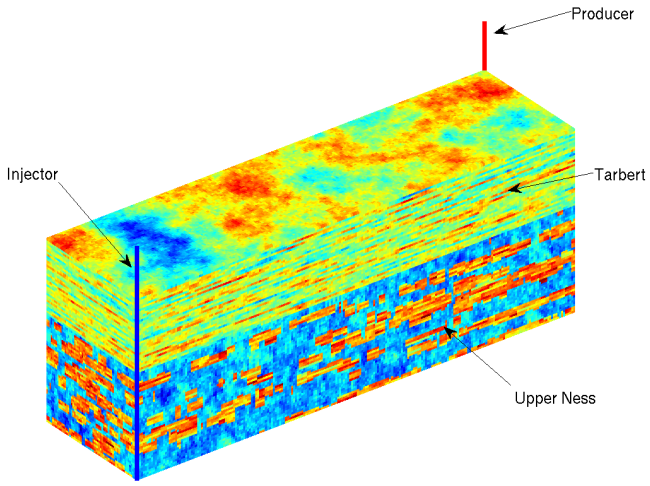

Figure 2.4: [6]This is a visualization the permeability in SPE10[20], a comparative test dataset for a reservoir.

Samples are gathered of the reservoir volume. This data contains porosity, permeability and volume structure. Flux values are calculated with the pressure equation and then applied in the flow simulation which describes the saturation transport of the second phase that is pumped into the reservoir.

As seen in Figure 2.4, there are two types of wells; injection and production. At the injection well water or gas is pumped into the system at high pressure. At the production well oil is forced out by the pressure difference between the reservoir and the production well.

2.4.2 Reservoir

The formation of an oil-reservoir happens over millions of years. First organic materials pile up, layer upon layer. When the pressure and temperature start to rise, the material slowly starts to decompose into hydrocarbons. Geological movement

in and around these areas of hydrocarbons can lead to some of them becoming trapped and forming a reservoir of crude oil. This reservoir is not pure liquid oil, it exists in small pores in the rock formations. This rock is our porous medium. Almost all rocks have pores and the distribution and volume fraction determines the flow properties in the rock.

The reservoir is typically modelled using a grid model that gives a geometric description of the rock formations and provides rock parameters that affect the flow of fluids (examples of parameters include permeability and porosity). Building such models is based on data, experience with, and study of similar formations and use of geostatistics.

2.4.3 Flow physics: the model

Buckley-Leverett(BL) is a mass transport equation used to describe flow through porous medium. It calculates how a phase flows in the system over time, based on flow velocity, porosity, and previously calculated saturation. The flow velocities are obtained through a pressure equation derived from mass conservation and Darcy's law. In its simplest form, the Buckley-Leverett equation can be written as:

$$\phi s_t + \overrightarrow{v} \cdot \nabla f(s) = q \tag{2.1}$$

Where $f(s)$ is the fractional flow, q is flow into and out of the system (wells), ϕ is porosity, s is average saturation of the cell, and $\overrightarrow{v}$ is total velocity given by Darcy's law.

$$\overrightarrow{v} = \frac{-k}{\mu} \nabla p \tag{2.2}$$

Where k is permeability, μ is viscosity and p is pressure.

Porosity: The rock porosity is the void volume fraction of the medium. Porosity is denoted as ϕ and $0 < \phi < 1$.

Saturation: Saturation is the fraction of void cell volume that is filled with water. Saturation is denoted as s and $0 < s < 1$.

Flux: Flux is the integral of the flow velocities over a finite surface.

$$\oint \overrightarrow{v} \cdot \overrightarrow{n} ds \tag{2.3}$$

In a regular 3D grid there is always six edges (all cells are perfect cubes). Should the grid be irregular the number of edges is > 4. This is because the shape of a cell is undetermined and a cell can have several neighbours on one edge.

Fractional flow: The fractional flow is the water fraction of the total flow and varies depending on the saturation of water. The function can also vary depending on various properties like viscosity and pressure. We have used the same function throughout this work.

$$f(s) = \frac{\lambda_w(s)}{\lambda_w(s) + \lambda_o(s)} \tag{2.4}$$

where $\lambda_w(s) = \frac{s^\alpha}{\mu_w} : (\alpha = 2, \mu_w = 1)$ and $\lambda_o(s) = \frac{(1-s)^\beta}{\mu_o} : (\beta = 2, \mu_o = 1)$, which gives the curve depicted in figure 2.5.

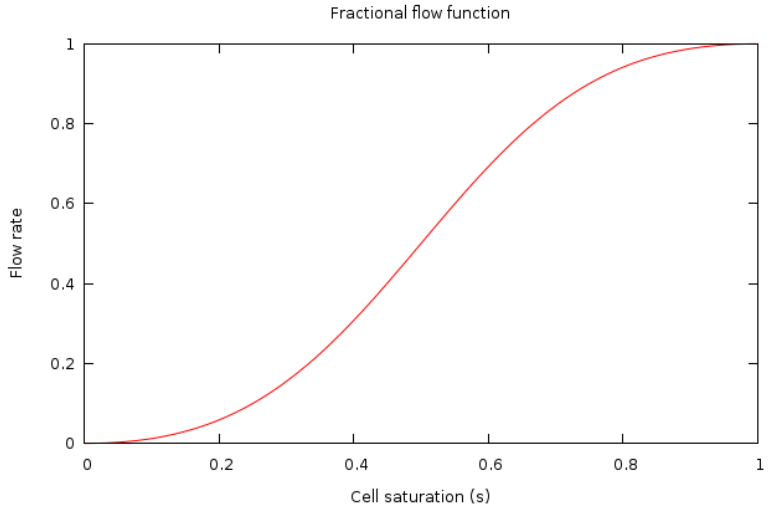

Figure 2.5: The fractional flow rate.

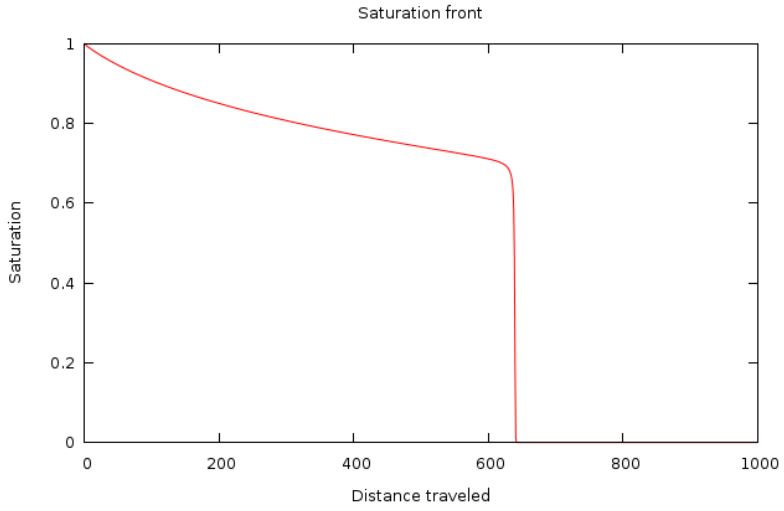

Figure 2.6: The saturation profile of a Buckley-Leverett model using Equation (2.4) as fractional flow rate.

A plot of the one-dimensional Buckley-Leverett solution clearly shows how the saturation front advances with a shock wave followed by a slow gradual increase to max saturation level. This is depicted in Figure 2.6.

2.4.4 Numerical analysis

To solve a partial differential equation on a computer, we have to deploy numerical methods. We use an explicit finite volume method for our computations, for which the unknown is the average saturation within each cell of the grid.

$$S_i^n = \frac{1}{|\Omega_i|}\int_{\Omega_i} S(x, y, n\Delta t)dxdy \tag{2.5}$$

The equation is discretized in space and time and we get the following expression:

$$\frac{\phi_i}{\Delta t}(s_i^{n+1} - s_i^n) + \frac{1}{|\Omega_i|}\sum_j \mathcal{F}_{ij}(s_i^n, s_j^n) = q_i \tag{2.6}$$

where we use an upwind approximation of the flux over each cell interface.

$$\mathcal{F}_{\alpha\beta} = \int [f(s_\alpha)\max(\overrightarrow{v}\cdot\overrightarrow{n_{\alpha\beta}}, 0) + f(s_\beta)\min(\overrightarrow{v}\cdot\overrightarrow{n_{\alpha\beta}}, 0)]ds \tag{2.7}$$

To fit these expressions into an algorithm for a 2D Cartesian grid, they are further formulated into the following expression:

$$\begin{aligned} s_{i,j}^{n+1} = s_{i,j}^n + \frac{\Delta t}{\phi_{i,j}}(q_{i,j}^n - \frac{1}{\Delta x\Delta y}[&\max(v_{i+1/2,j}, 0)f(s_{i,j}^n) + \min(v_{i+1/2,j}, 0)f(s_{i+1,j}^n) \\ &- \max(v_{i-1/2,j}, 0)f(s_{i-1,j}^n) - \min(v_{i-1/2,j}, 0)f(s_{i,j}^n)] \\ -\frac{1}{\Delta x\Delta y}[&\max(v_{i,j+1/2}, 0)f(s_{i,j}^n) + \min(v_{i,j+1/2}, 0)f(s_{i,j+1}^n) \\ &- \max(v_{i,j-1/2}, 0)f(s_{i,j-1}^n) - \min(v_{i,j-1/2}, 0)f(s_{i,j}^n)] \end{aligned}$$

Here we have used the midpoint rule to evaluate the flux integrals. In Figure 2.7

we depict a visual representation of the geometric topology of this calculation in three dimensions. This calculation involving all the closest neighbours is called a stencil. The basic structure of it is important in many fields of physics simulations.

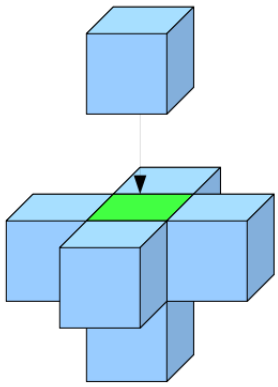

Figure 2.7: The topology of a stencil in a 3D grid. The blue cells are neighbouring cells, and the green cell is the local cell where we want to calculate a new value based on the values of neighbouring the cells

This calculation immediately becomes more difficult to handle when we move over to a corner-point grid (described in Section 2.4.6), where each cell can have a varying number of neighbouring cells.

CFL condition

In an explicit scheme, the value of Δt has to be carefully calculated in order to not destabilize the system. A destabilization can happen if the flow in the system combined with the size of Δt results in cells being filled over their capacity in one timestep. If this happens, the system will become unstable and produce unreliable values in and around that cell.

The CFL condition is a function that calculates how big Δt can be in order to

keep the system stable. We want Δt to be as big as possible to increase simulation speed, but at the same time not so big that it gives rise to unstable cells. The CFL condition depends on the velocities in the system and the derivative of the fractional flow function.

$$\Delta t < \min(\frac{V^i}{F^i_{in}} \times (max f')^{-1} \times c) \tag{2.8}$$

As the expression shows, the optimal Δt value is dependent upon the flux values, and therefore it needs to be recalculated every time the flux changes, or every update from the pressure equation.

2.4.5 High resolution scheme

The finite volume method introduced in 2.4.4 is the most commonly used model for reservoir simulation. However there can be certain caes where greater accuracy is needed. When shocks or discontinuities needs to be modeled more correctly we need to use a high resolution method.

In our particular case the saturation flows with a shock at the front. When the simple finite volume method is used this front will be smeared and not presented accurately in the simulation. Therefore we introduce a high resolution scheme using a second-order Runge-Kutta solver where we use bilienear reconstruction and a central-upwind flux as described by Hagen et. al[12] when they demonstrated the use of GPU for accelerating the shallow water equation. We will use the same high-resolution scheme for our reservoir simulation.

Second-order Runge-Kutta

The Runge-Kutta method we employ for evolving our solver is a total variation diminishing second-order Runge-Kutta method which is written as:

$$S_{ij}^{(1)} = S_{ij}^{(n)} + \sum_j \mathcal{F}_{ij}(S_i^{(n)}, S_j^{(n)})$$
$$S_{ij}^{(n+1)} = \frac{1}{2}S_{ij}^{(n)} + \frac{1}{2}[S_{ij}^{(1)} + \sum_j \mathcal{F}_{ij}(S_i^{(1)}, S_j^{(1)}))]$$

Quadrature rule

For a Runge-Kutta of second order, we use a fourth order Gauss quadrature to evaluate the edge fluxes.

$$\int_{-1/2}^{1/2} f(x)dx = \frac{1}{2}[f(\frac{-1}{2\sqrt{3}}) + f(\frac{1}{2\sqrt{3}})] \tag{2.9}$$

Which for fluxes in x-direction becomes

$$\mathcal{F}_{i\pm1/2,j}(t) = \frac{1}{2}[\mathcal{F}(\mathcal{Q}(x_{i\pm1/2}, y_{j+\alpha}, t)) + \mathcal{F}(\mathcal{Q}(x_{i\pm1/2}, y_{j-\alpha}, t))] \tag{2.10}$$

Where $\mathcal{Q}(x, y, t)$ is the bilinear reconstruction (2.11). To obtain the point values needed for the quadrature calculation we use a piecewise polynomial reconstruction.

Bilinear reconstruction

Because the Gaussian quadrature requires flux point values at the integration points for the rule, we need to reconstruct these point values through a piecewise polynomial reconstructions. In our case we use bilinear reconstruction.

$$\mathcal{Q}(x, y, t) = Q_{ij}^n + s_{ij}^x(x - x_i) + s_{ij}^y(y - y_j) \tag{2.11}$$

The slope values s_{ij}^x and s_{ij}^y are obtained from one-sided differences (slopes) based on the neighbouring cells. Note that the bilinear reconstruction becomes a trilinear reconstruction when we move from two dimensions to three dimensions.

To avoid introducing oscillations at discontinuities and local extremal points, one-sided slope estimates are averaged using a nonlinear limiter function. This gives

$$\Delta x s_{ij}^x = \phi(Q_{ij}^n - Q_{i-1,j}^n, Q_{i+1,j}^n - Q_{ij}^n) \tag{2.12}$$

$$\Delta y s_{ij}^y = \phi(Q_{ij}^n - Q_{i,j-1}^n, Q_{i,j+1}^n - Q_{ij}^n) \tag{2.13}$$

where the limiter function ϕ is written as

$$MM(a,b) = \frac{1}{2}(sgn(a) + sgn(b))min(|a|,|b|) \tag{2.14}$$

$$= \begin{cases} 0, \text{if } ab \leq 0, \\ a, \text{if } |a| < |b| \text{ and } ab > 0, \\ b, \text{if } |b| < |a| \text{ and } ab > 0, \end{cases} \tag{2.15}$$

2.4.6 Corner-point grid

Modern geological 3D models use a geometrical scheme called corner-point grid. These grid models are very agile and gives a very good representation of subsurface geology.

As described Aarnes et. al[2] the corner-point grid is constructed by inserting vertical pillars into the reservoir. These pillars are aligned as a Cartesian grid with one pillar in each point. Horizontal layers are then assigned to each sedimentary bed. The structure of a corner-point grid is depicted in Figure 2.8.

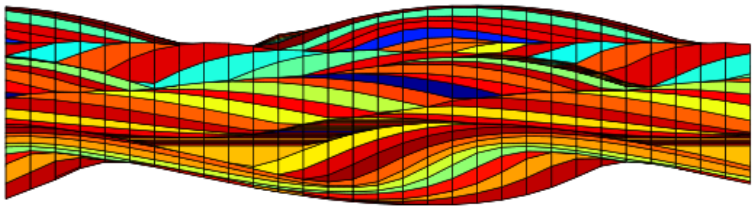

Figure 2.8: A corner-point grid viewed from the side. Taken from[2]

Reservoir simulation in this grid format can easily become a very complex operation. The problem we face when implementing a simple corner-point simulation on GPU, is the fact that cells have a varying number of neighbouring cells.

Chapter 3

Implementation

When exploring the use of GPU for a reservoir flow simulation, we have looked at several different topics that are interesting in this area. These topics are cooperation between CPU and GPU or heterogeneous computing, how to implement corner-point grids on the GPU, and a high resolution scheme to really demonstrate the power of the GPU. In this chapter we will describe the different prototype implementations.

3.1 The basic prototype model

The basic model is a 3D Cartesian grid, in which each cell has exactly six neighbours, the flux values are initialized at execution start and remain constant throughout the entire simulation time. The porosity can be set homogeneous, which means it is the same value in all cells, or it can be loaded from a binary file at execution start. The Entire grid is initialized with zero saturation. Δt, can be specified, and it will remain constant throughout the simulation. When Δt is not specified the solver will calculate a Δt from the CFL condition. When all the data is loaded into main memory it is copied to the GPU.

3.1.1 Memory scheme

The different values saturation, porosity, and flux are arranged in separate arrays, which again are arranged in three dimensions for easy indexing. The exception here is the flux, where each cell has six values, which we can cut down to three to avoid duplication. These three values are then again split into three different arrays, one for fluxes in x-direction, one for y-direction and one for z-direction. The saturation array is also padded with zero-values on all the edges of the grid. This is to avoid out-of-bounds errors when doing stencil operations.

The arrays on the GPU are bound to 3D texture references which provides caching to reduce latency, and the indexing becomes easier as a texture fetch operation for 3D textures uses coordinates as argument and not the array index. This is shown in Figure 3.1.

3.1.2 Initializing for GPU execution

The initialization splits the entire grid into equally sized thread blocks. This is because a kernel grid on the GPU cannot have thread blocks with varying sizes. To accommodate some of the recommendations for optimizations, such as having several active blocks per GPU core, the size of the blocks cannot be any larger than 4x4x4. This value is found through a simple equation from the CUDA manual [18]. By taking into account how much shared data each block needs, how many registers each thread occupies, and how many registers are available per core, we calculate how many blocks can occupy a core at any given time.

We found that the most effective way to deal with a grid that do not fit a 4x4x4 block kernel is to pad the grid with extra cells. This will result in some threads that don't do any relevant work. However the only other way to deal with this problem is to split the grid into different subgrids and having various block sizes in separate kernels. This approach proved to introduce more overhead than

a padding approach.

We want to limit data transfers between GPU global memory and GPU cores. By keeping the thread block as a perfectly sized cube in the grid, the amount of data that can be shared between threads in a block is maximized and the amount of total transmissions between global memory and core registers is minimized. This is because neighbouring cells rely upon each others data when doing the stencil calculations.

The saturation data is loaded as a block of 6x6x6 items. Note here that the saturation has a padding of one around the original thread block of 4x4x4. This padding is required when the stencil is computed. The fluxes are loaded as blocks of 5x4x4 for fluxes in x-direction, 4x5x4 for fluxes in y-direction and 4x4x5 for fluxes in z-direction. These blocks of data reside in shared memory. This is beneficial because neighbouring threads will need access to saturation data from neighbouring cells, and the flux is always shared between two cells, with the exception of boundary fluxes.

3.1.3 Computations

The main computing kernel on the GPU iterates the saturation values one timestep forward for each time it is executed. Each block preloads the data that is needed within each block, and then the computation is performed before the answer is written back to global memory. Each thread in a block is responsible for one cell and the stencil operation for that cell. The main computing part of the kernel is shown in Figure 3.1.

Before this part of the kernel can be executed, all the data is loaded from global memory to shared memory. This way, the data that is accessed multiple times does not require more than one global memory fetch. Shared memory is very limited and so not all data can be loaded to this area. The data that is most beneficial to preload in this manner is data that is accessed several times by different cells.

We also have to remember that data is only shared within each thread block. This makes the data required for the stencil calculations ideal for shared memory, because cells that are in close proximity to each other will have a high probability of being in the same thread block.

```
//Calculate flows over all edges in cell
//sat[] and flux[] are preloaded blocks of data

//Calculate flow in x direction
float flowX =
edgeFlow(fluxX[(threadIdx.x)*(bz)*(by)+(threadIdx.y)*(bz)+threadIdx.z],
fluxX[(threadIdx.x+1)*(bz)*(by)+(threadIdx.y)*(bz)+threadIdx.z],
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z+1],
sat[(threadIdx.x)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z+1],
sat[(threadIdx.x+2)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z+1]);

//Calculate flow in y direction
float flowY =
edgeFlow(fluxY[(threadIdx.x)*(bz)*(by+1)+(threadIdx.y)*(bz)+threadIdx.z],
fluxY[(threadIdx.x)*(bz)*(by+1)+(threadIdx.y+1)*(bz)+threadIdx.z],
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z+1],
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y)*(bz+2)+threadIdx.z+1],
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y+2)*(bz+2)+threadIdx.z+1]);

//Calculate flow in z direction
float flowZ =
edgeFlow(fluxZ[(threadIdx.x)*(bz+1)*(by)+(threadIdx.y)*(bz+1)+threadIdx.z],
fluxZ[(threadIdx.x)*(bz+1)*(by)+(threadIdx.y)*(bz+1)+threadIdx.z+1],
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z+1],
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z],
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z+2]);

float dXdYdZ = 0.5f;
float porosity = dt[0]/tex3D(por3DTexRef,bidx*bx+x,bidy*by+y,bidz*bz+z);
float q = 0.0f;

//Calculate stencil
float stencil = q + dXdYdZ*flowX + dXdYdZ*flowY + dXdYdZ*flowZ;

//Calculate result
float result =
sat[(threadIdx.x+1)*(bz+2)*(by+2)+(threadIdx.y+1)*(bz+2)+threadIdx.z+1] +
porosity*stencil;

//Write result to global memory
globalMemory[myCellAddress] = result;
```

Figure 3.1: Code extract from the main kernel

3.2 CPU - GPU: Heterogeneous prototype

During a reservoir simulation the pressure will change over time and thus changing the flow properties in the system. This change is calculated with a pressure solver that takes saturation and various properties like permeability and porosity as input and solves a system of equations. The result is a field of flux values that are used when evolving the simulation. The common way to arrange this is to solve pressure, then solve transport for a set time duration, before updating with new pressure values, see Figure 3.2. The time duration between each pressure should ideally be one transport step, but this is not feasible because pressure is expensive to solve. Therefore the time can vary from a day to a year in simulation time between each pressure step.

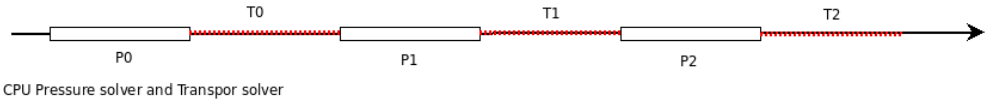

Figure 3.2: This figure shows how the pressure and transport is computed in a homogeneous computing model, where only the CPU is used.

When we free up the CPU by using the GPU instead we can solve pressure simultaneously with transport, known as heterogeneous computing. The downside is that the pressure will lag behind. When pressure is solved for a given time, the GPU simultaneously solves transport for that time. The lag will therefore be exactly one pressure step. We use an inhouse-developed pressure solver and link it with our GPU transport solver. To exchange data we write to binary files and signal with environment variables. This is not the ideal way, but it holds to demonstrate the concept.

The timelag introduced can give rise to some deviations in the final results. We address this issue in the our numerical tests described in Section 3.5.3.

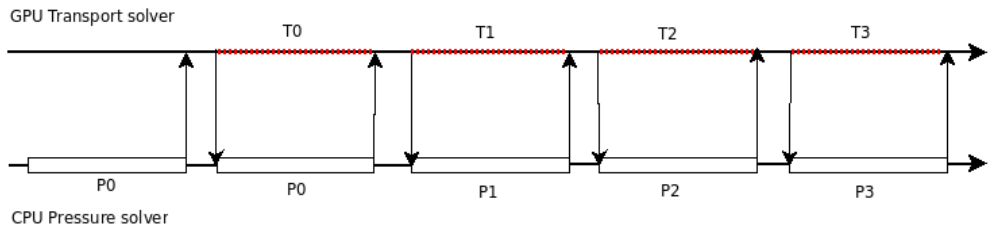

Figure 3.3: This figure shows how the CPU and GPU work in relation to each other in a heterogeneous computing model.

3.2.1 Computing CFL

In the heterogeneous model we need to recalculate Δt each time we update with new flux values. To speed up this calculation we use the GPU.

The calculations required needs to be performed for each cell in the grid. Figure 3.4 shows the setup of this kernel. We split the grid up as in the compute kernel and each cell then calculates what the ideal value for Δt is. This calculation is, as seen in Equation 2.8, based on the flow through the cell. The kernel then returns an array with equal size to the grid. We then do an all-reduce with the *min()* operator on this array. The result is what the maximum size for Δt can be. The reduce function is provided by the cudpp library[7], which is a library containing scan primitives such as bitonic search and reduce functions for CUDA.

```
// The CUDA internal variables have been renamed with the following scheme:
// bidx is blockIdx.x
// bx is blockDim.x
// x is threadIdx.x

// Load all flux values for each cell through texture references
float fluxX0 = tex3D(fluxX3TexRef,bidx*bx+x,bidy*by+y,bidz*bz+z);
float fluxX1 = tex3D(fluxX3TexRef,bidx*bx+x+1,bidy*by+y,bidz*bz+z);
float fluxY0 = tex3D(fluxY3TexRef,bidx*bx+x,bidy*by+y,bidz*bz+z);
float fluxY1 = tex3D(fluxY3TexRef,bidx*bx+x,bidy*by+y+1,bidz*bz+z);
float fluxZ0 = tex3D(fluxZ3TexRef,bidx*bx+x,bidy*by+y,bidz*bz+z);
float fluxZ1 = tex3D(fluxZ3TexRef,bidx*bx+x,bidy*by+y,bidz*bz+z+1);

// Load porosity through texture reference
float porosity = tex3D(por3DTexRef,bidx*bx+x,bidy*by+y,bidz*bz+z);

float f = 2.0f;

float fluxin = fabs(fluxX0) + fabs(fluxX1) +
        fabs(fluxY0) + fabs(fluxY1) + fabs(fluxZ0) + fabs(fluxZ1);

// The total flow into the cell is the sum of all flows divided by 2
// (Equal amounts flow into and out of the cell)
fluxin /= 2.0f;

// The cells ideal dt is the void volume (prorsity)
// divided by how much flow through the cell multiplied with
// max inclination (derivative) of fractional flow
float myDtValue = porosity/(fmax(fluxin,0.00001f)*f);

// The answer is multiplied with a safety factor (usually 0.9)
// and written to global memory.
// Note that myCellId is calculated from grid dimension sizes,
// block dimension sizes and thread id (not shown in here).
output[myCellID] = floor(value*safety);
```

Figure 3.4: Code extract from the kernel calculating the ideal Δt in each cell.

3.3 Corner-point grid prototype

The corner-point grid solver requires a total re-write from the Cartesian grid solver. The Cartesian grid solver is hard-coded to handle cells with exactly six neighbours. The irregular grid prototype will need to adapt each thread to handle its edges which can vary in number from cell to cell. In addition the data needs to be structured differently.

Instead of having one kernel with one thread per cell, we now use two separate kernels, where one is one-thread-per-edge and the other is one-thread-per-cell. We need each cell thread to collect all the edge values that belongs to that cell. The only way found to do this, was to create a mapping index that maps each cell to all its edges. This increases the amount of memory required, and also the amount of data that needs to be transferred between the GPUs cores and global memory.

In this model, we loose a lot of the GPU memory transfer optimizations. It is difficult to structure data to achieve coalesced access, or to minimize transfers by grouping cells in close proximity to each other together in blocks. This is because the amount of data per cell can vary and there is no way of varying shared memory on a per block basis.

On the GPU, we have very limited possibility when it comes to data structures. Writing to a pointer-based data structure such as a graph, tree, or stack is completely impossible, because race conditions would occur and threads would overwrite each other. We can read from these structures, but it will be very inefficient to find values by traversing the structure following the pointers. Therefore the best way is to construct non dynamic structures where you always know where the data is. Even if this requires more memory.

In our implementation the relation from an edge to its two cells is mapped by having two arrays containing cell ids, where the array index is the edge id. One of the arrays contain all the cells on the positive sides of the edges, while the other contains the cells on the negative sides. The positive side of a edge has its normal in the axis positive direction while the negative side has its normal in axis negative direction.

We also generate two arrays for a cell-to-edge relation. One array of int2 contains an index for where the edges for this cell can be found and how many edges the cell has. Then the second array lists out all the edges for each cell. This gives us a good deal of data duplication because all the edges are listed twice in this array.

```
__global__ void
calculateCellSaturations(float *saturation1, float dt)
{
        int cellID = threadIdx.x+blockDim.x*blockIdx.x;

        int2 edgeTablePointer = tex1Dfetch(pointersToEdgeTableTex, cellID);
        float myCellValue = 0;

        // Loop for summing up the edge flows
        for(int i = edgeTablePointer.x; i < edgeTablePointer.y+edgeTablePointer.x; i++)
        {
                int edgeID = tex1Dfetch(edgeTableTex, i);
                int posFace = tex1Dfetch(posCellIDTex, edgeID);
                int negFace = tex1Dfetch(negCellIDTex, edgeID);
                float flow = tex1Dfetch(flowTex, edgeID);

                if(posFace == cellID)
                        myCellValue += flow;

                if(negFace == cellID)
                        myCellValue -= flow;
        }

        // The rest of the kernel is similar to the general version

        float porosity = dt/tex1Dfetch(porosityTex,cellID);
        float q = 0.0f;

        float stencil = myCellValue-q;
        float localSaturation = tex1Dfetch(saturation0Tex, cellID);

        float result = localSaturation + porosity*stencil;

        saturation1[cellID] = result;
}
```

Figure 3.5: The second kernel from the corner-point grid calculation. Here all the edge flows are summed up

During execution we first launch a kernel that calculates the flow over each edge, by using its own flux and the saturation values for the two cells related to that edge. Next we execute a kernel that for each cell loops through all its edges and sums up the flow values to generate the stencil. This is shown in Figure 3.5. The loop cannot be split up into different threads, which is common in the CUDA API. This would create race conditions.

In a SIMD architecture like a GPU all the threads in a warp finishes at the

same time. When we use a loop of unknown length all the threads in a warp will have to wait for the thread with the most loop iterations. This can potentially slow down the kernel if many of the thread warps contains threads where many loop iterations are needed. In our case cells with high amounts of edges will lead to this effect. Especially if the grid system contains sparsely distributed cells with many edges.

3.4 High resolution prototype

The high resolution scheme is a lot more demanding in terms of memory and computations per kernel than the general scheme. In order to not exceed some of the hardware limitations we had to split the computation into several steps.

We split the Runge-Kutta stages into separate parts, and we also split out the bilinear reconstruction to a separate kernel. To evolve the simulation one step we then need to execute four kernels.

```
__device__ float minMod(float a, float b)
{
        int signa = a == 0 ? 0 : 1-2*signbit(a);
        int signb = b == 0 ? 0 : 1-2*signbit(b);
        float res = fmin(fabs(a),fabs(b));
        return res*(signa+signb)*0.5;
}
```

Figure 3.6: Code extract showing the limiter function

For the bilinear reconstruction, seen in Figure 3.7, we load 8x8x8 blocks of saturation values and reconstruct the three component gradient using 6x6x6 blocks of threads. These vectors are written to global memory. We then proceed to compute the Runge-Kutta step. This is done with thread blocks of size 4x4x4. We use the preloading to shared memory technique explained in Section 3.1.2 to load flux values, saturation values and the gradient vectors padded with a boundary of one as in Section 3.1.2.

```
// The buffer zone is accounted for
int x = threadIdx.x+1;
int y = threadIdx.y+1;
int z = threadIdx.z+1;

// Gradient reconstruction in x-direction
float rx =
minMod(sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z]-
        sat[(x-1)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z],
        sat[(x+1)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z]-
        sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z]);

// Gradient reconstruction in y-direction
float ry =
minMod(sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z]-
        sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y-1)*(blockDim.z+2)+z],
        sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y+1)*(blockDim.z+2)+z]-
        sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z]);

// Gradient reconstruction in z-direction
float rz =
minMod(sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z]-
        sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z-1],
        sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z+1]-
        sat[(x)*(blockDim.z+2)*(blockDim.y+2)+(y)*(blockDim.z+2)+z]);
```

Figure 3.7: Code extract from the kernel calculating the slope gradient between cells. Notice how the limiter function minMod is used to control the values

The point reconstructions have to be done in separate functions for each edge. This means that each cell executes six different reconstruction functions before the stencil is summed up and written back to global memory. The point reconstruction for edge x0 can be seen in Figure 3.8.

```
#define I_VALUE 0.2886751346f

__device__ float
CUW_X0(float sat0, float sat1, float4 grad0, float4 grad1, float flux)
{
        //x-1
        float fl = 0.25f*(
        fractionalFlow(sat0+grad0.x*0.5f+grad0.y*I_VALUE+grad0.z*I_VALUE)+
        fractionalFlow(sat0+grad0.x*0.5f-grad0.y*I_VALUE+grad0.z*I_VALUE)+
        fractionalFlow(sat0+grad0.x*0.5f+grad0.y*I_VALUE-grad0.z*I_VALUE)+
        fractionalFlow(sat0+grad0.x*0.5f-grad0.y*I_VALUE-grad0.z*I_VALUE));

        //x
        float fr = 0.25f*(
        fractionalFlow(sat1-grad1.x*0.5f+grad1.y*I_VALUE+grad1.z*I_VALUE)+
        fractionalFlow(sat1-grad1.x*0.5f-grad1.y*I_VALUE+grad1.z*I_VALUE)+
        fractionalFlow(sat1-grad1.x*0.5f+grad1.y*I_VALUE-grad1.z*I_VALUE)+
        fractionalFlow(sat1-grad1.x*0.5f-grad1.y*I_VALUE-grad1.z*I_VALUE));

        return fmin(flux,0)*fr+fmax(flux,0)*fl;
}
```

Figure 3.8: Code extract from the kernel reconstructing edgepoint values with the bilinear reconstructed gradient

The bilinear reconstruction and the Runge-Kutta step is done twice (second-order RK) for each transport stage.

3.4.1 CPU prototypes

For all the GPU prototypes we also made an equal CPU prototype. These prototypes where used for comparison in all of our tests. The CPU prototype was made as a template so that it was easy to switch between single and double precision. We also used OpenMP to split the most computationally demanding for-loops into threads.

3.5 Testing

We tested the different prototypes for both computing speed and the accuracy of results. This section explains the different testing techniques and parameters.

3.5.1 Test platform

Tables 3.1 and 3.2 give a description of the hardware used during testing.

Table 3.1: Desktop specifications

CPU	Intel i7 920@ 2.67GHz
Memory	6Gb DDR3 RAM
GPU	GeForce 260 XFX

Table 3.2: GPU specifications

Core clock frequency	1.24GHz
Number of multiprocessors	27
Number of cores	216
Global memory	1Gb
Max block size	512x512x64
Max threads per block	512
Registers per block	16384

3.5.2 Performance testing

All of the prototypes where measured for computing time per timestep computed. This was done by a simple timer that was started at the beginning of a computing step and stopped when the step was finished. The total compute time accumulated was averaged over the number of steps we had computed.

Today CPUs with two and four cores are common in desktop PCs and we therefore ran tests where only one core was used and tests where we used OpenMP to use all the cores simultaneously. The testing platform was, as seen in table 3.1, equipped with an iCore 920, which with its four cores and hyper-threading technology is ideal for multi-threaded computing.

The performance tests where performed using the SPE10[6] comparative dataset with zero-flow boundaries, production in cell 0 and injection in the last cell of layer one (cell 60x220x1). We ran tests using one layer, ten layers and all layers (85).

3.5.3 Numerical testing

To verify that the prototype solvers produced accurate results, we performed several numerical tests to prove our claims.

Accuracy testing

Hagen et. al[11] demonstrate that the results from a single precision solver of the shallow-water equation to a double precision solver does not differ higher than in the 6th to 7th digit. This is a good indication, and our flow simulation should not produce results far from these.

To test this we do a l_∞ norm calculation of our single precision GPU solver compared to a double precision CPU solver. We use the general prototype in these tests. The error values that we measure here should be equal across the different prototypes.

The l_∞ norm error calculation is expressed as the maximum value across vec-

tor X:

$$X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \tag{3.1}$$

$$|X|_\infty = \max_i |x_i| \tag{3.2}$$

Where x_i is the difference between the double and single precision value of cell i.

We measure the difference at pressure update. Pressure was set to follow simulation time. This means that if we update pressure every second day, we simulate two days of transport between each update. This way all the measurements should be as equal in value as possible when the conditions are equal.

We also do numerical comparisons between float an double precision where we eliminate as many factors as possible that can affect the final result. We use a constant flux field by cutting out the pressure solver completely and manually set the flux value. We make sure that the measurements are done at the place in time where a pressure update should have been if we had used a pressure solver. We also set a constant porosity.

Heterogeneous model

For the heterogeneous model, we needed to to verify that the time shift we introduced between the pressure solver and the transport solver would not add significant errors in the results. We therefore ran several tests where we checked how the results were affected. We looked at how the final results was influenced with

various time lag between pressure and transport. We also tried different viscosities to see how this could affect the result.

The measurements was performed in the production well. This is the point where fluids leave the system. We measure the saturation in this cell and plots the values over time. Measurements were performed at every pressure update. We used 64x64 grid with quarter-five spot well setup. This means a injection in lower left corner and production in upper right corner, with zero flow boundaries. We also ran tests through layer one of SPE10[6].

Chapter 4

Test results

4.1 Performance test results

The performance tests of the prototype solvers are presented in this section.

4.1.1 General prototype solver

Table 4.1: General prototype performance results

Grid size	Solver	Computing time	Speedup
60x220x1	Serial CPU	0.9ms	1.0x
	Threaded CPU	1.2ms	0.7x
	GPU	0.8ms	1.1x
60x220x10	Serial CPU	11.5ms	1.0x
	Threaded CPU	5.2ms	2.2x
	GPU	2.0ms	5.7x
60x220x85	Serial CPU	76.8ms	1.0x
	Threaded CPU	30.8ms	2.5x
	GPU	8.0ms	9.6x

We see from results presented in Table 4.1 that the GPU provides a significant performance increase. What we especially notice is that the GPU becomes more and more effective, compared to the CPU, as the problem grows. As explained in Section 3.5.2, the results presented here are averaged values of computing time per timestep.

4.1.2 High resolution solver

Table 4.2: High resolution prototype performance results

Grid size	Solver	Computing time	Speedup
60x220x1	Serial CPU	24.6ms	1.0x
	Threaded CPU	23.6ms	1.0x
	GPU	9.9ms	2.4x
60x220x10	Serial CPU	147.5ms	1.0x
	Threaded CPU	60.4ms	2.4x
	GPU	19.0ms	7.8x
60x220x85	Serial CPU	1179.2ms	1.0x
	Threaded CPU	355.4ms	3.3x
	GPU	100.9ms	11.7x

We see in Table 4.2 that the performance increase for the higher solution scheme is even better than what we observed for the general scheme in Table 4.1. The high resolution scheme requires more computations per cell so this difference was expected.

4.2 Numerical test results

4.2.1 Heterogeneous prototype

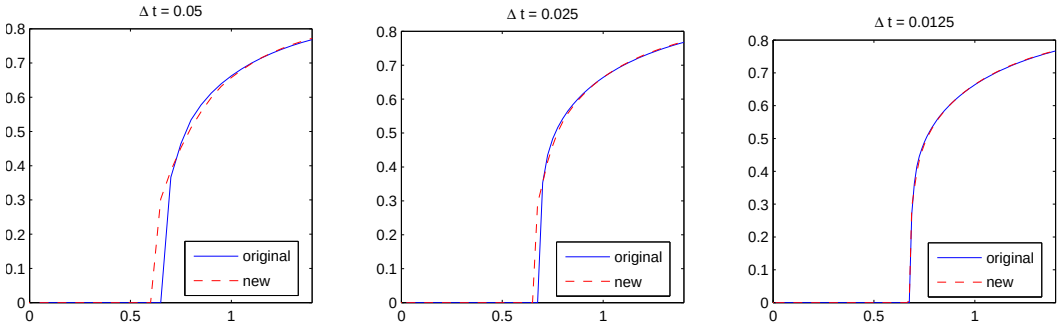

Figure 4.1: The plots show how the heterogeneous solver diverges from the serial solver as the window between each pressure solve is widened. This simulated in a homogeneous quarter five spot 64x64 grid. Note that Δt is the time between each pressure update.

The heterogeneous versus serial comparison test produced results close to what we expected. As seen in figure 4.1 the plots are equal in shape, but as the window between each pressure solve is widened they will start to diverge slightly from each other. The lag in pressure directly affects the speed of the front. When the window becomes too large, the two different fronts will arrive at the production well at different times. Figure 4.2 shows this effect even better, where we simulate flow through the first layer of the SPE 10 comparative dataset[20].

We see from the plots in Figure 4.3 that the heterogeneous version can introduce error if the viscosity of the injected phase becomes higher than the displaced phase. Here the timestep between each pressure solve is coarse, so we can reduce the time window between pressure updates to dampen this effect.

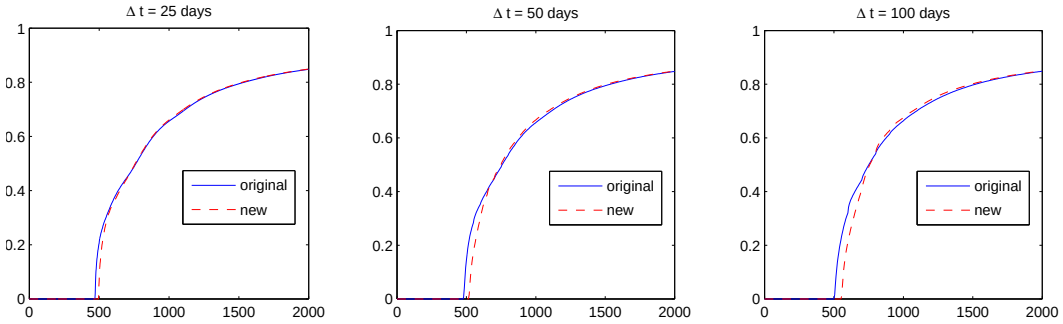

Figure 4.2: The plots show how flow differ between different pressure timesteps. This is simulated in layer one of the SPE10 comparative dataset. Note that Δt is the time between each pressure update.

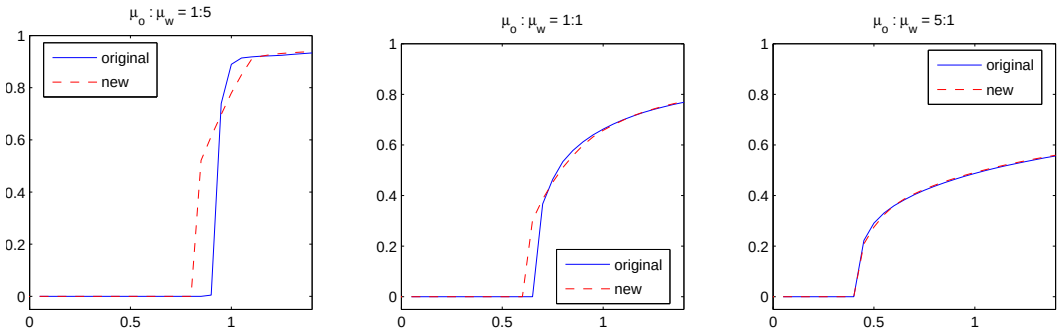

Figure 4.3: The plots show how the heterogeneous solver diverges from the serial solver as the viscosity ratio between fluids changes. This simulated in a homogeneous quarter five spot 64x64 grid

4.2.2 Double-precision versus single-precision

Here we present the results for the numerical tests that we performed to compare double versus single precision.

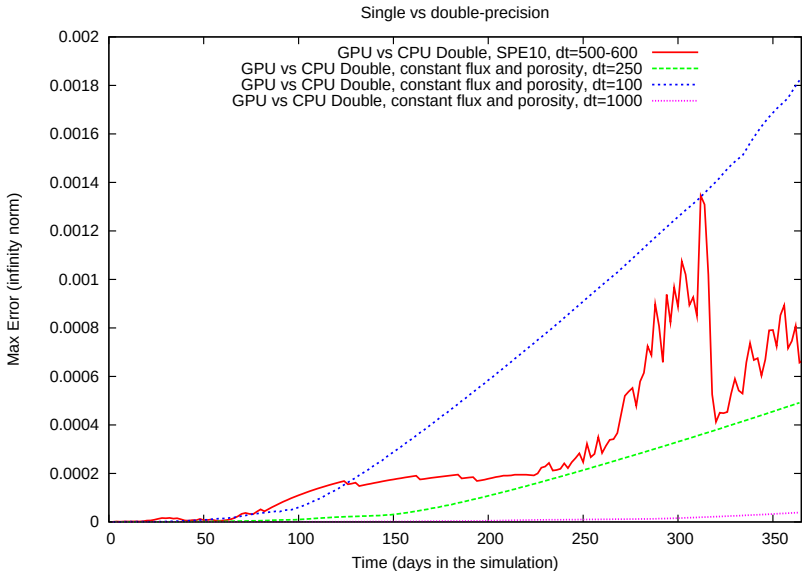

Figure 4.4: The plots show the max difference between single and double precision solvers.

In Figure 4.4 the dotted lines are homogeneous solvers with constant porosity and flux. The red line shows the max error between single and double precision solvers for layer one in SPE10. Notice how the size of Δt (the size of each transport step) is the determining factor that decides how fast the error accumulates. This is important because when going from using double precision to single precision the amount of timesteps that can be computed, before a significant error accumulates, is reduced. A double precision solver can run more steps because the the error added per timestep is many times smaller due to increased precision in the arithmetic operations.

In Figure 4.5 we see how a CPU single precision solver differ from a GPU singler precision solver. We see that the difference is no more than 10^{-6}. This error is to be expected because of different hardware implementations of the IEEE-754 floating point standard.

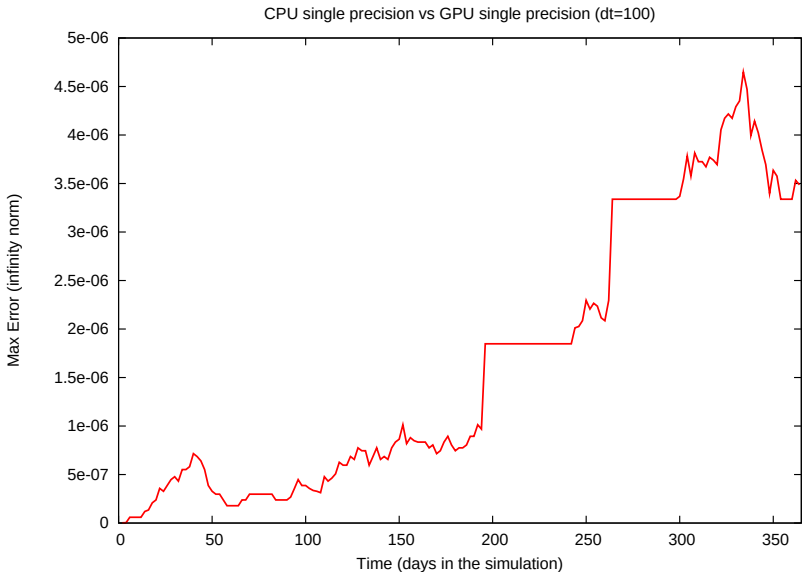

Figure 4.5: The plot show the max difference between a CPU single precision solver and a GPU single precision solver.

4.2.3 Mixed precision

To solve the error that arises from the use single precision instead of double precision Göddeke et. al[8][9] proposes a mixed precision scheme, where double precision is used some places to cancel out the most error contributing factors. We can test this out by doing some simple changes in our CPU implementation. We then run the saturation as double precision where the stencil and old saturation values are added in double precision.

As Figure 4.6 shows, this has an extremely positive effect. We reduce the

error by several orders of magnitude. The difference between mixed precision and double precision is smaller than the GPU to CPU single precision error.

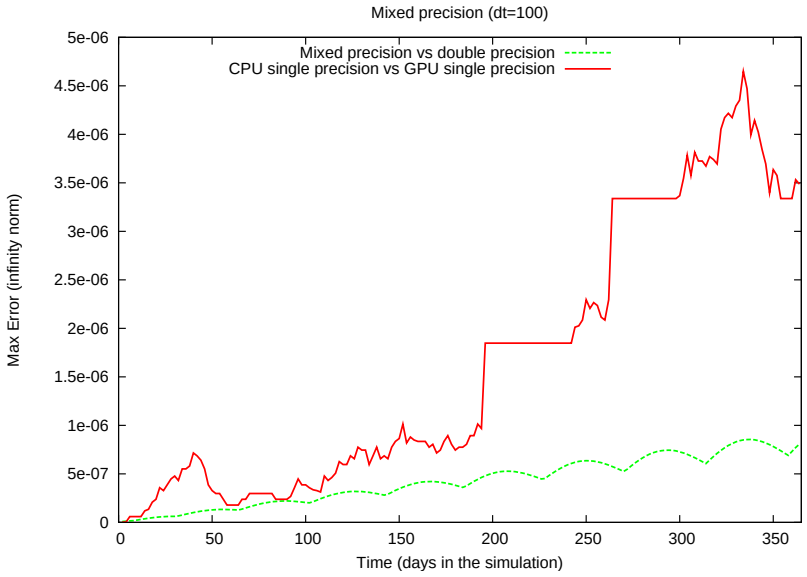

Figure 4.6: The plot show the max difference between a mixed precision solver and a double precision solver referenced against the GPU vs single precision error plot

The reason for this positive effect is that a small difference occur between the single and double precision solvers when the saturation values of the stencil computation is added together with the local cells old value. This error is removed when the saturation values are kept in double precision.

4.3 Corner-point grid prototype

The corner-point grid prototype was not tested extensively. We ran a corner-point format grid, that contained no irregularities, to verify that the prototype functioned correctly. The prototype worked correctly and it performed surprisingly well. On a 60x220x1 grid, it performed equally with the general prototype.

We do expect it to not perform this well when the grid becomes more irregular, but we did not have the chance to test this.

Chapter 5

Discussion

5.1 Results

In this section we will discuss the various results we obtained during testing. And we will also discuss the implementations and further work that can be done to improve them.

5.1.1 Performance

From our results we see that the CPU is able to keep up with the GPU when the problem size is small, but as the number of cells is increased the GPU performs more effectively. This is especially true for the high resolution model which is highly demanding in terms of flops.

We believe that the computing speed ratio between CPU and GPU will continue to grow proportionally with the problem size. This will continue until the problem size overflows the global memory of the GPU. When this happens we have to start swapping between host memory and global memory and this will

probably create a drop in performance.

We have not spent time on fine tuning and optimizations so we believe the performance of the GPU solvers can be increased even more. Especially the high resolution prototype should be possible to speed up more. This prototype takes up a lot of registers on the GPUs cores and so, parts of it is pushed to local memory. This is not ideal and should be avoided to get good performance.

We now use a lot of time to preload data into the GPUs cores. We have not had the chance to test if this preloading leads to more overhead than the time saved by reducing global memory fetches. There might be ways to avoid the preloading and only fetch data when needed that runs faster than our prototypes. The argument behind this theory is that the prefetching takes up so many registers in the kernel that the number of active blocks per core becomes significantly reduced.

On the other hand, by not preloading data into shared memory all the threads would have to fetch its own data, and could not rely on other threads sharing their data. This would at least quadruple the amount of memory fetches, but since not all the threads are involved in the preloading routine it would not add more operations per thread. It could however, add time in the form of latency. If not done correctly to avoid most of the latency, this could potentially create a bottleneck where a lot of time goes into waiting for memory fetches.

5.1.2 Precision

Our results show that a CPU single precision solver and a GPU single precision solver does not differ more than what we expected. Due to some different implementations of the float standard in the GPU hardware, they can differ by 10^{-7}.

Comparing against double precision also gives expected results. These results show that the number of timesteps computed on a GPU will have to be reduced compared against a double precision solver. This is because the iterative transport

solver will give a difference in accuracy between single and double precision. This will, over enough time steps, accumulate up to a difference which is unacceptable. We should also point out that this same effect exist while using double precision, compared against the actual real solution. However, here the added error each step is many orders of magnitude smaller. A double precision solver can therefore run more timesteps than a single precision solver without accumulating this error.

This error should however, be possible to correct. By running an algorithm at regular intervals it should be possible to restore the single precision result to match the double precision result. We try a slightly different approach and test a mixed precision method, which almost eliminates this accumulation error all together.

By keeping the saturation values in double precision and doing the summation of the stencil in double precision, we get a very satisfying result which is well within the tolerance threshold. However A GPU implementation of this scheme does require a restructuring of the current kernel. We must keep in mind that double precision operations on the GPU is expensive both in terms of performance and in terms of memory requirement. Should an error correction scheme, executed at regular intervals, prove to be equally effective, this will probably be easier to implement and still keep the performance increases the GPU provides.

We should however, not fail to mention that double precision is still faster on GPU than on CPU. The performance increase is smaller than single precision and the memory footprint will be larger, but the GPU should still outperform the CPU, though not to the same extent.

5.1.3 Heterogeneous computing model

The heterogeneous computing prototype really demonstrate what we believe is the strongest argument for using GPUs. We can eliminate the computing time of the pressure update almost completely. The numerical tests show that long intervals between each pressure update, or a high water to oil viscosity rate can

introduce deviations in the result when using this model. However most reservoir simulations done today are well within these boundaries.

The viscosity rate is a variable that cannot be changed without affecting the final results, but the pressure solve timestep can be set to a lower boundary of one pressure step per transport step. The only thing this will affect is that the simulation will need more computation time. Therefore if a high water to oil viscosity rate is needed, it is possible to compensate for the deviation by reducing the time window between each pressure solve.

5.2 Further development

There are still many aspects of reservoir flow simulation that needs to be tested further on the GPU. We leave many of the aspects of flow simulation totally unexplored. These aspects vary from introducing gravity into the model, to multicomponent flow, and further exploration and testing of corner-point grids.

5.2.1 Gravity

Gravity introduces a good deal of extra arithmetic operations and we believe that this could further increase the performance gap between the GPU and the CPU. While both will get a negative impact in performance , the CPU is much less able to tolerate such increases in operations per cell.

5.2.2 Corner-point grid

Our study of the corner-point grid implementation should be extended. We only had the chance to develop a simple prototype with the most general computation methods. It would be interesting to see if a high resolution corner-point scheme is

even possible to implement on a GPU. We would also like to see more study into how the performance is affected as the grid becomes more irregular.

The corner-point grid prototype is not well optimized with its large indexing tables. We would like to see more work being done to find ways to make use of CUDAs shared memory for the corner-point grid implementation. We have introduced a lot of memory fetching in order to solve the cell to edge and edge to cell relation. Some of these fetches can probably be avoided with better structuring of memory tables.

5.2.3 Multi-component flow

Multi-component flow is another feature that should be looked into and tested. Most of the industrial flow simulations requires two or three component systems (water, gas and oil). This will introduce a lot more data that need to be transferred and processed. The shared memory will come close to reaching its limit or even overstepping it. Thus one would have to either fetch all data into registers when needed, or do a gradual fetching procedure and switch between phases of fetching data and processing data inside the kernel.

On the other hand, a multi-component solver requires more arithmetic operations to evaluate flux functions. This will probably aid to increase the performance difference between the CPU and the GPU.

5.2.4 More advanced CFL condition

Another aspect that can be improved to speed up the simulation is a stronger CFL calculation. By taking into account how much saturation a cell has and using the slope value of fractional flow for this saturation, one could find a larger Δt. This would reduce the amount of steps needed, and directly speed up the simulation.

5.2.5 Scaling

One of the areas we did not explore at all was scaling. It is easy to add more GPUs into a system, but is it possible to run reservoir simulations across several GPUs and still retain good performance? We have to keep in mind here that this would require a good deal of data transfer between the GPUs.

Scaling in terms of problem size is also an area we left largely unexplored. How many cells can the GPU handle? We assume that the computing time per timestep will rise proportionally with the amount of cells in the system. At least as long as global memory can hold the entire dataset. According to our tests the CPU will have a much steeper rise in computing time than the GPU. Will this trend hold even after GPU global memory is filled and swapping needs to be introduced into the system?

Chapter 6

Conclusion

6.1 Summary of Results

Our performance tests show that the GPU quickly starts outperforming the CPU when the problem size grows. This holds for the general prototype, and the high resolution prototype even more so.

We see from the testing that the heterogeneous computing model works quite well. However, there are some very specific exceptions. Large time windows between pressure updates or high water to oil viscosity rates leads to errors in the results. However, the viscosity issue can be compensated for by running pressure updates more often.

From the precision testing we see that using single point precision drastically reduces the number of timesteps we can compute before the results start to deviate. However, we also see that the change that is needed to correct this is not very large, and is very feasible to implement on a GPU.

6.2 Conclusion

6.2.1 Performance at low cost

From our general prototype and the high resolution prototype we see that the GPU is vastly more capable than a CPU when the computing task is of a parallel nature. Many of the more demanding scientific simulations possess these basic characteristics that make them so easy to split into smaller parallel tasks.

Not only does a data-parallel processor, like the GPU, provide a great performance increase, but the parallel architecture also provides the most power efficient computing engines that can be used in high performance computing. Todays CPU waste a huge portion of its energy to heat emission, which again has to be removed to avoid overheating of CPU and other vital components. The GPU and other parallel architectures can avoid much of this heat generation by running at low frequencies, and still outperform the fastest CPUs in any parallel computing task.

6.2.2 Single vs double precision

The strongest argument against GPU accelerated simulations is its limited double precision performance. We show that running a reservoir simulation using single precision does not produce acceptable results when the number of iterations computed goes up beyond 10^4. However, we also show that by using a mixed precision solver, this problem is not an issue and this opens up the possibility of using GPUs.

In addition the trend seems to be that GPUs will slowly move over to the use of double precision as well as single precision. The performance for double precision on GPUs today is low compared to single precision, and so most of the benefits of using a GPU is not that high. But this will change as the need for double precision

increases.

6.3 Contributions

We have demonstrated an implementation of a reservoir simulation on GPU, both for a general finite volume scheme, but also for a more advanced high resolution scheme with bilinear reconstruction of point values. Further we have tested a heterogeneous reservoir simulation model where we solve pressure on the CPU and transport on the GPU. Finally we have proposed a computing model for corner-point grids.

This work forms a solid base to build further research and work into GPU-accelerated reservoir simulations. We have shown that the GPU is quite capable of providing increased performance, but that some problems still needs to be solved. We provide some insight into the the single versus double precision dilemma, but this problem still needs more study to overcome the GPUs limited double precision support. Our mixed precision tests show that this can be a viable solution if it is possible to implement effectively on the GPU.

Bibliography

[1] J. E. Aarnes, T. Gimse, and K.-A. Lie, *An introduction to the numerics of flow in porous media using Matlab*. Springer Verlag, 2007, pp. 265–306.

[2] J. E. Aarnes, S. Krogstad, and K.-A. Lie, "Multiscale mixed/mimetic methods on corner-point grids," *Computational Geosciences*, vol. 12, no. 3, pp. 297–315, sep 2008.

[3] ATI stream computing - technical overview. AMD. [Online]. Available: http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf

[4] (2008) IEEE standard for binary floating-point arithmetic. American National Standards Institute and Institute of Electrical and Electronic Engineers.

[5] A. R. Brodtkorp, C. Dyken, T. R. Hagen, J. M. Hjelemrvik, and O. O. Storaasli, "State of the art in heterogeneous computing," *submitted for publication*, 2009.

[6] M. A. Christie and M. J. Blunt, "Tenth SPE comparative solution project: A comparison of upscaling techniques," *SPE Reservoir Eval. Eng.*, vol. 4, pp. 308–317, 2001, url: `http://www.spe.org/csp/`.

[7] "CUDPP: CUDA data parallel primitives library," GPGPU. [Online]. Available: http://gpgpu.org/developer/cudpp

[8] D. Göddeke and R. Strzodka, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (part 2: Double precisionGPUs)," Technical University Dortmund, Tech. Rep., 2008.

[9] D. Göddeke, R. Strzodka, and S. Turek, "Accelerating double precision FEM simulations with GPUs," in *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Sep 2005.

[10] "GPGPU - general-purpose computations on graphics hardware," GPGPU. [Online]. Available: http://gpgpu.org

[11] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. O. Henriksen, "Visual simulation of shallow-water waves," in *Simulation Modelling Practice and Theory 13*. Elsevier B.V., 2005, pp. 716–726.

[12] T. R. Hagen, M. O. Henriksen, J. M. Hjelmervik, and K.-A. Lie, "How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine," in *Geometric Modelling, Numerical Simulation, and Optimization*. Springer Berlin Heidelberg, 2007, pp. 211–264.

[13] OpenCL specifications. Khronos. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[14] OpenGL 3.1 specifications. Khronos. [Online]. Available: http://www.opengl.org/registry/doc/glspec31.20090324.pdf

[15] Direct3D msdn documentation. Microsoft. [Online]. Available: http://msdn.microsoft.com/en-us/directx/default.aspx

[16] "Microsoft developer network, direct3d 10 graphics," Microsoft. [Online]. Available: http://i.msdn.microsoft.com/Bb205121.d3d10_pipeline_stages_so(en-us,VS.85).gif

[17] J.-M. Muller, "On the definition of ulp(x)," Institut National de Recherche en Informatique et en Automatique, Tech. Rep., 2005.

[18] CUDA programming guide. Nvidia. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf

[19] R. Ramanathan, "Multi-core processors: Making the move to quad-core and beyond," Intel Corporation, Tech. Rep., 2006.

[20] "SPE: Comparative solution project," Society of petroleum engineers. [Online]. Available: http://www.spe.org/web/csp//