NTNU – Trondheim
Norwegian University of
Science and Technology

A GPU Accelerated Simulator for CO2 Storage

Elisabeth K Prestegård

Master of Science in Physics and Mathematics
Submission date: Januar 2014
Supervisor: Helge Holden, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

The goal of this thesis has been to develop a fast simulator for large-scale migration of CO_2 in saline aquifers. We have also focused on being able to let the CO_2 storage atlas from the Norwegian Petroleum Directorate specify the reservoir properties. In order to meet the demands of simulating on large data sets combined with high performance, we have investigated the possibilities of using graphic processing units (GPUs) to accelerate the computations.

The Intergovernmental Panel on Climate Change, IPCC, has considered CO_2 to be one of the main factors influencing the climate changes of today. Capture and storage of CO_2 is one of the strategies which could reduce the amount of CO_2 released into the atmosphere. However, there are still uncertainties related to flow of CO_2 in saline aquifers. It is therefore necessary with fast simulators which can predict this behavior to minimize the risks involved in a storage project.

GPUs are initially designed to accelerate graphic operations. As opposed to standard CPUs, where most of the transistor capacity is used on advanced logic, the GPU uses most of its transistors on floating point operations in parallel. This results in that the theoretical upper bound for floating point operations are 7-10 times higher on the GPU than the CPU. Thus, GPUs have shown to be a strong tool when solving hyperbolic conservation laws using stencil based schemes, as a large amount of the computations can be parallelized.

In compliance with the storage atlases we have based our simulator on structured grids. Our numerical scheme consists of a finite volume method combined with an explicit Euler method.

Sammendrag

Målet med denne masteroppgaven har vært å utvikle en simulator for storskala migrasjon av CO_2 i berggrunnen, med fokus på å kunne kjøre raske beregninger på store datasett. Det har også vært lagt vekt på at Oljedirektoratets CO_2 lagringsatlas skal kunne være bakgrunnen for reservoar egenskapene. For å møte disse kravene, har vi sett på bruk av grafikkort (GPUer) for å akselerere beregningene.

The Intergovernmental Panel on Climate Change, IPCC, har vurdert CO_2 til å være en vesentlig faktor i dagens klimaendringer. CO_2 fangst og lagring er en av strategiene som kan redusere utslipp. Innenfor lagringsfasen er det fortsatt usikkerhetsmoment knyttet til flyt av CO_2, og man trenger derfor raske simulatorer som kan bidra med å minske risikoen ved eventuelle lagringsprosjekt.

Innenfor numerikk har man de siste årene sett et økt fokus på bruk av grafikkort. Graffikkort er i utgangspunktet designet for å akselerere grafiske operasjoner. I motsetning til standard CPUer, hvor det meste av transistorene brukes til avansert logikk, bruker grafikkortet det meste av sine transistorer til å utføre flyttallsberegninger i parallell. Dette gjør at man teoretisk sett har mulighet til å øke utføre flyttallsberegninger 7-10 ganger raskere på GPUer enn CPUer. For løsning av hyperbolske konserveringslover hvor man har stensil-baserte numeriske skjema, kan store deler av beregningene parallelliseres, slik at man kan utnytte ressursene på en GPU.

I samsvar med lagringsatlasene, er simulatoren vår basert på strukturerte grid. Vårt numeriske skjema er basert på finite volume metoder i rom, sammen med eksplisitt Euler i tid.

Preface

This report is my thesis for the Master's degree program at Industrial Mathematics, NTNU. It is written in the period of August 2013 - January 2014, at SINTEF ICT Applied Mathematics, and was based on a specialization project written in the spring of 2013.

First and foremost I would like to thank Halvor M. Nilsen and André R. Brodtkorb at SINTEF for all their help. They always took their time to answer any of my questions. I would also like to thank Helge Holden at NTNU for making the cooperation with SINTEF possible.

Elisabeth Prestegård
Oslo
January 19, 2014

Contents

Preface **V**

1 Introduction **1**

1.1 Why Graphical Processing Units . . . 2
1.2 Why Carbon Dioxide Capture and Storage . . . 3
1.3 Implementation Background . . . 4
1.3.1 SINTEF's MRST Module . . . 4
1.3.2 The Shallow Water Simulator . . . 6

2 Mathematical Background **9**

2.1 CO_2 Theory - Governing equations . . . 9
2.1.1 Mass balance and Darcy's law . . . 9
2.1.2 Reduction of Dimensions Through Vertical Integration 11
2.1.3 Hysteresis . . . 16
2.2 The Numerical Scheme . . . 17
2.2.1 Finite Volume Method and Spatial Discretization . . . 18
2.2.2 CFL Condition and Time Discretization . . . 20

3 GPU Programming Concepts **25**

3.1 GPU Architecture . . . 26
3.2 CUDA Programming Model . . . 27

4 A CO_2 Migration Simulator on the GPU **31**

4.1 Implementation . . . 32
4.1.1 C++ Interface . . . 32
4.2 GPU Implementation . . . 33
4.3 Numerical Results . . . 37
4.3.1 Verification of Simulator . . . 38
4.3.2 Coats Time Stepping versus MRST Time Stepping . . . 42
4.4 Performance Results on the GPU . . . 43
4.4.1 Optimizing the Flux Kernels . . . 46

4.4.2 Hardware Specific Optimization 51

5 Concluding Remarks **61**

Bibliography **63**

Chapter 1

Introduction

The goal of this thesis has been to develop a fast simulator for large-scale migration of CO_2, in saline aquifers. We also focused on being able to let the CO_2 storage atlas from the Norwegian Petroleum Directorate (NPD) specify the reservoir properties.

At SINTEF, there has over the last years been a development of simulators for CO_2 flow, the Matlab Reservoir Simulation Toolbox (MRST) [1,16]. We have considered the possibilities for further improvements in terms of runtime performance by using GPUs. We have based our GPU simulator on a simulator for the Shallow Water equations developed by Brodtkorb et al., [6,7]. This runs the computations solely on the GPU, and we have modified this to suit our numerical scheme.

The outline of this report is as follows: This chapter will first discuss exactly why GPUs might be applicable when aiming for improved performance within numerics, secondly the motivation behind simulating CO_2 storage. We then describe the CO_2 simulator MRST and the Shallow Water simulator, and why they are applicable to us. The next chapters will go through the theory involved when simulating CO_2 flow in saline aquifers; governing equations, the idea behind vertical integration and the numerical scheme. Then we go on to discussing the basics of GPUs; a description of GPU architectures exemplified by NVIDIA's Fermi Architecture and an introduction to the concepts in CUDA, the programming model we have based our GPU implementation on. Finally, we discuss the results of our simulator, both in terms of numerical results and performance results, along with comments for further work.

1.1 Why Graphical Processing Units

In scientific computing we see a constant demand for increasing the performance of numerical solutions. Means of achieving this have typically been to increase the clock speed of the central processing unit (CPU). However, this frequency is limited by power and heat restrictions, thus the introduction of *multicore* processors [10]. Now, we rather see an increased focus on improving the parallelism when aiming to improve performance [10].

GPUs have the last years been considered a great tool for parallel computing. Initially designed to do the large amount of calculations required for graphical computations, the very idea behind their architecture is to execute simple sequences of code in parallel. GPUs today consists of multiple stream processors, each operating in a SIMD kind of way. SIMD, *single instruction, multiple data*, is just that; a single instruction applied to multiple data elements.

Regarding the actual speedup gain by using GPUs instead of CPUs, a comparison of theoretical peak performance between NVIDIA GPUs and Intel CPUs can be seen in Figure 1.1. As of 2012, the performance gap was approximately seven times for both gigaflops and bandwidth [5]. Thus, with suitable algorithms the potential of increased performance due to parallelism is clear.

On the market today, there are three main suppliers of GPUs: Intel, AMD

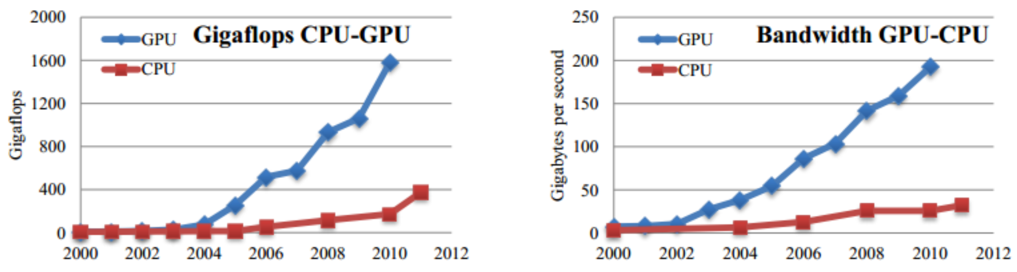


Figure 1.1: Theoretical peak performance of NVIDIA GPUs in comparison with Intel CPUs. Picture is from [5]

and NVIDIA. Within the low-performance market Intel have the largest share of the market, while AMD and NVIDIA provide most of the GPUs on the high performance market [5]. Within academics, NVIDIA provide the most widespread GPUs, and is also what we have based our implementation on. There are three languages suitable for implementation on NVIDIA GPUs - NVIDIA CUDA, DirectCompute and OpenCL. As stated by Brodtkorb et al., [5], CUDA is the most advanced language out of these three, and also what we have done our implementation in.

1.2 Why Carbon Dioxide Capture and Storage

The Intergovernmental Panel on Climate Change, IPCC, stated in their Third Assessment Report [18], that human activities had, and would continue to, influence the atmospheric composition. Also, they stated, the largest contribution to greenhouse gases from humans was CO_2. This is a result of combustion of fossil fuels such as coal, oil and natural gas. It was predicted that the emissions of CO_2 due to fossil fuel burning would continue being the dominating influence on the atmosphere during the 21^{st} century. CO_2 capture and storage (CCS) can play a vital role in lowering the emission of CO_2. IPCC states that CCS may be responsible for half of the emission reduction this century.

CCS consists of three main steps; first CO_2 is captured by separating it from industrial and energy related sources, then transported to a storage location, and here it ideally will remain away from the atmosphere for a long time [18]. Even though capturing is considered the most costly process, storage is the most uncertain part. Knowledge about CO_2 storage is based upon experience gained from the oil and gas industry, but the technology itself is new, and fast computations are therefore important to evaluate the risks involved when storing CO_2 [18].

There are two main ways of storing CO_2 today, ocean storage and geological storage. Ocean storage will typically be either releasing CO_2 into the ocean; 800 meters or more below surface where it dissolves, or releasing it onto the surface bed where a CO_2 "lake" will develop. Geological storage on the other hand refers to the methods involving use of deep underground geological formations. Here CO_2 can either be injected into drained oil and gas reservoir, into existing reservoirs as a unit to increase recovery of oil and gas, or into saline formations [18]. In the North Sea Basin, deep saline aquifers and depleted oil and gas fields are regarded as the formations with the greatest potential for CO_2 storage, [11]. In our work, we have focused on CO_2 storage in saline aquifers.

When planning a geological storage project it is essential to predict the distribution of CO_2 such that injection can be maximized while the risk of leakage is kept minimal. There are several mechanism which traps CO_2 in a geological storage site: First of all, when injecting CO_2 more than 800 meters below surface, CO_2 will be in a supercritical or liquid state. CO_2 will then be much less dense than brine, resulting in an upward flow, which can be prevented from flowing back to the surface if it is kept down by an impermeable cap rock. Secondly we have solution trapping, which occurs as CO_2 dissolves in water. As it is dissolved, it is no longer a separate phase, such that forces which originally drives it upwards is eliminated. Mineral trapping can occur if dissolved CO_2 reacts with minerals, potentially forming carbonate minerals. This could be the most permanent way of storage, but can potentially take thousands of years to accomplish. Finally we have residual trapping; as the CO_2 flows through the reservoir, some of it will be trapped in the pore space due to capillary forces [13, 18].

In Norway one has in the last years seen an increased focus towards CCS. The Norwegian Petroleum Directorate published in 2013 a CO_2 storage atlas for the Norwegian Sea [11], concluding that in the Norwegian Sea alone, one could store 5.5 giga ton of CO_2 - a hundred times more than the actual emissions from Norway were in 2012. When the report was written, two CO_2 storage project was in place on the Norwegian Continental Shelf. One is the Utsira formation, where approximately 1 million tonnes of CO_2 from the gas production at Sleipner Vest Field has been injected every year since 1996 [11].

1.3 Implementation Background

1.3.1 SINTEF's MRST Module

At SINTEF, there is currently a development of a numerical CO_2 Laboratory, funded by CLIMIT, the Norwegian research program for accelerating the commercialization of CCS. The CO_2 Laboratory is a module within the open source Matlab Reservoir Simulation Toolbox (MRST) [1, 16]. The toolbox includes demonstrations of simulation methods and modeling concepts.

The theory behind the MRST simulator is based upon the same governing equations as in Section 2.1. The result is a coupled system, with the first partial differential equation (PDE) referred to in literature as the *pressure* equation and the second one referred to as the *transport* or *saturation* equation. A typical approach when solving this system is the IMPES formulation: implicit pressure, explicit saturation. The idea is to let the elliptic pressure equation be solved by means of an implicit method, using either the initial data or the results from previous time steps to update the current global pressure in the reservoir. Following this, an explicit method is used to solve the (nonlinear) saturation equation, where the pressure from the previous equation is used as a constant, instead of a variable.

One of the differences between our simulator and the MRST, is the fact that while MRST supports simulation on unstructured grids, we have implemented ours using structured, or *cartesian*, grids. By operating solely on a cartesian grid, we hope to better take advantage of the speed-up in performance the GPU can offer. By keeping the grid structured, it is first of all easier to split the domain for parallel computing. Secondly, by not having to deal with all the extra information about grid cell neighbours etc., we reduce the amount of memory which needs to be stored in global memory, and transferred during simulation. The latter will have an impact on performance, see Section 4.4. Using cartesian grids is also in compliance with the CO_2 storage atlas from NPD, which is what we aim to be able to simulate on.

In the MRST module, support for simulating both on a 3D model and a 2D model is implemented. The 2D model is a vertically integrated version of the 3D model, thus being computationally cheaper on each iteration. During the work with MRST, Ligaarden and Nilsen compared 3D modeling of CO_2 storage with a 2D model based upon an assumption of vertical equilibrium (VE) [17]. They found that in cases with a low z-resolution, a VE model is more accurate than a full 3D model, where capillary effects are small. Also, there is a stronger decoupling between the pressure equation and the transport equation when considering the VE model, especially for the post injection scenario. We have thus based our work on the VE model.

Finally, as the pressure equation is elliptic, it is thus not as suitable for parallel implementation, hence not as suitable for being solved on the GPU. There is currently ongoing research regarding parallel solvers for elliptic PDEs, see for instance Esler et al., [9]. Another approach in our simulator would perhaps be to let the CPU solve the pressure equation. However, this is outside the scope of interest in this thesis, and we have thus chosen to leave this for further work. For the time being, we therefore rely on the results from MRST regarding the updated pressure.

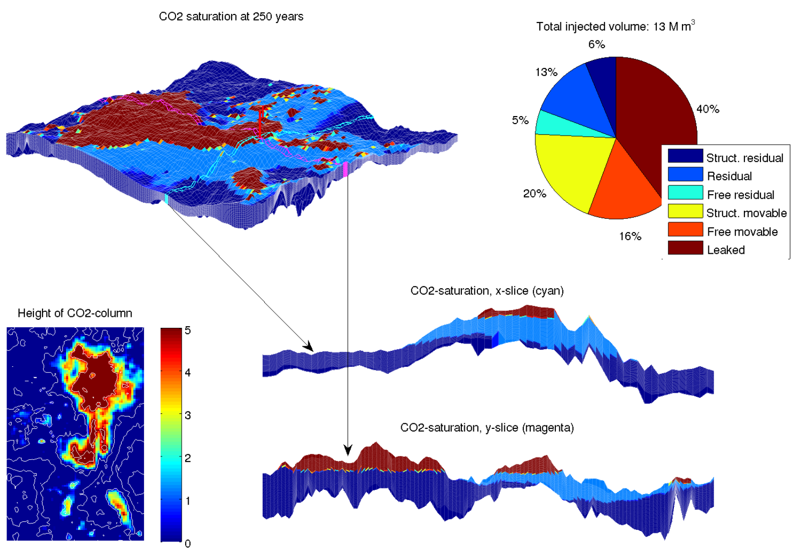


Figure 1.2: An example of simulating the Sleipner Field by the MRST Toolbox.

1.3.2 The Shallow Water Simulator

Our GPU accelerated simulator is based upon the Shallow Water Simulator implemented by Brodtkorb et al., [6, 7]. They solved an explicit Kurganov Petrova scheme for the shallow water equations. The simulator is written using C++ and NVIDIA CUDA [23]. The C++ code handles data allocation, deallocation and initialization, plus the data transfer between the CPU and the GPU, while the GPU which handles all of the computations. We will briefly describe why this simulator has been applicable to us.

The shallow water equations, derived from the Navier-Stokes equations, can model phenomena such as river flows, tidal waves and dam breaks. The equations are given as follows:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hu \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}$$

i.e., a system of hyperbolic conservation laws, where h represents the depth of water, and B, the bottom topography. When written in a more compact form:

$$Q_t + F(Q) + G(Q) = H_B(Q, \nabla B) + H_f(Q)$$

Here Q represents the vector of conserved variables, while F and G represents horizontal flux. H_B and H_f are the bed slope and bed shear stress source terms.

The simulator is based on a Kurganov-Petrova scheme [14], with a spatial discretization as follows:

$$\begin{aligned} \frac{\mathrm{d}Q_{ij}}{\mathrm{d}t} &= H_f(Q_{ij} + H_B(Q_{ij}, \nabla B) \\ &\quad -\big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big] \\ &= H_f(Q_{i,j}) + R(Q)_{ij} \end{aligned}$$

Now, what we see is that the shallow water simulator is based upon a system of conservation laws, and discretized in a way based on computing the fluxes over cell *faces*. When modeling CO_2 storage, the governing equations are based around not a system, but a scalar conservation law, see Section 2.1. And although we have a different numerical scheme, namely a finite volume method based upon the first order upwinding method, we will also base the flux computations on cell face values.

Brodtkorb et al. implemented both an Euler scheme and a second order Runge-Kutta scheme. Thus, both of the simulators will be explicit schemes, with a CFL restriction (see Section 2.2.2) on the time step, which needs to be updated on each time step for optimal performance.

The underlying grid structure of the shallow water simulator is, as in ours, a cartesian one. With some adjustments we could therefore follow the same approach in terms of domain decomposition. (The adjustments is due to the fact that our scheme is first order, as opposed to the second order scheme implemented in the Shallow Water Simulator.)

As a result of the factors mentioned above, the modeling of CO_2 and the shallow water equations follow the same basic outline when implemented on the GPU.

Chapter 2

Mathematical Background

In our simulator, we have aimed to model the migration of CO_2 in a reservoir with an impermeable cap rock. We have based our equations on an assumption of a sharp interface between the two phases; CO_2 and brine, and vertical equilibrium within each phase. Our main variable of interest is the height of CO_2 across the reservoir. The following sections will describe how to get the governing equation, together with the numerical scheme implemented.

2.1 CO_2 Theory - Governing equations

2.1.1 Mass balance and Darcy's law

The basic equations for transport of fluid in porous rock is the law of mass balance and Darcy's law. We first describe the law of mass balance:

If we let q denote the amount of some quantity, say mass or energy, the change over time for q over any domain Ω, will be determined by the flux f over the boundaries $\partial\Omega$, and also by any external source or sink s. If we let $\mathbf{x} \in \mathbb{R}^3$, this conservation law can mathematically be described (on integral form) as:

$$\int_\Omega \frac{\partial q}{\partial t}\,\mathrm{d}\mathbf{x} = -\oint_{\partial\Omega} f\cdot\nu\,\mathrm{d}A + \int_\Omega s\,\mathrm{d}\mathbf{x} \tag{2.1}$$

Or, written as a partial differential equation (PDE):

$$\frac{\partial q(\mathbf{x})}{\partial t} + \nabla\cdot f(\mathbf{x}) = s(\mathbf{x}) \tag{2.2}$$

When relating this to the conservation of volume for fluid in a porous medium;

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = \Psi \tag{2.3}$$

Here $\mathbf{u}$ is the velocity and ρ the density of the fluid, while Ψ denote the density function of any external source or sink. Rock porosity, ϕ is the pore volume fraction of the rock, i.e. the "empty" spaces, such that $0 \leq \phi \leq 1$. Porosity is dependent of the reservoir pressure, p, and the rock's compressibility is related by $c_r = \frac{1}{\phi}\frac{\mathrm{d}\phi}{\mathrm{d}p}$ [3].

In 1856, the French engineer Henri Darcy found empirically that the velocity for flow in porous media is related to the gradient of the fluid pressure and gravity. This law can be thought of as an analogue to Fourier's law of heat conduction and Ohm's law of electrical conduction, however with two driving forces rather than just one [3]:

$$\mathbf{u} = \frac{-\mathbf{k}}{\mu}(\nabla p + \rho g \mathbf{e}_z) = -\frac{\mathbf{k}}{\mu}(\nabla p - \rho\mathbf{g}) \tag{2.4}$$

Here we let $\mathbf{k}$ represent permeability, ∇p the pressure gradient, and μ the viscosity of the fluid.

Permeability is the rock's ability to transport fluid. In SI units it is measured in m^2, however, it is also common to measure it in Darcy (D), where $1\mathrm{D} \approx 0.987{\cdot}10^{-12}\ \mathrm{m}^2$. $\mathbf{k}$ is a positive definite tensor, such that the fluid will flow in the direction of the pressure gradient.

(2.3) and (2.4) can also be extended to multiphase flow. If we disregard the interaction between components within each phase, equation (2.3) can for each phase be written as:

$$\frac{\partial(\rho_\alpha\phi s_\alpha)}{\partial t} + \nabla(\rho_\alpha\mathbf{u}_\alpha) = \Psi_\alpha \tag{2.5}$$

where α is an index for the phase you are considering, i.e. brine or CO_2 in our case. The saturation of a phase, s_α, is the volume fraction of each phase in the porous media, meaning that

$$\sum_\alpha s_\alpha = 1, \qquad \alpha \in \{CO_2,\ \text{brine}\} \tag{2.6}$$

Darcy's law for multiphase flow can be written as [3]:

$$\mathbf{u}_\alpha = -\frac{k_{r\alpha}\mathbf{k}}{\mu_\alpha}(\nabla p_\alpha + \rho_\alpha g \mathbf{e_z}) = -\frac{k_{r\alpha}\mathbf{k}}{\mu_\alpha}(\nabla p_\alpha + \rho_\alpha \mathbf{g})$$

Here relative permeability; $k_{r\alpha}$, has been taken into account. This is due to the fact that for multiphase flow, the actual permeability for a phase at a given location will depend on the saturation of the other phases at this location, but also how the phases interact with the rock. This means that the total permeability of phase α is $\mathbf{k}_\alpha = \mathbf{k}k_{r\alpha}$ [3].

By introducing phase mobility; $\lambda_\alpha(s_\alpha) \equiv \frac{k_{r\alpha}(s_\alpha)}{\mu_\alpha}$, we get the following:

$$\mathbf{u}_\alpha = -\lambda_\alpha \mathbf{k}(\nabla p_\alpha - \rho_\alpha \mathbf{g}) \tag{2.7}$$

2.1.2 Reduction of Dimensions Through Vertical Integration

A problem when modeling CO_2 - brine aquifers, is that the parameters included in our models will range over large scales, both temporal and spatial.

On the temporal scale we might have, on the largest, processes spanning centuries. This includes for instance the post injection period. Following this, we find the time scale of injection which typically have a lower end of 1 year. On the other side of the scale, we find the period of capillary equilibrium, with an approximate upper end of only a week [21].

For the spatial scale, we will typically have aquifers spanning kilometers wide - in each direction. On the other hand, the CO_2 - brine interface on a pore scale might be of order 10^{-6} meters [21]. The first scale, the *macro* scale, is important due to definition of domain and computational capacity. On the latter, the *micro* scale, we will find the processes of mass transfer, relative permeability, etc. We thus need approximations which can capture processes on both sides of the length-scale we are operating in.

As we want to be able to simulate the changes our entire reservoir, we want a model represented on the macro scale, both spatially and temporal. The idea is to vertically integrate our model, and thus incorporate the behavior on a micro scale into the macro scale equations by means of heterogeneous multiscale methods (HMM). This we will do by defining a compression- and a reconstruction operator, denoted by $\mathcal{C}$ and $\mathcal{R}$. The compression operator will compress the 3D information of a variable into a 2D variable, typically by means of integration. The reconstruction operator will give approximations to the fine-scale parameters based on the results of the coarse scale model, thus closing our model. An important concept in HMM is consistency, i.e., if we take the combined operator ($\mathcal{RC}$), our coarse function will remain unchanged.

In the following, we will use uppercase letters to denote our coarse-scale variables, while the reconstructed fine-scale variables will be denoted by hat. Also, the variables ξ_T and ξ_B will be used to denote the upper and lower boundaries of our formation.

First, let us start with a compression operator for mass, which is the conserved variable:

$$M^i = \mathcal{C}_{M^i} m^i \equiv \sum_\alpha \int_{\xi_B}^{\xi_T} \rho_\alpha \phi s_\alpha m_\alpha^i \,\mathrm{d}x_3 = \int_{\xi_B}^{\xi_T} m^i \,\mathrm{d}x_3$$

M^i will satisfy a coarse-scale conservation law:

$$\frac{\partial M^i}{\partial t} + \nabla_{\shortparallel} \cdot \mathbf{F}^i = \Psi_\Sigma^i$$

where

$$\Psi_\alpha^i = \int_{\xi_B}^{\xi_T} \psi_\alpha^i \,\mathrm{d}x_3 + \Psi_B^i - \Psi_T^i$$

$$\mathbf{F}^i = \mathbf{e}_{\shortparallel} \cdot \sum_\alpha \int_{\xi_B}^{\xi_T} \rho_\alpha \mathbf{u}_\alpha m_\alpha^i \,\mathrm{d}x_3$$

As we can see, the coarse scale variable $\mathbf{F}^i$ is a function of fine-scale variables through $\mathbf{u}_\alpha$. We will thus need to define reconstruction operators for phase pressure and saturations.

The Dupuit Assumption

Spatial averaging of pressure is difficult, as pressure is not an additive variable. What we will do is choose our compressor operator for pressure based on subsampling rather than vertical integration, and define the sampling as the pressure at a datum; $x_3 = \xi_P$, such that

$$P_\alpha = \mathcal{C}_{P_\alpha} p_\alpha \equiv p_\alpha(x_3 = \xi_P) \tag{2.8}$$

Now, we will also take into account the Dupuit assumption, which is an assumption of no-flow perpendicular to the boundaries of our formation, i.e.:

$$0 \approx \mathbf{u}_\alpha \cdot \mathbf{e}_3 = \lambda_{\alpha,3}(s_\alpha) k_3 \left(\frac{\partial p}{\partial x_3} - \rho_\alpha \mathbf{g}\mathbf{e} \right)$$

This will be satisfied if

$$0 = \int_{\xi_P}^{x_3} \left(\frac{\partial p}{\partial x_3} - \rho_\alpha \mathbf{g}\mathbf{e} \right) \mathrm{d}x_3'$$
$$\Rightarrow \quad p_\alpha(x_3) - p_\alpha(\xi_P) = (\mathbf{g}\mathbf{e}) \int_{\xi_P}^{x_3} \rho_\alpha \, \mathrm{d}x_3'$$

This gives us our reconstruction operator for fine-scale pressure, where the superscript D denotes that it is based on a Dupuit assumption:

$$\widehat{p}_\alpha = \mathcal{R}^D_{P_\alpha} \equiv P_\alpha + (\mathbf{g} \cdot \mathbf{e}) \int_{\xi_P}^{x_3} \rho_\alpha \, \mathrm{d}x_3$$

The Sharp Interface Assumption

When injecting CO$_2$ at depths below 800 m, CO$_2$ will be in a supercritical form [20]. In a saline aquifer, the difference in density for CO$_2$ and brine will thus be of magnitude 100, such that CO$_2$ will be the less dense fluid, and typically flow above brine, along the cap rock. The depth of the interface region between CO$_2$ and brine can be quite thin, depending on material properties and in particular the capillary pressure function. We will for simplicity assume that this interface region is a sharp interface, with the assumption of vertical equilibrium within each phase. A visualization can be seen in Figure 2.1. As CO$_2$ flows across the reservoir it will displace the current phase, brine, although not completely. Some of the brine will remain trapped in the pores, and we refer to this as the residual brine.

We now introduce the coarse scale variables we need in our sharp-interface model. First we have the coarse-scale porosity;

$$\Phi = \mathcal{C}_\Phi \phi = \int_{\xi_B}^{\xi_T} \phi \, \mathrm{d}x_3$$

As the porosity is assumed known on the fine scale, i.e., a parameter rather than a variable, we do not need a reconstruction operator.
The coarse-scale saturation is given as

$$S_\alpha = \mathcal{C}_{S_\alpha} s_\alpha = \int_{\xi_B}^{\xi_T} \phi s_\alpha \, \mathrm{d}x_3$$

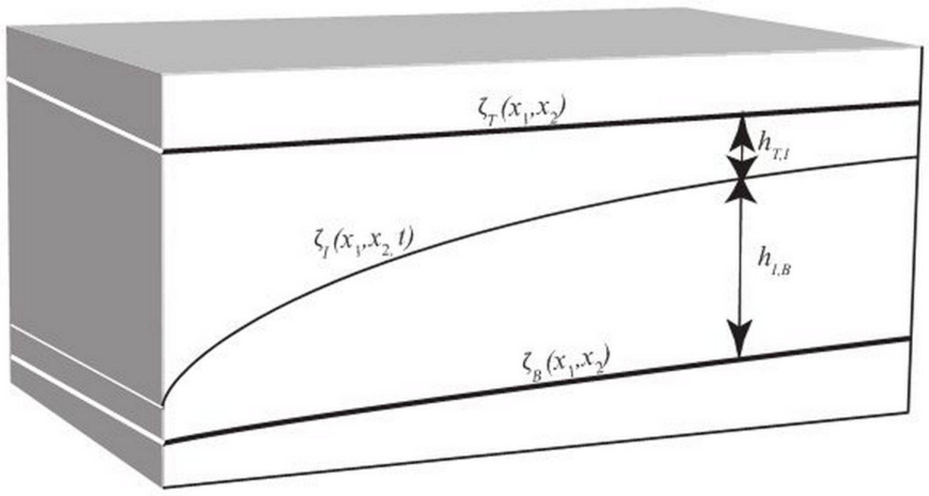


Figure 2.1: Our aquifer, with the two fluids separated by a sharp interface labeled ξ_I. CO_2 will be found in the upper layer and brine in the lower. Figure is from Nordbotten and Celia, [21].

As we can see, the coarse scale representation still give us the equation that $\sum_\alpha S_\alpha = 1$. We also vertically integrate the flux contribution;

$$\mathbf{U}_\alpha = \int_{\xi_B}^{\xi_T} \mathbf{e}_{\shortparallel}\cdot\mathbf{u}_\alpha \, \mathrm{d}x_3 = -\int_{\xi_B}^{\xi_T} \lambda_{\alpha,\shortparallel}(\widehat{s_\alpha})\mathbf{k}_{\shortparallel}(\nabla_{\shortparallel}\widehat{p_\alpha} - \rho_\alpha \mathbf{e}_{\shortparallel}\cdot\mathbf{g}) \, \mathrm{d}x_3$$

We have that

$$(\nabla\widehat{p_\alpha} - \rho_\alpha \mathbf{e}_{\shortparallel}\cdot\mathbf{g}) = \nabla_{\shortparallel} P_\alpha + \rho_\alpha(\mathbf{g}\cdot\mathbf{e})\nabla_{\shortparallel}(x_3 - \xi_P) - \rho_\alpha \mathbf{e}_{\shortparallel}\mathbf{g}$$

And by introducing

$$\mathbf{K} = \int_{\xi_B}^{\xi_T} \mathbf{k}_{\shortparallel} \, \mathrm{d}x_3$$
$$\mathbf{G} = \mathbf{e}_{\shortparallel}\mathbf{g} + (\mathbf{g}\cdot\mathbf{e}_3)\nabla_{\shortparallel}\xi_P$$
$$\mathbf{\Lambda}_\alpha(S_\alpha, \widehat{s_c^t}) = \int_{\xi_B}^{\xi_T} \lambda_{\alpha,\shortparallel}\widehat{s}_\alpha, \widehat{s_c^t})\mathbf{k}_{\shortparallel} \, \mathrm{d}x_3 \mathbf{K}^{-1}$$

We thus have the coarse scale conservation given by

$$\Phi\frac{\partial S_\alpha}{\partial t} + \nabla_{\shortparallel}\mathbf{U}_\alpha = \Upsilon_\alpha \tag{2.9}$$

or

$$\Phi\frac{\partial S}{\partial t} - \nabla_{\shortparallel}(\mathbf{\Lambda}_\alpha(S_\alpha, \widehat{s_c^t})\mathbf{K}(\nabla_{\shortparallel} P_\alpha - \rho_\alpha \mathbf{G})) = \Upsilon_\alpha$$

Now, in the sharp interface mode, the idea is that the capillary fringe is small enough to be neglected. This gives us the reconstructed saturation values:

$$\widehat{s}_\alpha = \begin{cases} s_{\alpha,B} & x_3 \leq \xi_I(\mathbf{x}) \\ s_{\alpha,T} & x_3 > \xi_I(\mathbf{x}) \end{cases}$$

We also introduce the variable h, which is the height between different interfaces, which is indicated by the subscripts. For instance, $h_{T,B} = \xi_T - \xi_B$. We can now represent the compressed saturations by means of h;

$$S_\alpha = s_{\alpha,B} + \frac{h_{T,M}}{h_{T,B}}(s_{\alpha,T} - s_{\alpha,B}) \tag{2.10}$$

This gives the coarse scale equation

$$\Phi\frac{(s_{\alpha,T} - s_{\alpha,B})}{h_{T,B}}\frac{\partial h_{T,M}}{\partial t} - \nabla_{\shortparallel}\cdot\left[\mathbf{\Lambda}_\alpha(h_{T,M})\mathbf{K}\left(\nabla_{\shortparallel} P_\alpha - \rho_\alpha \mathbf{G}\right)\right] = \Upsilon_\alpha \tag{2.11}$$

where

$$\mathbf{\Lambda}_\alpha(h_{T,M}) = \int_{\xi_B}^{\xi_T - h_{T,M}} \lambda_{\alpha,\shortparallel}(s_{\alpha,B})\mathbf{k}_{\shortparallel}\,\mathrm{d}x_3\mathbf{K}^{-1} + \int_{\xi_T - h_{T,M}}^{\xi_T} \lambda_{\alpha,\shortparallel}(s_{\alpha,T})\mathbf{k}_{\shortparallel}\,\mathrm{d}x_3\mathbf{K}^{-1}$$

Now, as we can see from (2.11), we need an expression for the phase pressure, as it is $h_{T,M}$ which is the variable of primary interest to us. We solve this by keeping in mind that $\sum_\alpha S_\alpha = 1$, and sum (2.11) for each of the phases. This gives us

$$\nabla\cdot\mathbf{U}_\Sigma = \Upsilon_\Sigma \tag{2.12}$$

(2.12) is often referred to as the *pressure equation* in literature. ((2.11) is usually referred to as the saturation or transport equation). The pressure equation can be solved with respect to $\mathbf{U}_\Sigma$ in numerous ways. However, as we aim to have a simulator implemented on a GPU, we eventually want an explicit scheme suitable for parallelization. The fact that (2.12) is an elliptic equation makes it thus less suitable for GPU implementation. There is currently ongoing research regarding parallel solvers for elliptic PDEs, see for instance Esler et al., [9], but this is outside the scope of interest for this project. So, for now, we will just assume we have solved (2.12).

In the reconstruction of the phase pressure, we chose to let P_α be given at a sampling height, see (2.8). We now choose this sampling height to align with the reservoir top, such that $\xi_P \equiv \xi_T$. This gives the expression

$$\begin{aligned} \mathbf{U}_\Sigma = &- \mathbf{\Lambda}_b \mathbf{K} \left(\nabla_{\shortparallel} P_T - \rho_c(\mathbf{g}\cdot\mathbf{e})\nabla\xi_T - \Delta_\alpha\rho(\mathbf{g}\cdot\mathbf{e})\nabla\xi_I - \rho_b \mathbf{e}_{\shortparallel}\mathbf{g}\right) \\ &- \mathbf{\Lambda}_c \mathbf{K} \left(\nabla_{\shortparallel} P_T - \rho_c(\mathbf{g}\cdot\mathbf{e})\nabla\xi_T - \rho_c \mathbf{e}_{\shortparallel}\cdot\mathbf{g}\right) \end{aligned} \tag{2.13}$$

If we then solve (2.13) for $\nabla_{\shortparallel} P_T$:

$$\begin{aligned} \nabla_{\shortparallel} P_T = &\frac{1}{\mathbf{\Lambda}_\Sigma}\big(- \mathbf{K}^{-1}\mathbf{U}_\Sigma + (\mathbf{g}\cdot\mathbf{e})[\Delta_\alpha\rho\mathbf{\Lambda}_b\nabla(\xi_T - h) + \\ &(\mathbf{\Lambda}_b + \mathbf{\Lambda}_c)\rho_c\nabla\xi_T] + (\mathbf{e}_{\shortparallel}\cdot\mathbf{g})[\rho_b\mathbf{\Lambda}_b + \rho_c\mathbf{\Lambda}_c]\big) \\ \nabla_{\shortparallel} P_T = &\frac{-\mathbf{K}^{-1}}{\mathbf{\Lambda}_\Sigma}\mathbf{U}_\Sigma + \frac{1}{\mathbf{\Lambda}_\Sigma}(\mathbf{g}\cdot\mathbf{e})\Delta_\alpha\rho\mathbf{\Lambda}_b\nabla(\xi_T - h) + \\ &\rho_c\nabla\xi_T + \frac{1}{\mathbf{\Lambda}_\Sigma}(\mathbf{e}_{\shortparallel}\cdot\mathbf{g})[\rho_b\mathbf{\Lambda}_b + \rho_c\mathbf{\Lambda}_c]\big) \end{aligned}$$

This gives the equations

$$\Phi\frac{(s_{c,T} - s_{c,B})}{h_{T,B}}\frac{\partial h_{T,M}}{\partial t} + \nabla_{\shortparallel}\cdot\mathbf{U}_c = \Upsilon_c \tag{2.14a}$$

$$\Phi\frac{(s_{b,T} - s_{b,B})}{h_{T,B}}\frac{\partial h_{T,M}}{\partial t} + \nabla_{\shortparallel}\cdot\mathbf{U}_b = \Upsilon_b \tag{2.14b}$$

where

$$\mathbf{U}_c = \frac{\mathbf{\Lambda}_c}{\mathbf{\Lambda}_\Sigma}\big(\mathbf{U}_\Sigma - \mathbf{K}\mathbf{\Lambda}_b(\mathbf{g}\cdot\mathbf{e}_3)[\Delta_\alpha\rho\nabla(\xi_T - h)] - \mathbf{K}(\mathbf{e}_{\shortparallel}\cdot\mathbf{g})\Delta_\alpha\rho\mathbf{\Lambda}_b\big) \tag{2.15a}$$

$$\mathbf{U}_b = \frac{\mathbf{\Lambda}_b}{\mathbf{\Lambda}_\Sigma}\big(\mathbf{U}_\Sigma + \mathbf{K}\mathbf{\Lambda}_c(\mathbf{g}\cdot\mathbf{e}_3)[\Delta_\alpha\rho\nabla(\xi_T - h)] + \mathbf{K}(\mathbf{e}_{\shortparallel}\cdot\mathbf{g})\Delta_\alpha\rho\mathbf{\Lambda}_c\big) \tag{2.15b}$$

2.1.3 Hysteresis

When injecting CO_2 into our system, the idea is, as previously mentioned, that the CO_2 will flow along the caprock and above the brine. However, the brine might become completely surrounded by the invading CO_2. This leads to residual brine in the pore space, and, as it will have no connecting paths to the rest of the brine, it will remain trapped and immobile. As the CO_2 plume moves, we will have a corresponding effect for the CO_2, with residual CO_2 being trapped in the pores behind the plume. A visualization can be seen in Figure 2.2.

$$\widehat{s}_\alpha \begin{cases} s_{\alpha,B} & x_3 \leq \xi_R(\mathbf{x}) \\ s_{\alpha,R} & \xi_R \leq x_3 \leq \xi_M(\mathbf{x}) \\ s_{\alpha,M} & x_3 > \xi_M(\mathbf{x}) \end{cases}$$

ξ_R is taken to be the furthest extent (over time) of the mobile region, such that

$$\xi_R = \min_{t' \in [0,t]} \xi_M(t')$$

This trapping mechanism is important, and will reduce the available pore space volume. Thus, the equation (2.14) will be written as

$$\Phi \frac{\Delta s_\alpha}{h_{T,B}} \frac{\partial h_{T,M}}{\partial t} + \nabla_{\shortparallel} \cdot \mathbf{U}_\alpha = \Upsilon_\alpha \tag{2.16}$$

where

$$\Delta s_\alpha = \begin{cases} s_{\alpha,T} - s_{\alpha,B} & \text{if} \quad h_{M,R} = 0 \\ s_{\alpha,T} - s_{\alpha,R} & \text{if} \quad h_{M,R} > 0 \end{cases}$$

2.2 The Numerical Scheme

In nonlinear conservation laws one might experience discontinuities in the solution, even though the initial data are smooth [15]. If there is a discontinuity present, the PDE (2.2) does not hold, and one will have to rely on the integral conservation law; (2.1) [15].

The finite volume methods (FVM) are numerical methods based on solving the integral form rather than the differential equation. As opposed to finite difference methods which is based on pointwise approximations at grid points, FVM approximate the total *integral* of the conserved quantity over each grid cell. This ensures that the physical quantities are preserved within the numerical domain.

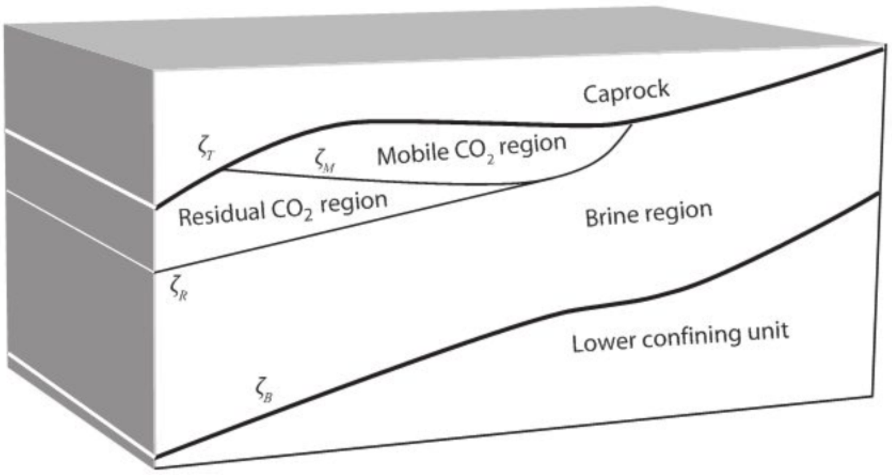


Figure 2.2: Our aquifer, with the two fluids separated by a sharp interface labeled ξ_I. Residual CO_2 will be found behind the mobile CO_2 front, and residual brine will be found in the mobile region. Picture is from [21].

The conserved variable in (2.1), namely q, is updated based on numerical flux functions that approximate the correct fluxes. We have chosen to do this by means of a first order upwind method. The results in an ordinary differential equation (ODE) per grid cell, which is solved using an explicit Euler method.

Implementing higher order schemes both in time and space will be left for further work.

2.2.1 Finite Volume Method and Spatial Discretization

We will be operating on grid cells, defined by

$$\mathcal{C}_{i,j} = \left[x_i - \frac{\Delta x}{2}, x_i + \frac{\Delta x}{2}\right] \times \left[y_j - \frac{\Delta y}{2}, y_j + \frac{\Delta y}{2}\right]$$

For any time step, denoted by n, we let Q^n_{ij} approximate the average value over cell $\mathcal{C}_{i,j}$ at time $t = t^n$:

$$Q^n_{i,j} \approx \frac{1}{\Delta x \Delta y} \int_{x_i - \frac{\Delta x}{2}}^{x_i + \frac{\Delta x}{2}} \int_{y_j - \frac{\Delta y}{2}}^{y_j + \frac{\Delta y}{2}} q(x, y, t_n)\,\mathrm{d}y\,\mathrm{d}x = \int_{\mathcal{C}_{i,j}} q(x, y, t_n)\,\mathrm{d}\mathbf{x}$$

If we now consider (2.14a), we abbreviate the notation and write it in a more compact way:

$$\Phi\frac{\partial h}{\partial t} + \nabla\cdot\mathbf{f}(h) = \Upsilon_c \tag{2.17}$$

So, when integrating over (2.17), the averaged value in each cell is approximated by

$$\int_{\mathcal{C}_{i,j}} \frac{\partial\Phi(\mathbf{x},t)h(\mathbf{x},t)}{\partial t}\,\mathrm{d}\mathbf{x} + \int_{\mathcal{C}_{i,j}} \nabla\cdot\mathbf{f}(h,\mathbf{x},t)\,\mathrm{d}\mathbf{x} = \int_{\mathcal{C}_{i,j}} \Upsilon_c\,\mathrm{d}\mathbf{x} \tag{2.18}$$

Now, by letting $F(h,\mathbf{x},t) = \frac{\partial}{\partial x}\mathbf{f}(h,\mathbf{x},t)$, and $G(h,\mathbf{x},t) = \frac{\partial}{\partial y}\mathbf{f}(h,\mathbf{x},t)$, a spatial discretized version of our equation on a Cartesian grid can be written as

$$\frac{\partial\Phi_{i,j}h_{i,j}}{\partial t} = -\big[F(h_{i+1/2,j}) - F(h_{i-1/2,j)}\big] - \big[G(h_{i,j+1/2}) - G(h_{i,j-1/2})\big] + \Upsilon_{i,j}$$

where we let $F(h_{i,j+1})$ etc. be approximations of the flux over the different faces:

$$F(h_{i,j+1}) \approx \frac{1}{\Delta y}\int_{y_j-1/2}^{y_j+1/2} F(h(x_{i+1/2},y,t))\,\mathrm{d}y \tag{2.19}$$

Upwinding Scheme

We will use face upwind scheme to approximate the flux over each face. An upwind method will assume that the flow of each phase can only be in one direction, when considering the direction over one cell.

We will determine the upwind direction independently for each phase by considering the coarse scale phase velocities from (2.15). The mobilities of each phase will be taken from the upwind cell. If we consider the southern face, this gives us:

$$\mathbf{\Lambda}^c_{i,j-1/2} = \begin{cases} \mathbf{\Lambda}^c_{i,j} & \text{if } \mathbf{U}_\Sigma > 0 \quad \text{and } -\big[\nabla(\xi_T - h)(\mathbf{g}\cdot\mathbf{e}_3) + (\mathbf{e}_{\shortparallel}\cdot\mathbf{g})\big] > 0 \\ & \text{or if } \quad \mathbf{U}_\Sigma - \mathbf{\Lambda}^b_{i-1/2,j}\mathbf{K}\Delta_\alpha\rho[\nabla(\xi_T - h)(\mathbf{g}\cdot\mathbf{e}_3) + (\mathbf{e}_{\shortparallel}\cdot\mathbf{g})] > 0 \\ \mathbf{\Lambda}^c_{i,j-1} & \text{otherwise} \end{cases}$$

$$\mathbf{\Lambda}^b_{i,j-1/2} = \begin{cases} \mathbf{\Lambda}^b_{i,j} & \text{if } \mathbf{U}_\Sigma > 0 \quad \text{and } \big[\nabla(\xi_T - h)(\mathbf{g}\cdot\mathbf{e}_3) + (\mathbf{e}_{\shortparallel}\cdot\mathbf{g})\big] > 0 \\ & \text{or if} \quad \mathbf{U}_\Sigma + \mathbf{\Lambda}^c_{i-1/2,j}\mathbf{K}\Delta_\alpha\rho[\nabla(\xi_T - h)(\mathbf{g}\cdot\mathbf{e}_3) + (\mathbf{e}_{\shortparallel}\cdot\mathbf{g})] > 0 \\ \mathbf{\Lambda}^b_{i,j-1} & \text{otherwise} \end{cases}$$

The mobilities over the other faces are found similarly.

For each cell, the source term, $\Upsilon_{i,j}$ consists of what is produced in that cell and what flows out, which again is determined by the *fractional flow* function. The fractional flow for CO_2, $fw(s_c)$ can be written as

$$fw(s_c) = \frac{\mathbf{\Lambda}_c}{\mathbf{\Lambda}_c + \mathbf{\Lambda}_b}$$

The source term is therefore discretized as $\Upsilon^n_{i,j} = \max(q, 0) + \min(q, 0) \cdot fw(s_c)$.

Thus, to sum up, our final discretized scheme can be written as:

$$h^{n+1}_{i,j} = h^n_{i,j} - \frac{\Delta t}{\Phi \Delta x \Delta y}\big[F(h^n_{i+1/2,j}) - F(h^n_{i-1/2,j)} + G(h^n_{i,j+1/2}) - G(h^n_{i,j-1/2}) + \Upsilon^n_{i,j}\big] \tag{2.20}$$

2.2.2 CFL Condition and Time Discretization

We integrate the spatially discretized equation (2.20) with respect to time, using a first order explicit Euler scheme. If we assume that the rock is incompressible, this gives us

$$\begin{aligned} h^{n+1}_{i,j} &= h^n_{i,j} - \frac{1}{\Phi_{i,j}\Delta x \Delta y}\Delta t \Big[\big[F(h^n_{i+1/2,j}) - F(h^n_{i-1/2,j})\big] - \qquad (2.21) \\ &\qquad \big[G(h^n_{i,j+1/2}) - G(h^n_{i,j-1/2})\big] - Q^n_{i,j}\Big] \\ &= h^n_{i,j} + \Delta t R(h^n)_{i,j} \qquad (2.22) \end{aligned}$$

Explicit schemes are suitable for parallelization, but it does impose a restriction on the time step, Δt. This is known as the CFL Condition. It was in 1928 that Courant, Friedrichs and Lewy stated a necessary, (however not always sufficient), condition for the stability of any explicit method [15]:

CFL Condition *A numerical method can be convergent only if its numerical domain of dependence contains the true domain of dependence of the PDE,* *at least in the limit as Δt and Δx go to zero.*

In our simulator we have implemented two ways of calculating the CFL conditioned time step. The first is implemented in the MRST simulator [1, 16], and we will thus refer to this method as *MRST time stepping*. This is, as we will see, a condition which is only saturation dependent through the velocities $\mathbf{U}_\Sigma$. Therefore, it needs only be modified when we compute a new solution to the pressure equation. (The pressure equation, being solved implicitly, has no time step restriction associated with it. Therefore, we will typically require to update this more seldom than the transport equation [12].)

The second way of calculating the time step is based upon the work of Coats [8]. This requires more computations per iteration than the MRST method, as this depends on the phase velocities $\mathbf{U}_c$, and $\mathbf{U}_b$. However, when basing our time step restriction on the current state of our system, this will result in a time step closer to the largest one permissible. With a larger time step, we will require fewer time steps, but it might also result in less numerical diffusion.

Which method best suited in our simulator will depend on how much the increased computational cost in Coats will affect the computational time per iteration, and how big the reduction in number of time steps is.

CFL Condition, MRST

The well known CFL Condition for IMPES is given as, [8]:

$$\frac{\Theta_i \Delta t}{\Phi \Delta x \Delta y} \leq 1$$

Here, Θ represents some function of rates and reservoir and fluid properties.

The changes in the flux function will be determined by the changes in the gravity driven or the pressure driven velocities. The change in the pressure driven velocity can be expressed as:

$$v_p' = \left[\mathbf{U}_\Sigma \frac{\partial}{\partial s} f_w\right] + q,$$

while in the gravity driven velocity this can be expressed as:

$$v_g' = \left[\frac{\partial}{\partial s}\left(f_w \mathbf{\Lambda}_b\right) \frac{\partial}{\partial x} \xi_T\right] \mathbf{K} \Delta\rho \mathbf{g} - f_w \mathbf{\Lambda}_b \mathbf{K} \Delta\rho \mathbf{g} \left[\frac{\partial}{\partial x} h_{T,M}\right] \tag{2.23}$$

This results in the CFL condition:

$$\Delta t \leq \frac{1}{2} \min \left\{ \frac{\Phi \Delta x \Delta y}{\max \left\{ v'_{p,x} + v'_{p,y}, v'_{g,x} + v'_{g,y} \right\}} \right\} \tag{2.24}$$

We will briefly make some comments on how we have implemented this: First of all, the parabolic part (the last term of equation (2.23)) can be quite tedious to implement, and we have not done so. Therefore, if the CO_2 plume is very steep, and this term dominates, one will have to use Coats time stepping, see the section below.

When disregarding the parabolic term, we see that the only saturation dependency we have, is for the derivative of the fractional flow, or the global velocity computed in the pressure equation; $\mathbf{U}_\Sigma$. For the derivative of the fractional flow; f_w with respect to saturation, we do as implemented in MRST [1, 16]: We use a maximum of the fractional flow function to estimate a safe time step. This is conservative, but then the time step restriction needs only be updated on the first iteration following an update of the pressure equation.

CFL Condition, Coats

In general, estimate the theoretical, optimal time step is difficult for nonlinear equations. We thus follow the more practical approach suggested by Coats [8]. This means that we apply the analysis using constant coefficients equal to local values and consider each subregion separately. Finally we choose a globally dominant time step based on the over all lowest restriction.

For simplicity in notation, we will only derive the result for the 1D case, but the 2D result will be discussed at the end of the section.

The equation expressing conservation of mass of CO_2 in cell i in 1D is given by

$$\frac{\Phi_i}{\Delta t}(h_i^{n+1} - h_i) = F_{i_1}(h_{i-1}^n, h_i^n) - F_i(h_i^n, h_{i+1}^n) \tag{2.25}$$

We now introduce ϵ to denote the error between the exact solution and our numerical solution at time $t = t_0 + n\Delta t$, such that $\epsilon_i = h_i^n - \tilde{h_i^n}$. Here, $\tilde{h_i^n}$ denotes the exact solution. We now express (2.25) twice, once with h_i^n and once with $\tilde{h_i^n}$. The difference between them can be written as:

$$\begin{aligned}
\frac{\Phi_i}{\Delta t}(\epsilon_i^{n+1} - \epsilon_i^n) &= \big[F_{i-1}(h_{i-1}^n, h_i^n) - F_{i-1}(\tilde{h_{i-1}^n}, \tilde{h_i^n})\big] \\
&\quad - \big[F_i(h_i^n, h_{i+1}^n) - F_i(\tilde{h_i^n}, \tilde{h_{i+1}^n})\big] \\
&= \big[F_{i-1}(h_{i-1}^n, h_i^n) - F_{i-1}(h_{i-1}^n - \epsilon_{i-1}^n, h_i^n - \epsilon_i^n)\big] \\
&\quad - \big[F_i(h_i^n, h_{i+1}^n) - F_i(h_i^n - \epsilon_i^n, h_{i+1}^n - \epsilon_{i+1}^n)\big]
\end{aligned}$$

Using the first Taylor terms, this can be written as

$$\begin{aligned}
\frac{\Phi_i}{\Delta t}(\epsilon_i^{n+1} - \epsilon_i^n) &= \epsilon_{i-1}^n \frac{\partial F_{i-1}}{\partial h_{i-1}} - \epsilon_i^n \Big(\frac{\partial F_{i-1}}{\partial h_{i-1}} + \frac{\partial F_i}{\partial h_i}\Big) - \epsilon_{i+1}^n \frac{\partial F_i}{\partial h_{i+1}} \\
&= a_i \epsilon_{i+1}^n - b_i \epsilon_i^n + c_i \epsilon_{i-1}^n
\end{aligned}$$

By using Von Neumann stability analysis, (see appendix in the article by Coats [8]), the resulting stability condition is

$$\frac{\Delta t}{\Phi}(a_i + b_i + c_i) \le 2a_i + c_i \le b_i$$

As $a_i + c_i = b_i$, this simplifies to

$$\frac{\Delta t}{\Phi}\Big(\frac{\partial F_i}{\partial h_i^n} - \frac{\partial F_i}{\partial h_{i+1}^n}\Big) \le 1 \tag{2.26}$$

For the calculation of $\frac{\partial F_i}{\partial h_i} - \frac{\partial F_i}{\partial h_{i+1}}$, we differensiate equation (2.15). There are four potential outcomes, as we have two cases of countercurrent and two cases of cocurrent flows. (Brine and CO_2 can either flow in the same direction or in opposite directions.) We will do the calculations using the upstream weighting, and for the cause of an example, assuming $\Lambda_b = \Lambda_b(h_{i+1})$, $\Lambda_c = \Lambda_c(h_i)$. Thus, to simplify notation, we only use an asterix on the phase mobilities to represent differentiated with respect to h_{i+1} when differentiating Λ_b, and with respect to h_i when differentiating Λ_c. We have also introduced $\Lambda_t = \Lambda_b + \Lambda_c$.

$$\begin{aligned}
\frac{\partial F_i}{\partial h_i} - \frac{\partial F_i}{\partial h_{i+1}} &= -\mathbf{K}\Delta\rho g\Big[\frac{\Lambda_b'\Lambda_c\Lambda_t - \Lambda_b\Lambda_c\Lambda_b'}{\Lambda_t^2} - \frac{\Lambda_b\Lambda_c'\Lambda_t - \Lambda_b\Lambda_c\Lambda_c'}{\Lambda_t^2}\Big](\nabla\xi_t - \nabla h) + ... \\
&= -\mathbf{K}\Delta\rho g\big[\Lambda_b'\frac{\Lambda_c^2}{\Lambda_t^2} - \Lambda_c'\frac{\Lambda_b^2}{\Lambda_t^2}\big](\nabla\xi_t - \nabla h)\mathbf{K}\Delta\rho g\Lambda_b\frac{\Lambda_c}{\Lambda_t}\frac{2}{\Delta x}
\end{aligned}$$

The same procedure can be done for the other countercurrent case, as well for the two cocurrent cases, but the final result will be the same. After some rewriting, the expression can be written as

$$\frac{\partial F_i}{\partial h_i} - \frac{\partial F_i}{\partial h_{i+1}} = \frac{\Lambda_c}{\Lambda_b \Lambda_t} |\mathbf{U}_{b,i}| \Lambda_b' + \frac{\Lambda_b}{\Lambda_c \Lambda_t} |\mathbf{U}_{c,i}| \Lambda_c' + \mathbf{K} \Delta\rho g \Lambda_b \frac{\Lambda_c}{\Lambda_t} \frac{2}{\Delta x} \tag{2.27}$$

The stability analysis outlined in this section is as mentioned only for one dimension. However, as discussed in [8], when advancing to two (or three) dimensions, one only has to add an additional term of identical form for each additional dimension.

To conclude on this chapter, we now have the mathematical background we need for our simulator: We have the governing equations, the numerical scheme and the time step restriction. The actual implementation will be discussed in Chapter 4.

Chapter 3

GPU Programming Concepts

In this chapter we will describe the architecture of NVIDIA GPUs, and introduce the CUDA programming model.

Increasing the performance of a computer has been a goal ever since the computers were introduced. Traditionally, increasing a processor's clock speed, has been considered one of the main ways in achieving this. However, due to heat and power restrictions, manufacturers also had to consider other means. [26].

Late in the 80's, in order to release some of the workload of the CPUs, special processors were introduced to take care of the continuously increasing amount of graphics operations. In 1999, NVIDIA introduced their first graphical processing unit (GPU), the GeForce 256 graphics card. While GPUs originally were supposed to do the required calculations for a better visual experience, they would also turn out to be useful in non-graphical, scientific applications. As the multi-core CPUs originally are constructed to perform simultaneous execution of multiple applications, they are designed to execute a single thread very fast, and will spend much effort on complex logic. GPUs on the other hand, rather gain performance benefits by executing multiple threads, and are optimized to do fast computations instead of complex logic. [6], [25]. As a result of this, algorithms intended for non-graphical applications, can benefit from the architecture of the GPUs, if they are suitable for parallelization.

As one started to exploit the non-graphical use of the GPU, one had to use graphic libraries, such as OpenGL in order to have calculations done [25].

But then, in 2007, NVIDIA released CUDA-SDK , a parallel computing language dedicated to use on GPUs. Based around C, C++ and Fortran, developers could now proceed without use of the traditional graphic processing languages.

3.1 GPU Architecture

Optimizing GPU accelerated code is not a trivial task. The performance will highly depend on memory transfer, and how many threads of execution one choose to run at the same time. This in turn will depend on the underlying architecture of the GPU. The following description is based on specifications for the Fermi Architecture. For a more detailed description, we suggest reading NVIDIAs own paper on the architecture [22], and "GPU programming strategies and trends in GPU computing" by Brodtkorb et al. [5] which the following section is a short summary off. Or, for a description of the newer GPU architecture available from NVIDIA, we suggest reading about the Kepler Architecture [24].

In a Fermi-based architecture, one can find up to 512 accelerator cores, called CUDA cores. Each of these cores have an integer arithmetic logic unit and a floating point unit, executing one integer or floating point instruction per clock cycle. These 512 CUDA cores are divided into 16 streaming multiprocessors (SMs), giving each multiprocessor a total of 32 cores each. The multiprocessors also have 16 load / store units each, making it possible to calculate source and destination addresses for 16 threads per clock. Transcendental instructions, such as sine and square roots, are executed by four Special Functions Units, which execute one instruction per thread, per clock.

Each SM operates in a SIMD - single instruction multiple data, kind of way. They execute 32 threads (a *warp*) simultaneously. Multiple warps can be active at the same time, thus reducing latency from computations or memory transfer.

A visualization of the Fermi architecture can be found in Figure 3.1.

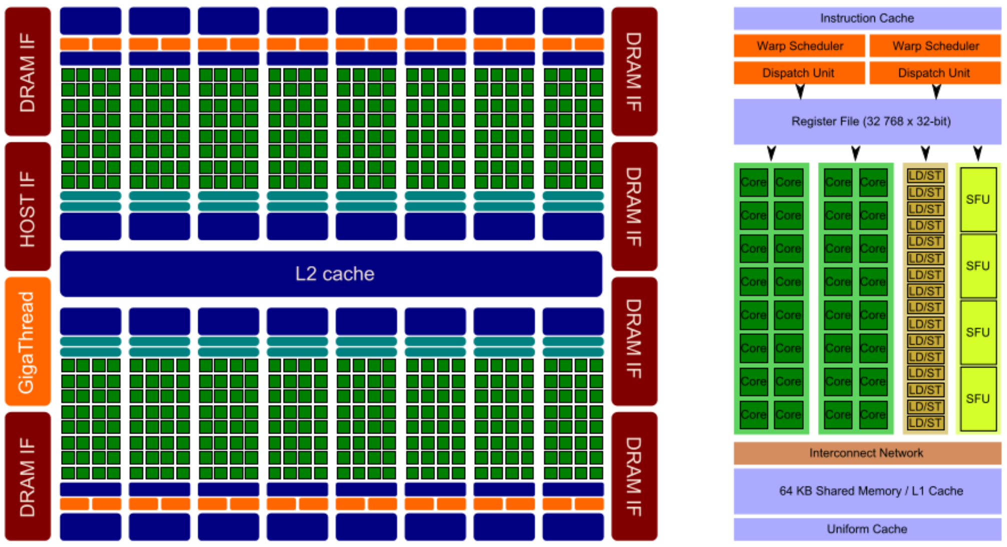


Figure 3.1: To the left we see the Fermi architecture with its 16 multiprocessors surrounding the L2 cache available to all the cores. The green represents the execution units while the light blue represents local register file and L1 cache. To the right we see *one* of the SMs contained in the architecture on the left. The Figure is from Brodtkorb et al., [5].

Memory Organization

Memory units on GPUs are divided into three areas; registers, shared memory and global memory. Each thread within a multiprocessor will have it's own private register, which are the fastest memory unit out of the three. Shared memory, being the second fastest memory unit, is accessible to all threads within one multiprocessor. Lastly, the slowest memory type is the main memory of the GPU, also referred to as *global memory*. A visualization of the memory hierarchy can be found in Figure 3.2.
The Fermi Architecture also include L1 and L2 caches, which operate in a way comparable to the CPU caches, but it also have caches related more to traditional graphics operations.

3.2 CUDA Programming Model

With the introduction of CUDA - "compute unified device architecture", executing programs on the GPU became possible without knowledge of graphical

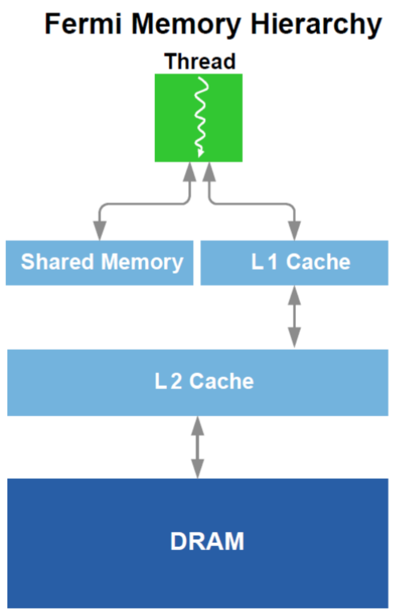


Figure 3.2: The Fermi architecture memory hierarchy. The picture is from NVIDIAs Whitepaper, [22]

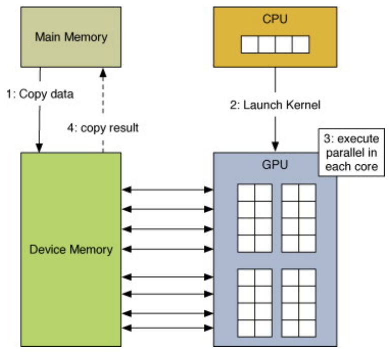


Figure 3.3: The CUDA processing flow. Picture is from Milo et al. [19].

programming concepts. The CUDA processing flow can be seen in Figure 3.3. And as the figure shows, the main idea is to initialize the data on the host (the CPU), copy the data to the device (the GPU), execute the GPU program, and copy the data back to the host. A function running on the GPU will be referred to as a *kernel* [25]. During kernel execution on the GPU, the CPU can continue doing its own computations, and then synchronize with the GPU as results are needed.

The CUDA programming model organize the threads in a hierarchy; a specified number of threads are gathered in *blocks*, which again are gathered in *grids*, see Figure 3.4 [25]. When executing a kernel, you launch it on one grid at the time, thus letting each block be handled by *one* single multiprocessor. This provides the threads within each block access to the same shared memory, as discussed in the previous section. To take advantage of the fact that the SMs can execute one warp at the time, we would therefor want the number of threads within each block to be a multiple of 32.

As indicated in Figure 3.4, each thread is assigned a unique thread id within a block, and each block is given a unique block id within the grid. Thus making it possible for threads executing the same code to work on different data.

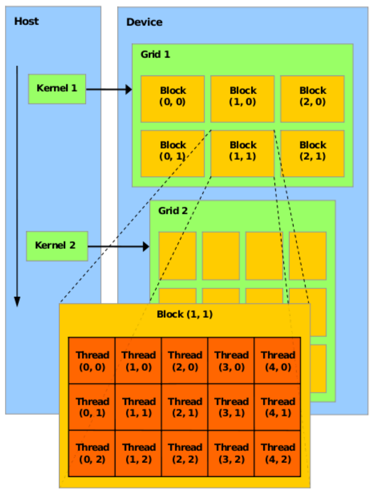


Figure 3.4: The CUDA thread hierarchy. Each grid consists of multiple blocks, which again consists of multiple threads. Each *block* is executed on a streaming processor, making it possible for threads within the same block to access the same shared data.

When optimizing the performance for code run on GPU, all of the parameters mentioned above have to be taken into account. If frequently used data is kept in shared memory, this reduces the cost of memory transfer. When deciding on number of warps per streaming multiprocessor, we want to keep the number as high as available register and shared memory allows [7].

Chapter 4

A CO_2 Migration Simulator on the GPU

We will in this chapter present our CO_2 simulator on the GPU. The first sections give an overview of how we have implemented the simulator. We describe the program flow; the C++ interface and the CUDA kernels. Following this we present the results. This includes verifying our results in comparison with the MRST simulator, but we also compare the two different dynamic time step approaches. (Coats time stepping and MRST time stepping, see Section 2.2.2.) This is followed by an analysis of performance when using a GPU. We describe the different parameters we found to affect the speed-up, including grid size, block configuration etc.

For simplicity, we state the equation we aim to solve once again:

$$\Phi \frac{\Delta s_c}{h_{T,B}} \frac{\partial h_{T,M}}{\partial t} + \nabla_{\shortparallel} \cdot \mathbf{U}_c = \Upsilon_c \tag{4.1}$$

where we recall that $\mathbf{U}_c$, the phase velocity of CO_2, also includes the term $\mathbf{U}_\Sigma$, which is the solution from the *pressure* equation. As previously mentioned, solving the elliptic pressure equation is outside the scope of interest in this thesis and we leave this for further work. Thus we rely on MRST to get this variable.

It has been shown by Ligaarden and Nilsen [17] that when using a Vertical-Equilibrium model, there is a stronger decoupling between the pressure equation and the transport equation for the migration phase as opposed to a 3D model. We have therefore implemented support for (4.1) where we neglect the contribution from the pressure equation. This will simplify the computations, but also result in us reading fewer variables from global memory. The impact this has on performance is further discussed in Section 4.4.

We also have the discretized scheme:

$$\begin{aligned} h_{i,j}^{n+1} &= h_{i,j}^{n} - \frac{\Delta t}{\Phi \Delta x \Delta y}\left[F(h_{i+1/2,j}^{n}) - F(h_{i-1/2,j)}^{n}) + G(h_{i,j+1/2}^{n}) - G(h_{i,j-1/2}^{n}) + \Upsilon_{i,j}^{n}\right] \\ &= h_{i,j}^{n} + \frac{\Delta t}{\Phi \Delta x \Delta y} R(h)_{i,j}^{n} \end{aligned} \tag{4.2}$$

On each time step, Δt, is found by

$$\Delta t \leq \frac{r}{\Phi \Delta x \Delta y}$$

where the CFL restriction r is found using either the approach of Coats or MRST.

4.1 Implementation

Our simulator is based on the work of Brodtkorb et al. [6, 7], and is written using C++ and NVIDIA CUDA [23]. The C++ code handles the initialization process and data transfer, but all of the computations are taken care of solely by the CUDA kernels on the GPU [6, 7]. We can also store the results from our simulator using NetCDF [2].

4.1.1 C++ Interface

Our C++ interface is responsible for all data allocation and deallocation, as well initialization and data transfer between the CPU and GPU [6, 7]. It is also the CPU which invokes the CUDA kernels.

A visualization of the program flow can be seen in Figure 4.1. After initializing the simulator class, we first check whether to solve the pressure equation or not. (As previously mentioned, this does not have to be solved on every iteration, [12].) We then enter the step function. This is where all of the computations are performed, i.e., the CUDA kernels are invoked. As we use

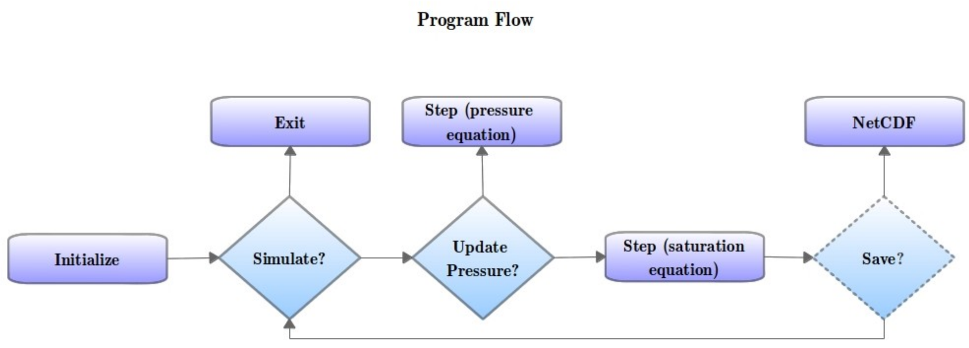


Figure 4.1: Our program flow. Initialization is done on the CPU, while the step function run one time step on the GPU. We can also exit the simulation loop and visualize using netCDF. Picture is a modified version of the one found in [7].

a dynamic time step, we cannot know how many time steps to perform in advance. Hence, we check after each iteration how far in the process we have gotten. This also gives us the opportunity to write output data to NetCDF files if we want to.

Our simulator will always assume a rectangular domain. This makes the domain decomposition, see Figure 4.4, much easier. For realistic cases however, this is not necessarily the case. Therefore, a parameter indicating whether a cell is within the actual domain or not is needed. A visualization can be seen in Figure 4.2. We have chosen to indicate cells outside the boundary by a negative height of the reservoir. This approach reduces the amount of memory we have to read from global, as opposed to storing the parameter as an extra variable.

Every value stored is in single precision, meaning that data transfer and arithmetic operations will execute approximately twice as fast [7]. For a detailed description of how this affects the numerical results compared to MRST which use double precision, see Section 4.3.1.

4.2 GPU Implementation

In Figure 4.3, we show the CUDA kernels involved in the step function. When simulating using Coats' time stepping, the kernels are executed in the

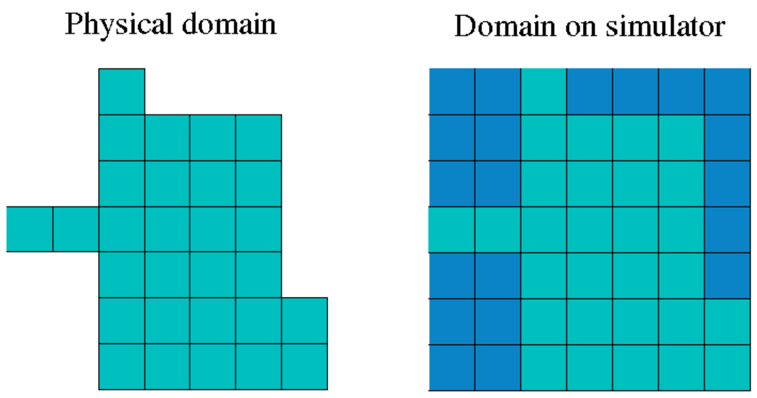


Figure 4.2: To the left we have a non-rectangular domain, which on the right hand side has been fitted to a rectangular domain on the simulator. The cells outside the original domain are indicated with a negative reservoir height.

order visualized in a), while for the MRST approach they are executed in the order visualized in b). For Coats time stepping, we see that we first of all do the flux calculations, before we find the maximum time step allowed, based on the CFL condition. Then we move on to performing the Euler time step, and finally we impose the boundary conditions. For the MRST approach on the other hand, we only need to calculate the maximum time step on the *first* iteration following an update from the pressure equation. Thus, whenever this is *not* the case, the arithmetics involved are simpler.

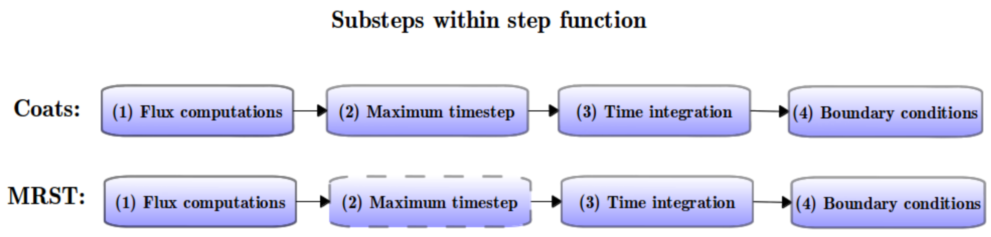


Figure 4.3: The CUDA kernels in the step function from Figure 4.1. a) shows the order of the kernels when simulating with the dynamic time step using Coats' approach, while b) shows the order when simulating using the approach of MRST. Notice the dotted line in b), indicating that this kernel is invoked only on the first iteration following the pressure equation.

Before we launch our kernels, we do a domain decomposition, as illustrated by Figure 4.4. Our global domain is decomposed into blocks which can be executed independently. Since we are in need of neighbouring cells when computing the flux over the interfaces, the blocks each have local ghost cells. These overlap with the neighbouring, local boundary cells of the other blocks. As opposed to the original simulator [6,7], we have a first order scheme, both in time and space. When decomposing our domain, we therefore need only two global ghost cells in each direction, versus four in the original simulator. This gives us a larger amount of available shared memory per block, and also a reduction in number of cells we have to process.

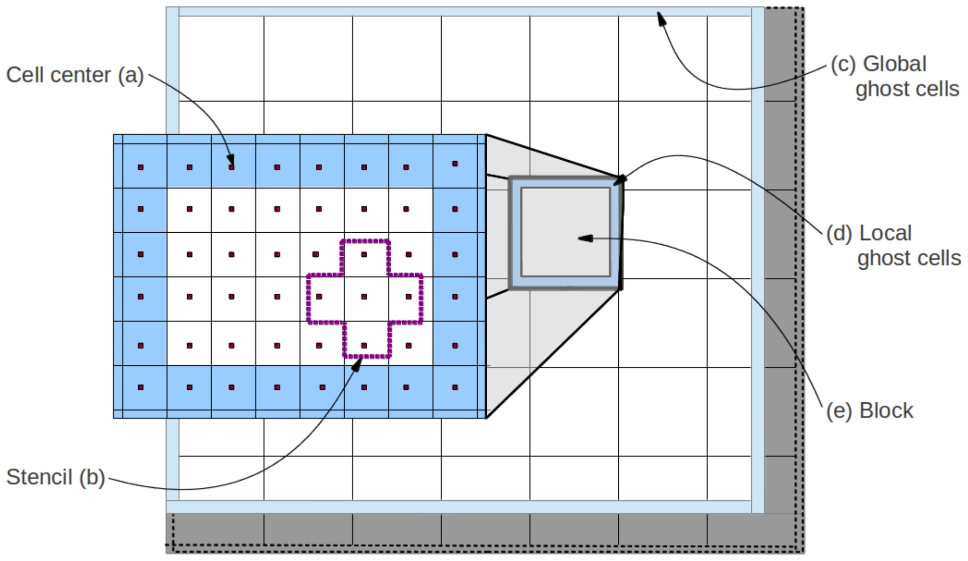


Figure 4.4: Our domain decomposition. The global domain is partitioned into blocks (e). They each have local ghost cells (d), to make sure the stencil (b) kan access cell information from other blocks. All of our variables are stored in cell centers (a), which the interface flux is calculated from.

In the following, we will describe the implementation of our kernels. As the flux kernel is the most expensive one, see Table 4.1, this is described a bit more thorough. All of the kernels are also based upon the work by Brodtkorb et al., [7].

Flux Computation Kernel

This is the kernel where we compute the flux contributions F and G from equation (4.2). As in the original simulator, we start by reading all the required variables from global to shared memory. This includes height and historical maximum height of CO_2, the reservoir properties; topography, height, porosity, permeability and information regarding well(s), along with the fluxes computed in the pressure equation - namely $\mathbf{U}_\Sigma$. $\mathbf{U}_\Sigma$ have contributions for both the x- and the y-direction. We also need to store F and G in shared memory. However, as shared memory is a limited resource we have chosen to store F and G in $\mathbf{U}_{\Sigma,x}$ and $\mathbf{U}_{\Sigma,y}$ respectively, *after* these have been used in their required computations. We only have to make sure we synchronize the threads. This means that within a block of threads, none of the threads can do any further computations before they have *all* reached the synchronization spot in the kernel. If not, we would (possibly), read and write simultaneously to the same variable. It complicates the kernel, but allows us to use less of the shared memory. This again allows each streaming multiprocessor to keep more blocks active at the same time, which improves the performance. For an overview on the effects of this approach see Section 4.4.2.

Before writing the result back to global memory, F and G are added together into R, to reduce the memory transfer [7]. In case of Coats time stepping, or whenever needed by MRST time stepping, the flux kernels also compute the CFL restriction - r, on each face. We then use shared memory reduction to find the minimum r within each block. This results in that we need only to write one value for r per block to global memory. Also, the maximum time step kernel need only check one value per block.

Maximum Time Step Kernel

The maximum time step kernel is a simple kernel very similar to the one found in the Shallow Water Simulator [6, 7]. This kernel consists of only one block, with n threads. Depending on the size of the grid, the flux kernels may consist of hundreds of blocks. Now, each thread within this single block strides through the dataset containing r. Therefore, thread t_0 considers r from the flux kernel blocks $t_0 + kn$, thread t_1 considers r from blocks $t_1 + kn$ etc. Once all the variables has been read into shared memory, we use shared memory reduction to find the minimum r across all the n threads.

Finally, Δt is computed based on the restriction in r. If we simulate during injection, Δt is compared to the maximum value we can have before we need to solve the pressure equation again.

Time Integration Kernel

In this kernel we perform the time integration based on the explicit Euler method. Here we are not limited by shared memory, and we do not need any local ghost cells. This is a result of us storing R for each cell in the flux kernels, as opposed to storing F and G separately [6, 7].

We read both the height and historical height of CO_2, along with R and Δt into per-thread register. They are then updated, before the height and historical height is written back to global memory.

Boundary Condition Kernel

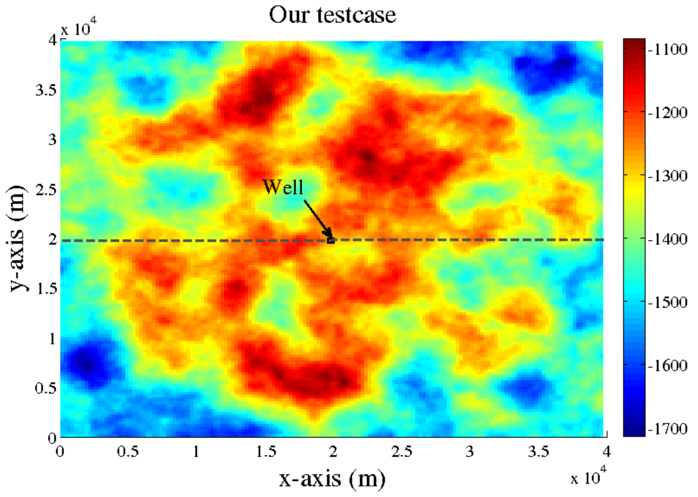
For the boundary conditions we have used global ghost cells, see Figure 4.4. We have a first order scheme, and therefore we require one global ghost cell in each direction. As we run this kernel at the end of each full time step, we do not need to treat the boundary cells any different than other cells in the flux computation kernel [6, 7]. We have focused on implementing a no-flux boundary condition, and we do this simply by mirroring the values in the neighbouring cell.

In the case of a non-rectangular domain, we have as previously mentioned, indicated the boundary by a negative height of the reservoir, see Figure 4.2. In this case, the fluxes F and / or G is always set equal to zero, but then this is taken care of in the flux kernels.

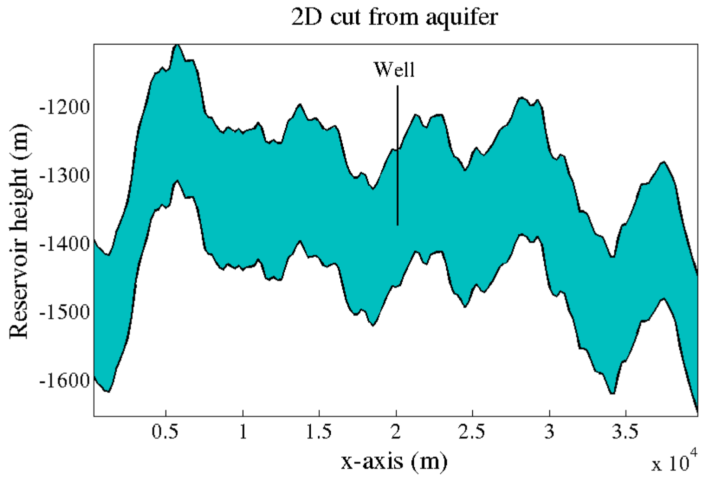
4.3 Numerical Results

In this section we consider the numerical results from our simulator. We start by comparing our results with the MRST simulator. Following this we compare the two different time step approaches.

4.3.1 Verification of Simulator



(a)



(b)

Figure 4.5: The test case for Section 4.3.1. a) Shows the reservoir seen from above, with the well indicated. The stapled line shows where the figure in b) is taken from. The case is from "runSlopingAquiferBig.m" in MRST, [1, 16]

Before anything else, we want to verify that our simulator produces the correct results. We compare our results to the results provided by MRST. However, while MRST uses double precision, we run our simulator using single precision. As previously mentioned, when using single precision on the GPU, data transfer and arithmetic operations will execute roughly twice as fast [7]. As our aim has been increased performance, we have thus chosen this approach.

The test case we have chosen is a sloping aquifer.[1] This test case consists of 160 x 160 cells, where each cell spans 250 m in every direction. A visualization can be seen in Figure 4.5. The upper figure shows the domain seen from above, while in the lower one we show a 2D cut from the side. This cuts through the axis where the well is located.

In Figure 4.6 we show how CO_2 migrates across the reservoir. First during 100 years of injection, and secondly during 100 years after injection has ended. Especially for the post-injection case, we can see how the structure of the reservoir traps the CO_2 .

Regarding the accuracy, we first start off by discussing the results in Figure 4.8. This plot shows the *maximum* error in height of CO_2 between MRST and the GPU Simulator. We found this error to be located in areas where the two velocities which contributes to R; gravity driven and pressure driven, were close in magnitude - not in direction. This produces an error which is more visible when simulating with single precision as opposed to double precision. Another factor which is also affecting this, is that when we update the height of CO_2, we compute $\frac{\Delta t}{\Phi \Delta x \Delta y} R$. However, since we need to update the pressure after a given period of time, we require that $\Delta t = \min\{\Delta t, \text{time until pressure is updated}\}$. If Δt then is small, this will also result in a difference compared to the double precision simulator.

Even though the maximum error might appear big, note that in Figure 4.7 that after 100 years of injection, the cells actually containing CO_2 are the same in both simulators. This shows that even though there are differences, (in this case, never larger than 0.4 m) the *behavior* of the simulators are the same.

To conclude, we see that when using single precision to improve the performance, we do loose accuracy in comparison with using double precision. If one is to simulate CO_2 storage, one would therefore need to consider whether this accuracy loss is significant when one take into account the simplifications made. It is also worth mentioning that for newer GPU architecture, Fermi, there has been implemented the FMA (fused multiply-add instruction) for single precision [22]. This reduces the error when doing a multiply and add instruction, by having a single rounding step instead of two separately, which is

[1] See the example "runSlopingAquiferBig.m" in MRST, [1,16]

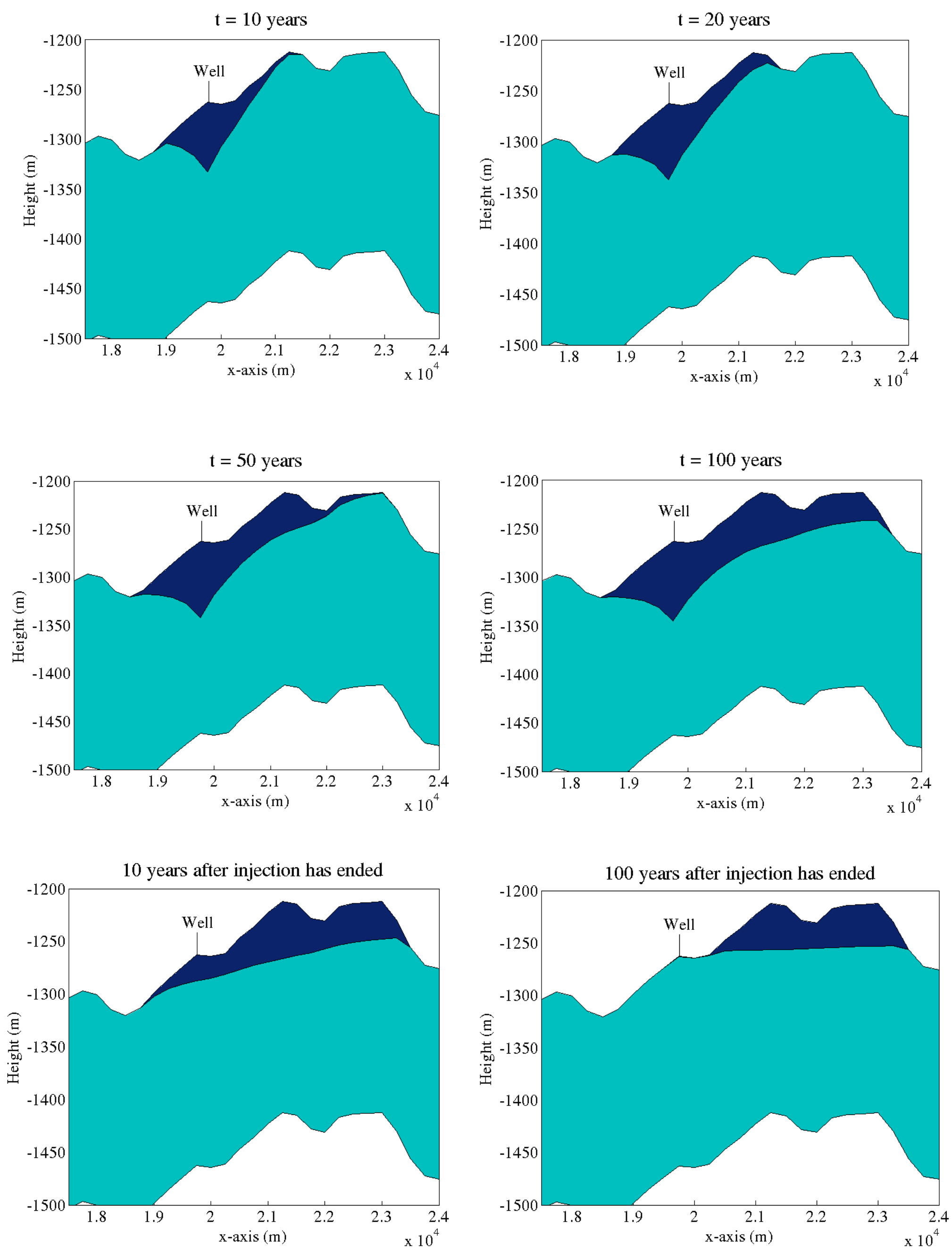
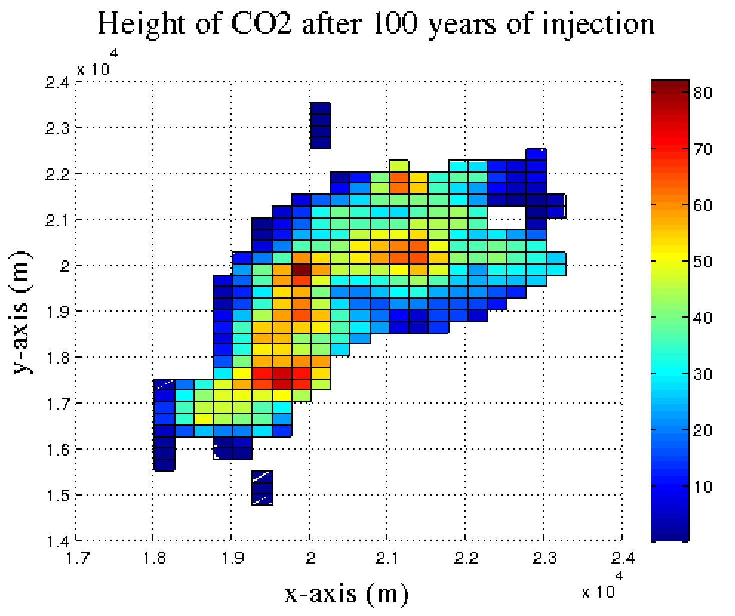


Figure 4.6: A 2D cut from the test case in Figure 4.5, during 100 years of injection and 100 years of migration. The green area is the aquifer, initially filled with brine, while the blue area is the CO_2 which we can see migrates across the reservoir.



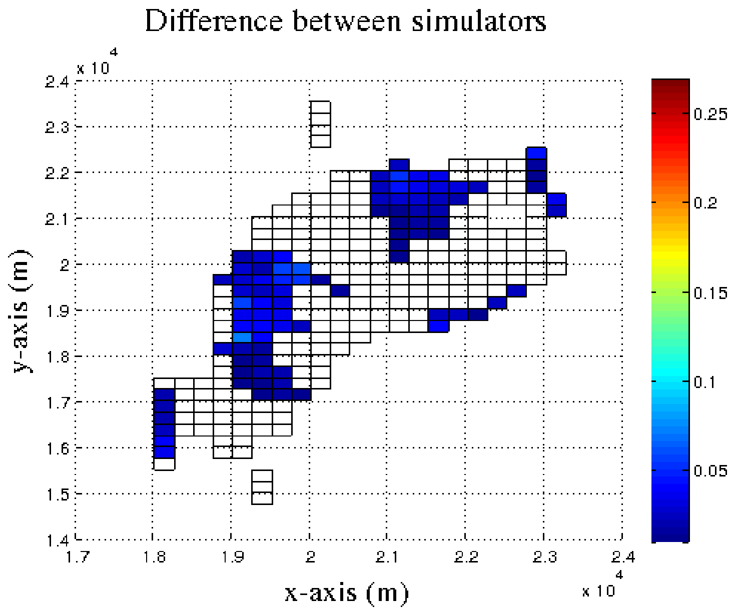


Figure 4.7: Top: The distribution of CO_2 in the reservoir, after 100 years of injection. Bottom: The cells where the error between our simulatar and the MRST simulator is $> 1e-2$.

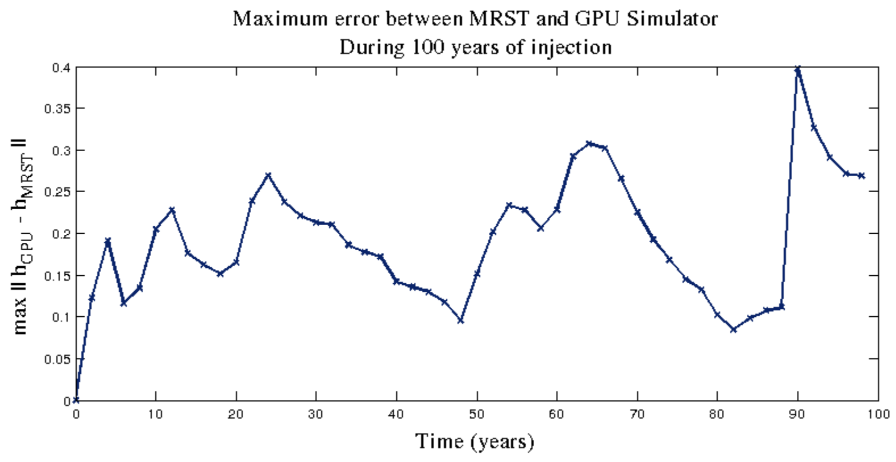


Figure 4.8: The maximum error during 100 years of injection. The maximum error is located where the contributions from the gravity and pressure driven velocity differ in direction but are close in magnitude.

the case in MAD (multiply-add) instructions. MAD is what is implemented in the our NVS 3100M device. It could thus be interesting, if one has a newer GPU available, to see if this could reduce the error when using single precision.

4.3.2 Coats Time Stepping versus MRST Time Stepping

In our simulator, we have implemented support for simulating using both MRST time stepping (Section 2.2.2) and Coats' time stepping (Section 2.2.2). We will now briefly motivate the reason for this.

In the MRST module, one simulates the VE model using the CFL condition referred to as "MRST time stepping" [1,16]. This, as mentioned in Section 2.2.2, will only need to be updated after each update of the pressure. For the segregation part of the problem, this is (relatively) straightforward. And, if the CO_2 plume is not too steep, the segregation part will also be the the dominating term in the CFL condition [17]. However, if the parabolic term is the dominating term in the transport equation, one needs to find the eigenvalues of the system, in order to determine the time step restriction. This can be a tedious part to implement, and as it is outside the scope of interest for this thesis, we have not focused on this. Therefore, if simulating with MRST time stepping on the GPU simulator, this might give instabilities in the results.

Coats time stepping requires us to find the maximum allowed time step after *each* iteration. It is therefore less strict, as it will adapt better to the state of the system.

In Figure 4.9, we see the difference when simulating using MRST time stepping and Coats' time stepping. The test case is the Johansen formation [17], and we simulate 60 years of migration. The initial conditions are based on 30 years of injection. This results in the segregation part dominating the time step restriction, as required.

What we can see from the figure, is that Coats time stepping will complete the simulation the fastest. This is the result of that it only requires 25 time step to complete the 60 years, as opposed to 38, for the MRST approach. Thus, even though the flux kernels require fewer computations when simulating with MRST, the fact that the time strep restriction is more conservative hides this performance gain.

As we have seen that the Coats time stepping results in a faster simulator, we have for the remaining of the results simulated on this approach. It also allows us to simulate for all cases; not just when the segregation part is dominating.

4.4 Performance Results on the GPU

What we eventually want when having a GPU based simulator, is performance gain. There are multiple factors which can affect how well our simulator performs. Not only should the algorithms be optimized, hardware specifications will also play a vital part. This includes optimizing the block size configuration; having kernels which are balanced in terms of memory transfers and computational costs; and being able to fully utilize the GPU resources.

In the following section, we have based all of our runtime values on migration

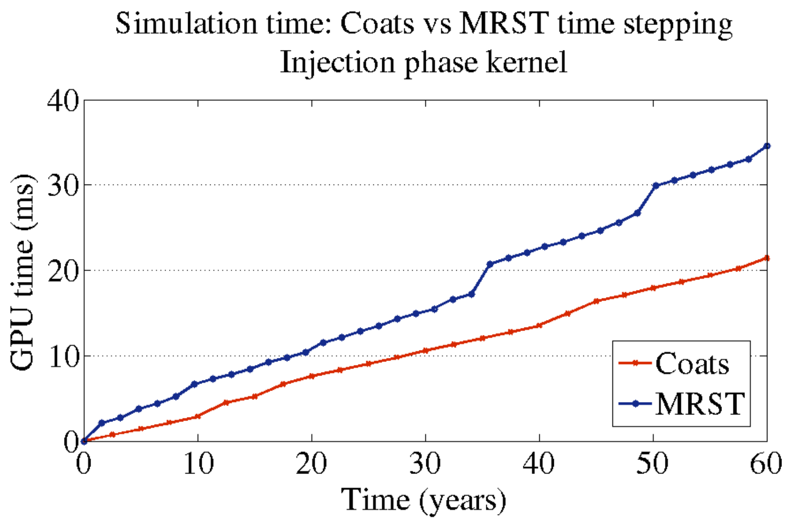


Figure 4.9: Simulating 60 years of migration on the Johansen formation, using MRST's way of computing time steps (blue) and Coats way of computing time steps (red).

on the Utsira formation[2]. The Utsira formation is located outside the west-coast of Norway, and there is currently an ongoing CO_2 storage project here on parts of the formation, i.e., the Sleipner field [11]. Therefore, this example is especially interesting. In Figure 4.10 we have included a 2D overview of the formation. As the GPU simulator assumes a rectangular domain, we see from the figure that on this case, a large amount of the domain is outside the physical boundaries.

Unless otherwise specified, this grid consists of 592x1698 cells, or 1 005 216 cells. Due to memory restrictions using MRST, we have not been able to simulate injection on this case. Thus, the initial height of CO_2 has been created by means of a Gaussian distribution.

Also recall that Ligaarden and Nilsen [17] found that when simulating during the post-injection period, there was a stronger decoupling between the pressure equation and transport equation when simulating based on a vertical equilibrium model, as opposed to a full 3D model. They suggested that this could be taken advantage of for performance purposes. We have therefore also implemented support for a kernel which neglects the contri-

[2]See the example "utsiraMigrationFromSleipner.m" [1, 16] for more information regarding this example.

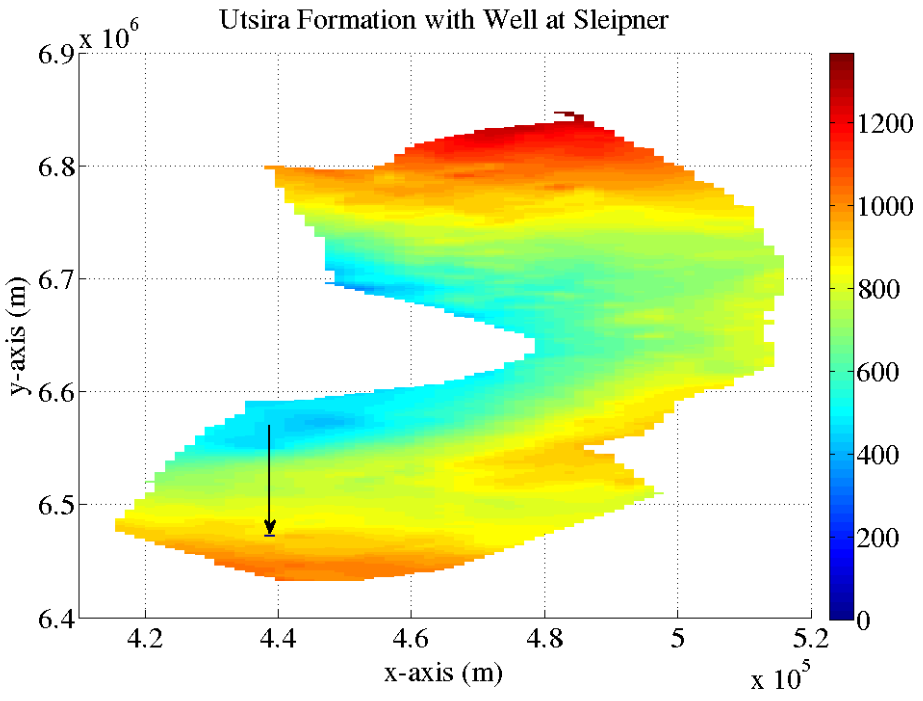


Figure 4.10: A 2D plot of the Utsira formation. The black arrow indicate the well at the Sleipner field, where CO_2 injection is currently ongoing.

butions from the pressure equation. This will then require to read fewer variables from global memory and it will also require fewer computations. Hence, in the following section, we refer to the two different kernels as the *injection kernel* and the *post-injection kernel*. The latter might also be referred to as the *migration* kernel.

Even with the simplified computations in the migration kernel, the flux kernels are by far the most expensive, as can be seen in Table 4.1. This section therefore focus mainly on the flux kernels, as this is where any optimizing will have the largest impact.

Table 4.1: Percentage of total simulation time on the Utsira formation

	Injection kernel	Post-injection kernel
Flux kernel	88.7 %	79.3 %
Time integration kernel	9.7 %	16.4 %
Boundary condition kernel	0.4 %	0.8 %
Maximum time step kernel	< 0.1	0.1 %

4.4.1 Optimizing the Flux Kernels

For a GPU kernel, there are mainly three bottlenecks one can encounter. The kernel may be limited by instruction throughput, memory throughput or latencies [4]. The communication between the CPU and the GPU, along with synchronizations, may also be dominating bottlenecks. We will start by looking into our flux kernels to see if they are bounded by bandwidth, or arithmetic operations.

Math Bound versus Memory bound Kernels

To profile CUDA kernels one can use the profiler tool developed by NVIDIA; CUDA Visual Profiler . To identify how many arithmetic operations a kernel performs, one can simply look at the instruction-to-byte ratio. However, it is not always the case that the profiler reports accurate figures in this matter [4]. We have therefore chosen to instead follow the method described in "GPU Computing in Discrete Optimization", which were found to be more accurate by Brodtkorb et al. [4]. The idea is to create one kernel which only performs the arithmetics from the original kernel, and one which only performs the memory transfers. When comparing the runtime for each of these, we get an estimate of how well the overlap between memory operations and arithmetic operations are.

The memory kernel reads all the required variables from global memory into shared memory and adds them together. (The reason we add them together, is to make sure the compiler does not regard them as useless, and overrides the instruction to read them.) They are then stored in R, before written to global memory again.

For the arithmetic version of our kernel, we commented out all of the required readings from global memory, and just performed the arithmetics. The compiler will strip away everything which does not add to the final result. This is handled by putting all load / store operations into a condition which at runtime always will be evaluated to false.

For the math kernel, we have two different results, depending on the current state of the system: If there is CO_2 in the cells we are computing on, you have to complete all the computations. If there is no CO_2, we do not have to complete the computations - we know that the contributions to R is zero, and we know that this interface will not be the interface limiting the time step. Thus, the ratio between the math kernel and the memory kernel will change as the system changes.

In Figure 4.11 we show the different runtimes between the arithmetic and

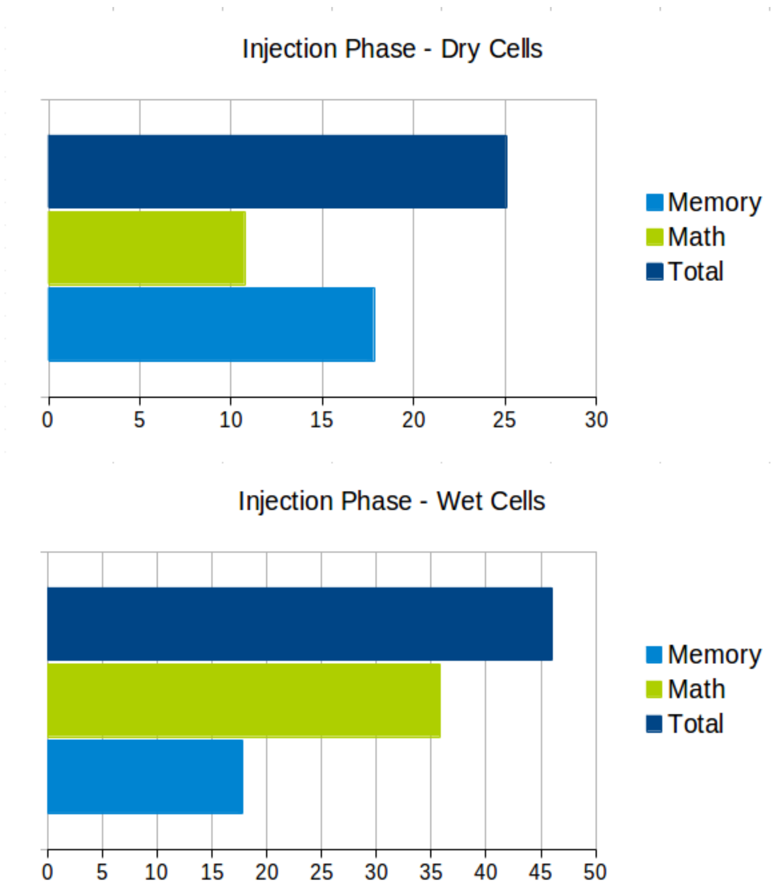


Figure 4.11: Runtime (in ms) for the math kernel, arithmetic kernel in comparison with total time using the flux kernel for injection. Top: Simulation on dry cells. Bottom: Simulation on wet cells.

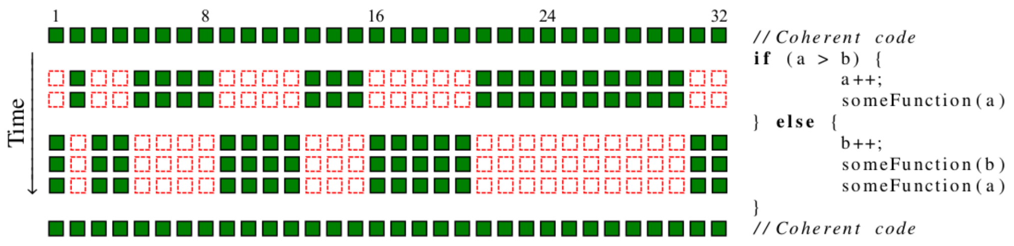
the memory reads for the injection kernel. The migration kernel is shown is Figure 4.13. We consider the injection kernel first.

What we actually want, is a kernel which is well balanced between the math and the memory transfers. However, in our case we see that in the wet cells, our kernel is bound by the arithmetic. This means that it is the amount of computations which mainly contributes to the total runtime.

One factor which may improve performance for the arithmetic within wet cells, is to remember the way a GPU executes its instructions: 32 neighbouring threads, called a *warp*, execute the exact same instructions. This is one of the advantages of a GPU. However, if we within the code have branching - typically represented by an if-else statement, *all* of the threads will execute the same code. For instance, if our code looks like this:

if $CO_2 > 0$: function-a()
else : function-b()

Then, if *all* of the threads in the warp contain CO_2, only function-a() will be called. If, on the other hand, some of the threads do not contain CO_2, then each thread will call function-a() *and* function-b(), before masking out what is not relevant to the particular case [4]. An illustration can be seen in Figure 4.12. The result of this, is that in the worst-case scenario, we might slow down our code by a factor of 32.



```
//Coherent code
if (a > b) {
        a++;
        someFunction(a)
} else {
        b++;
        someFunction(b)
        someFunction(a)
}
//Coherent code
```

Figure 4.12: Branching of code within a 32-wide warp the GPU. As we can see, all of the threads needs to perform the same computations, but the result is masked out whenever they do not apply. The Figure is from Brodtkorb et al., [4].

To avoid this kind of branching, one can try to sort the domain beforehand. In the case of the Utsira formation, this could for instance involve sorting based on cells which are within the physical domain or not. Then we would maximize the amount of warps which evaluate to false when considering whether we need to do the computations or not. Another technique is to let the CPU perform the branching: Then we can use templates to create two kernels; one for those which are true, and one for those which are false [4].

Another way of reducing the cost of the math kernel could be to lower the amount of computations, by reading more variables to / from global memory.

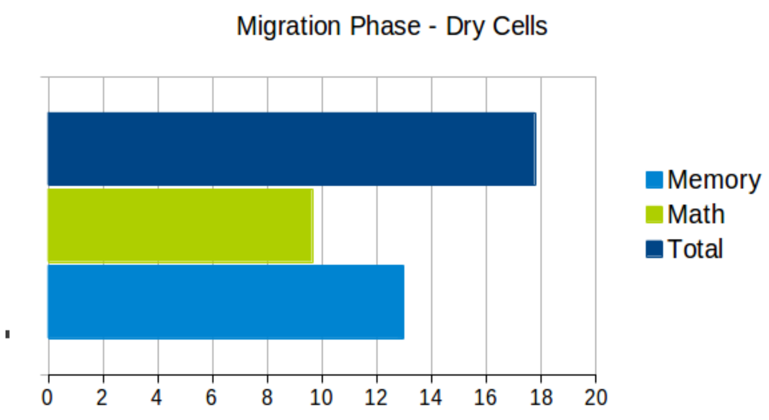
This would, however, then become another costly (and unnecessary) memory transfer in the dry cells.

For the dry cells, we see that it is the memory transfers which is responsible for most of the total runtime. But, an interesting feature is that even here, the arithmetic contribute to a large amount of total runtime. This is the result of two things in particular: First, we still need to compute the minimum CFL restriction r - any dry cell cannot know whether the surrounding cells are dry. Secondly, we have to still *check* if the cell is dry, and when we compute R from R_x and R_y, we also have to check if the neighbouring cells are within the physical domain or not. (Recall that the physical domain might be non-rectangular, even if the final grid is not.) Thus, in total, we see from the figure that dry cells require 54 % of the total runtime that wet cells have.

For the post-injection case in Figure 4.13, we see the same trend as in the injection case. The memory reads are cheaper - not surprisingly: we read fewer variables. For the arithmetic, on the other hand, these are cheaper in wet cells. But, for dry cells, they cost the same as in the post-injection. Thus, in this case, the dry cells actually take as much as 60 % of the time that the wet cells do.

Hence, these results shows that for wet regions we can mainly improve performance by improving our computational algorithms. It also shows that even dry regions contribute largely to our computational costs. Mainly in terms of memory transfer, but also in terms of arithmetic. This suggests that we can benefit from having a way of checking if there actually is any CO_2 in the block we are currently operating on. This will especially have a large effect on formations which mainly are dry, or when simulating on non-rectangular formations which result in large numerical grids. Both which is the case on the Utsira formation.

When trying to further improve our kernel, we have focused on trying to reduce the costs connected to the dry cells. Although we have seen that there is a great potential to reduce the costs in wet cells as well, we found that for most of the cases we have considered, the amount of dry regions are much larger than the wet regions. So for now, we leave the improvement of the wet regions for further work.



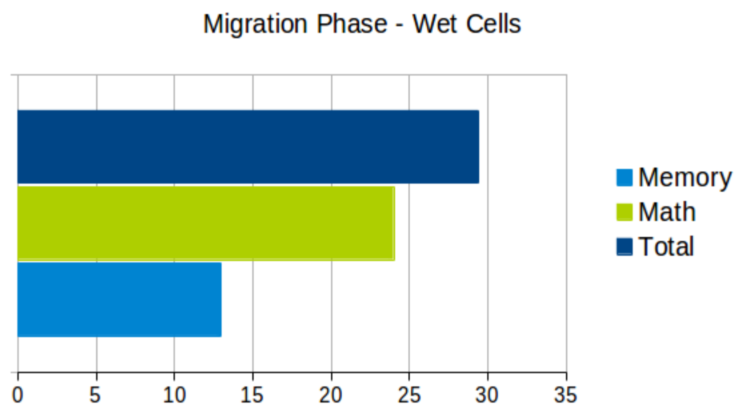


Figure 4.13: Runtime (in ms) for the math kernel, arithmetic kernel in comparison with total time using the flux kernel for migration. Top: Simulation on dry cells. Bottom: Simulation on wet cells.

Early Exit in Dry Blocks

In the following, we describe the implementation and results of an approach where we have tried to reduce the costs connected to regions without CO_2. It is based on the work done by Brodtkorb et al. in the original simulator [6,7], and we refer to this approach as an "early-exit" option within dry blocks.

To implement support for early exit in dry blocks, we let both the flux kernel and time integration kernel have access to a vector D, of size $nb_x \times nb_y$. Here, nb indicates number of blocks. Now, in the time integration kernel, we use global reduction to check if any of the cells in the current block contain any CO_2, or if there is any flux over any faces. If there is, we indicate this by letting $D(bx, by) = 1$, and otherwise, $D(bx, by) = 0$. Here we have used bx

and *by* indicate the current block. In the next time step, when we enter the flux kernel on block (bx, by), we need to make sure none of the surrounding blocks contain any CO_2. Thus we let a thread read 5 values from D - the value corresponding to its own block, but also the blocks on its left, right, top and bottom. If none of the above contain any CO_2, we exit the kernel for this block. Otherwise, we continue with the memory transfer and computations. When using the early exit option, it is also important to keep the block size configuration (see Section 4.4.2) the same for the flux kernels and the time integration kernel.

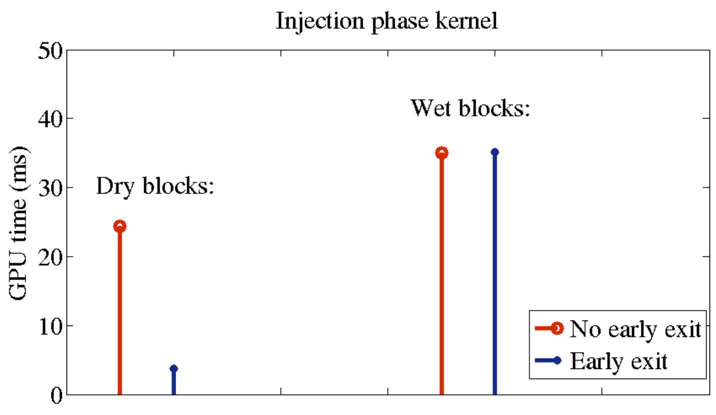
An overview of the average runtimes can be seen in Figure 4.14. It shows the runtime per time step. As we can see, the early exit option increases slightly the simulation time in blocks where there is CO_2, both for the injection and the post-injection kernel. This makes sense, as we need to read an additional variable from global memory. However, the reduction in runtime for dry blocks is in comparison drastic. We can also see the largest change in the injection kernel. This is as expected, as even for dry blocks this kernel is the most expensive.

In Figure 4.15, we have shown the result of simulating 750 years of migration on the Utsira formation. For each of the cases we run the computations using the optimal block size configuration (see Section 4.4.2).

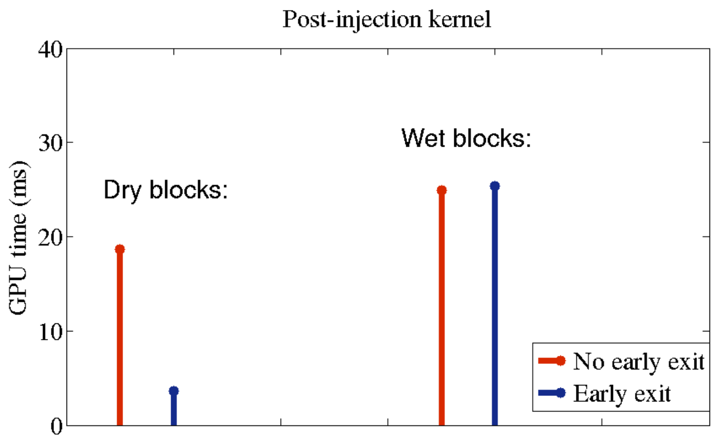
This plot also clearly shows why we have chosen to implement two different flux kernels: If we consider the case where early exit option is disabled, we see that the migration kernel has a 25 % reduction in overall runtime!
To conclude, the performance gain will be the largest when the CO_2 occupies a small fraction of the reservoir. It might also decrease as time evolves, as we will then typically have CO_2 in more cells - even if this was not the case with these initial conditions.

4.4.2 Hardware Specific Optimization

We recall from Section 3.1, that different GPUs have different architecture. As a result of this, different configurations will optimize for different GPUs. We have been simulating using the NVIDIA GT218 device, and have thus aimed to optimize for this. In this section we will start of by describing how different block size configurations affects the performance. We will also consider *when* we are really capable of fully taking taking advantage of the GPU resources, in terms of grid sizes.

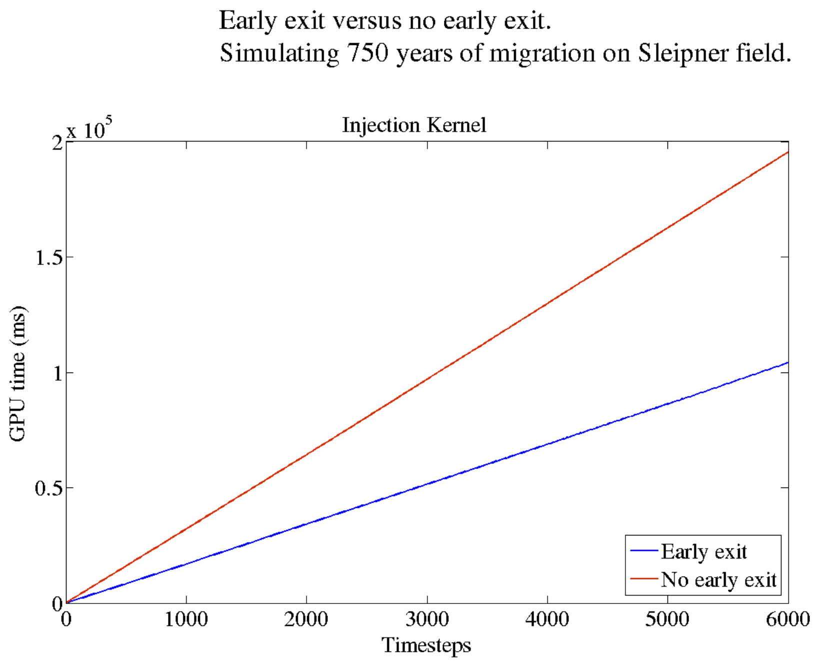


(a)

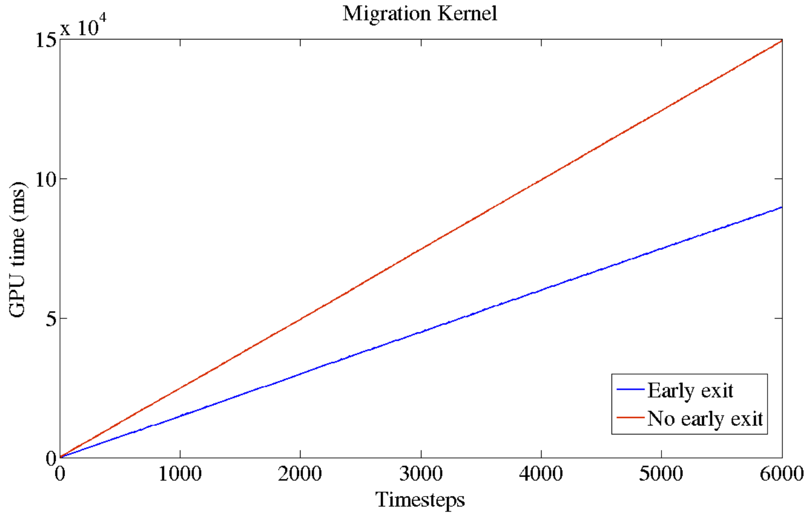


(b)

Figure 4.14: Difference in run time when using the early exit option and not. a) Shows the injection case, while b) shows the migration phase.



(a)



(b)

Figure 4.15: Total runtime on the GPU, when simulating 750 years of migration on the Utsira formation. The runtime is measured after every second year.

Optimal Block Size Configuration

As described in Section 3.1, a CUDA kernel is executed in parallel across a set of parallel threads. These threads are organized by the programmer (or compiler) into blocks, which again is organized into grids of blocks. Each block has a per-block shared memory space used for communication amongst threads. When a block is "active", i.e., a thread within this block has started executing the kernel, this block will stay "active" until all the threads within this block has finished. It is the SM (streaming multiprocessor) that executes one or more blocks in parallel. They execute 32 threads simultaneously, called a *warp*. All of the threads within a warp belong to the same block. Thus, to reduce latencies, we want to keep our block size a multiple of 32.

On the NVS 3100M device, we have 2 multiprocessors available. Each of these can keep 32 warps active at the same time. That is, a total of 1024 threads per SM. (Also recall that in the flux kernels, we let one thread operate on each grid cell.) Increasing the number of warps kept active, will increase the occupancy. The occupancy is a measure of how the streaming multiprocessor can hide memory latencies. When a warp stalls, the processor will instantaneously switch to another warp. We therefore want as many warps per SM as we can get. However, this might be limited by shared memory usage, register usage and block size.

Another aspect to consider when aiming for optimal block size, is that keeping as much of frequently used data in shared memory as possible improves the performance. Also, as we use local ghost cells within each block (see Figure 4.4), we want to keep the size of shared memory as square as possible. This minimized the ratio between internal cells and ghost cells.

Each of our streaming multiprocessor have 16 KB shared memory available, and each block also have maximum 16 KB shared memory available. Thus, we cannot have block size configuration which requires more than this. As previously mentioned; for the injection kernel we store nine variables per thread cell, and for the migration kernel this is reduced to eight.

In table 4.2, we show the average duration for the two types of flux kernels which we have implemented. The results are obtained using NVIDIAs profiler tool; nvprof. The test case is the same as previously, the Utsira formation consisting of 592×1698 cells. It should also be mentioned that the exact duration of the kernels changes based on the actual domain and current state of simulation. We saw this in the previous section; cells containing CO_2 take more time to handle than cells without CO_2 . However, the trend is the same, and we include the table to give an overview. The simulation is also run without the early-exit option we presented in the previous section. This is to best see the difference for the different block configurations.

Table 4.2: Average duration and shared memory usage for the two flux kernels using different block configurations on the Utsira formation.

Block config	Flux compuations			
	Injection kernel		Post-injection kernel	
	Avg. duration	Shared mem.	Avg. duration	Shared mem.
8x8	31.958 ms	3616	23.473 ms	3216
16x8	29.845 ms	6496	22.353 ms	5776
16x10	26.499 ms	7792	19.979 ms	6928
15x11	29.712 ms	7972	22.685 ms	7088
16x12	38.590 ms	9088	18.483 ms	8080
16x16	32.754 ms	11680	24.086 ms	10384
20x16	31.730 ms	14272	24.039 ms	12688
22x16	32.524 ms	15568	24.023 ms	13840
24x16	-	-	21.138 ms	14992

As we can see from the table, the optimal block configurations in the two flux kernels are not the same. We first consider the injection kernel.

We found the optimal block configuration for the injection phase to be blocks with 16x10 (= 160) threads. Each block will then use approximately 7.7 KB shared memory. This allows each multiprocessor to run 2 blocks, and thus keeping 10 warps active at the same time.

In the case with 20x16 threads per block, we also get 10 warps active at the same time. However, we notice that the average duration of the kernel is approximately 8 % higher in this case. This is a result of the fact that we now have all the threads, and thus all the warps, active within one block. As a consequence, each time we synchronize the threads in this block, we also synchronize across all the active warps.

Due to the amount of shared memory we require per cell in our grid, the 22x16 configuration gives the most active warps we can have per SM. We note, however, that this one is slower than the 20x16, even though they both have all the warps within one block. This is the result of the block configuration being less square, thus we require a higher number of total ghost cells.

We can also see the result of having a block configuration which is not a multiple of 32 in the 15x11 case. Even if two blocks can be kept active, it cannot fully utilize the resources, and is therefore slower than the 16x10 case.

For the migration kernel, the same parameters affects the optimal block configuration. However, due to a smaller amount of shared memory, the optimal block configuration in this case is the 16x12 case. This allows us to run 2 blocks per streaming multiprocessor, such that each can keep 12 warps active.

We now look into the effect of implementing the injection kernel the way we did. Recall from Section 4.2, that the values $\mathbf{U}_{\Sigma,x}$ and $\mathbf{U}_{\Sigma,y}$ are only needed in the x- and y-directions respectively. So, after finishing computing the x-direction, we synchronize over the block, and let F be stored in the array which used to hold $\mathbf{U}_{\Sigma,x}$. The same goes for the y-direction. If however, we had stored F and G explicitly in shared memory, we would require 11 variables per grid cell, thus also requiring more shared memory per block. However, we would not need to synchronize across the threads in the block. In Table 4.3 we show the different runtimes for each of these approaches. The chosen block configurations in this table includes the block sizes which were optimal for either one of them.

Table 4.3: Average duration and shared memory usage for the two injection kernels, using their optimal block size configurations.

Block	Injection kernel			
config.	9 values per grid cell		11 values per grid cell	
	Avg. duration	Shared mem.	Avg. duration	Shared mem.
16x10	26.499 ms	7792	32.952	9520
16x8	29.845 ms	6449	29.220	7936

The 16x10 block size is the optimal block size when having 9 values in shared memory per grid cell, and, as previously mentioned, allows for two blocks to be kept active per SM. However, for the case with 11 values per grid cell, due to the required shared memory, we can on this configuration only keep 1 block active per SM. That explains the large difference in the runtime. For the block size configuration 16x8, the difference between the runtime is in favor of the kernel having 11 values in shared memory. This is the result of the required synchronization within the block. In total, however, we see that we gain in performance when reducing the required shared memory.

To conclude on the block size configuration for the flux kernels, we see that keeping as many blocks as possible active at the same time, is what improves the performance. We also see that in our case, this is mainly restricted by shared memory requirements. Thus this results in that for the injection kernel, we gain performance when reusing the arrays $\mathbf{U}_{\Sigma,x}$ and $\mathbf{U}_{\Sigma,y}$, even though this requires synchronization across the block.

We also briefly discuss the block sizes used for some of the other kernels: For the time integration kernel, we are not limited by shared memory, nor do we need to compute on any of the ghost cells. Thus, we can use an block size configuration of 32x16. This results in each block having 512 threads, which is the maximum number of threads per block. We get a 100 % occupancy as a result of this, as we maximize the bandwidth utilization [7]. If, on the other hand, the dry exit option is chosen, we have to do so on the cost of block size in this kernel. We would then require to use the same block size here as in the flux kernels.

The kernel which is responsible for computing the maximum time step, based on the restriction r, is also not limited by shared memory. Each thread here strides through the dataset containing minimum r computed by the flux kernels. If the we use a block size which results in fewer threads than number of elements in the dataset, we lower the occupancy. If we use too many, some of the warps will give useless feedback [7]. Thus, as in the original simulator we use tempates to create multiples of the maximum time step kernel: One kernel for 1,2,4,$\cdots$, 512 kernels. At runtime, the optimal kernel is chosen.

Absolute Performance

We have tested the absolute performance of our implementation for different domain sizes. This we have done to see how large the domain needs to be for the GPU to fully utilize its processing capabilities.

In Figure 4.16 we see the result of simulating using the injection kernel on the Utsira formation. As we recall from previous sections, we do have quite a large difference between computing wet and dry cell. Thus, the actual numbers on the y-axes will change, depending on the state of the system.

However, what is mainly of interest in this graph is the slope of the curve. We see that the performance, in terms of cells computed on per second, will increase until we reach approximately 5 million cells. (The reason the slope is not smooth is due to the fact that when we refine our grid in order to get the desired gridsize values, the ratio between number of cells in the x- and y-direction might be slightly different, thus affecting the global padding of our domain.)

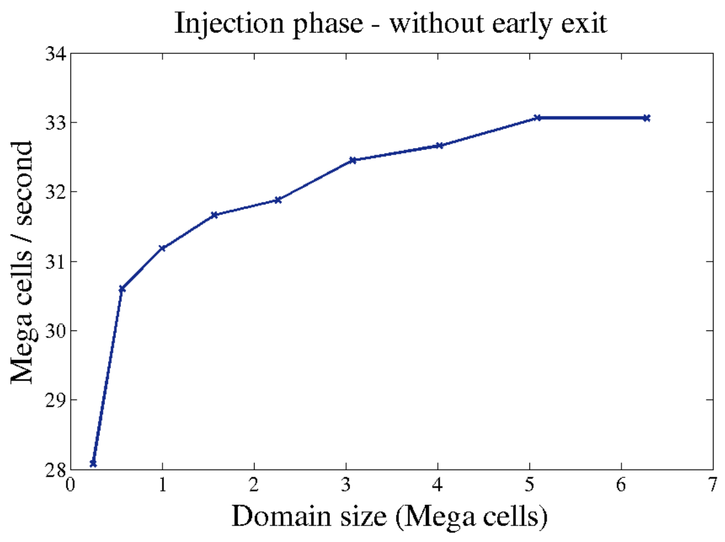
For the early exit option, when comparing the values on the y-axes, we see (again) that this option clearly performs better than the original kernel. Here, we can see that the curve is more affected by the mentioned refinement in the x- and y-direction, as this might affect how many blocks are considered wet and dry.

MRST versus the GPU Simulator

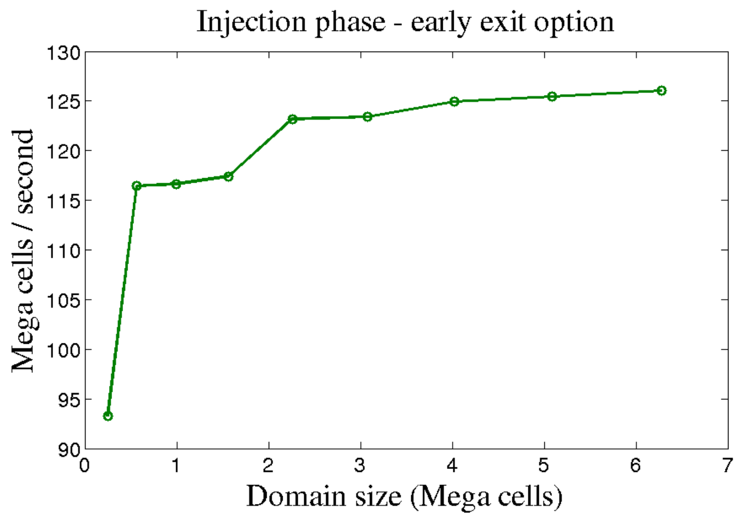
To end this chapter, we will make some remarks about the performance of our simulator in comparison with MRST. The aim of this thesis was, after all, to see whether using GPUs could improve the runtime when simulating large scale migration of CO_2 in saline aquifers.

A direct comparison of the runtimes of our simulator and MRST is not a fair comparison. First of all, MRST is based upon unstructured grids. Secondly, MRST is written using Matlab - a completely different programming language. And third, MRST uses double precision, while we only have single precision. What we can say, however, is that we have definately seen a huge advantage in simulating on structured grids. The largest cases we have considered have consisted of more than 6 million grid cells. This is more than 6 times larger than what we were able to simulate using MRST on our computer. And although MRST can support simulation on a larger variety of grids, we were mainly concerned with being able to perform simulations on the storage atlases from NPD, which as of today are on a structured format.

However, in Table 4.4 we give the runtime per iteration for the two different



(a)



(b)

Figure 4.16: Absolute performance (per time step) of our simulator, as a function of domain size when using the injection phase kernel. a) Shows the performance without the early exit, while b) shows the performance with the early exit. Note the difference between the y-axes.

Table 4.4: Runtime per iteration on Utsira, a comparison between MRST and GPU Simulator

Simulator	Runtime
MRST	218.20 ms
GPU, using injection kernel without early exit	2.534 ms
GPU, using injection kernel with early exit	1.306 ms
GPU, using migration kernel without early exit	1.889 ms
GPU, using migration kernel with early exit	1.164ms

simulators. The test case is as previously the Utsira migration. However, as we also needed to simulate with MRST this time, the grid consists only of 148x424 cells - i.e., far below where we have seen that the GPU simulator fully utilize its resources.

From the table it is quite clear that the combination of simulating on structured grids, using single precision and a GPU to perform the computations, clearly performs faster than the MRST simulator.

Chapter 5

Concluding Remarks

The goal of this thesis was to develop a fast simulator for large-scale migration of CO_2 in saline aquifers. We also focused on being able to simulate on the CO_2 storage atlases from the Norwegian Petroleum Directorate.

Based on the framework from the GPU accelerated simulator for the Shallow Water equations [6, 7], we have implemented a CO_2 storage simulator. We have shown that we can simulate on the storage atlases, and in particular we have done so on the Utsira Formation. We can solve the transport equation in the IMPES formulation; solving the pressure equation was outside the scope of interest in this thesis and we leave this for further work. At the moment we therefore rely on the results from other simulators, for instance the MRST simulator [1, 16].

To fully take advantage of the GPU's resources, we have based our simulator on structured grids. This reduces the memory requirements when specifiying the reservoir properties, and lowers the costs related to memory transfers. The simulator also uses single precision in all of its computations; yet another choice which reduced memory related costs. We considered the effect this had on the accuracy by comparing our results with MRST: For areas where the pressure driven and gravity driven velocities where close in magnitude, not in direction, this affected the height within the cells. But, we also saw that the overall behavior, in terms of which cells were contained CO_2 or not, was the same.

Our simulator is based on the Vertical-equilibrium model [17]. We have implemented a first order scheme, which is based on a finite volume method in space, and the explicit Euler method in time.

The runtime of our simulator was mainly affected by the kernel responsible for the flux computations. We reduced the runtime by implementing an early-exit approach for dry blocks [6, 7]. This approach resulted in that we on the Utsira formation had a reduction in runtime which was close to 50 %. For areas which contain CO_2 , some work still remain in order to optimize the flux kernel. We briefly discussed methods which could be used in this matter - actual implementation we leave for further work.

We also found that our simulator was mainly limited by shared memory access. If increasing the available shared memory, we will allow more blocks per streaming multiprocessor, hence improving occupancy. To solve this, one can either run the simulation on newer GPUs (which typically have mor memory available per block and per streaming multiprocessor), or implement a solution which requires fewer variables stored in the flux kernel. This was the case when we implemented a kernel where the contribution from the pressure equation was neglected.

Finally, we found that the combination of using a GPU to perform the computations, having structured grids, using single precision and implementing the simulator in C++ / CUDA reduced the runtime per iteration with a factor $\sim$ 100 compared to MRST simulator. This was on a case consisting of only 60 000 cells, and our simulator had the best performance when we reached 5 million cells.

Thus, based on the results we have had, we believe it is worth continuing the development of a GPU accelerated simulator for CO_2 storage in saline aquifers.

Bibliography

[1] Matlab Reservoir Simulation Toolbox. `http://www.sintef.no/Projectweb/MRST/`, 2013. [2013 - 05 - 05].

[2] NetCDF (network Common Data Form). `http://www.unidata.ucar.edu/software/netcdf/`, 2014. [2013 - 01 - 03].

[3] J.E. Aarnes, T. Gimse, and K.-A. Lie. An introduction to the numerics of flow in porous media using Matlab. In *Geometric Modelling, Numerical Simulation, and Optimization*, pages 265–306. Springer, 2007.

[4] André R Brodtkorb, Trond R Hagen, Christian Schulz, and Geir Hasle. Gpu computing in discrete optimization. part i: Introduction to the gpu. *EURO Journal on Transportation and Logistics*, pages 1–29, 2013.

[5] A.R Brodtkorb, T.R Hagen, and M.L. Sætra. GPU programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 2012.

[6] A.R. Brodtkorb and M.L. Sætra. Explicit shallow water simulations on GPUs: Guidelines and best practices. 2012.

[7] A.R. Brodtkorb, M.L. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55:1–12, 2012.

[8] K.H. Coats. Impes stability: Selection of stable timesteps. *SPE Journal*, 8(2):181–187, 2003.

[9] K.P. Esler, V. Natoli, and Samardzic A. GAMPACK (GPU accelerated algebaric multigrid package. ECMOR XIII – 13th European Conference on the Mathematics of Oil Recovery, 2012.

[10] M. Garland, S. LeGrand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *Micro, IEEE*, 28(4):13–27, 2008.

[11] E. Halland, W.T. Johansen, and F. Riis(eds.). CO2 Storage atlas - Norwegian Sea. 2013.

[12] Fernando Sandro Velasco Hurtado, Clovis Raimundo Maliska, and AFC SILVA. A variable timestep strategy for accelerating the impes solution algorithm in reservoir simulation. In *Proceedings of the XXVII Iberian Latin American Congress on Computational Methods in Engineering. Belém, Brasil: UFPA*, page 14, 2006.

[13] R. Juanes, E.J. Spiteri, F.M. Orr, and M.J. Blunt. Impact of relative permeability hysteresis on geological CO2 storage. *Water Resources Research*, 42(12), 2006.

[14] A. Kurganov and G. Petrova. A second-order well-balanced positivity preserving central-upwind Scheme for the Saint-Venant system. *Communications in Mathematical Sciences*, 5(1):133–160, 2007.

[15] R.J. LeVeque. *Finite Volume Methods for Hyperbolic Problems.* Cambridge university press, 2002.

[16] K.-A. Lie, S. Krogstad, I.S. Ligaarden, J.R. Natvig, H.M. Nilsen, and B. Skaflestad. Open-source Matlab implementation of consistent discretisations on complex grids. *Computational Geosciences*, 16(2):297–322, 2012.

[17] I. Ligaarden and H.M. Nilsen. Numerical aspects of using vertical equilibrium models for simulating CO2 sequestration. In *Proceedings of ECMOR XII–12th European Conference on the Mathematics of Oil Recovery, EAGE, Oxford, UK*, 2010.

[18] B. Metz, H.C. Davidson, M.L. de Coninck, and L.A. Meyer(eds.). IPCC Special report on carbon dioxide capture and storage: Prepared by working group iii of the intergovernmental panel on climate change. *IPCC, Cambridge University Press: Cambridge, United Kingdom and New York, USA*, 2, 2005.

[19] F. Milo, M. Bernaschi, and M. Bisson. A fast GPU based, dictionary attack to openPGP secret keyrings. *Journal of Systems and Software*, 84(12):2088–2096, 2011.

[20] J.M. Nordbotten and M.A. Celia. Similarity solutions for fluid injection into confined aquifers. *Journal of Fluid Mechanics*, 561:307–327, 2006.

[21] J.M. Nordbotten and M.A. Celia. *Geological Storage of CO2: Modeling Approaches for Large-Scale Simulation.* Wiley, 2011.

[22] NVIDIA. Fermi: NVIDIA's next generation CUDA compute architecture. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[23] NVIDIA. CUDA C programming guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2012.

[24] NVIDIA. Kepler GK110: NVIDIA's next generation CUDA compute architecture. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[25] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13):1490–1508, 2011.

[26] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-purpose GPU Programming.* Addison-Wesley Professional, 2010.