NTNU – Trondheim
Norwegian University of
Science and Technology

A CUDA Back-End for the Equelle Compiler.

Håvard Heitlo Holm

Master of Science in Physics and Mathematics

Submission date:	June 2014
Supervisor:	Helge Holden, MATH
Co-supervisor:	Atgeirr Flø Rasmussen, Sintef ICT

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

As parallel and heterogeneous computing becomes more and more a necessity for implementing high performance simulators, it becomes increasingly harder for scientists and engineers without experience in high performance computing to achieve good performance. Even for those who knows how to write efficient code the process for doing so is time consuming and error prone, and maintaining and implementing changes in such code requires huge effort. By providing tools for automated utilization of parallel hardware, such efforts could be restricted and experts in numerical methods could spend their time on expressing better methods rather than on implementation details.

In this thesis we present a CUDA back-end for the Equelle compiler. Equelle is a domain-specific language designed for writing simulators of partial differential equations, and is under development at SINTEF ICT. The language provides natural syntax for describing finite volume methods, and lets the compiler take care of high performance. The back-end presented in this thesis allows programs written in Equelle be compiled to execute on graphics processing units (GPUs), without requiring the user to have any knowledge in GPU programming.

We have verified correctness of the CUDA back-end by applying it to Equelle simulators for the shallow water equations and both explicit and implicit methods for the heat equation. Good performance have been shown for all three simulators, and we discuss what should be done next to achieve even higher performance.

Sammendrag

Parallelle og heterogene beregninger blir stadig mere nødvendig for å implementere effektive simulatorer med høy ytelse. Dette gjør det vanskeligere for forskere og ingeniører uten erfaring innen parallellprogrammering å oppnå høy ytelse på sine programmer. Selv for de som kan denne typen programmering er det tidkrevende å implementere gode programmer, og vedlikehold og utvidelse av programvaren er ressurskrevende. Ved å tilby programvare for automatisk å utnytte parallell hardware kan disse ressursene omprioriteres, og forskere innen numeriske metoder kan bruke tiden sin på å utvikle nye metoder istedet for å måtte jobbe på detaljnivå med implementasjonen.

I denne masteroppgaven presenterer vi en CUDA back-end til Equelle kompilatoren. Equelle er et domene-spesifikt programmeringsspråk laget for å lage simulatorer av partielle differensialligninger, og er for tiden under utvikling ved SINTEF IKT. Språket har en naturlig syntaks for å uttrykke numeriske metoder av typen "finite volume methods", og kompilatoren har ansvaret for at simulatorene får høy ytelse. CUDA back-end'en vi presenterer her lar programmer skrevet i Equelle bli kompilert til å kjøre på GPUen (graphics processing unit) istedet for på CPUen, uten at brukeren trenger å vite noe om hvordan man programmere disse.

Vi har verifisert at den nye back-end'en oppfører seg som forventet ved å bruke den til å kompilere en simulator for bølger og vannflyt i grunt vann, bassert på "the shallow water equations". Vi har også kjørt simuleringer ved å kompilere Equelle implementasjoner av varmeligningen for både eksplisitte og implisitte metoder. Ytelsen har vist seg å være god for alle de tre simulatorene, og vi har diskutert hvordan ytelsen kan forbedres ytterligere.

Preface

This thesis is written as the final part of my Master's degree in Industrial Mathematics, a specialization within Applied Physics and Mathematics at the Norwegian University of Science and Technology (NTNU). It is written for the course *TMA4910 - Numerical Mathematics, Master Thesis* during the period from 13th of January to 10th of June 2014.

The thesis has been written at SINTEF ICT, Applied Mathematics, and has been related to their project *Simulators That Writes Themselves.* SINTEF has also provided hardware used for my performance analysis. I would therefore like to thank Atgeirr Flø Rasmussen and Johan Seland for proposing this subject for my specialization project and master thesis, and for providing help and guidance throughout this year. Special thanks to Atgeirr for all his guidance, and for his enthusiasm when things work according to plan. I would also like to thank Helge Holden at NTNU for making this cooperation possible.

Lastly, a big thank you goes to my fellow students at Matteland, for making the time here memorable.

Håvard Heitlo Holm
Trondheim
10th June 2014

Contents

Chapter 1

Introduction

In this thesis we will expand the Equelle compiler by offering a compiler back-end for executing simulators written in Equelle on graphics processing units (GPUs). Equelle is a Domain-Specific Language (DSL) for writing simulators of partial differential equations in their discrete numerical form, and is under development as a research project at SINTEF ICT. The goal of the language is to provide a programming environment in which it is easy to express finite volume methods, and where the compiler takes care of high performance. The goal of this thesis is to implement GPU support in the compiler, so that users of Equelle can take advantage of the computational power of GPUs without having to know anything about how to program GPUs themselves.

We will use the CUDA programming model to implement the back-end, and we will through this thesis discuss the implementation choices and design of the back-end. We demonstrate correctness by running simulators for the heat equation using both explicit and implicit methods, and fir the shallow water equations. Implicit methods are treated by automatic differentiation, which we have implemented by operator overloading. We also show that we achieve great performance, and especially so for simulators on large grids.

Even though we through this thesis are able to execute Equelle programs on the GPU and achieving good speedup, we still observe huge performance potentials in the compiler. We therefore discuss how we can improve performance by using a profiler to identify weaknesses in our design, and to monitor program flow and memory management. From this discussion we give suggestions for the future development of the compiler.

Some parts of this thesis are based on work done during the Specialization Project[20] which is considered to be preparation work for the master thesis. During that work we looked at different DSLs and learned to use Equelle by writing the shallow water simulator used in this thesis. Most of Sections 2.2, 2.3 and 3.1, as well as some of Section 2.1, are therefore taken from that project, and edited and improved to fit this format. As we show that the CUDA back-end gives us the exact same numerical results for the shallow water simulations, we have also used illustrations from the specialization project to show examples of the water flow. These are shown in Figures 5.4 and 5.5.

Organization of this Thesis

We start by introducing concepts from the scientific fields on which we build our new CUDA back-end in Chapter 2. We give an introduction to the need for and use of heterogeneous computing, and the challenges faced by programmers in order to fully utilize the computational power on parallel platforms and accelerator hardware. We continue by briefly showing the main concepts behind the finite volume method for numerically solving partial differential equations (PDEs). The concept of Domain-Specific Languages (DSLs) are then introduced, followed by a short overview of how compilers work.

Chapter 3 gives an introduction of the DSL Equelle. We here describe the idea behind the language and what problems it can be used to solve. In order to understand the rest of the thesis we also explain some of the functionality and built-in functions integrated in Equelle. We then go through the organization of the Equelle compiler and what we mean by the front-end and back-end.

In Chapter 4 we present the methods, tools and design choices we have built our CUDA back-end library on. We describe the data types used for the most important Equelle types, and implementation details of the most central functionality of the language. We also explain automatic differentiation and how it can be used to build linear systems arising from implicit methods for solving PDEs.

The back-end is then validated in Chapter 5, where we compile three different Equelle simulators by using the new back-end and running them on the GPU. The simulators are two versions of the heat equation, one solved by an explicit method and the other implicitly, and the shallow water simulator. We also conduct experiments to look at the performance of the back-end.

Chapter 6 is devoted to discuss the design choices, the performance and how we can make the back-end perform even better. We do a step-by-step optimization procedure targeting explicit formulations, and show which factors seem to be the major bottleneck in all three simulators. We then discuss where the future development should be focused and suggest some improvements to both the compiler front-end and back-end, before we finish with some concluding remarks in Chapter 7.

In order to help the reader build the software we have developed, we have added some build instructions in Appendix A. Here we have also included information on how to download the same version as we have performed our tests on, in order to make reproducibility easier. We have written an extensive documentation of the software, and have therefore added instructions on how it can be generated as well.

Appendix B is used to describe the numerical methods which our test simulators are based on. We have also added listings to show the complete heat equation simulators written in Equelle.

Chapter 2

Background

2.1 Heterogeneous Computing

Parallel computing is not a new concept. Since the 90s the fastest supercomputers in the world have consisted of thousands of processors, and the MPI (Message Passing Interface) programming model for writing programs suitable for huge clusters is well established. What is new in parallel computing is the extent of its use. Traditionally, parallel computing was only used in scientific computing. Now however we will have a hard time finding a computer without multi-core processors. Even smart-phones and tablets have dual- or quad core processors, and are therefore target architectures for parallel computing.

On top of this, specialized acceleration hardware such as the graphics processing unit (GPU) has during the last decade developed from being used for graphics only to now performing high performance general purpose operations. The term *heterogeneous computing* refers to using more than one type of computational hardware, such as the CPU and GPU combined to do computations, opposite to only running the application on the CPU. This section gives an overview of the development of parallel hardware and accelerators, and the challenges faced by software developers to keep up with their development.

2.1.1 From Serial to Parallel Computers

The modern way of developing faster computers for both regular and high performance computing, differs a lot from how this development has traditionally been. The traditional way to develop a new and better series of CPUs was to increase the numbers of transistors on a chip, as well as their speeds, in order to increase their clock frequency. The clock frequency describes how many instructions a CPU can perform per unit of time. As the power consumption by a micro processor has a cubic relation with the frequency [21], this development came to an end as the frequency reached the limit of what the CPU cooling systems could handle. In order to keep on developing faster computer systems, the vendors had to use other strategies.

The new strategy for developing more powerful processors came from placing multiple processor cores on the same chip and to get them to cooperate on executing the programs running on the computer. However, in order for a single program to be able to take advantage of multiple cores, the program must be written differently compared to a traditional program. How the program should be parallelized has to be described by the programmer, as this cannot in general be done with automated tools.

Before the introduction of multi-core CPUs, old programs got a performance boost for free as they where executed on new and faster CPUs. Since the clock-frequency now stays the same, this *free lunch* is no longer available, and programs written for serial execution have to be changed in order to perform better on new hardware. In some cases, it might be enough to add some lines of compiler directives, so that the compiler can parallelize computationally intensive parts of the program, but in other cases the entire program has to be redesigned. This is one of the major challenges that comes with the development towards multi-core processors.

Parallel systems can be divided in two fundamentally different platform models, where the multi-core CPUs in modern laptops and smart phones come in the one group and traditional supercomputers come in another. The difference is found in how the computational cores access memory, and whether memory accessed by one core is visible to the others.

Traditional computer clusters consists of several processors connected through a network. Each of these processors have their own memory, and in order for one processor to read data written by another processor the data has to be explicitly sent over the network. This is called **distributed memory systems**, and the industry standard for writing programs suitable for clusters is the Message Passing Interface (MPI) API. When these programs are executed, there will actually be executed one process for each processor, which run independent from each other. The only points in the program where processors have to wait for each other are during communicative function calls where the processes send and receive data across the network.

The multi-core processors on the other hand, are designed so that there are multiple computational cores on the same chip. All of them access the same memory, so that if one core write to memory, this is information that all other cores can read as well. Such systems are said to have **shared memory architecture**. These can be programmed by creating *threads* inside a program where each thread run on a different core. The programmer can then create threads which later on are joined to one control flow within the program. In other words, all functionality comes from only one process. There are several APIs available for writing such programs, where the perhaps most used one is OpenMP. OpenMP is a high level interface where the programmer tells the compiler what to parallelize, while the alternative pthreads API is a low level interface where the programmer explicitly needs to create and destroy threads and take care of the partitioning of the problem between the threads.

These two fundamentally different models requires different ways of thinking about parallelization, but they can also be combined. Since most clusters consists of nodes of multi-core CPUs, one can write applications using MPI to distribute the problem across the network, and use OpenMP or pthreads within each MPI process. For a deeper discussion on these programming models, see the book by Pacheco [25].

2.1.2 GPUs and CUDA

Parallel to the development of clusters and multi-core CPUs, the gaming industry used a lot of resources in developing faster and more powerful graphics processor units, GPUs. Their goal is to optimize graphical computations, and are designed to best fit the nature of such problems. In visualization and computer graphics the goal is often to compute an output image based on an algorithm and some input data. Most often, the nature of these algorithms are such that any pixel in the output can be computed from the input independently of any other pixels in the output. In other words, all output pixels can be computed at the same time, making the algorithms easy to parallelize. Such algorithms are said to be *embarrassingly parallel.*

The development of more powerful GPUs have been very rapid, driven from the demands of the huge gaming industry and their need to get better and more detailed graphics. In order to produce smooth graphics, 30 images per second have to be generated and displayed, each with an average of 1.4 million pixels. So not only can 1.4 million elements be computed in parallel, but there is a huge amount of data that have to be loaded and written for each of the 30 frames within a second. Therefore, the GPUs have developed to optimize total throughput, rather than the speed of each single operation. The devices are therefore ideal for embarrassingly parallel problems [8].

It was originally not intended for the GPU to provide general purpose computations, however given their huge throughput, such computations were attempted by researchers in the early 00s. Larsen and McAllister [23] did a proof of concept study in 2001 showing how matrix multiplication could be done on the GPU by tricking the GPU to believe it was applying textures to images. After this, researchers kept experimenting on using graphical APIs to trick the GPU to do scientific computing by expressing it in terms of the graphical primitives. An example of this is also a MATLAB interface for certain numerical linear algebra algorithms expressed in the OpenGL API by Brodtkorb [7].

CUDA was the first programming language for doing general purpose programming on GPU, released by NVIDIA in 2007. The CUDA language is based on C/C++, and made it therefore easy for researchers familiar with C programming to take advantage of the computational powers of GPUs. Applications written in CUDA can only be executed on GPUs produced by NVIDIA. Other programming platforms, such as OpenCL, came later offering more portable programs which can be run on a wider range of GPUs. The software developed for this thesis, is written in CUDA, and we will therefore not look deeper into the OpenCL model here.

CUDA provides functions that allocate memory on the GPU (the *device*) and for copying data between the CPU (the *host*) and the device. In order to do computations on the device, we write special functions called *kernels.* These kernels are mapped to a *grid* of *blocks* of *threads*, where the kernel body is executed on all threads specified by the programmer. Each thread has its unique id within each block, and each block has its unique id inside a grid. Since programs we launch on the GPU should be highly parallel in nature, we can use these IDs to specify on what data element each thread should operate. A schematic overview of this hierarchy is shown in Figure 2.1.

The size of the grid and blocks are given in the function call, and they should be chosen to fit both the hardware restrictions and problem sizes in order to achieve best performance. The execution model is a combination of the single-instruction multiple-data (SIMD) model from Flynn's taxonomy [17], and the single-program multiple-data (SPMD) model. The blocks are executed by the SPMD model, where each block runs the same program (the kernel body) with different data. Within each block, groups of 32 threads operate in *warps.* Each warp is executed in SIMD fashion, where all instructions are done simultaneously by all threads.

The memory hierarchy of a GPU is depicted in Figure 2.2. Each thread has its own private registers which can not be accessed by any other threads. Each block share a fast but limited memory block called *shared memory.* This can be seen as a programmable cache, and is useful for applications which have regular memory access patterns. The

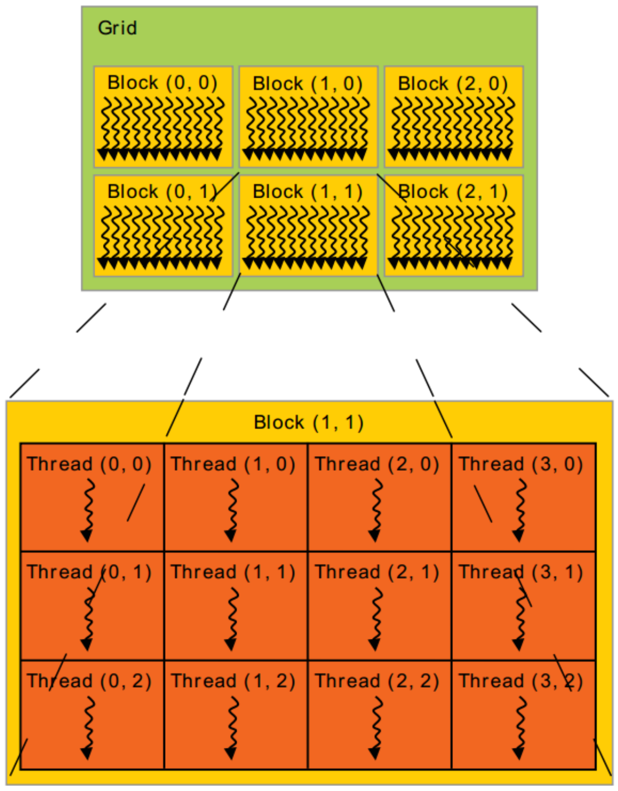


Figure 2.1: A CUDA kernel is launched on a set of threads which are grouped into blocks. All the blocks make up the grid. All blocks contain the same number of threads given in a 1, 2 or 3 dimensional structure, and the size of the grid is given as a 1, 2 or 3 dimensional structure as well. Source: Nvidia [1].

shared memory can only be accessed by its own block. Then there is the global memory which is accessible from all threads in all blocks. Functions such as `cudaMalloc` for allocation of memory, and `cudaMemcpy` for copying data between host and device, operate on this global memory.

In addition to the generality of the CUDA programming language, there exist several libraries for often used applications and algorithms. Some that are used in this thesis are the cuSPARSE and Cusp libraries for numerical linear algebra, and the Thrust library providing datatypes and algorithms such as sorting, scanning and reductions. There are also libraries for fast Fourier transforms and random number generation as well, called cufft and curand respectively. For further reading about CUDA, see the online programming guide[1], the book by Sanders and Kandrot[31], or Brodtkorb[8].

CUDA Example

We will now give a small example of a program using CUDA for adding two arrays, in order to further introduce and explain some important functionality and terminology of the language. The program is shown in Listing 2.1 and is not very complex, but it will serve our purpose as most kernels written for this thesis are quite small.

CUDA is built around a C-like environment where the user is responsible for allocating

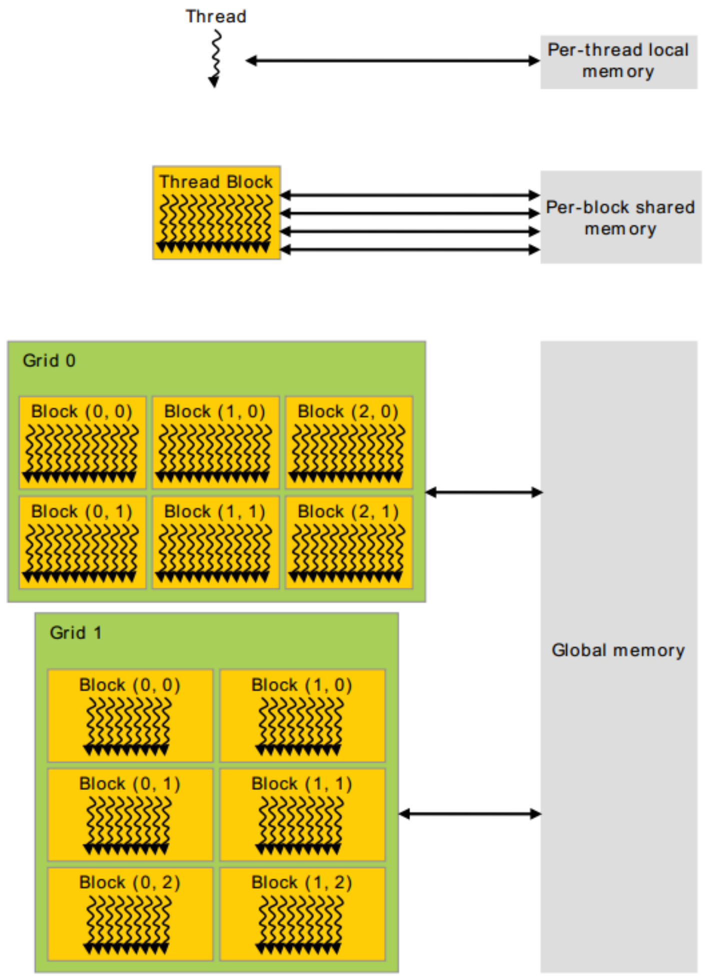


Figure 2.2: The memory hierarchy of GPUs as seen by CUDA. Source: Nvidia [1].

the required memory on the device before it is used. The user is also responsible for deallocating it as well in order to prevent memory leaks. Similarly as the C-functions `malloc` and `free`, CUDA uses `cudaMalloc` and `cudaFree` for this, as seen in lines 26 and 43-44. All functions belonging to the CUDA runtime returns a status flag of the type `CudaError_t`, which may be used to check if the functions executed successfully.

After allocating memory, we might want to fill the device array with values from host memory. This will be the case if the initialization process is serial by nature, or if we read the data from file. Copying of data stored on the device is done with a call to `cudaMemcpy(dest, src, size, direction)`. We copy the `size` bytes starting at memory address `src` over to the memory block of the same size starting at `dest`. The direction is specified by using a flag to tell the function if we want to copy from host to device memory as in line 28, or the opposite as in line 40. This function can be used as a device-to-device or host-to-host function as well.

Now, we look at the kernel implementation of `add_kernel` at line 9, starting with the qualifier `__global__`. The qualifier specifies to the compiler that this function should be compiled to run on the device instead of the host, but that we can still use the host to call it. The qualifier `__device__` on the other hand, as used for the function `myID` at line

4, tells the compiler that this function is executed on the device only, and that it is only possible to call it from another function running on the device.

All launched threads will execute the code given in the kernel independently of each other. In order to figure out which thread we are in, we can access the predefined variables which are used in the `myID` function. The variable `threadIdx` gives the thread's index inside its block, while `blockIdx` gives the index of the block we are in. Both the variables have three members, `x`, `y` and `z`, since CUDA allow 3D kernels. The variable `blockDim` returns the number of threads inside the block in the given direction, and there is a similar function for `gridDim` for the number of blocks in each direction. The function `myID` will therefore return a unique index for each thread in a kernel launced on a 1D grid of 1D blocks.

The kernel is called from line 36, where the triple angle bracket syntax is used for specifying the number of threads and blocks we want to use. The first argument is the number of threads per block, and the second one is number of blocks in the grid. Each of the arguments can be of type `dim3` defining 3D Cartesian structures. We only specify one dimension in this example. Since all blocks are of the same size, the total number of threads will be a multiple of the block size. The grid size is therefore chosen to be such that if we have a data size n such that $n-1$ is a multiple of the block size, we still launch enough kernels for the last data element to have its own thread as well. This is why we need to do the if-test at line 11, as we often launch too many threads and we do not want to access unallocated memory blocks.

More examples of CUDA kernels can be seen in Chapter 4.

Listing 2.1: CUDA example program for adding two arrays. Includes both a global kernel and device function.

```
 1 #include <cuda.h>
 2 
 3 // Device function for finding each thread's ID
 4 __device__ int myID() {
 5     return threadIdx.x + blockIdx.x*blockDim.x;
 6 }
 7 
 8 // Implement kernel for adding two arrays of size n
 9 __global__ void add_kernel( double* a, const double* b, const int n) {
10     int id = myID();
11     if ( id < n ) {
12         a[id] += b[id];
13     }
14 }
15 
16 int main (int argc, char** argv) {
17     const int n; // Size of array
18     const int threads_per_block; // Define number of threads per block
19 
20     double *a_host, *b_host; // host pointers
21     /* Allocate memory and init host arrays - omitted */
22 
23     // Declare device pointers
24     double *a_dev, *b_dev;
25 
26     CudaError_t stat = cudaMalloc( (void**)&a_dev, sizeof(double)*n);
27     if ( stat != cudaSuccess) { /* Error allocating memory - Exit
```

```
        program */ }
    stat = cudaMemcpy(a_dev, a_host, sizeof(double)*n,
        cudaMemcpyHostToDevice);
    if ( stat != cudaSuccess) { /* Error copying memory - Exit program
        */ }

    /* Same for b_dev - Allocate memory and copy data */

    // Set up and launch kernel
    const dim3 block_size( threads_per_block );
    const dim3 grid_size((int)((n + threads_per_block -
        1)/threads_per_block));
    add_kernel<<<block_size, grid_size>>>(a_dev, b_dev, n);

    // Copy data from device to host
    // Omitting error checking
    cudaMemcpy(a_host, a_dev, sizeof(double)*n, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(a_dev);
    cudaFree(b_dev);
    /* Freeing host memory - omitted */

    return 0;
}
```

2.1.3 Challenges in Parallel Computing

The GPU is not the only accelerating hardware available for doing heterogeneous high performing computing. Other examples are the FPGA (Field Programmable Gate Array), initially used for discrete logic but also applied to signal processing and as accelerator in high-performance computing, and the Cell BEA which was the main engine in the Play Station 3 gaming console [9]. The Xeon Phi coprocessor released by Intel in 2012 is another highly parallel hardware architecture which also is designed to be able to run existing code as it is based on the regular x86 architecture. However, in order to obtain optimal performance, the existing applications have to be retuned to fit the physical limitations of the hardware.

With the different memory models and APIs for writing parallel programs for multi-core CPUs and clusters, and the continuous development of more powerful and flexible GPUs and other acceleration hardware, programmers face a huge challenge in order to keep software optimized with respect of this. Software developers have to deal with

- several different programming models,
- portability issues between different hardware types, and
- maintenance required to gain performance of new hardware, or even new versions of the same hardware.

An important challenge within High Performance Computing is therefore to find a way to make it easy for programmers, scientists and engineers to utilize the power of new hardware without having to cope with the problems listed above. One solution is to make compilers which automatically transform programs to match a specified hardware, but this is a tremendously huge task, and for now considered to be out of reach. Instead of considering all possible programs however, a step in the right direction is to find ways of automatically translate certain types of programs into parallel code. This can be done by choosing a narrow problem domain and analyse the common data structures and operations used to solve those problems. By using properties of the data structures and the natural operations on them, it is feasible to automatically create parallel code.

In this thesis we will consider the Domain-Specific Language (DSL) Equelle, which is currently under development by SINTEF ICT. The goal with the DSL is to provide a programming language for writing simulations of certain partial differential equations (PDEs) where the user is freed from the concern of implementation details related to high performance. This is done by providing a compiler that translates the application into accelerated hardware specific code. This is a step in the right direction in order to resolve the issues listed above.

2.2 Numerical Partial Differential Equations

Partial differential equations (PDEs) and their solutions have been studied for centuries, and analytic solutions are only available for a small subset of all PDEs arising in science and engineering. Even though analytical solutions are out of reach it is often the case that numerical methods help us obtain approximate solutions.

Numerical methods are usually designed by using a discretization of the domain of the PDE, and there are three classes of methods that are most commonly used. The discrete domain takes form of a grid or mesh, and the goal is to find approximate solutions, for example at each node or cell in the grid. The different classes of methods are:

Finite Difference Methods The derivatives are approximated by combining Taylor expansions on the nodes. These are then used to find a stencil that gives the approximate solution in each node based on its neighbours.

Finite Element Methods The PDE is considered on integral, or weak, form. Each cell is assigned a set of basis functions and the functions in the PDE are evaluated in terms of these basis functions. The solution will then be found as coefficients on the function space spanned by this basis.

Finite Volume Methods This method is applied to PDEs based on conservation laws. The rate of change in each cell has to match the production by any source term inside that cell as well as the fluxes through the cell's boundaries. The solution is then found as an approximate value in each cell.

In this thesis we will only encounter numerical schemes based on the finite volume method, and therefore take a closer look at it. For further reading on finite volume methods, the book by LeVeque [24] can be consulted.

2.2.1 Finite Volume Methods

We consider a hyperbolic PDE on its conservative form

$$\frac{\partial}{\partial t} q(\mathbf{x}, t) + \nabla \cdot \mathbf{f}(q(\mathbf{x}, t)) = 0, \tag{2.1}$$

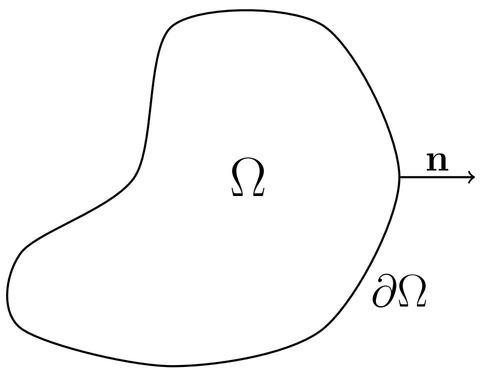


Figure 2.3: A domain Ω with boundary $\partial\Omega$ having an outward pointing unit normal $\mathbf{n}$.

where $q(\mathbf{x}, t)$ is some conserved quantity in position $\mathbf{x}$ at time t. The equation is considered without source term, and $\mathbf{f}$ is a flux function and depends on q. We will use this equation to describe a general finite volume method on a domain Ω, as shown in Figure 2.3, with boundary $\partial\Omega$ and an outward pointing unit normal $\mathbf{n}$.

We start by integrating the equation over the domain and move the time derivative outside the integral as

$$\frac{\partial}{\partial t}\int_\Omega q(\mathbf{x}, t)\,\mathrm{d}\Omega + \int_\Omega \nabla \cdot \mathbf{f}(q(\mathbf{x}, t))\,\mathrm{d}\Omega = 0. \tag{2.2}$$

On the second term, we use the divergence theorem which states that

$$\int_\Omega \nabla \cdot \mathbf{f}(q(\mathbf{x}, t))\,\mathrm{d}\Omega = \int_{\partial\Omega} \mathbf{f}(q(\mathbf{x}, t)) \cdot \mathbf{n}\,\mathrm{d}\gamma, \tag{2.3}$$

and Equation (2.2) becomes

$$\frac{\partial}{\partial t}\int_\Omega q(\mathbf{x}, t)\,\mathrm{d}\Omega + \int_{\partial\Omega} \mathbf{f}(q(\mathbf{x}, t)) \cdot \mathbf{n}\,\mathrm{d}\gamma = 0. \tag{2.4}$$

Let $|\cdot|$ denote an element's natural size, so that $|\Omega|$ will denote the volume of Ω if $\Omega \in \mathbb{R}^3$, and its area if $\Omega \in \mathbb{R}^2$. In other words, $|\Omega| := \int_\Omega \mathrm{d}\Omega$. Since we want to end up with a numerical method, we approximate the value $q(\mathbf{x}, t)$ inside of Ω at time t_n as

$$Q_\Omega^n \approx \frac{1}{|\Omega|}\int_\Omega q(\mathbf{x}, t_n)\,\mathrm{d}\Omega. \tag{2.5}$$

To be able to use Q_Ω^n we integrate equation (2.4) in time from t_n to t_{n+1},

$$\int_{t_n}^{t_{n+1}} \frac{\partial}{\partial t}\int_\Omega q(\mathbf{x}, t)\,\mathrm{d}\Omega\,\mathrm{d}t + \int_{t_n}^{t_{n+1}} \int_{\partial\Omega} \mathbf{f}(q(\mathbf{x}, t))\,\mathrm{d}\gamma\,\mathrm{d}t = 0, \tag{2.6}$$

and get

$$\left(Q_\Omega^{n+1} - Q_\Omega^n\right)|\Omega| + \Delta t F_{\partial\Omega} = 0, \tag{2.7}$$

where $\Delta t = t_{n+1} - t_n$ is the time step. $F_{\partial\Omega}$ represent a numerical approximation of the flux out of the boundary, and has to be chosen according to the nature of the flux term $\mathbf{f}$ and the geometry of the boundary. In other words, $F_{\partial\Omega}$ is an approximation of

$$F_{\partial\Omega} \approx \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \int_{\partial\Omega} \mathbf{f}(q(\mathbf{x}, t)) \cdot \mathbf{n} \, \mathrm{d}\gamma \, \mathrm{d}t. \tag{2.8}$$

Solving Equation (2.7) for the next time step, Q_{Ω}^{n+1}, we will therefore end up with some numerical method of the form

$$Q_{\Omega}^{n+1} \approx Q_{\Omega}^{n} - \frac{\Delta t}{|\Omega|} F_{\partial\Omega}. \tag{2.9}$$

Since $F_{\partial\Omega}$ depends on q in the time interval $[t_n, t_{n+1}]$, we can use any ODE (ordinary differential equation) method to approximate the flux function:

Forward Euler Using the already obtained solution Q_{Ω}^{n} from the previous time step to approximate $F_{\partial\Omega}$, so that $F_{\partial\Omega} = F_{\partial\Omega}^{n}$.

Backward Euler Use the unknown solution of the next time step, Q_{Ω}^{n+1}, in the flux term, so that $F_{\partial\Omega} = F_{\partial\Omega}^{n+1}$.

Crank-Nicolson By combining the two methods above we get the Crank-Nicolson method, with $F_{\partial\Omega} = \frac{1}{2}\left(F_{\partial\Omega}^{n} + F_{\partial\Omega}^{n+1}\right)$.

The Forward Euler method is an explicit method, meaning that the updated solution Q_{Ω}^{n+1} can be computed straight forward only based on values from time t_n. The other two are implicit methods as Q_{Ω}^{n+1} depends on values at t_{n+1}, and to obtain a solution we are required to solve a system of linear- or non-linear equations. The benefit of using implicit methods is that it is easier to obtain stability of the solution. Another option is to keep the equation in semi-discrete form by not integrating in time, and use a Runge-Kutta method to solve the remaining ODE.

2.3 Domain-Specific Languages

As we saw in Section 2.1, it is becoming increasingly harder to keep high performance software at optimal speed with regards of new hardware and programming models. When it comes to writing computationally intensive software it is not enough to be an expert of the application field, but it is also necessary to have knowledge of parallel programming, or how to use accelerators such as GPUs. Di Pietro et.al. [14] list four areas of expertise that are needed in the field of scientific computing in order to solve a problem in an optimal way. These are

Modelling Using the laws of physics to create a mathematical model describing the problem at hand. It is often the case that the problem can be described as a partial differential equation (PDE) with suitable initial- and boundary conditions.

Discretization The PDE has to be transformed from continuous form to a discrete approximation in order to be solved numerically by a computer. Numerical methods such as finite element (Galerkin methods), finite differences or finite volume methods are the most widely used ones, and can formulate discrete representations of a wide range of PDEs. The resulting problem is often to solve a system of linear equations.

Solution Use of numerical packages and libraries that will solve the discrete problem, by using the properties associated with the obtained matrix to optimize the solution in terms of correctness, stability and efficiency.

Software design Writing the numerical packages used to find the solution, by using both low-level data structures and algorithms that are tuned to both the programming language and the underlying hardware architectures.

It is often not the case that one person has expert knowledge in all these areas, and in order to solve problems more efficiently, a software system should be developed so that the interaction between the various domain experts could be limited. This can be done by creating a **Domain-Specific Language (DSL)**. A DSL is a programming language designed for solving problems in a narrow problem domain. It should have a syntax which feels natural to an expert in the respective domain, so that the domain expert can easily express himself. The writing of a program in a DSL should therefore take significantly less time for a domain expert, compared to the effort needed for expressing the same program in a general-purpose programming language.

Since all programs within a domain usually share some form of algorithmic design, datatypes and memory access patterns, the DSL compiler can perform a lot more aggressive optimizations and error checking of the code, compared to what a general-purpose language compiler would be able to do. A good DSL for scientific computing will therefore cover the software design and much of the solution step from the above list, and therefore leave the user to focus more on the modelling and discretization. Since writing programs in the DSL should be faster and easier than implementing the same functionality in a general-purpose language, it should also allow the user more time to test different discretizations without too much effort.

There are several well known domain-specific programming languages that are widely used by scientists. Matlab is a widely used DSL for programming numerical algorithms and linear algebra. Typesetting of academic papers are often done in LaTeX or TeX, and queries in relational databases are commonly written in SQL. In order to write compilers for DSLs, one is likely to use two DSLs as well, namely Lex for the lexical analysis and Yacc for the parsing. Even more examples are found in [34].

Creating a DSL is not without drawbacks [34], and it requires planning, quality implementation and proper maintenance in order to be successful. First of all when planning a DSL, it might sometimes be hard to find the proper scope of the language. The language needs to be flexible enough for the user to express all necessary operations, but still be restricted enough for the compiler to be able to do aggressive optimization and error checking. It is therefore important to have proper balance between the domain-specific and the general-purpose aspects.

Another challenge is to provide the user with good tools for developing applications in the language. Since online resources of new and small languages tend to be limited, it is important to provide good documentation and user training. New Integrated Development Environments (IDEs) with syntax highlighting, auto-completion and integrated debuggers are other tools that increase productivity and user-friendliness, and should therefore be provided. As we saw in the previous section, the hardware development is fast, and long term maintenance is important in order to keep the compilers up to date for maximal performance.

Some of these challenges can be easily overcome by implementing the DSL within a general-purpose language. The domain-specific parts are in these cases often represented by classes, functions and operator overloading, while all the flexibility of the host language is still preserved. Also, the DSL can then take advantage of already existing IDEs and debuggers, and it is often enough to provide a simple plug-in for the IDE to provide syntax highlighting of new keywords. When a DSL is designed as a part of an already existing language, it is called a **domain-specific embedded language**.

Since we in this thesis will consider the DSL Equelle, which targets finding solutions of PDEs solved by finite volume methods, it is natural to mention two other DSLs in related domains which have some of the same goals. Liszt and Halide are two relatively new DSLs that both provide programming environments where users write their program independently of the hardware they will run on. The two compilers are then responsible for translating the code to be run on a given hardware.

Liszt [11, 13] is the language which has a domain closest to the one of Equelle. It is designed for solving PDEs related to turbulent flow on unstructured meshes, by combining operations on cells, faces and vertices. Liszt is embedded inside the object-functional programming and scripting language Scala, and compiles to an intermediate C++ code, which can be specified to target different hardware architectures. They offer back-ends for shared memory platforms using pthreads, for clusters and distributed memory platforms using MPI, and for GPUs using CUDA.

In the field of image processing we find the DSL Halide [26, 27] which provides a simple syntax for expressing such algorithms, as well as strategies for how to execute them. Complex image processing programs often consist of several layers of stencils and graphs, where the performance depends heavily on the strategy for evaluating them. What Halide provides is a language where the algorithms and strategies are expressed separately at a high level. This makes it easy and little time consuming to test different strategies in order to find the best one for the given algorithms.

2.4 Compilers

Compilers are computer programs intended to translate a text written in one programming language (the *source language*) to another language (the *target language*). The target language can be assembly language, machine code or executable programs tailored for the given architecture on which it is compiled, as is the case with the C compiler `gcc`, or byte code which is created to run on virtual machines on all architectures, as is the case for java. Alternatively, the target language can be another high-level programming language, as we will see is the case with Equelle. Here is a very brief explanation of how a compiler works, and the reader should refer to [5] for a longer discussion on compiler design.

A compiler usually consists of three steps as depicted in Figure 2.4. The front-end checks that the source code is correct given the set of rules that makes up the language. The program is then optimized in terms of speed and resources, before the same program is generated in the target language. The front-end again consists of several steps in order to check for different types of errors in the source code.

Consider the source code line

```
int a = (b + 5) * (c - 4);
```

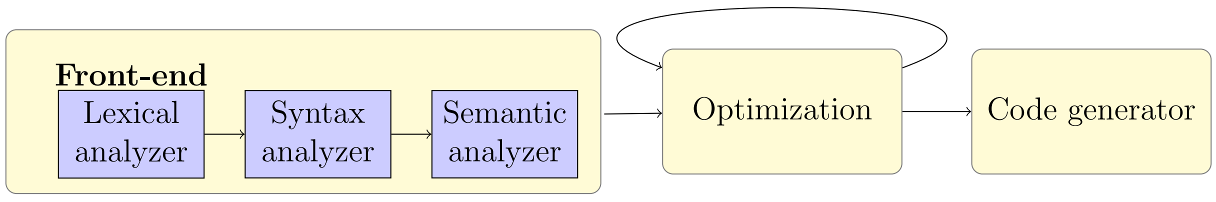


Figure 2.4: A schematic view of a typical compiler.

The *lexical analyzer* is in charge for recognizing what each word in the input means, such as recognizing keywords, numbers or variable names, matching them in *tokens*. In the above example, `int` is a keyword defining the data type of a variable, while `a`, `b` and `c` are variable names, `+`, `-` and `*` are arithmetic operations, and 4 and 5 are numbers.

The next step is a *syntax analyzer* which matches the stream of tokens to the *grammar* of the given language. The compiler use this grammar to build a syntax tree matching the input, for our example as shown in Figure 2.5. An example of a syntax error would be if we had forgotten a arithmetic operation or the first operand in front of the multiplication. The syntax analyzer is also in charge of filling the *symbol table*, which consists of the variables declared in the program together with their types and additional information needed by the rest of the compiler. A variable is added to the symbol table when the syntax analyzer encounters a declaration, and similarly the syntax analyzer looks up the variable in the table when the variable is used.

The *semantic analyzer* are in charge of type checking, ensuring that all data types are compatible with their use. For instance, if `b` was a string, the above example would not make sense and give a compiler error. Since the different parts of the front-end checks for different kind of errors, this also makes it easier to create more readable error messages.

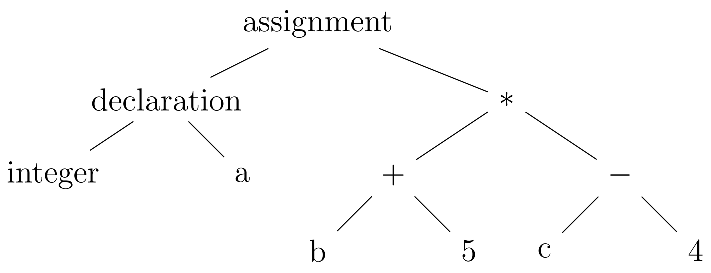


Figure 2.5: A possible syntax tree of the above example.

After the semantic analysis, the compiler is left with an *abstract syntax tree*, such as the one in Figure 2.5 .This is where the optimization phase of the compiler is done. There are plenty of ways a compiler can do optimization, for example with the respect to execution speed, memory usage, power consumption and so on, depending on what the goal of the program is.

The most important part of a optimizing compiler is nonetheless to conserve correctness. Making a program run faster is no use if the resulting program does not behave as expected. It is also important to balance the optimization level with the time it takes to optimize, as it is hard to keep a productive work flow if the compilation process becomes too time consuming.

The last step of a compiler as shown in Figure 2.4, is the code generation step. This is also known as the *back-end* of a compiler. The code generator goes through the nodes of the abstract syntax tree, and output the source program in the form of an optimized equivalent target program. This main focus of this thesis will be on the back-end of the Equelle compiler which has CUDA as target language.

Chapter 3

Equelle

Equelle [15] is a Domain-Specific Language (DSL) designed for writing highly efficient simulators of partial differential equations solved by finite volume methods. The language is under development at Sintef ICT and it is expected that future release versions will cover a wider functionality than what is presented here. We will therefore in this chapter give an overview of the state of the language at the time of writing, as this defines the initial state for this master thesis. The compiler is an Open Source project, and the source code is therefore freely available on GitHub[16].

The need for a language such as Equelle has grown from the increasing complexity of writing high performance code for scientific computing. Writing a finite volume method in C/C++ requires a huge amount of error prone indexing for looping over all cells and accessing values in the correct neighbour cells and faces. Once the problem becomes more complex than the simplest examples, the code will become hard to read and debug for anyone but the code author. Add optimizations to the code and wrap everything inside a parallel API, and the program becomes even larger, more complex and harder to read. It will require large effort to change the underlying numerical method, or even look for small index bugs. Not to mention the big effort needed to rewrite the entire program from e.g. MPI to CUDA.

The goal for Equelle is to provide a portable programming model tailored for solving problems using the finite volume method. Since the normal way of expressing a numerical method is to write an equation for the value of one cell based on the neighbouring cells and faces given a general cell in the grid, the statements and operations in Equelle are designed in the same manner. The programmer can define subsets of cells or faces in order to compute values by different rules depending on whether the cell is at the boundary or not, just to mention one example. A program written in Equelle will therefore be completely free from indices for faces and cells. We will show this in more detail soon.

The programs are written independently of the grid, and therefore require the numerical method to be expressed in a general way. The programming model is based on the assumption that there is a grid, and that the specified operations will solve the equation regardless of which grid is provided at run-time. This gives portability in the sense that a single equation can be solved by one program on several grids, with different initial and boundary conditions.

The second form of portability is that the user should not worry about efficiency and parallelism, as this is the compiler's task. The goal for Equelle on this end is to provide several compiler back-ends, so that the Equelle source code can be compiled to efficient code for a series of different hardware, such as multi-core CPUs, cluster of CPUs or GPUs, depending on what is available on the user's system. A simulator written in Equelle can therefore in principle be compiled to any computational platform, without a single change in the Equelle source code. As new platforms for high performance computing are developed, we can add new back-ends to the Equelle compiler and in that way reuse the existing simulators written in Equelle by only a recompilation.

We will in this chapter give a presentation of the Equelle language, so that we later can talk about the implementation of its functionality. As of this writing, a draft of *Equelle Reference Manual*[29] exists as a $\TeX$ document in the GitHub directory of the language, and this will be an important source for Section 3.1, where we will describe some syntax and functionality. We will later in Section 3.2 give an overview of the Equelle compiler, where we also discuss the intermediate representation which provides the basis for new back-ends in Section 3.2.1.

3.1 Programming in Equelle

We will now present an overview of the types, syntax and some of the built-in functionality of the Equelle language. For a more thorough presentation the Reference Manual[29] should be consulted. In order to get some context around the concepts we explain in this chapter, we show the full Equelle implementation of the heat equation in Listing B.1 in Appendix B.1. This is the example simulator given on the Equelle website[15] and is therefore not written for this Thesis. It is however a very useful example, as it makes use of most of the functionality offered by Equelle. The simulator will also be used in Chapter 5, where we test the new back-end.

As mentioned above, Equelle is designed to do computations on general grids consisting of cells separated by faces. The geometry of the grid is provided by the user at run-time, and therefore the Equelle syntax is not grid-specific. Different domains are accessed through built-in functions such as `AllCells()` and `BoundaryFaces()`, and more specific subsets can also be specified by a combination of built-in functions and a set of run-time provided cell or face indices.

Variables in simulations suited for Equelle are typically data sets with one element for each cell or face in the entire or just parts of the grid. The data types provided in the language are therefore suited for this need. Variables are declared as a combination of a regular type and the domain the variables are `On`. For example we declare a variable `myVar` with the line

```
myVar : Collection Of {Scalar, Vector, Bool} On ...
        {AllCells(), BoundaryCells(), InteriorFaces(), [etc]}
```

where we only use one of the types given in each curly bracket.

The language is strongly typed, meaning that the type of a variable can not be changed after it is declared. This makes the compiler capable to easily throw an error if we try to do operations that do not make sense, such as adding some value defined on `InteriorFaces()` with a value defined on `BoundaryCells()`. Since this would not make sense even in the cases where these sets are of the same size, the user is able to detect such bugs at compile time instead of experiencing cryptic simulation results in run-time.

The language offers a series of built-in functions which performs operations on the grid. In general a grid is defined in 2 or 3 dimensions and consists of non-overlapping cells. In

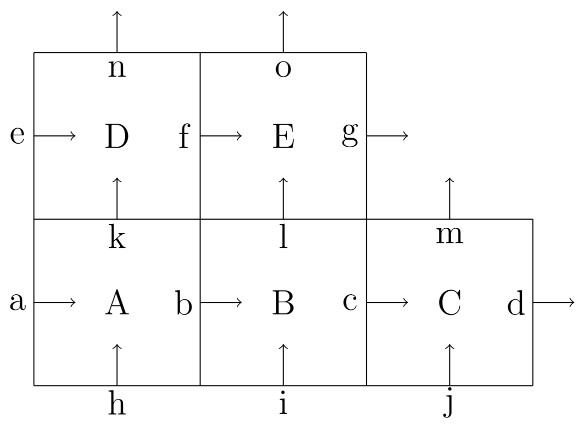


Figure 3.1: Example 2D grid where capitalized letters are cell identifiers and lower case letters identify faces. The arrows show the orientation of each faces.

Table 3.1: Some built-in functions in Equelle operating on the grid showed in Figure 3.1.

Built-in functions	Return values
`BoundaryFaces()`	$\{a, d, e, g, h, i, j, m, n, o\}$
`InteriorFaces()`	$\{b, c, f, k, l\}$
`FirstCell(InteriorFaces())`	$\{A, B, D, A, B\}$
`SecondCell(InteriorFaces())`	$\{B, C, E, D, E\}$
`IsEmpty(SecondCell(BoundaryFaces()))`	$\{0, 1, 0, 1, 0, 0, 0, 1, 1, 1\}$

3 dimensions, each cell is bounded by the surface faces which then again is bounded by edges. The edges are line segments between vertices. Each face has an orientation given by an attached unit normal vector, and this orientation plays an important role in some of the built-in functions.

Some of the built-in grid related functions can be seen in Table 3.1, where the functions are evaluated on the grid shown in Figure 3.1. The function `BoundaryFaces()` can be used as a set holding boundary conditions, and it is also easy to imagine how subsets of `BoundaryFaces()` can be used to treat different types of boundary conditions on different parts of the domain.

In finite volume methods we almost always need to compute some sort of flux value across each face in the domain using values stored at each cell. Such an operation gives one flux value for each face in the domain, and to access the cells at each side of the face, `FirstCell` and `SecondCell` functions are used. These take a set of faces as input and return one cell corresponding to each of those faces. The functions use the faces' unit normals to decide which of the two cells are the first and the second. The direction of the normal is used such that the vectors points from the `FirstCell` to the `SecondCell`, as can be seen in the table. The function `isEmpty` is also handy as it can be used to get the orientation of faces on the boundary, checking if the argument is a legal cell or not.

The face orientations are also used in the built-in functions for finding gradients and divergence. The function `Gradient` takes a `Collection Of Scalar On AllCells()` as input and compute the discrete gradient across each interior face based on the face's orientation. The gradient function could also have been written in Equelle as

```
Gradient(u) = (u On SecondCell(InteriorFaces())) ...
              - (u On FirstCell(InteriorFaces()))
```

The size of the cells, and so the distance between the two values, are not used in the evaluation of the function, so if `Gradient` is to be used for a first order approximation of the derivative of `u` on the faces, we also need to explicitly use these distances in the evaluation. The function `Centroid` takes a set of grid elements as input and returns the centre coordinate of these. An approximation of the derivative of `u` across the interior faces can therefore be written as

```
first = FirstCell(InteriorFaces())
second = SecondCell(InteriorFaces())
derivative = Gradient(u) / |Centroid(second) - Centroid(first)|
```

Here we also use the syntax for finding the natural size of grid elements. The notation $|\cdot|$ is used for finding the length of vectors, as well as sizes of cells and faces. Since the grid can be either of 2 or 3 dimensions, function names based on volume and area could be misleading, and the syntax used here therefore provides more general expressions. This use can also be seen in Listing B.1 on lines 21 and 27.

The `Divergence` function is constructed in a similar manner. It is implemented to take values at faces as input and return for each cell a sum of its faces' values relative to the face orientations. If the value at face i is given by the function $f_i(u)$ and the face orientation is given by the unit normal $\mathbf{n}_i$, we can write the result from the divergence function as

$$D_{\text{cell}} = \sum_{i \,\in\, \text{Faces(cell)}} f_i(u)(\mathbf{n}_i \cdot \mathbf{n}), \tag{3.1}$$

where $\mathbf{n}$ is the outward unit normal for the cell. The `Divergence` function can be seen in a context in line 56 in Listing B.1.

The critical point in solving numerical PDEs is often the treatment of the time derivative. Given an explicit formulation of the problem, the solution at the next time step can be found directly from the previous as $Q^{n+1} = \tilde{F}(Q^n)$, where $\tilde{F}$ represent the right hand side from Equation (2.9). The solution can then be programmed directly by the basic functionality of the language. However, the user has to be careful using small enough time steps to ensure stability of the solution.

In order to obtain a stable solution using larger step sizes, an implicit method can be used instead. Such methods are formulated as $Q^{n+1} = \tilde{F}(Q^{n+1}, Q^n)$ as discussed in Section 2.2, and results in a (non-)linear system of equations that needs to be solved. In Equelle this is done by the `NewtonSolve` function, or by `NewtonSolveSystem` if the problem is a system of PDEs. The programmer is required to formulate the problem in terms of a residual function and provide an initial guess, which in these Equelle functions are used by a Newton method for solving the problem.

Another important thing to have in mind when writing a simulator in Equelle is that all variables are immutable by default. In order to reassign values to a variable, the variable has to be explicitly declared with the `Mutable` keyword. Typical variables that we want to be `Mutable` are variables for storing the result for each time step, such as the declaration of `u0` in line 62 in the heat equation.

The `On` and `Extend` Operations

The `On` and `Extend` operations are some of the most central operations in Equelle. They are used to map variables between different sets on the grid, without the user having to worry about error prone indexing. Both operations can be used in two different ways, and we will take a look on both of them.

The `Extend` operator can first of all be used to create a `Collection` of uniform values on a set, as can be seen in line 26 in Listing B.1 stating

```
bf_sign = IsEmpty(FirstCell(bf)) ? (-1 Extend bf) : (1 Extend bf)
```

Here we create two sets defined on the set `bf`, where all elements are -1 and 1 respectively. We then use the ternary if function in combination with `IsEmpty`, so that `bf_sign` becomes -1 if the `FirstCell` of a `bf` face is outside the boundary and 1 otherwise.

The other use of `Extend` maps a `Collection Of Scalar` to a superset by inserting zero for the new elements in the `Collection`. A good illustration of this is found on line 54 in Listing B.1, where the fluxes for the `InteriorFaces()` and the `BoundaryFaces()` have been computed separately and we want them combined in a new variable. Since the two flux variables are defined on different sets we are not allowed to add them as is, but by extending both to the set of `AllFaces()` by inserting zeros we are able to get all the fluxes in the same variable:

```
fluxes = (ifluxes Extend AllFaces()) + (bfluxes Extend AllFaces())
```

The two uses of the `On` operator is slightly more complex. The first use is as a *restrict to* operation, which performs the opposite operation of the last use of `Extend`. Given a `Collection` defined on a set, we can use the `On` operator to extract a subset of the `Collection` copied to a new variable, or to be used in some calculations. For instance, in line 31 from Listing B.1 stating

```
dir_sign = bf_sign On dirichlet_boundary
```

we have that `bf_sign` is a `Collection Of Scalar On BoundaryFaces()` which is evaluated `On dirichlet_boundary`, which is a subset of `BoundaryFaces()` provided by the user. After evaluating this line, `dir_sign` becomes the subset of `bf_sign` that corresponds to the faces stored in `dirichlet_boundary`.

The other use of `On` is as an *evaluate on* operation. The right hand set do not need to be a domain with unique cells or faces, but can be a `Collection` on which the left hand side should be evaluated. If `u` is a variable defined as `Collection Of Scalar On AllCells()` then the following line is perfectly legal

```
first_u = u On FirstCell(InteriorFaces())
```

Since a `Cell` can be `FirstCell` to multiple faces, the right hand side is not a domain, but can rather be looked at as a set of `Cell` IDs. Since each element in `u` corresponds to one `Cell` ID as well, we fill `first_u` with the elements in `u` corresponding to the `Cell` ID in the set. This is therefore not a one-to-one operation, but rather of type one-to-many.

The Parameter File

The parameter file is given as argument to the executable Equelle simulator, and is the file containing all runtime variables needed by the program. This is where the grid is defined and where the user provide simulation input such as for example initial conditions.

The grid can be given in two different ways. The first is as a Cartesian grid where the user defines `nx`, `ny` and, if the grid is given in three dimensions, `nz` as well as `dx`, `dy` and `dz` for cell sizes. The Equelle program will then construct the grid from these parameters and store it as an unstructured grid. The alternative is to read the grid from file by using

```
grid_filename=myGrid
```

in the parameter file.

The parameter file also gives us the opportunity to give the user control over other aspects of the simulator execution as well. This can for example be information related to linear solvers, such as which solver and preconditioner to use, and what the accuracy of the solution should be. It can also state whether the arguments to the `Output` function of type `Collection Of Scalar` should be written to file or to screen.

3.2 The Equelle Compiler

The Equelle compiler consists of two distinct steps, namely the front-end and the back-end. In short, the front-end translates the Equelle source code to C++ code using types and classes from the requested back-end. The C++ file is then linked with the back-end library and compiled to an executable. The back-end is designed as a library for each computational platform, and therefore the user has to specify which platform the Equelle program should be compiled to in the front-end step. The entire process is shown in Figure 3.2.

The front-end is very similar to the typical compiler front-end described in Section 2.4. The compiler performs lexical and syntax analysis in order to classify each token in the source code and matches them to the programming statements from the grammar in order to build the abstract syntax tree. The semantic analyzer then assures that the operations requested in the source code are allowed with respect to their types as described in the previous section.

Currently the Equelle compiler does not perform optimizations on the abstract syntax tree. We have chosen to include optimization in Figure 3.2 nonetheless, as it is expected to be included in the future. In Section 6.3.1 we discuss how doing optimizations in the front-end potentially may give some performance gains, but it is however not within the scope of this thesis to implement.

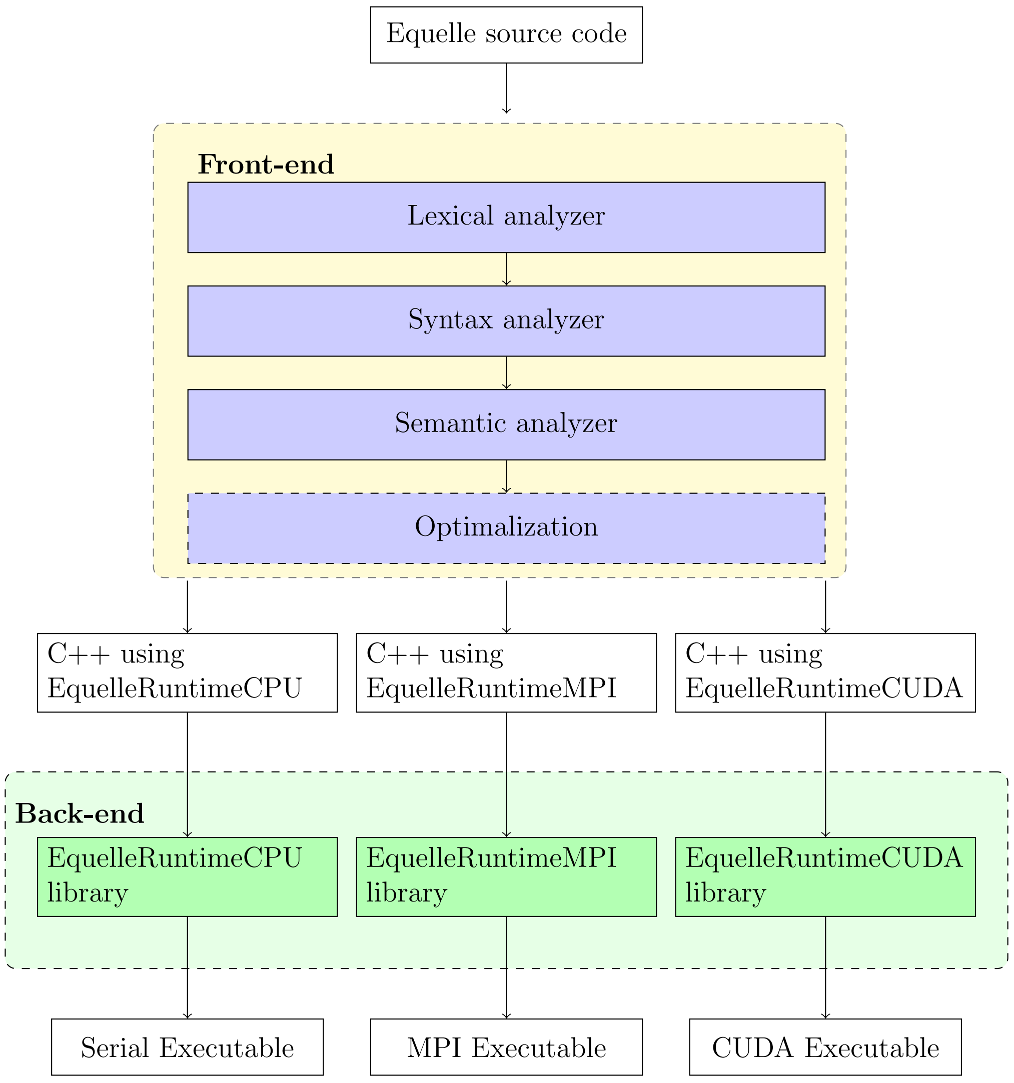
There are several possible outputs available from the front-end, depending on which hardware we want to deploy the simulator on. The choice of back-end is done by adding a flag when we compile the Equelle source code, and we get C++ files as intermediate code. These files include different libraries from the Equelle back-end, depending on the targeted platform. We will now take a closer look at the intermediate representation.

3.2.1 Intermediate Representation

As we have already mentioned, the intermediate code produced by the Equelle compiler's front-end is a C++ program. It is translated from the Equelle source code in a line wise one-to-one fashion. The Equelle functions and types are easily recognized in the C++ program, and the readability is therefore quite high. We use classes and type definitions to handle the Equelle types, so that for example a `Collection Of Scalar` variable becomes an instance of the C++ class `CollOfScalar` in the intermediate code.

The Equelle functionality is handled in two ways. The built-in functions and grid operators are all wrapped in an `EquelleRuntime` class, so that all Equelle function calls are replaced with a call to the corresponding member function from `EquelleRuntime`. Arithmetic operations and comparison expressions on the other hand, are handled by operator overloading.

An example of the intermediate code is shown in Listing 3.2, where the Equelle function `computeBoundaryFluxes` given in Listing 3.1 is translated by the front-end. The first


Equelle source code
Front-end
Lexical analyzer
Syntax analyzer
Semantic analyzer
Optimalization
C++ using EquelleRuntimeCPU
C++ using EquelleRuntimeMPI
C++ using EquelleRuntimeCUDA
Back-end
EquelleRuntimeCPU library
EquelleRuntimeMPI library
EquelleRuntimeCUDA library
Serial Executable
MPI Executable
CUDA Executable
Figure 3.2: The Equelle compiler

thing we notice is how the function call is written in the same way in C++ as in Equelle. This comes from the C++11 lambda syntax, making it possible to define functions inside a given scope. Notice also how variable names are preserved, and the member functions in the Equelle runtime are easily recognized. The front-end also inserts a conservative amount of parentheses. The complete intermediate code can be seen in the Appendix in Listing B.3, where the complete Equelle program is given in Listing B.1.

Listing 3.1: An Equelle function computing boundary fluxes taken from the heat equation simulator.

```
computeBoundaryFlux : Function(u : Collection Of Scalar On AllCells())
                        -> Collection Of Scalar On BoundaryFaces()
computeBoundaryFlux(u) = {
     u_dirbdycells = u On (bf_cells On dirichlet_boundary)
     dir_fluxes = (btrans On dirichlet_boundary) * dir_sign * ...
                  (u_dirbdycells - dirichlet_val)
    -> dir_fluxes Extend BoundaryFaces()
}
```

Listing 3.2: The intermediate code for the Equelle source code shown in Listing 3.1. The Equelle runtime class is here represented by the variabel `er`.

```
std::function<CollOfScalar(const CollOfScalar&)> computeBoundaryFlux = [&](const
    CollOfScalar& u) -> CollOfScalar {
    const CollOfScalar u_dirbdycells = er.operatorOn(u, er.allCells(),
        er.operatorOn(bf_cells, er.boundaryFaces(), dirichlet_boundary));
    const CollOfScalar dir_fluxes = ((er.operatorOn(btrans, er.boundaryFaces(),
        dirichlet_boundary) * dir_sign) * (u_dirbdycells - dirichlet_val));
    return er.operatorExtend(dir_fluxes, dirichlet_boundary, er.boundaryFaces());
};
```

This intermediate representation has several advantages. The most important is that it makes it easy to provide new back-ends. By wrapping all the Equelle functionality inside Equelle-like C++ classes, we are able to define these classes according to the requirements of each back-end. A back-end will therefore be represented by a library consisting of the `EquelleRuntime` class and the classes for the Equelle types. In order to create a new back-end, we create a new library using the same names on the classes and giving them the same user interface. The implementations of the classes and their functions are then back-end specific.

Since all back-end specific code is abstracted by libraries with the same user interface, we can produce intermediate code for different back-ends with only minimal differences. The only differences between the different back-ends will be

- use of a class `EquelleRuntime<back-end>` as the main runtime variable and in function declarations.
- the need to `#include` different back-end specific libraries.
- usage of different namespaces that separates the back-end libraries from each other. This is also why we safely can give the Equelle classes the same names in the different back-ends.

All these differences are code that are independent of the contents of the Equelle source file, and the generated code itself is therefore exactly the same for any back-end.

Another advantage is that the intermediate code is very easy to generate. Since every Equelle operation has a corresponding member function or operator in the `EquelleRuntime` class, the intermediate representation becomes very readable. It can therefore be written by humans as well as the compiler. Because of this, it is easy for users to expand their Equelle programs to do things that is currently not in the scope of the language. Examples of this can be to add visualization of the result inside the main loop, add timing on specific parts of the simulator, or make function calls to other C++ code. The readable intermediate representation also makes it possible to use Equelle as a domain-specific *embedded* language, or to easily wrap the Equelle simulator inside a larger program.

There are also disadvantages by using this intermediate formate. It restricts the front-end's possibility to do Equelle specific or back-end specific optimizations. In Section 6.3.1 we will illustrate what kind of optimizations we might want the front-end to perform.

Chapter 4

CUDA Back-End

This chapter will give a presentation of the CUDA Back-End for the Equelle compiler. We will start by going through the starting point and the design choices that the back-end is built on, before we dig into the details. We will also explain the development process, and the methods and software developments techniques used throughout the project.

At the starting point of this master project, Equelle had only a serial back-end. The compiler was designed for correct evaluation of all functionality that should be in the language, so that prototypes and test simulators could be written, among others the shallow water simulator written in the Specialization Project [20] done as a preparation for this thesis. The format of the intermediate code as well as the idea of how to add new back-ends (as described in Section 3.2.1) were already planned, and we decided to design the CUDA back-end according to this.

4.1 Development Process

Development Tools and Techniques

The Equelle project has from the beginning been an Open Source project managed in a Git system. Git is a **version control software** which also makes distributed software development easy, letting several people easily contribute to the same project. One of the most useful features in Git is that it is easy to go back to a previous version of your project. In case you do some changes at one point that later do not seem like a good idea after all, the version control system makes it easy to restore that part of the code. Equelle is found on the Git based hosting service GitHub [16], where the CUDA back-end developed through the work of this master project is found under the directory

```
equelle/backends/cuda
```

One problem in software developing is to prove correctness of the software. This is important for several reasons. The most obvious reason is that the software has to be correct in order to fill its purpose. A second reason is that during development, finding bugs are easier if the developers have proof of which functions behave correctly and which do not.

A widely used method for coping with this is the method of **unit testing**. The idea behind this method is that for every function that is added to any part of the software, a series of tests are written such that if these tests are passed then the function will work for all possible cases. For each time new functionality is added to our back-end all tests for all functions are executed, so that if a new functionality breaks correctness of an already implemented function, we will find out immediately and even more important we see exactly what breaks in which function. This makes debugging easy and serves as a proof of correctness for software releases.

In this project we have used unit tests to *some* extent. Since all functions developed for the CUDA back-end are either CUDA kernels, CUDA library calls, or functions that set up and launch kernels, and most of them deeply depend on a grid, creating special cases are hard. The way we have made tests are by first using a grid which is small enough that we can compute the results by hand, and then check that the CUDA back-end finds the same results. Later we have used the serial back-end for comparison on a larger grid, ensuring that our new back-end compute the same results as the serial back-end. Even though these tests do not include special cases, we get the confirmation that the primary use of the functions work as they are supposed to. We run these tests every time we do changes in the code, and that way make sure that we do not break correctness of functions by changing other parts of the program.

The library developed in this project is not just a concept study, but is meant to be an integrated part of the Equelle software. This means that it is important for the source code to be well organized and structured, as well as following "best practice". It has to be easy to navigate through the source code folders, and the code has to be commented such that other programmers easily can extend the library, make optimizations or fix bugs. We have therefore made an effort in writing an extensive **documentation** for the library, so that it can be read by documentation generators such as `doxygen`. See Appendix A for how to get the documentation as a pdf document or as html.

Profilers are the primary tool for code optimization, and are used for analysing the work flow and resource consumption of computer programs. They let the user check the number of times instructions or functions are called, what the program does during any time of the execution, and how much time is spent on each task. The most useful information we can get from a profiler is what the main bottlenecks or hotspots of a program are. These are the main limiting factors when it comes to performance. We want to know what functionality is the most time consuming part of the program, and why is it so.

In order to see why this information is important, consider a program consisting of parts a and b where a counts for 95% of the program's execution time. We get very limited benefit by optimizing part b, since no matter how well we do it we can obtain no more than a 5% gain in performance. If a, on the other hand, was optimized to only run twice as fast, the entire program would execute in almost half the time! Most programs are more complex than this example, and it is not always straight forward to find which part is most resource demanding, and this is why we use a profiler.

During the development process for this thesis, we first implemented the functionality so that the back-end would produce the correct results for all functionalities. Then we used the Nvidia Visual Profiler on a test simulator in order to detect performance bottlenecks. When we got an overview of which parts of the back-end that were limiting the performance of the simulator, we considered how we could redesign those functions in order to come up with better solutions.

We have used three external CUDA libraries in the implementation of this back-end, namely **Thrust**, **cuSPARSE** and **Cusp**. Thrust[4] is a C++ template library providing generic algorithms and data-structures similarly to the C++ Standard Template Library, but with the additional possibility to use the GPU. The cuSPARSE library[2] provides functions for basic linear algebra for sparse matrices on the GPU. Our application will depend on its functions for sparse matrix - sparse matrix additions and multiplications especially. The third library, Cusp[6], contains CUDA implementations of some iterative solvers of linear systems, as well as some preconditioners. This is used for implementing the Equelle function `NewtonSolve` which is needed for implementing implicit methods. Thrust and cuSPARSE are available inside of the complete CUDA package, but Cusp has to be downloaded separately. The use of these libraries therefore add one extra dependency to the Equelle compiler.

Overall Design

We base the CUDA back-end on the intermediate representation described in Section 3.2.1. We therefore implement a library containing the same types as defined there, and with an `EquelleRuntimeCUDA` class taking care of the built-in functions from Equelle. We make sure to give the runtime class the exact same user interface as is given to `EquelleRuntimeCPU`, which implements the serial back-end.

We implement the minimal required changes to the front-end, so that the CUDA intermediate code uses the `EquelleRuntimeCUDA` class instead of `EquelleRuntimeCPU`. We also wrap the entire CUDA back-end in the new `equelleCUDA` namespace, and make the front-end generate code for using it instead of the serial back-end's namespace. We also edit the list of headers and libraries that has to be included in the C++ file.

Almost all operations in Equelle are of an embarrassingly parallel nature. For every function that returns a `Collection Of <something>`, each element in that collection is independent of any of the other elements in the output. This means that all operations are well suited for execution on the GPU. The most straight-forward way to do this is therefore to make each such function make a call to a CUDA kernel or library function which performs this operation.

There is however one reason for why we might not want to implement the back-end according to the given intermediate code. The reason is that launching a CUDA kernel comes with an overhead. The way high-performance code for GPUs should be written is with as low a number of kernel launches as possible. Combining kernels to lower the total number of kernel calls is a commonly used optimization technique for GPU computing and is called kernel fusion.

So why do we still design the compiler the way we do? First of all, if we want to have multiple operations inside the same kernel, we have to generate the intermediate code in a completely different way. That would require adding back-end specific optimization techniques to the front-end. Instead of generating function calls for Equelle functions, the front-end would have to analyse which code blocks that would not need global synchronizations, and generate tailored CUDA kernels for these code blocks. The process of implementing this in the compiler could be a topic for a master thesis by itself, and is therefore outside of our scope. We will however briefly look at why this is an attractive feature in Section 6.3.1.

Since the intermediate code is designed such that we can implement any back-end to match it, it comes with a lot of new type names. Since a `Collection Of Scalar` could be an array stored on the CPU, the GPU, or distributed across several nodes in a cluster, it has its own type `CollOfScalar` in the intermediate code. That way it is up to the back-end designer to say what a `CollOfScalar` is in any given back-end. The same goes for single scalar values and strings, which are called `Scalar` and `String` respectively in the intermediate code.

Because of this, all types from the intermediate code are represented as classes or standard types through the `typedef` keyword. The Equelle functionality is then implemented through member functions in the runtime class, which again use calls to member functions in the other classes in our back-end. The exception from this is the CUDA kernels, which can not be implemented as members of a class.

Even though CUDA kernels have to be implemented as independent functions, we still want to organize our code in a way that shows that certain kernels in principle belongs to certain classes. We therefore wrap the kernels inside namespaces called `wrap<class>`. For instant, the CUDA kernels needed by the member functions of the class `DeviceGrid` are found in the namespace `wrapDeviceGrid`.

Since all data collections are stored linearly in memory, and since Equelle is meant not to require structured grids, we use one dimensional block and grid sizes for all kernels. We implement the back-end to use a fixed number of 512 threads for each block, and then launch the smallest number of blocks that still is sufficient to do the computations. Most kernels are implemented such that each thread is responsible for one element in the output, and we find the required number of blocks by

$$num\_blocks = floor\left(\frac{collection\_size + block\_size - 1}{block\_size}\right). \tag{4.1}$$

If the collection size is a multiple of the block size, we get a number which is almost one block too much, but round it down to the matching number of blocks. If however we have a data size that is one more than a multiple of the block size, we will wind up with one extra block, and thus launch 511 more threads than strictly required. This is why each CUDA kernel needs to check that its thread ID is less than the collection size we want to evaluate, and this is standard procedure in GPU computing.

Instead of repetitively rewriting Equation (4.1), we create a `struct` holding two variables of the `dim3` type. This struct, named `kernelSetup`, is initialized by the number of threads we want to launch. This helps us avoid potential errors caused by typos, as well as making it faster to program new kernels.

The same principle is followed for the code needed for finding the identity of a CUDA thread. We therefore implement the function `myID()` similarly to what we did in the example from Listing 2.1.

Memory Management

In order to get high performance it is important to minimize the data transfer between the host and device memory. Since most data types in Equelle consist of arrays of the same size as the number of cells or faces, and we here provide functions where they are modified or used in calculations by the GPU, we should keep all such data arrays in the GPU memory only. Having such arrays in host memory should only be done in cases where there are some interaction with the screen or file system, as well as initialization of the grid.

An Equelle program usually have the following structure:

1. Initialize or read the grid.
2. Read input data.
3. Definitions and calculations of global variables or grid subsets.
4. Declaration and implementation of functions containing pure calculations.
5. Iterate through some time steps, from where the above functions are called.
6. Writing results to screen or to file for each time step.

From the above list the only parts that requires the data arrays to be accessed by the CPU are number 1, 2 and 6.

Initialization of the grid can be done in two ways. Either by building a Cartesian grid from input parameters `nx, ny, nz, dx, dy, dz`, or by specifying a file containing coordinates from which the grid is created. In both cases we use the CPU to construct the grid and the neighbour relations between cells. When these calculations are done, we transfer the data arrays to the GPU.

For input and output functions we rely on the CPU to read or write the given data to or from file, and we therefore require copying data between device and host. The input variables are simply read by the CPU and transferred straight to the device, and output is done by a straight copy back to the host followed by writing to the file or the screen.

In order to store data we use the CUDA and Thrust APIs, depending on what we are storing. The CUDA API provides C like memory management, which requires manually allocation and deallocation of memory before and after use. It is done with the functions `cudaMalloc` and `cudaFree`. This is efficient and its low level interface makes it easy to create functions using the data array. The drawback is that it may be error prone, as it might cause memory leaks. The CUDA API is used in the data arrays storing `Collection Of Scalar`, where allocation is only done in constructors, and freeing of memory is implemented in the destructor. Doing this in a safe and correct way solves the potential memory leak problem.

The class used to store `Collection Of Face` and `Collection Of Cell` uses the Thrust container class `thrust::device_vector<int>`. Here we do not have to worry so much about memory leakage, but rather deal with a more verbose set of function calls in order to get access to the raw data buffers. The reason why the Thrust library was used for these collections is that the Thrust library provides algorithms that can prove useful for processes such as sorting, finding subsets, etc, which make more sense for a set of indices than for a set of scalar values.

The CUDA API is also used for data transfer between host and device, by the function `cudaMemcpy`. As well as the pointers to the beginning of the source and receiving data blocks, and the amount of data, the function takes a flag as input, indicating which direction the data is copied. This also gives us the ability to easily copy data between different memory locations within the GPU.

All variables and class members which are just a single variable and not part of an array are stored on the host. Since the program is designed such that we call a lot of kernels, the program control will often be at the CPU. Even though it is the GPU that do all the hard work and big computations, the CPU is in charge of what is executed on the GPU and when. In order to do this, the CPU will have to do some small evaluations between the kernels, and sending a few extra scalar parameters in a kernel call is not performance critical.

The only data array that is stored entirely on the CPU is data of the loop controlling type `Sequence Of Scalars`. The values here are read sequentially one at each loop iteration, and therefore comes in between the kernel calls. It would therefore not make sense to have these values stored on the device.

We will now start to describe the library which is the CUDA back-end the Equelle compiler consists of.

4.2 The Equelle Type `Collection Of Scalar`

One of the most important classes in the CUDA back-end, is the class `CollOfScalar` which represents the data type `Collection Of Scalar` from Equelle. Objects of this class are involved in the majority of Equelle function calls, and this is where all intermediate and final results are stored. It is implemented with few member functions other than constructors, destructor and get-functions. We also overload arithmetic and comparison operator to operate on this class.

The most interesting feature of this class is that it has an automatic functionality for storing not only the values of the variable for the given set of cells or faces, but also its Jacobian matrix. To understand the need for this functionality, we need to take a look on how we find implicit solutions and the concept of Automatic Differentiation.

4.2.1 Implicit Solution

When we write simulators using an implicit method, we get a system of non-linear equations. For a cell Ω_i the equation from a finite volume method will be on the form

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{|\Omega_i|} F_i^{n+1}, \tag{4.2}$$

as we saw in Equation (2.9). Here Q_i^n denotes the main variable in cell Ω_i for time t_n, and F_i^{n+1} denotes the total flux through the boundary of the cell, $\partial\Omega_i$, calculated from Q^{n+1}. Unless the flux term is very simple, F will be a non-linear function depending on the neighbour cells of Ω_i. Assuming a grid with N cells, we collect all the main variables for the same time step in a vector $\mathbf{Q} = [Q_1, Q_2, \dots, Q_N]^T$. We then end up with a system of non-linear equation which has to be set up and solved for each time step,

$$\mathbf{Q}^{n+1} = \mathbf{Q}^n - \mathbf{F}(\mathbf{Q}^{n+1}). \tag{4.3}$$

This problem is solved by using a numerical method for finding the root of the residual function $\mathbf{r}(\cdot)$ defined by

$$\mathbf{r}(\mathbf{Q}^{n+1}) := \mathbf{Q}^n - \mathbf{Q}^{n+1} - \mathbf{F}(\mathbf{Q}^{n+1}). \tag{4.4}$$

The approach taken by the serial back-end is to solve (4.4) by using a Newton method, where given an initial guess $\tilde{\mathbf{Q}}^0$ (typically equal to $\mathbf{Q}^n$) we use the iterative method

$$\tilde{\mathbf{Q}}^{k+1} = \tilde{\mathbf{Q}}^k - J_{\mathbf{r}}(\tilde{\mathbf{Q}}^k)^{-1}\mathbf{r}(\tilde{\mathbf{Q}}^k), \quad \text{for } k = 0, 1, 2, ..., \tag{4.5}$$

where $J_{\mathbf{r}}(\tilde{\mathbf{Q}})$ denotes the Jacobian matrix to the vector $\mathbf{r}(\tilde{\mathbf{Q}})$. In order to solve for each iteration we write the above equation as a system of linear equations,

$$J_{\mathbf{r}}(\tilde{\mathbf{Q}}^k)\mathbf{x}^k = \mathbf{r}(\tilde{\mathbf{Q}}^k), \tag{4.6}$$

where $\mathbf{x}^k = \tilde{\mathbf{Q}}^k - \tilde{\mathbf{Q}}^{k+1}$. We thereby solve (4.6) for $\mathbf{x}^k$ using a suitable linear solver, then find $\tilde{\mathbf{Q}}^{k+1}$ from

$$\tilde{\mathbf{Q}}^{k+1} = \tilde{\mathbf{Q}}^k - \mathbf{x}^k. \tag{4.7}$$

We use a stopping criteria for the Newton method by comparing $||\mathbf{r}(\tilde{\mathbf{Q}}^k)||_2$ by some user specific tolerance. Suppose we pass this criteria after j iterations, we let $\mathbf{Q}^{n+1} = \tilde{\mathbf{Q}}^j$.

The problem we are left with is how to generate the Jacobian matrix. The Jacobian matrix has the form

$$J_{\mathbf{r}}(\mathbf{Q}) = \begin{bmatrix} \frac{\partial r_1(\mathbf{Q})}{\partial Q_1} & \frac{\partial r_1(\mathbf{Q})}{\partial Q_2} & \cdots & \frac{\partial r_1(\mathbf{Q})}{\partial Q_N} \\ \frac{\partial r_2(\mathbf{Q})}{\partial Q_1} & \frac{\partial r_2(\mathbf{Q})}{\partial Q_2} & \cdots & \frac{\partial r_2(\mathbf{Q})}{\partial Q_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial r_N(\mathbf{Q})}{\partial Q_1} & \frac{\partial r_N(\mathbf{Q})}{\partial Q_2} & \cdots & \frac{\partial r_N(\mathbf{Q})}{\partial Q_N} \end{bmatrix} \tag{4.8}$$

meaning that $(J_{\mathbf{r}}(\mathbf{Q}))_{i,j} = \frac{\partial r_i(\mathbf{Q})}{\partial Q_j}$ is the residual function for the ith cell differentiated with respect to the solution of the jth cell. If we have a first order method, where the flux through a face only depends on the two cells the face separate, then each residual function will only depend on the variables of the neighbouring cells. Therefore, most of the entries in the Jacobian matrix will be zero, but it would still be unproductive to make the user find all the functions describing the non-zero derivatives we are left with. A solution is the method known as Automatic Differentiation.

4.2.2 Automatic Differentiation

The technique known as Automatic Differentiation (also referred to as Algorithmic Differentiation), or AD in short, consists of the use of dual numbers in order to obtain both the function value and the value of the function's derivative by using a set of programmable tools. It seems like the technique evolved during the '70s and '80s without any distinct inventor. The books by Rall[28] and Griewank[18] are however widely recognized textbooks on the topic.

AD should not be confused by the more known techniques of symbolic derivation or finite differences. Approximating derivatives by using finite differences is done by looking at a point $x + h$ close to x, and finding the Taylor approximation at $x + h$. By choosing h small enough, we can neglect higher order derivatives, and thus get an approximative value of the derivative. Symbolic derivation is done on the algebraic expression for a function, using the chain rule and derivation rules to find the algebraic expression of the derivative. Both these techniques have certain drawbacks, and none of them are identical to AD. Symbolic derivation often uses a lot of memory and results in re-computation of the same expression multiple times. Finite differences on the other hand brings an approximation error into the calculations.

When using AD, we carry numerical data of both the value and the derivative for every variable and write every basic function or operator for such variables to operate on both values. For instance, consider variables x and y with derivative values x' and y', which we want to add and store in variable z. The resulting value is as always $z = x + y$, but what about the resulting derivative? From the rules of derivation we know that derivation is a linear operation, and therefore

$$z' = (x + y)' = x' + y'. \tag{4.9}$$

Considering the same variables and multiplication, we would get

$$\begin{aligned} z &= xy \\ z' &= (xy)' = x'y + xy'. \end{aligned} \tag{4.10}$$

Remember that all x, y, x' and y' are scalar values, and we therefore store only two numbers for each of the variables.

The way we automatically get the derivative value in the computations of a function $f(x)$ is by operator overloading. Let `AD` be a data type with two members of type `double` so that `AD = {double AD.val, double AD.der}`. We then overload the multiplication operator as follows:

Listing 4.1: Pseudo code of overloading of the multiplication operator, in order to do automatic differentiation.

```
AD operator*(const AD lhs, const AD rhs) {
    AD out;
    out.val = lhs.val * rhs.val;
    out.der = lhs.der * rhs.val + lhs.val * rhs.der;
    return out;
}
```

The same technique can be used in the implementation of other common mathematical functions as well, such as $\sqrt{\cdot}$, where

$$\begin{aligned} z &= \sqrt{x} \\ z' &= \frac{1}{2\sqrt{x}} x'. \end{aligned} \tag{4.11}$$

Since x do not denote the primary variable, but any variable with a pre-calculated derivative value, we need to use the chain rule.

Having overloaded all operators and standard mathematical functions, we can write new mathematical functions which use the data type `AD` just as we would for any normal function written for normal data types. A typical function header will be

```
AD f(AD x);
```

In order to get any calculations we need to initialize some variables manually, either as constants inside our function, or as the primary variable given as input to `f`. For these cases we need to explicitly state what their derivative values are, as well as their primary value, and all other variables will get their values and derivatives based on these ones.

First by considering scalar constants, we immediately realize that their derivatives are zero. Second we consider the primary variable. The primary variable is defined as the variable all derivatives are with respect to. Hence, if we let the primary variable be called x, the derivative of this variable will be $\frac{\mathrm{d}x}{\mathrm{d}x} = 1$. Thereby, a scalar constant would have the values $\{c, c'\} = \{c, 0\}$ while a primary variable would have the values $\{x, x'\} = \{x, 1\}$, where x and c would be any values suitable for the application.

Multi-Variable Case

So how does this extend to a situation where we have multiple variables, such as for example a discretized PDE on a grid? Let us define the primary variable as $\mathbf{x} = [x_1, x_2, ..., x_N]$, and consider two other variables $\mathbf{Q}$ and $\mathbf{U}$ defined on the same set as $\mathbf{x}$, with AD types $\{\mathbf{Q}, \mathbf{Q}'\}$ and $\{\mathbf{U}, \mathbf{U}'\}$. The AD type will still be consisting of a value and a derivative as before, however we now need the value to be a vector, and the derivative to be a matrix,

$$\mathbf{Q} = [Q_1, Q_2, ..., Q_N]$$
$$\mathbf{Q}' = \begin{bmatrix} \frac{\partial Q_1}{\partial x_1} & \frac{\partial Q_1}{\partial x_2} & \cdots & \frac{\partial Q_1}{\partial x_N} \\ \frac{\partial Q_2}{\partial x_1} & \frac{\partial Q_2}{\partial x_2} & \cdots & \frac{\partial Q_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial Q_N}{\partial x_1} & \frac{\partial Q_N}{\partial x_2} & \cdots & \frac{\partial Q_N}{\partial x_N} \end{bmatrix}.$$

The variable $\{\mathbf{U}, \mathbf{U}'\}$ is defined similarly.

We now want to define arithmetic operations in term of the AD type similarly to what we did for the single variable case. We start by considering addition here as well. We have component wise additions, and are therefore interested in the gradient vector of $\nabla(Q_i + U_i)$ for each variable, $i = 1, 2, ..., N$. This operation is linear, and we therefore have the same rules for addition in the multi-variable case as in the single variable case

$$\begin{cases} \mathbf{V} & = \mathbf{Q} + \mathbf{U} \\ \mathbf{V}' & = \mathbf{Q}' + \mathbf{U}' \end{cases} \tag{4.12}$$

where $\mathbf{V}$ is found by vector addition and $\mathbf{V}'$ by matrix addition.

Multiplication has to be treated with a bit more care. Since we consider each variable as a collection of values defined on grid subset, we are actually interested in element-wise multiplication. Therefore, the resulting values from a multiplication will be

$$\mathbf{Q} \cdot \mathbf{U} = [Q_1U_1, Q_2U_2, ..., Q_NU_N] \tag{4.13}$$

Let $\{\cdot\}_i$ denote a row of a matrix, letting us write the i'th row of the Jacobian matrix as

$$\{(\mathbf{Q} \cdot \mathbf{U})'\}_i = \left[\frac{\partial}{\partial x_1} Q_iU_i, \frac{\partial}{\partial x_2} Q_iU_i, ..., \frac{\partial}{\partial x_N} Q_iU_i\right]. \tag{4.14}$$

Looking at element j from this row, we get that

$$\frac{\partial}{\partial x_j} Q_iU_i = U_i \frac{\partial Q_i}{\partial x_j} + Q_i \frac{\partial U_i}{\partial x_j}. \tag{4.15}$$

We are therefore able to write each row as

$$\{(\mathbf{Q} \cdot \mathbf{U})'\}_i = U_i \{\mathbf{Q}'\}_i + Q_i \{\mathbf{U}'\}_i, \quad \text{for} \quad i = 1, 2, ..., N. \tag{4.16}$$

This means that we need U_i and Q_i to be multiplied with the i'th row of $\mathbf{Q}'$ and $\mathbf{U}'$ respectively. This can be done by setting up diagonal matrices from the values of $\mathbf{Q}$ and $\mathbf{U}$, and multiply with the diagonal matrices from the left, as

$$(\mathbf{Q} \cdot \mathbf{U})' = \text{diag}(\mathbf{U})\mathbf{Q}' + \text{diag}(\mathbf{Q})\mathbf{U}'. \tag{4.17}$$

The implementation of this `AD` operation can be written in pseudocode as shown in Listing 4.2.

Listing 4.2: Pseudo code for overloading of the multiplication operator, in order to do automatic differentiation in a multi-variable environment.

```
AD operation*(const AD& Q, const AD& U) {
    AD out;
    out.val = elementwiseMultiplication(Q.val, U.val);
    Matrix diagQ = diagonalFromVector(Q.val);
    Matrix diagU = diagonalFromVector(U.val);
    out.der = diagU * Q.der + diagQ * U.der;
    return out;
}
```

Similar rules can be found for division and square root as well. For an element-wise division operation, the Jacobian element at row i, column j becomes

$$\frac{\partial}{\partial x_j}\frac{Q_i}{U_i} = \frac{U_i\frac{\partial Q_i}{\partial x_j} - Q_i\frac{\partial U_i}{\partial x_j}}{U_i^2}. \tag{4.18}$$

By the same arguments as in Equations (4.15), (4.16) and (4.17), we create diagonal matrices of the elements of $\mathbf{Q}$, $\mathbf{U}$ and element-wise $1/\mathbf{U}^2$, as

$$\left(\frac{\mathbf{Q}}{\mathbf{U}}\right)' = \text{diag}\left(\frac{1}{\mathbf{U}^2}\right)\left(\text{diag}(\mathbf{U})\mathbf{Q}' - \text{diag}(\mathbf{Q})\mathbf{U}'\right). \tag{4.19}$$

Same argument for element-wise square root gives us

$$\left(\sqrt{\mathbf{Q}}\right)' = \text{diag}\left(\frac{1}{2\sqrt{\mathbf{Q}}}\right)\mathbf{Q}'. \tag{4.20}$$

When we have defined the rules for all arithmetic operations and basic mathematical functions, we only need to define constants as AD variables with Jacobians only consisting of zeros, and the primary variable with the identity matrix for as Jacobian.

4.2.3 Implementation

We implement our class `CollOfScalar` as a multi-variable AD class, and need therefore member variables to hold a vector or array for the values, and a sparse matrix for derivatives. Both these members should use the device memory to hold all data arrays. We create two new classes `CudaArray` and `CudaMatrix` for this purpose, implementing them with a suitable user interface which makes it easy to include the functionality of AD.

The Class `CudaArray`

The `CudaArray` class is our main container class for storing values on a grid domain. The class contains a device array for holding the object's values, alongside the size of the array, kernel launch size parameters, and a CUDA error checking variable. The class does not have any special member functions besides constructors, data access functions, and a function which copies the device array to host memory. The latter function is for the Equelle function `Output`, and debugging and testing use.

The main Equelle functionality needed for this class however, is operator overloading. We want to use the standard arithmetic operations `{+, -, *, /}` to do the corresponding element-wise operation on the data stored on the GPU. We will implement CUDA kernels for doing the element-wise operations, and overload the operations to create a `CudaArray` object for the result, set up a call to the CUDA kernel with the correct device pointers, before returning the result. We use multiplication as an example to illustrate this, and the implementation is seen in Listing 4.3.

Listing 4.3: Overloading of the multiplication operator for `CudaArray`

```
 1 CudaArray operator*(const CudaArray& lhs, const CudaArray& rhs) {
 2     CudaArray out = lhs;
 3     kernelSetup s = out.setup();
 4     multiplication_kernel <<<s.grid, s.block>>>(out.data(), rhs.data(),
           out.size());
 5     return out;
 6 }
 7
 8 __global__ void wrapCudaArray::multiplication_kernel(double* out,
 9                                                      const double* rhs,
10                                                      const int size)
11 {
12     int index = myID();
13     if ( index < size ) {
14         out[index] = out[index] * rhs[index];
15     }
16 }
```

The code operates in two steps. First, creating a new variable as a copy of `lhs`, and then calling the kernel which computes the element-wise sum. Note here that `wrapCudaArray` is the namespace containing the CUDA kernels related to `CudaArray`, and is therefore not a class. The if-test on line 13 makes sure that we stay inside of the allocated memory for both arrays, since we are likely to deploy more CUDA threads than we need, as discussed in Section 4.1.

We also add functions for reductions. This is done by copying the data array into a Thrust `device_vector` variable and then calling Thrust functions that match the reduction we want to perform. The reduction functions we then implement in `CollOfScalar` only call the corresponding function in its `CudaArray` member.

The class `CudaMatrix`

In order to store sparse matrices containing the derivatives of the scalar collections we define on the grid, we create a class `CudaMatrix` suitable for our use of matrix operations. Since variables in Equelle rarely are connected to other cell variables than its neighbours the matrices will for sure be sparse, and we therefore want an efficient way of storing them. The storage scheme should also fit with the data structures which are used by the libraries we need for both matrix arithmetic and linear solvers. This is why we use the **compact sparse row-major format**, or CSR-format for short, to store our matrices.

The CSR-format is defined by combining three linear arrays to hold the information of all the non-zero elements in the matrix. Given an $n \times m$ matrix with nnz number of non-zero values, the three arrays are:

Values `val` Stores the nnz non-zero values of the matrix, by holding all non-zero values for each row continuously in memory.

Column index `colInd` Storing the column index of the corresponding value in the `val` array.

Row pointer `rowPtr` Stores the indices in the two other arrays which represents the beginning of each row. It also contains where the last row ends, so that this array is of size $n+1$, and `rowPtr[i+1] - rowPtr[i]` represent the number of non-zero values for row i.

Given the following matrix as example,

$$\begin{bmatrix} 1.2 & 0.0 & 0.0 & -0.2 \\ 0.0 & 4.8 & 6.0 & 3.2 \\ 1.1 & 0.0 & 0.0 & 0.0 \\ 0.0 & -3.4 & 5.3 & 0.0 \end{bmatrix}$$

the corresponding storage in CSR-format will be:

```
val = [1.2, -0.2, 4.8, 6.0, 3.2, 1.1, -3.4, 5.3]
colInd = [0, 3, 1, 2, 3, 0, 1, 2]
rowPtr = [0, 2, 5, 6, 8]
```

The three arrays are stored on the device, and on the host we store the number of rows and columns, the number of non-zeros, error handling variables and a cuSPARSE description. We design the class's constructors such that implementing other functions in Equelle will be as easy as possible. The following is the most important constructors:

`CudaMatrix()` Empty matrix, without any allocated device memory. We will come back to the meaning of an empty matrix.

`CudaMatrix(const int size)` Identity matrix of the given size.

`CudaMatrix(const CollOfScalar& coll)` Diagonal matrix of the same size as `coll`, where the diagonal elements are the same as the values stored in the given collection.

`CudaMatrix(const CudaArray& array)` Diagonal matrix of the same size as `array`, where the diagonal elements are the same as in `array`.

`CudaMatrix(const CollOfBool& coll)` Diagonal matrix of the same size as `coll`. The elements are 0.0 or 1.0 entries according to the input.

`CudaMatrix(const thrust::device_vector<int>& vec, const int size)` Restriction matrix with `vec.size()` rows and `size` columns, based on the indices from `vec`.

We will take a closer look at restriction matrices in Section 4.3.2, and show how the boolean matrices are used in Section 4.4.2.

The class itself does not contain a lot of member functions. We implement functions to access the raw data buffers, a check for whether it is empty, and a function to copy the matrix to the host. The latter is primary for testing purposes. In addition, we implement a function to get the transposed. The need for the transpose is discussed in Section 4.3.2, and we look more into the implementation details in Section 6.1.1.

The rest of the functionality of this class is through the overloaded operators. They are added as `friend` functions to the class, so that they have direct access to its private members. When adding or multiplying sparse matrices, we might get more non-zero elements then we had in the original matrices. The operations are implemented by functions from cuSPARSE, which offers a two-step process for these operations. Parts of the sparse matrix multiplication code can be seen in Listing 4.4. We have omitted the parts of the code which validates the input parameters and catch errors from the cuSPARSE and CUDA function calls. The two main function calls to cuSPARSE also require long lists of input parameters, which we also have omitted for readability. The complete parameter lists can be seen in the cuSPARSE documentation guide [2].

Listing 4.4: Overloading of the multiplication operator for `CudaMatrix` by using the two step procedure from the cuSPARSE library. The main cuSPARSE function calls requires a long list of input variables, which is omitted here for readability.

```
 1 CudaMatrix equelleCUDA::operator*(const CudaMatrix& lhs,
 2                                   const CudaMatrix& rhs)
 3 {
 4     /* Check for empty matrix */
 5     /* Check for correct sizes */
 6
 7     // Create an empty matrix of correct sizes.
 8     CudaMatrix out; out.rows_ = lhs.rows_; out.cols_ = rhs.cols_;
 9
10     // Step 1) Find nonzero pattern of output
11     cudaMalloc( (void**)&out.csrRowPtr_, (out.rows_+1)*sizeof(int));
12     int *nnzTotalDevHostPtr = &out.nnz_;
13     cusparseSetPointerMode(CUSPARSE, CUSPARSE_POINTER_MODE_HOST);
14     cusparseXcsrgemmNnz( /* input parameters omitted */
15                          out.csrRowPtr_, nnzTotalDevHostPtr );
16
17     // Set number of non-zeros and allocate memory for the other arrays.
18     out.nnz_ = *nnzTotalDevHostPtr;
19     cudaMalloc( (void**)&out.csrVal_, out.nnz_*sizeof(double));
20     cudaMalloc( (void**)&out.csrColInd_, out.nnz_*sizeof(int));
21
22     // Step 2) Multiply the matrices:
23     cusparseDcsrgemm( /* input parameters omitted */
24                       out.csrVal_, out.csrRowPtr_, out.csrColInd_ );
25
26     return out;
27 }
```

Listing 4.4 shows how we immediately are able to allocate the memory for `csrRowPtr` as it does not depend on the number of non-zeros. We then make the cuSPARSE function call at line 15 to get the number and pattern of non-zeros in the result, where lines 13 and 14 are required set up. We are then able to allocate memory for the arrays containing the values and column indices, before calling the second step of the procedure at line 24. The process of getting the correct value for `out.nnz_` is system dependent, and line 13 is therefore a simplification. The complete code can be seen on Github in the file.

`.../equelle/backends/cuda/src/cudaMatrix.cu`

Addition of two sparse matrices follows a similar cuSPARSE pattern, which really computes $C = \alpha A + \beta B$, for scalar α and β. The two step function is only implemented once in the function `cudaMatrixSum`, where $\alpha = 1.0$ and β is taken as input. The reason for this is that `cudaMatrixSum` then can be used for addition using $\beta = 1.0$ and subtraction with $\beta = -1.0$.

As mentioned in the constructor list above, we include the possibility to create empty matrices from the default constructor. We define an empty matrix as a matrix which is of a legal size according to any arithmetic operation, but containing only zeros. This means that if we have matrices A and B, and B is empty, their sum is $A + B = B + A = A$. Multiplication by an empty matrix from both the left or right results in an empty matrix. These rules are implemented in the overloaded operators.

Implementing `CollOfScalar`

After creating good interfaces for the classes `CudaArray` and `CudaMatrix`, it is quite straight forward to overload the arithmetic operators for `CollOfScalar` to use automatic differentiation. We simply follow the pseudo code in Listing 4.2 for the example class `AD`, as well as the evaluation rules for derivatives. The implementation of multiplication for `CollOfScalar` is shown in Listing 4.5.

Listing 4.5: Overloading of the multiplication operator for `CollOfScalar` using automatic differentiation. We compute the derivatives as long as at least one of the input parameters contains derivatives.

```
CollOfScalar equelleCUDA::operator*(const CollOfScalar& lhs,
                                    const CollOfScalar& rhs)
{
    CollOfScalar out;
    out.val_ = lhs.val_ * rhs.val_;
    if ( lhs.autodiff_ || rhs.autodiff_ ) {
        out.autodiff_ = true;
        CudaMatrix diag_u(lhs.val_);
        CudaMatrix diag_v(rhs.val_);
        out.der_ = diag_v*lhs.der_ + diag_u*rhs.der_;
    }
    return out;
}
```

We check the boolean `autodiff_` member if the input variables have Jacobian matrices or not. The member variable is `false` by default, so if none of the input variables have derivatives, we do not give the output any derivative either. Otherwise we create diagonal matrices from the input's values, and follow Equation (4.17). Note that we compute the derivative of the output even though one of the input values do not have a derivative. This creates no problems for us because of the way we treat empty matrices in arithmetic operations.

4.3 The Grid

The grid is the most important data structure in any Equelle program. All other data types rely on the grid, as the number of cells and faces gives the sizes of other collections, the `Vector` type has same dimension as the grid, and most of the built-in Equelle functions use grid relations of some sort. We therefore need a smart way of storing the grid, which also gives us access to the information that we will need. This section considers the grid storage scheme that we use, and also implementation of some of the central grid related functions available in Equelle.

Since Equelle programs are not limited to structured Cartesian grids, we treat any grid as unstructured grids. This is similar to the serial back-end, which uses a struct of type `UnstructuredGrid` from the OPM (Open Porous Media) Initiative[32] software. Since the serial back-end is able to do all operations needed by Equelle by using this data structure, we have decided to implement a class with the same member arrays as `UnstructuredGrid`, but where the arrays are stored on the GPU. Since the grid is stored on the GPU, or the device, we name the class `DeviceGrid`.

The first data we store in this class are some simple integers denoting the size of the other arrays and the size of the grid. These are `dimensions_`, total number of cells and faces named `number_of_cells_` and `number_of_faces_` respectively, as well as the size of the `cell_faces_` array, named `size_cell_faces_` that will be described below. Since these are simple integers, we keep them on the host. The arrays stored on the device are the following:

- `cell_centroids_` Contains the coordinates of the centroid of every cell. The size depends on the dimension of the grid, as it stores three values per cell in 3D grids, but only two values per cell if the grid is 2D. The size of the array will therefore be `number_of_cells_ * dimensions_`.

- `face_centroids_` Contains the coordinates of the centroid of every face, and therefore `number_of_faces_ * dimensions_` elements of type double.

- `cell_volumes_` Stores the volume of every cell. For structured grids, this will be an array consisting of the `number_of_cells_` copies of the same number.

- `face_areas_` Stores the area of every face.

- `face_normals_` The non-normalized normal vector for each face. In order to get the normal vectors in unit length, we have to divide them by `face_areas_`. The size of this array will be `dimensions_ * number_of_faces_`.

- `cell_facepos_` This array contains indices describing the range in the `cell_faces_` array belonging to each cell. It can also be used to find the number of faces per cell, as cell *i* has `cell_facepos_[i+1] - cell_facepos_[i]` faces.

- `cell_faces_` Contains indices of the faces surrounding all the cells in the grid. The face indices for cell *i* is found in the interval `cell_faces_[cell_facepos_[i]]` to `cell_faces_[cell_facepos_[i+1] - 1]`. The size of this array is found in the last element of `cell_facepos_`.

- `face_cells_` This is the inverse array of `cell_faces_` and lists the cells on each side of every face. The array is `2 * number_of_faces_` long, and gives the two cells such that the normal vector points from the first to the second cell. Hence `FirstCell` of face *f* is `face_cells_[2*f]`, while `SecondCell` of face *f* is `face_cells_[2*f+1]`. If a face is on the boundary either the first or second value will be `-1`, denoting *no cell*.

In addition to holding the grid data, this class is given the interface for Equelle functions that are heavily depending on the grid. For instance, we use this class to find subsets of faces and cells, `FirstCell` and `SecondCell` for faces, and `On` and `Extend` operations. It is straight-forward to copy data such as sizes of cells and faces, centroids and normals to the variables that request them, as these results are already stored in the class.

In order to describe the interface however, we need to give a small explanation of the types `CollOfCell` and `CollOfFace`. These types are implemented as different versions of a template class `CollOfIndices<int codim>`, where `codim` is the codimension. The codimension of a subspace L of a vector space V is given as

$$\text{codim} L = \dim V - \dim L. \tag{4.21}$$

This means that given a 3-dimensional grid where cells are also given in 3-dimensions, the codimension of a cell is 0. A face on the other hand, is a 2-dimensional subset of the grid, and therefore has codimension 1. Considering a 2-dimensional grid, we see that cells becomes 2-dimensional as well, and faces only 1-dimensional. However, the codimension of cells and faces are the same. We therefore define the following:

```
typedef CollOfIndices<0> CollOfCell;
typedef CollOfIndices<1> CollOfFace;
```

and when we talk about `CollOfIndices` we refer to any of the two.

The class `CollOfIndices<codim>` makes no use of the codimensional template. The class is simply an interface towards a Thrust device vector storing indices, and the `codim` denotes simply if these indices are cell indices or face indices. We also note that in cases where we operate on `AllCells()` or `AllFaces()`, we will get a `CollOfIndices` containing $[0, 1, 2, ..., N-1]$, where N is number of cells or faces. We say that these collections are full, and instead of storing an array with `indices[i] = i` we simply set a flag telling us that the collection is full, combined with a size parameter to tell us how many indices the collection should contain. For collections that are not full, we require that the stored indices are sorted.

The reason that we want to separate `CollOfFace` and `CollOfCell` is that some Equelle functions are only legal for a set of faces but not for a set of cells, and opposite. Allowing both kind of indices for operations where only one of them makes sense could easily create strange bugs in the back-end, and for any future users.

We will go through some of the functions to illustrate how easily we get the different Equelle functions from the grid arrays. Most of the functions are very straight-forward and embarrassingly parallel, and we are therefore able to just give the algorithms by only using the mentioned `DeviceGrid` arrays for a single output element, instead of listing the kernels in their full form.

4.3.1 Implementation of Equelle Grid Functions

Some central built-in functions in Equelle are the functions to specify subsets of cells and faces in the grid, and common subsets are the interior and boundary sets. These are also functions that are not straight forward to parallelize because of two reasons:

1. We do not know how large the results will be.

2. We need the result to be sorted, in order to use `Collection Of Scalar` variables related to these sets.

The first issue can be resolved by computing the number of boundary cells and faces during the grid construction. This would add a one time cost, but could potentially help us during each function call. The second reason is a bigger problem. The process of finding out if cell i is a boundary cell can be done without dependencies of the other cells, but we can not know where in the sorted result we should put i.

The solution we have chosen is as follows, using `BoundaryCells()` as an example.

- Create a Thrust device vector of size `number_of_cells_` where all elements are `number_of_cells_`. Since indices have base zero, `number_of_cells_` is a non-legal index.
- Call a CUDA kernel using one thread for each cell in the grid, which checks if the cell is a boundary cell or not. If the cell is a boundary cell we let the corresponding element in the device vector become the cell's index, otherwise we keep it unchanged.
- Use the Thrust algorithm `remove_if` to extract all elements of the vector that do not have the value `number_of_cells_`.

The resulting device vector will then contains the sorted indices of the boundary cells in the grid. The same method also applies for `BoundaryFaces()`, `InteriorFaces()` and `InteriorCells()`.

The kernels are based on the information stored in the `DeviceGrid` arrays. Note first that the Equelle functions `FirstCell` and `SecondCell` are able to do direct look-up in the array `face_cells_` based on their `Collection Of Face` input. Since this array contains the value `-1` where there are no cells, we can use this to look for cells outside the boundary. Listing 4.6 shows the complete kernel for finding the boundary faces. Finding `InteriorFaces()` is done similarly, but assigning to `b_faces` only if both comparisons are false.

Listing 4.6: CUDA kernel for finding the faces located at the grid boundary.

```
__global__ void boundaryFacesKernel( int* b_faces ,
                                     const int* face_cells ,
                                     const int number_of_faces)
{
    const int face = myID();
    if (face < number_of_faces) {
        if ( (face_cells[2*face] == -1) || (face_cells[2*face + 1] ==
            -1) ) {
             b_faces[face] = face;
        }
    }
}
```

In order to find `BoundaryCells()` we have to check if any of the faces surrounding the cell is a boundary face. In the array `cell_faces_` we have access to these face indices, where the array `cell_facepos_` describes where in `cell_faces_` we have to look for the faces for each cell. For every cell c we can therefore loop through `cell_faces_` from position `cell_facepos_[c]` to position `cell_facepos_[c+1]`, and check if any of the faces are boundary faces.

Listing 4.7: CUDA kernel body for finding the cells located at the grid boundary. The array `b_faces` is the result and is initialized with `number_of_faces_`.

```
 1 const int cell = myID();
 2 if ( cell < number_of_cells_) {
 3     bool boundary = false;
 4     int face;
 5     for (int f_i = cell_facepos_[cell]; f_i < cell_facepos_[cell + 1]; f_i++){
 6         face = cell_faces_[f_i];
 7         if ((face_cells_[ 2*face ] == -1) || (face_cells_[ 2*face +1] == -1)){
 8             boundary = true;
 9         }
10     }
11     if (boundary) {
12         b_cells[cell] = cell;
13     }
14 }
```

The body of the kernel for finding boundary cells is shown in Listing 4.7. We assume that each cell is not on the boundary in line 3 and change this boolean if this assumption turns out not to be true. In order to find `InternalCells()`, we implement the same algorithm but use `if (!boundary)` on line 11 instead.

4.3.2 Implementing `On` and `Extend`

In Section 3.1 we described the functionality of the `On` and `Extend` operators, and we will here take a look on how we implement them. The `EquelleRuntimeCUDA` class will consist of four functions in order to cover all uses of these functions. The member functions for the `Extend` operator match the two different uses of the operator as described before:

1. `Extend` a `Scalar` to make a `CollOfScalar` of uniform values matching a given grid set. This one is straight forward to implement by using a constructor of `CollOfScalar` matching this need, and we will therefore not give it any more attention.

2. `Extend` a `CollOfScalar` defined on one grid set to a superset by inserting values of zero representing the new elements.

The `On` operator is implemented by two member functions as well:

3. As *restrict to* operator, returning a subset of a variable given the relationship between the set the variable is originally based on and the subset of it representing the elements we request.

4. As *evaluate on* operator, where we create a variable based on a mapping defined by a `Collection` of `Cells` or `Faces`, which do not need to be a domain. The `Collection` can hence consist of non-unique elements.

Listing 4.8 shows the function definitions, listed in the same order as the descriptions above. Note that we are using template functions, since the sets can be either `CollOfFace` or `CollOfCell`. Template functions allow us to just make one implementation covering both codimensions instead of implementing the same functions twice. The drawback of using template functions is that we are not allowed to call CUDA kernels from them. This is because they have to be implemented in header files, which are included from both CUDA files and regular C++ files, and therefore are read by both the `nvcc` and `gcc` compilers. The solution is to send the arguments on to non-template functions, by using only the `device_vector<int>` member from the `CollOfIndices` type. We have written the code so that the `EquelleRuntimeCUDA` class sends the arguments to `DeviceGrid` member template functions, which then uses grid information to call non-template functions from the `wrapDeviceGrid` namespace which set up and call the CUDA kernels.

Listing 4.8: Function headers for the `EquelleRuntimeCUDA` member functions implementing the `On` and `Extend` operators.

```
    template<int codim>
    CollOfScalar operatorExtend(const Scalar& data,
                                const CollOfIndices<codim>& set);

    template<int codim>
    CollOfScalar operatorExtend(const CollOfScalar& data_in,
                                const CollOfIndices<codim>& from_set,
                                const CollOfIndices<codim>& to_set)
                                    const;

    template<int codim>
    CollOfScalar operatorOn(const CollOfScalar& data_in,
                            const CollOfIndices<codim>& from_set,
                            const CollOfIndices<codim>& to_set);

    template<int codim_data, int codim_set>
    CollOfIndices<codim_data> operatorOn( const
        CollOfIndices<codim_data>&
                                           in_data,
                                          const
                                              CollOfIndices<codim_set>&
                                              from_set,
                                          const
                                              CollOfIndices<codim_set>&
                                              to_set);
```

The complexity of implementing number 2, 3 and 4, highly depends on the input sets, which we will illustrate here by using the *restrict to* operation from case 3. We will also misuse the index operator for readability. First consider a function from the Equelle language,

```
sub_u = u On BoundaryFaces()
```

where we let `u` be a `Collection Of Scalar On AllFaces()`. It is translated into the C++ line

```
CollOfScalar sub_u = er.operatorOn( u, er.allFaces(), er.boundaryFaces() );
```

Since `AllFaces()` is a complete set, meaning that each face index matches its array index, it is straight forward to get a copy of only the elements in `u` that match the faces from `BoundaryFaces()`. If `f` is the face in position `i` in `BoundaryFaces()`, we just have to compute

```
sub_u[i] = u[f]
```

since `f` is the face in position `f` in `AllFaces()`.

If we then use the following line

```
my_u = sub_u On myBoundary
```

where `myBoundary` is a user specified subset of `BoundaryFaces()`, the mapping is not as straight forward. If `f` is the face in position `i` in `myBoundary`, we will not find face `f` in position `f` in `BoundaryFaces()`, as both sets are non-complete subsets. In order to assign the correct value to position `i` in `my_u`, we need to search in `BoundaryFaces()` to find `f`, and use the obtained index to read from `sub_u`. This search have to be done for each thread assigning to `sub_u`, which would lead to a huge amount of reads from global memory and huge differences in the threads' workloads. We therefore find another approach for this problem.

The solution we choose for doing a subset to subset `On` operation is by creating a temporary variable by extending the input to a complete set using the `Extend` operator, and then do a `On` operation from the complete temporary variable. Instead of having each thread search through the `from_set` variable, we choose to allocate a significant amount of temporary memory instead. This may not be an optimal solution, but the profiler will help us detect if this operation will be performance critical.

In summary, in order to compute `my_u` as defined above, the back-end will do the following operation:

```
temp = sub_u Extend AllFaces()
my_u = temp On myBoundary
```

This covers the use of `On` as described in point 3, but in order to fully understand all parts of it we need to understand the second use of the `Extend` operator as well.

The goal of the *extend by zeros* operation is to create a new `CollOfScalar` object of the same size as `to_set`, which consists of the elements of `data_in` where `to_set` is also found in `from_set`, and where the rest of the elements are zero. The variable names here are taken from the function header from Listing 4.8. The main challenge in implementing this in CUDA is to find out which elements in `to_set` are also found in `from_set`. The challenge becomes easier by assuming that the `to_set` is a complete set.

Listing 4.9: Naive implementation of the kernel for the `Extend` operator.

```
// Number of threads in total should be at least to_size
__global__ void extendKernel( double* out_data,
                              const double* in_data,
                              const int* from_set,
                              const int from_size,
                              const int to_size)
{
    int id = myID();
    if ( id < to_size ) {
        out_data[id] = 0;
        __syncthreads();
        if ( id < from_size ) {
            out_data[from_set[id]] = in_data[from_set[id]];
        }
    }
}
```

A naive way to do this is shown in Listing 4.9. The idea is that we initialize all elements in the output to zero, and then overwrite the elements specified by the `from_set` by the values from `in_data`. The problem with this code is that the CUDA function `__syncthreads()` is only able to synchronize threads within a block, and that we have no control on how far the execution of other blocks are. If `from_set[id]` matches an `id` held by a thread in another block, we cannot know which of the blocks will first write to that memory location. Therefore we could easily be in the situation that the correct value would be written to `out_data`, followed by another block "initializing" it to zero.

CUDA does not offer any functionality to synchronize between all blocks from a function running on the device. The only global synchronization method available is to split the functionality between two kernels, and that is what we have to do to implement the `Extend` operation correctly. The two correct kernels are shown in Listing 4.10.

Listing 4.10: Correct implementation of the two kernels needed for the `Extend` operator.

```
__global__ void extendKernel_step1( double* out_data,
                                    const int to_size)
{
    int id = myID();
    if ( id < to_size ) {
        out_data[id] = 0;
    }
}

__global__ extendKernel_step2( double* out_data,
                               const double* in_data,
                               const int* from_set,
                               const int from_size)
{
    int id = myID();
    if ( id < from_size ) {
        out_data[from_set[id]] = in_data[from_set[id]];
    }
}
```

The implementation of `Extend` above works as long as `to_set` is a complete set so that the indices of `out_data` correspond to the set indices. In case we have a situation where we want to map a small subset to a larger subset, we use the same method as for the `On` operator. Suppose we use the same variables as earlier, the Equelle line

```
sub_u = my_u Extend BoundaryFaces()
```

would be executed the same way as

```
temp = my_u Extend AllFaces()
sub_u = temp On BoundaryFaces()
```

The last use of the `On` operator, operating on `CollOfIndices` is implemented similarly to how we have implemented the operators for `CollOfScalar`. Because of the two-step procedure required to map a subset `On` a subset, we also need to implement a version for `Extend` for collections. Since the implementations depend on look-ups in arrays, the only difference needed to operate on grid entities are to operate on integers instead of doubles.

`On` and `Extend` of the derivatives

The description above mentions how we implement the `On` and `Extend` operators to work on the values stored in `CollOfScalar` objects, but not how they operate on possible derivatives stored in the objects.

Recall that if we consider a variable $\mathbf{U}$ defined on the set of all cells of the grid, named Ω, the data "belonging to" cell $i \in \Omega$ are the value U_i and its derivatives with respect to all primary variables $\mathbf{x}$, $\frac{\partial U_i}{\partial x_j}, \forall j \in \Omega$. The values in other cells differentiated with respect to x_i are not related to U_i. This means that if we are interested in $\mathbf{U}_{sub}$ defined as $\mathbf{U}$ on a subset of the grid $\Omega_{sub} \subset \Omega$, then we are also interested in the derivatives $\frac{\partial U_i}{\partial x_j}$, where $i \in \Omega_{sub},\ j \in \Omega$.

When we apply the operators `On` and `Extend` to variables with derivative matrices, we apply the rules from their values to the rows of the matrices containing the derivatives. Take the grid in Figure 4.1 as an example, where $\Omega = \{1, 2, ..., 8\}$ represent all cells, and the marked cells represents $\Omega_{sub} = \{2, 3, 6, 7\}$. Using $\{\cdot\}_i$ to denote the i'th row of a matrix, the derivative of $\mathbf{U}_{sub}$ will be

$$\mathbf{U}_{sub}.\mathtt{der} = \begin{bmatrix} \{\mathbf{U}.\mathtt{der}\}_2 \\ \{\mathbf{U}.\mathtt{der}\}_3 \\ \{\mathbf{U}.\mathtt{der}\}_6 \\ \{\mathbf{U}.\mathtt{der}\}_7 \end{bmatrix}.$$

The first idea for how to implement this would be to find the rows in the input that should be in the output and copy all elements from those rows, similarly to what we did to the values. The problem is that the compact storage scheme and the variable number of non-zero values per row in the sparse matrices makes this task hard to match the GPU's architecture. Before copying row i we need to know how many rows with index less that i are part of the new set as well, and the total number of non-zeros in these. The solution we have used is to apply *restriction matrices.*

A restriction matrix is a matrix which when multiplied from the left maps selected rows from the right-hand-side matrix to be part of the answer. Using the grid from Figure 4.1 as example, we get the derivatives for Ω_{sub} from Ω by

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \{\mathbf{U}.\mathtt{der}\}_1 \\ \{\mathbf{U}.\mathtt{der}\}_2 \\ \{\mathbf{U}.\mathtt{der}\}_3 \\ \{\mathbf{U}.\mathtt{der}\}_4 \\ \{\mathbf{U}.\mathtt{der}\}_5 \\ \{\mathbf{U}.\mathtt{der}\}_6 \\ \{\mathbf{U}.\mathtt{der}\}_7 \\ \{\mathbf{U}.\mathtt{der}\}_8 \end{bmatrix} = \begin{bmatrix} \{\mathbf{U}.\mathtt{der}\}_2 \\ \{\mathbf{U}.\mathtt{der}\}_3 \\ \{\mathbf{U}.\mathtt{der}\}_6 \\ \{\mathbf{U}.\mathtt{der}\}_7 \end{bmatrix}. \tag{4.22}$$

Using the overloaded multiplication operator from the `CudaMatrix` class, cuSPARSE helps us find both the new non-zero pattern and copy the correct data into the result.

Constructing a restriction matrix in CSR-format is convenient and easy. First note that we always will have one non-zero value for each row, and all these values will be 1. This makes it easy to initialize the `rowPtr` and `val` arrays. Secondly, the 1 on a given row is in the column with the same index of the row from the original matrix which we want to copy. Given a `On` function call from a complete set to a `CollOfIndices<codim> to_set`

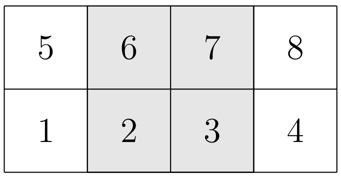


Figure 4.1: Grid to show `On` and `Extend` examples.

containing the indices of the subset, the column indices of the 1's will become the indices in `to_set`.

The `Extend` operation can be done using the same technique, only with a prolongation operator instead of restriction. The mapping from Ω_{sub} to Ω can be done using matrix-matrix multiplication as

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \{\mathbf{U}\texttt{.der}\}_2 \\ \{\mathbf{U}\texttt{.der}\}_3 \\ \{\mathbf{U}\texttt{.der}\}_6 \\ \{\mathbf{U}\texttt{.der}\}_7 \end{bmatrix} = \begin{bmatrix} -0- \\ \{\mathbf{U}\texttt{.der}\}_2 \\ \{\mathbf{U}\texttt{.der}\}_3 \\ -0- \\ -0- \\ \{\mathbf{U}\texttt{.der}\}_6 \\ \{\mathbf{U}\texttt{.der}\}_7 \\ -0- \end{bmatrix}. \tag{4.23}$$

Constructing this matrix is not as straight forward as for the restriction matrix, since we have some rows with no non-zero values making it harder to create the array `rowPtr`. However, since the prolongation matrix is the transpose of the restriction matrix for the reverse mapping, we can construct it as a restriction matrix using the `from_set`, and use the transpose flag before multiplying the matrix.

4.4 Other Equelle Functions

We will here give some details relating three more Equelle functions, which do not fit in any of the two previous sections. We start with the `Gradient` and `Divergence` operators, before we look at ternary if statements.

4.4.1 The `Gradient` and `Divergence` Operators

Most of the communication between cells happens through the Equelle functions `Gradient` and `Divergence`. As mentioned in Section 3.1 these functions calculate the discrete analogue to what we usually mean with gradients and divergence. They simply implement the directional sum of the surrounding values from the input relative to the output.

The `Gradient` function takes a set of values `On AllCells()` and produce a result `On InteriorFaces()`. Each face therefore uses the values from the two cells it separates. `Divergence` operate in the opposite direction, computing a value `On AllCells()` based on input `On AllFaces()` or `On InteriorFaces()`.

Both of the functions make use of the definitions of `FirstCell` and `SecondCell` based on the direction of the unit vector of each face. A positive `Gradient` value means that the value stored in SecondCell is larger than the value stored in FirstCell, and Divergence is implemented as the sum of values with outward unit normals minus the sum of the values with inward unit normals.

Implementing these functions to operate on some CollOfScalar are done by using the arrays stored in the DeviceGrid class. Recall that the array cell_faces_ contains the face indices surrounding each cell, based on the range given in the cell_facepos_ array. The opposite is found in face_cells_, where face_cells_[2*i] and face_cells_[2*i +1] gives the face index for the FirstCell and SecondCell respectively for a face i. We will therefore implement the Gradient and Divergence functions by using arrays in CUDA kernels and launching one thread for each output element.

The CUDA kernel used to compute the Gradient is shown in Listing 4.11. Since Gradient is only defined on InteriorFaces(), this set needs to be computed before calling the kernel, and is here represented by the int_faces input array of size size_out. The parameter grad is the result array, and cell_vals is the input values defined on AllFaces(). The kernel finds the face index of the thread's interior face in line 6, and uses the face_cell array to know where to look up in cell_vals in order to compute the result in line 8.

Listing 4.11: CUDA kernel for computing the Gradient Equelle function. Requires one thread for each element in InteriorFaces().

```
 1 __global__ void wrapEquelleRuntimeCUDA::gradientKernel( double* grad,
     const double* cell_vals, const int* int_faces, const int*
     face_cells, const int size_out)
 2 {
 3     const int i = myID();
 4     if ( i < size_out ) {
 5         // Compute face index:
 6         const int fi = int_faces[i];
 7         //grad[i] = secondface[fi] - firstface[fi]
 8         grad[i] = cell_vals[face_cells[fi*2 + 1]] -
               cell_vals[face_cells[fi*2]];
 9     }
10 }
```

The Gradient is relatively straight forward to compute, since each InteriorFaces() separates two and only two cells. The Divergence on the other hand is a bit more complex, as the number of faces surrounding a cell can vary from cell to cell for unstructured grids. Finding all faces surrounding a cell can be done by looping over the cell_faces_ range defined by cell_facepos_. We also need the direction of the unit normals, or in other words, find out whether the cell we are looking at is a FirstCell or SecondCell for each face. We therefore need the face_cells_ array in this kernel as well.

Also, note that we only implement a kernel for computing the Divergence from values defined on AllFaces(), even though it is also legal to give values on only InteriorFaces() as input. By only implementing a kernel for flux values on AllFaces() we can use the complete set property of the input, where the index of the flux array corresponds with the index of the face it belongs to. If the input fluxes had been a subset we would need some kind of search for each thread in order to find the index of a given face's value. For function calls to Divergence with input defined only on InteriorFaces() we use Extend to AllFaces() on the input before calling the kernel. This will cause some extra adding of zeroes by some of the threads, but will not affect the performance.

Listing 4.12: CUDA kernel for computing the `Divergence` Equelle function. Requires one thread for each cell in the grid.

```
 1 __global__ void divergenceKernel( double* div,
 2                                   const double* flux,
 3                                   const int* cell_facepos,
 4                                   const int* cell_faces,
 5                                   const int* face_cells,
 6                                   const int number_of_cells)
 7 {
 8     const int cell = myID();
 9     if ( cell < number_of_cells ) {
10         double div_temp = 0; // total divergence for this cell.
11         int factor, face;
12         // Iterate over this cells faces:
13         for ( int i = cell_facepos[cell]; i < cell_facepos[cell+1]; ++i
               ) {
14             factor = -1; // Assume normal inwards
15             face = cell_faces[i];
16             if ( face_cells[face*2] == cell ) { // if normal outwards
17                 factor = 1;
18             }
19             // Add contribution from this cell
20             div_temp += flux[face]*factor;
21         }
22         div[cell] = div_temp;
23     }
24 }
```

The implementation of the `Divergence` kernel is shown in Listing 4.12. The loop beginning on line 13 iterates over the faces surrounding a given cell, checks the direction of the normal vector relative to the cell and either adds or subtracts the correct flux value.

`Gradient` and `Divergence` with Automatic Differentiation

In order to see how derivatives behave in the context of `Gradient` and `Divergence` functions, consider a variable $\mathbf{U} = \mathbf{U}(\mathbf{x})$ as a function of a primary variable $\mathbf{x} = [x_1, x_2, ..., x_n]$. Consider the face f separating cells a and b, and the `Gradient` $\mathbf{G}$ on the `InteriorFaces()`, such that

$$G_f(\mathbf{x}) = U_b(\mathbf{x}) - U_a(\mathbf{x}). \tag{4.24}$$

The derivatives of $\mathbf{G}$ on face f is clearly

$$\frac{\partial G_f(\mathbf{x})}{\partial x_i} = \frac{\partial U_b(\mathbf{x})}{\partial x_i} - \frac{\partial \nabla U_a(\mathbf{x})}{\partial x_i}. \quad \text{for } i = 1, 2, ..., n \tag{4.25}$$

We can therefore consider the f'th row of the Jacobian of $\mathbf{G}$ as the difference between the b'th and a'th rows of the Jacobian of $\mathbf{U}$. Since these rows are likely to have different non-zero patterns, it is hard to write efficient kernels for this in a general setting. We therefore implement this operation as matrix-matrix products.

The serial back-end uses matrix-matrix products in order to implement derivatives of the `Gradient` and `Divergence` operators as well. The matrices representing the operators are stored in the `EquelleRuntimeCPU` class, and are constructed by the `Opm::HelperOps` class from the autodiff-module in OPM. We follow a similar approach and implement a

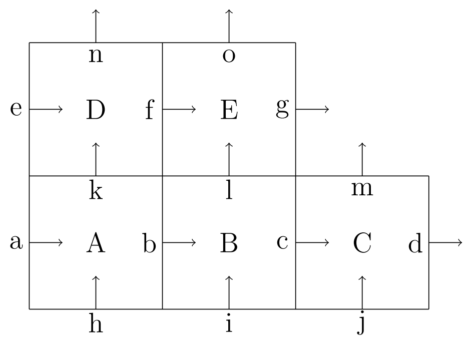


Figure 4.2: Example 2D grid where capitalized letters are cell identifiers and lower case letters identify faces. The arrows show the orientation of each faces.

class `DeviceHelperOps` and make it member of the the `EquelleRuntimeCUDA` class. The `DeviceHelperOps` then constructs the matrices similarly to the `Opm::HelperOps` class, but stores them on the GPU as `CudaMatrix` objects instead. The construction is serial by nature, so there is no use in constructing the matrices using the GPU. The process is a one-time cost, and we are therefore not concerned about letting the CPU compute these matrices.

In order to see what the matrices will look like, we once again consider the grid used in Section 3.1 which is also shown in Figure 4.2 for convenience. The `InteriorFaces()` will be $\{b, c, f, k, l\}$, and the `Gradient` operation on each of them corresponds to

$$\begin{aligned} grad(b) &= B - A \\ grad(c) &= C - B \\ grad(f) &= E - D \\ grad(k) &= D - A \\ grad(l) &= E - B \end{aligned} \tag{4.26}$$

This can be written in matrix form as

$$\texttt{Gradient}\left(\begin{bmatrix} b \\ c \\ f \\ k \\ l \end{bmatrix}\right) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \\ D \\ E \end{bmatrix} \tag{4.27}$$

where we see the form of the matrix representing the `Gradient` operator.

The `Divergence` operator can be represented by a matrix in a similar fashion. Both matrices are descriptions of the neighbour relations in the grid, and they can be used for computing the values as well as derivatives, by a matrix vector product. The profiler however shows that the cuSPARSE sparse matrix times vector function becomes a hotspot in simulators using explicit solutions. We therefore use the kernels described earlier in this Section for finding the values based on these operators, and the cuSPARSE function for multiplying two sparse matrices for their derivatives.

4.4.2 Ternary If Operator

The last operator from Equelle we will look at is the ternary if operator. Its syntax in Equelle is similar to C/C++ ternary if syntax,

```
result = predicate ? iftrue : iffalse
```

where `predicate` is a boolean expression resulting in a `Collection Of Bool`, and `iftrue` and `iffalse` are collections of the same type of `result`. All collections are required to be of the same size, as the operation is interpreted as an element-wise ternary if operation. In the intermediate code, we get the three input variables as arguments in a function call.

As the operator is embarrassingly parallel, it is implemented by setting up a CUDA kernel where each of the elements in the output is found by evaluating the corresponding boolean in the `predicate` array and assigning from `iftrue` or `iffalse` accordingly. If we have `CollOfScalars` with derivatives however, we need a different approach. This is because we do not have any guarantee that the derivatives of `iftrue` and `iffalse` has the same non-zero pattern.

Similarly to the `On` and `Extend` operators, the ternary if operator will be a row-wise operator for the derivatives, and it will be implemented in terms of matrix arithmetic. Let A and B be the derivative matrices for `iftrue` and `iffalse` respectively, and let P be a diagonal matrix such that

$$P_{i,i} = \begin{cases} 1.0 & \text{if} \quad \texttt{predicate}[\texttt{i}] \text{ is } \texttt{true} \\ 0.0 & \text{if} \quad \texttt{predicate}[\texttt{i}] \text{ is } \texttt{false}. \end{cases} \tag{4.28}$$

We can then express the derivative of the result, C, of a ternary if operation as

$$C = PA + (I - P)B. \tag{4.29}$$

In order to build the matrix P we use a `CudaMatrix` constructor which implements Equation (4.28) from a `CollOfBool` variable.

4.5 Linear Solvers

We have up until this point looked into the implementation details of most of the functionality in Equelle, and the treatment of derivative values for helping us use implicit methods in simulators. As we saw in Section 4.2.1, we implement the implicit simulators by defining a residual function, and then call the `NewtonSolve` function in Equelle with the residual as input. We then get the evaluation of the derivatives of the residual by the automatic differentiation rules we have implemented, and all that is left is to solve the linear system,

$$Ax = b. \tag{4.30}$$

The task of implementing the linear solver is complex. In order for the solver to match the CUDA back-end we need to use solvers which can utilize the massively parallel hardware. Since Equelle lets us write simulators which result in different types of matrices, we also need the method to be general enough to not rely on specific matrix properties.

Solvers of linear systems can be divided in the following two groups; direct methods and iterative methods. Direct methods are algorithms that are able to solve the linear system in a finite amount of steps. Most direct methods consist of computations with a sequential dependency pattern, and are therefore not suited for implementation on GPUs.

Iterative methods rely on approximations of the solution such that the approximation becomes increasingly better. By checking the norm of the residual from the approximate solution, we are able to detect when the obtained solution is *close enough* to the exact solution, and use that as the final solution. This is often a required method when A is large and sparse, as direct method becomes too time and memory consuming. By using matrix properties such as symmetry and positive definiteness, the methods can be expressed by less computations and with a better convergence rate. It is often the case that faster methods require stricter matrix properties in order to converge at all.

In order to improve the convergence rate for iterative methods it is possible to use a preconditioner M acting like an approximation of the inverse of matrix A. The idea is that

$$M^{-1}Ax = M^{-1}b \tag{4.31}$$

is easier to solve with a given iterative method than Equation (4.30). There are several techniques for defining preconditioners, often based on the nature of the problem, non-zero fill pattern, or size of diagonal elements. It is therefore often the case that a good preconditioner for one matrix is not a good preconditioner for another, which makes it hard to provide a single preconditioner that is good for any problem.

The Cusp Library

Since it takes a huge effort to implement linear solvers efficiently in CUDA, we make use of the Cusp library[6]. It offers functions for both solvers and preconditioners implemented in CUDA. The solvers are a selection of different Krylov subspace methods, and since linear solvers are not the main focus of this thesis, we will only give a minimal explanation of them. For a more thorough introduction and discussion on iterative solvers and preconditioners, see [30]. The provided linear solvers provided by Cusp are the following:

CG The Conjugate Gradient method is one of the best known iterative methods. In order to obtain convergence the matrix A is required to be symmetric and positive definite (SPD). Symmetric matrices satisfy $A^T = A$, and a matrix $A \in \mathbb{R}^{n\times n}$ is positive definite if $\mathbf{x}^T A\mathbf{x} > 0$ for any non-zero vector $\mathbf{x} \in \mathbb{R}^n$. The method was at first suggested as a direct method, as it gives the exact solution in n iterations given that the above requirements are met. It is however often the case that the solution is good enough within few iterations.

GMRes Generalized Minimal Residual works by minimizing the residual vector with respect to the given Krylov subspace. The only requirement on A is that the matrix is square and non-singular, meaning that A^{-1} exists.

BiCGStab The BiConjugate Gradient Stabilized method[33] is an increasingly popular method for solving non-symmetric linear problems. It is based on other non-symmetric versions of CG such as BiCG and CGS (CG Squared), by resolving the issues CGS has with sensitivity to rounding errors. The method requires that A is non-singular.

The above solvers can be used directly on Equation (4.30), or on Equation (4.31), and we can also apply one of the following preconditioners provided by Cusp:

Table 4.1: An overview of which solvers and preconditioners from the Cusp library that are implemented in the CUDA back-end for Equelle.

	None	**Diagonal**	**Ainv**	**Smooth agr.**
CG	yes	yes		
GMRes	yes			
BiCGStab	yes	yes		

Diagonal The diagonal preconditioner extract the diagonal elements D of A, and use $M^{-1} = D^{-1}$, providing a scaled version of the linear system. The application of the preconditioner is inexpensive, but also of limited effect. The diagonal preconditioner has biggest impact to the required number of iterations on problems where the original system consists of rows of various scales.

Approximations of A^{-1} Cusp provides three different inverse approximation preconditioners, with a various number of parameters for accuracy and memory consumption. Two of the preconditioners are intended for SPD matrices only, while the third one is for non-symmetric cases.

Smoothed aggregation Algebraic multigrid (AMG) preconditioner based on smoothed aggregation.

The class `LinearSolver` has been created in order for the Equelle CUDA back-end to provide a simple interface towards solving linear systems. We have not implemented combinations of all the methods and preconditioners offered by Cusp. The most important combinations are the solvers without preconditioners, in order to show a proof of concept on the linear solver part of the back-end. Table 4.1 shows the combinations that are implemented at the current stage, however adding a new combination of solver and preconditioner to the back-end is relatively easy.

In order to choose what method we should use to solve the linear systems arising in Equelle, we initialize the `LinearSolver` class with user provided input parameters at runtime. Adding the lines

```
solver=GMRes
preconditioner=none
```

to the parameter file makes the program use the GMRes method without preconditioners. We do not require the user to give such parameters on input, and the default option is to use the BiCGStab method with a diagonal preconditioner. The reason we have chosen BiCGStab as default is because it works for both symmetric and non-symmetric matrices, and has faster convergence than GMRes.

We have also added the option to use the CPU for solving the linear system. In some cases we might not be able to find the solution using the methods provided by Cusp, and in other situations we might get very slow convergence with any of the provided methods regardless of which of these preconditioners we use. In those cases we might want to use another preconditioner which might have a sequential build phase, or that we want to use a direct solver instead. We therefore offer the option to use the wider range of solvers offered by the CPU, with the same interface as the serial back-end, as this makes the CUDA back-end more flexible.

We will consider a case in Section 5.2 where we are unable to get acceptable convergence with the Cusp methods, and therefore use the CPU solver to validate the rest of the back-end.

Chapter 5

Numerical Experiments and Performance Testing

We have now in the previous chapter seen how we have implemented the CUDA back-end for the Equelle compiler. This chapter will describe some test cases where we validate the back-end, showing how the simulators behave similarly on our new back-end compared to the serial back-end. We will show simulator results for mainly the heat equation and the shallow water equations, as described in Appendix B.1 and B.2 respectively.

We will in Section 5.1 look at explicit and implicit solutions of the heat equations on structured grids, before we validate the correctness on an unstructured grid in Section 5.2. In Section 5.3 we will show how the shallow water simulator written for the Specialization Project[20] can make use of the CUDA back-end without a single change in the Equelle source code. We will also show some performance measurements in Section 5.4.

We have used the following two GPUs for testing the back-end:

- Nvidia Tesla K40
- Nvidia NVS 5200M

The NVS is a laptop GPU with 1 GB of global memory. It is based on the Fermi architecture, and have a memory bandwidth of 14.4 GB/s. The core clock rate is 625 MHz, and it has 96 CUDA cores.

The Tesla on the other hand is based on the Kepler architecture, and is a pure compute GPU without the possibility to render graphics. It has 12 GB of global memory, and a memory bandwidth of 288 GB/s. The clock frequency on the Tesla is 706 MHz, which is not a lot more than the NVS, but since the Tesla is designed for double precision floating points, while the NVS is designed for single precision with capability to use double precision, this number is not directly comparable. The Tesla also has far more CUDA cores than the NVS, with its 2880 [3].

5.1 Heat Equation on Structured Grid

The heat equation is stated as

$$\frac{\partial u}{\partial t} - k\nabla^2 u = 0 \tag{5.1}$$

and is used both as an example simulator on the Equlle website[15] and as a test case during the development of this back-end. The Equelle program is therefore not written as part of this Thesis. Appendix B.1 gives an overview of how the finite volume method is used and implemented, and also shows the Equelle source code for the method for both the implicit and explicit case. We will here look at the simulators on a 3-dimensional Cartesian grid.

We start by considering the explicit simulator. We choose a grid of size $19 \times 24 \times 20$ unit cube cells, and use a time step size of $\Delta t = 0.5$. The Dirichlet boundary is chosen to be at $x = 0$ and $x = 19$ as

$$\begin{aligned} u(0, y, z, t) &= 50z, & &\text{for } 0 \leq y \leq 24,\ 0 \leq z \leq 20,\ t \geq 0, \\ u(19, y, z, t) &= 1000 - 50z & &\text{for } 0 \leq y \leq 24,\ 0 \leq z \leq 20,\ t \geq 0. \end{aligned} \tag{5.2}$$

The initial conditions are chosen as

$$u(x, y, z, 0) = 500, \quad \text{for } 0 < x < 19,\ 0 \leq y \leq 24,\ 0 \leq z \leq 20. \tag{5.3}$$

We compile the explicit heat equation simulator using the new CUDA back-end, and provide the above grid information, initial and boundary conditions as program input. The simulation is run for 150 time steps, and the results can be seen in Figure 5.1.

By compiling the simulator using the serial back-end as well, we can use the same input data in order to run the simulation on the CPU for comparison. We then see that the output from the last time step from the CUDA back-end and the serial back-end are exactly the same for all cells.

We now turn to the implicit method, and compile it using the CUDA back-end as well. We use the same parameter file as for the explicit case, meaning that we use the default BiCGStab solver with a diagonal preconditioner. The only difference is that we now use larger time steps, $\Delta t = 4$. The results are shown in Figure 5.2, and are similar to the results for the implicit case. The maximum difference for the 20th iteration when comparing to the same simulation using the serial back-end is $8.3 \cdot 10^{-8}$. This is expected as we use 10^{-8} as default tolerance for the linear solver.

5.2 Heat Equation on Unstructured Grid

Since we have designed the CUDA back-end to use unstructured grids, we will here apply the implicit heat equation simulator on the domain of the Norne oil field. A grid model of the reservoir consisting of almost 45 000 cells has previously been used to show correctness for unstructured grids on the serial back-end, and we use it here for the same reason.

The grid with the result is shown in Figure 5.3. We have applied Dirichlet conditions by extracting the faces which have approximately lowest x values and set the temperature on them to constant 0. Similarly, boundary faces with highest x coordinates were given Dirichlet value of 1000. We use the same initial conditions as before, with $u(x, y, z, 0) = 500$.

The linear system arising from the `NewtonSolve` function for this case turns out to be very hard to solve. Using the default parameters for maximum number of iterations and any of the three solvers from Table 4.1, the method diverges. This can be caused by two reasons:

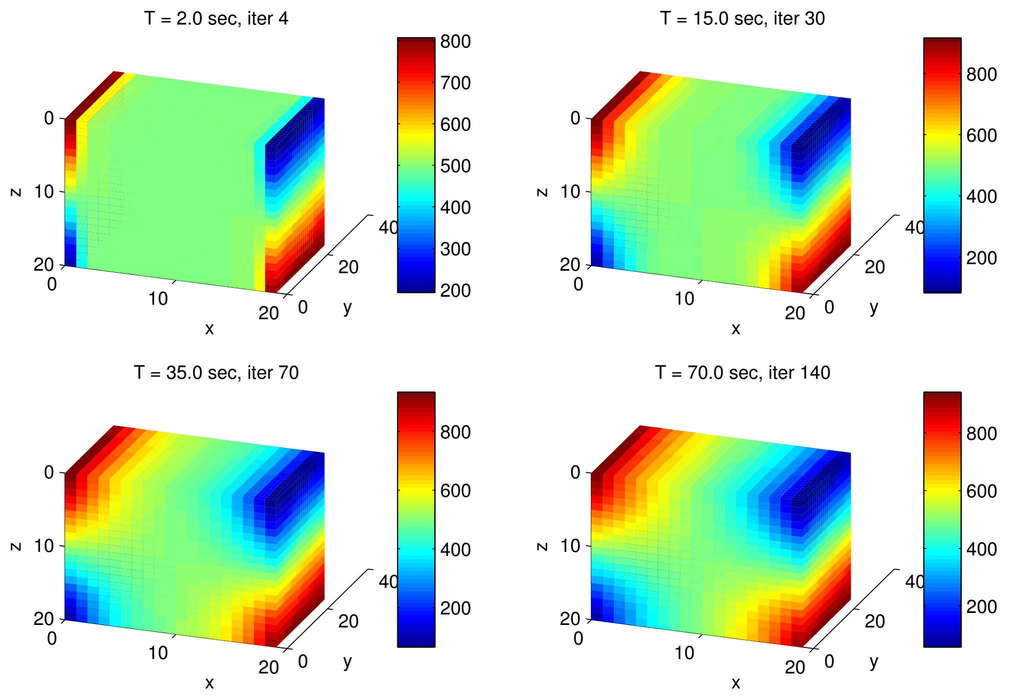


Figure 5.1: Explicit solution of the heat equation, using initial data from Equation (5.3) and boundary conditions from Equation (5.2).

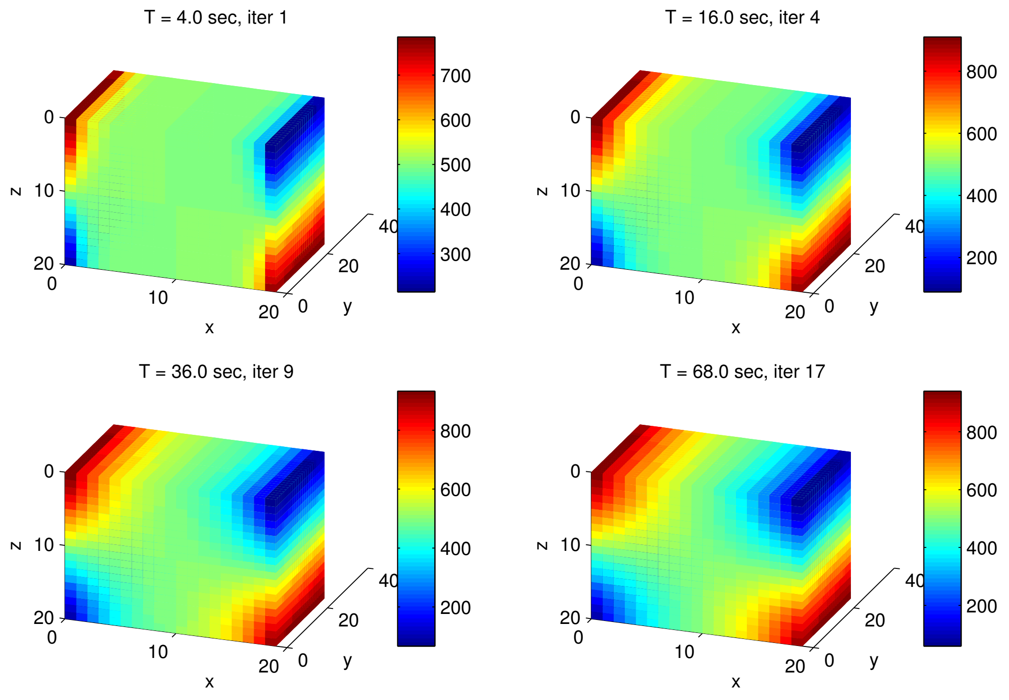


Figure 5.2: Implicit solution of the heat equation, using initial data from Equation (5.3) and boundary conditions from Equation (5.2).

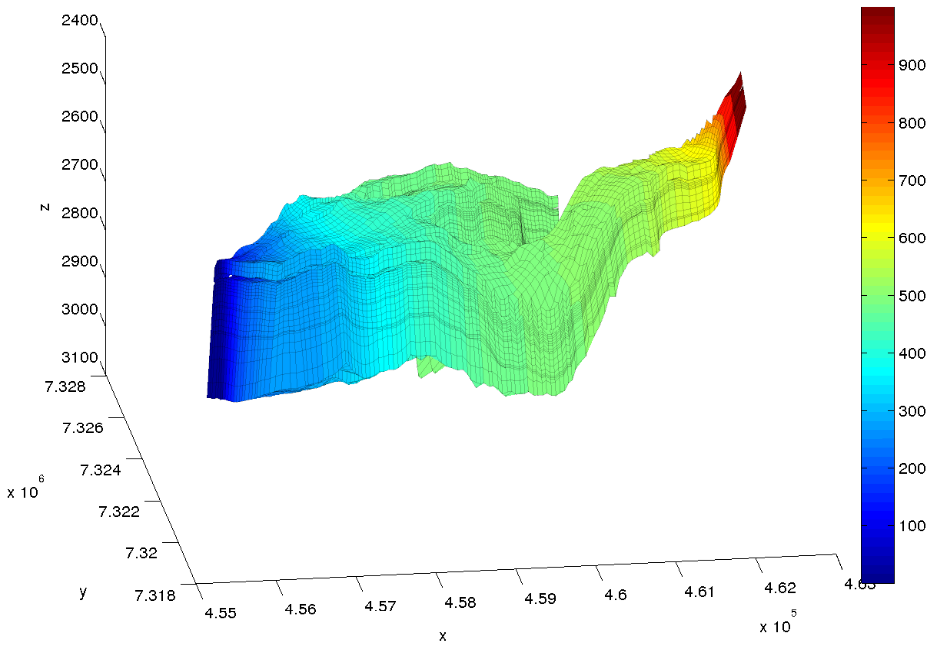


Figure 5.3: Heat equation solved in the Norne oil field. Dirichlet boundary conditions are applied to the rightmost and leftmost faces as shown in this figure.

1. The linear system is of such a nature that in order to solve it we need better methods which are not available in the CUDA back-end.

2. The linear system has been created with errors, resulting in an impossible-to-solve system.

In order to check which reason is causing our problem, we let the CPU solve the linear system, while the CUDA back-end constructs the matrix using automatic differentiation. The linear system is then solved by the following steps:

1. Copy A and b from device to host.

2. Solve $Ax = b$ by the methods available for the CPU.

3. Copy x from host to device.

The solutions we get from the CPU-solver leads to the results we see in Figure 5.3, and comparing the solution with the serial back-end gives a maximum error less that 10^{-8}. This suggest that the first problem suggested above was the reason we did not get correct results. If the construction of the matrix had been wrong, we should not get correct results from solving the linear system with the CPU. This validates correctness of the CUDA back-end applied to unstructured grids.

By doing some more testing we also achieved convergence by using the GMRes method provided by Cusp. This was done by drastically increasing the maximum iteration count for the linear system by using

```
solver_max_iter=100000
```

in the parameter file. The GMRes method was also implemented with a restart for every 100'th iteration. The simulator then computed the correct answers in 3 hours and 20 minutes. In comparison, by letting the CPU solve the linear system we only require about 20 seconds on the entire simulation. This shows us that the GPU is not always the best option for implementing fast code.

5.3 Shallow Water Equation

In the Specialization Project[20] done as a preparation to this master thesis, we developed an Equelle simulator for the shallow water equations for varying bottom topography and dry states. The shallow water equations are given as

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} \tag{5.4}$$

where h is the water depth, u and v are velocities in x and y directions respectively, g is gravity, and $B = B(x, y)$ is the description of the bottom topography. The terms hu and hv represent mass fluxes in their respective directions. An overview of the numerical method used to write the simulator is shown in Appendix B.2.

The equations describe how waves behave under a free surface when the motion is due to gravity only. It is derived from the Navier-Stokes equations by assuming that the vertical velocity is negligible compared to the horizontal velocities, and it was first described as the Saint Venant system[12].

By using the same data sets as in [20], we want to reproduce the results found there with as little effort as possible. We use the same scripts for generating the bottom topography

$$B(x, y) = B_1(x, y) + B_2(x, y), \tag{5.5}$$

where $B_1(x, y)$ is the basic form defined as

$$B_1(x, y) = \frac{x}{150} + \frac{y}{100} + 0.3 \sin\left(\frac{2\pi x}{85}\right) + 0.2 \sin\left(\frac{2\pi y}{85}\right), \quad , 0 \le x, y \le 100, \tag{5.6}$$

and $B_2(x, y)$ is a radial hump given by

$$B_2(x, y) = \begin{cases} 0.3 \sin\left(\frac{r-32.5}{10}\pi\right), & \text{for } 32.5 \le r \le 42.5 \\ 0 & \text{otherwise} \end{cases} . \tag{5.7}$$

where $r = \sqrt{(x-50)^2 + (y-50)^2}$. The bottom is most easily seen in Figure 5.5.

In order to reproduce the results from [20], we compile the Equelle program for the shallow water equations using the flag for the CUDA back-end, compile the C++ intermediate code and link it to the back-end library, and run the program using the bottom topography from Equation (5.5) and the initial conditions we want to use. The results for a radial dam break in a filled basin are seen in Figure 5.4, and the results using partially filled basin with dry states are shown in Figure 5.5. By comparing the numerical results to the results from the serial back-end, we see that the numbers only differs on a machine epsilon level. Since the numerical results are as good as identical, we have used the figures from [20] here as well.

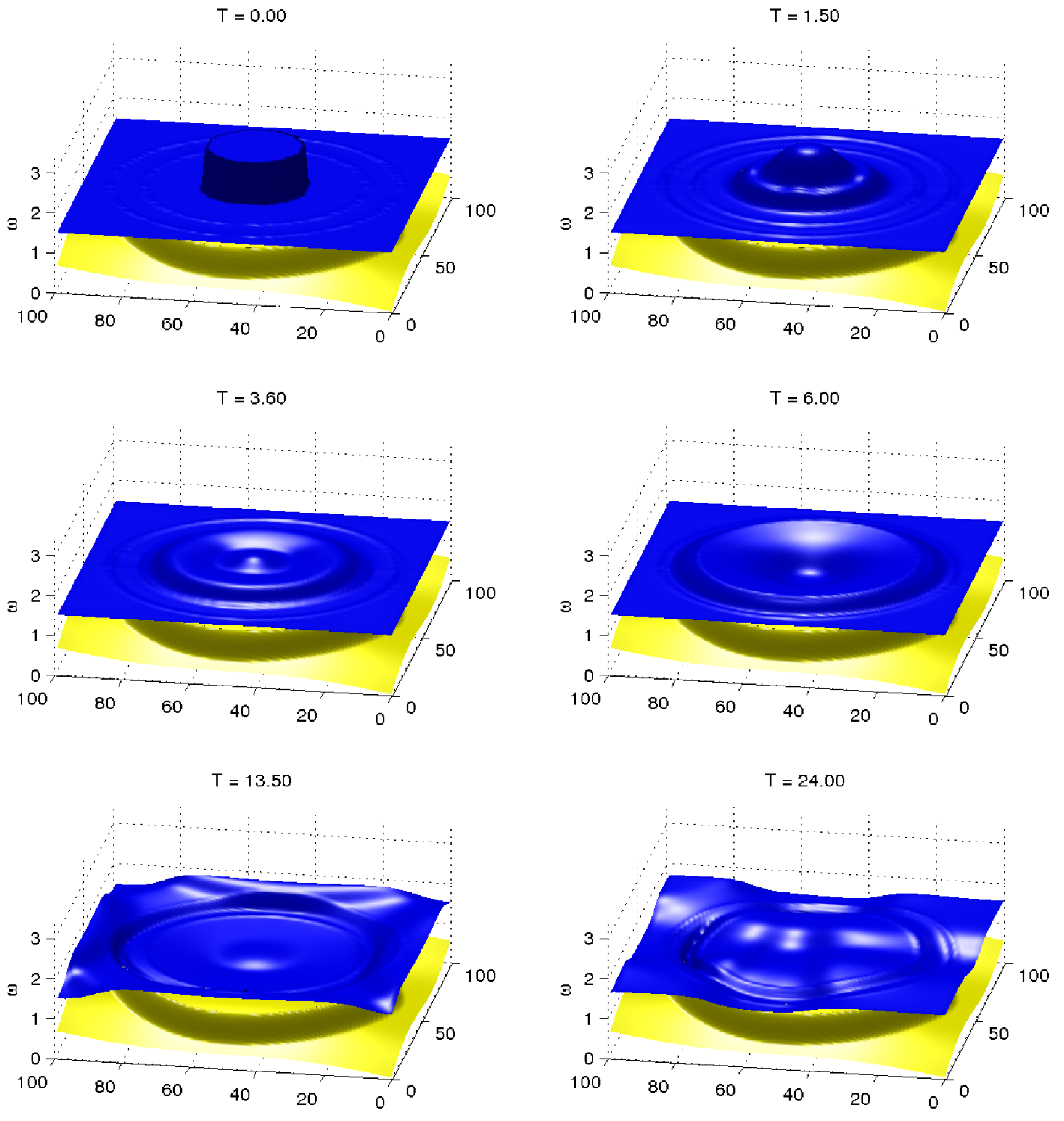


Figure 5.4: Simulation of a radial dam break on the topography described in Equation (5.5).

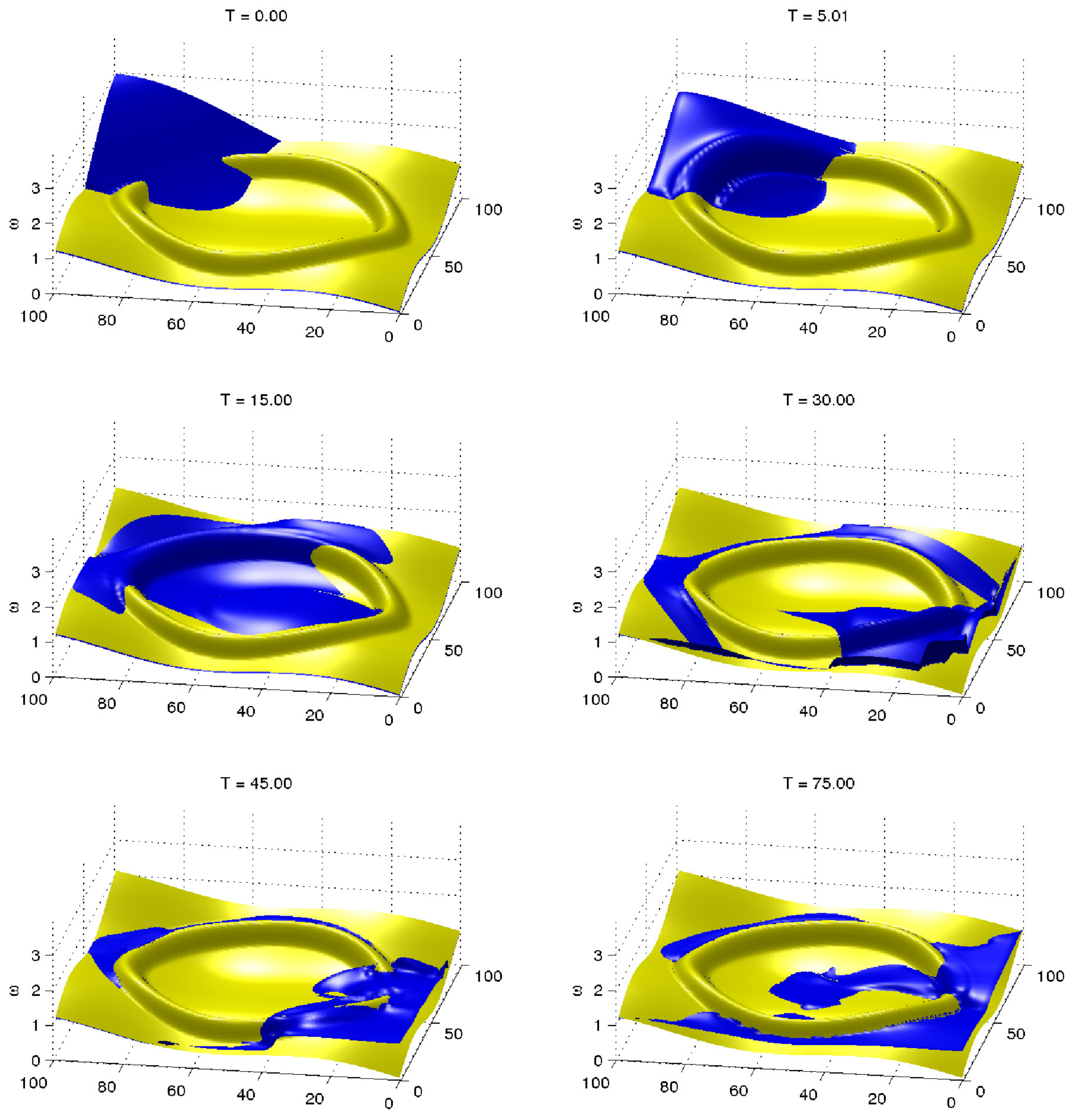


Figure 5.5: Simulating a body of water flowing from one corner on the topography described in Equation (5.5).

Table 5.1: The grid sizes used in order to test performance of the explicit heat equation.

Num cells (10^6)	0.125	0.25	0.5	1	1.5	2	2.5	3	3.5	3.75
nx	50	50	100	100	150	200	250	300	350	375
ny	50	100	100	100	100	100	100	100	100	100
nz	50	50	50	100	100	100	100	100	100	100

5.4 Performance Testing

We have now seen that our new CUDA back-end is able to produce correct results for both simple explicit simulators and the more complex evaluation of implicit simulators by using AD on the GPU. The back-end also handles unstructured grids as we saw in Section 5.2. In this Section we will therefore look at the performance of the CUDA back-end.

It is important to note that any comparison with the serial back-end will give a somewhat misleading speed-up. The serial back-end is not tuned for performance, and comparing high performance parallel code with unoptimized serial code gives us limited information. That said, we will still compare the performance of the CUDA back-end to the serial back-end to some extent, because they are built on the same principles and ideas. The generated part of the intermediate code, and hence the function calls to the Equelle runtime class, are exactly the same. This means that performance limiting factors such as limited reuse of variables and lots of temporary variable use in complex expressions are found in both back-ends. Comparing the two back-ends might therefore not be as misleading as first indicated.

Explicit Heat Equation

Our first performance test case is based on the explicit heat equation on Cartesian grids. The use of Cartesian grids makes it easy to generate test cases of different grid sizes while still being able to generate face indices on which we put Dirichlet conditions. All tests will therefore be scaled versions of the simulation shown in Figure 5.1.

The tests were conducted by editing the intermediate code by adding timing inside the for-loop. Since most simulators applied to real-world problems are likely to have a huge amount of time steps, we are here not interested in the initialization phase of the program. We therefore use 120 time steps in each simulation, where the reported timings are started at iteration 10 and ended at iteration 110. This gives us timing results per 100'th iteration and will represent the performance in the long run. The grid configurations for each problem size is given in Table 5.1.

We compare different kinds of hardware by using the CUDA back end on two different GPUs, the Tesla K40 and the NVS 5200M, and the serial back-end on an Intel Xeon E5-2620 CPU, running at 2.10GHz. The results are gathered in Figure 5.6. Plot A) shows a comparison of the run times on the three different processors, while B), C) and D) focuses on one processor at a time. Since the computational complexity of explicit solutions are linear with respect to the number of cells, we have also added graphs of execution time per 1 million cells in those plots.

Plot A) in Figure 5.6 shows how the GPUs instantly outperform the CPU with the

Table 5.2: Comparison between the CUDA back-end and the serial back-end for the extreme cases in terms of best and worst speed-up.

	NVS		Tesla	
Num cells (10^6)	0.125	1.5	0.125	3.75
Serial back-end (s)	49	78.14	49	85
CUDA back-end (s)	13.6	11.11	7.7	1.02
Speed up	3.6	7.0	6.3	**83**

current back-ends. We also see how large the difference becomes between the two GPUs. Since the Tesla has a lot more computational cores than the NVS, it is able to process a lot more data simultaneously. The Tesla is also designed for double precision arithmetic while the NVS primarily is designed for single precision. The dramatically higher performance of the Tesla is therefore expected. Note also how we do not include data for the NVS for cases of more than 1.5 million cells, as the device runs out of memory.

As mentioned, the computational complexity of the explicit heat equation is linear, which means that the relative time for computing 1 million elements should be constant for any problem size. As we see in Plot B) the serial back-end confirms this relationship quite well.

The GPUs on the other hand seem to perform better per element for larger cases than for small cases. This might have different reasons, where kernel overhead is one candidate. Since each kernel launch requires a small overhead regardless of the number of blocks and threads, this overhead becomes less significant if each kernel have more data to process. Hence computational time per element goes down. Another reason might be that if the problem size is not big enough, the GPU might have more computational cores than the program is able to use at once, meaning that some resources are idle at all time for smaller problems. When the test cases become larger we also see that processing time per element becomes more and more constant.

By comparing the CUDA back-end to the serial back-end, we see that the obtained speed-up depends on the problem sizes. For small problems, which in this case is 125 000 cells, the speed up will not be as impressive as for the largest cases. Table 5.2 summarize the performance gains in using the CUDA back-end instead of the serial back-end. On the laptop GPU, the NVS, we experience a performance gain with a factor of between 3.6 and 7, but by using the Tesla we measure up to 83 times speed up!

As mentioned earlier, we have to stress that the serial back-end is not optimized and that these numbers therefore are not directly comparable. But we would also like to point out that the optimizations done in the CUDA back-end is without changing anything in the compiler front-end, and that both back-ends therefore are affected by the performance limitations brought by this design.

Implicit Heat Equation

In order to measure performance of an implicit method we use the implicit simulator for the heat equation to collect runtimes for different grid sizes. Since there is a lot more work involved in finding the next time step of an implicit method than for an explicit method, we here measure execution time for only 10 iterations, where we let the program

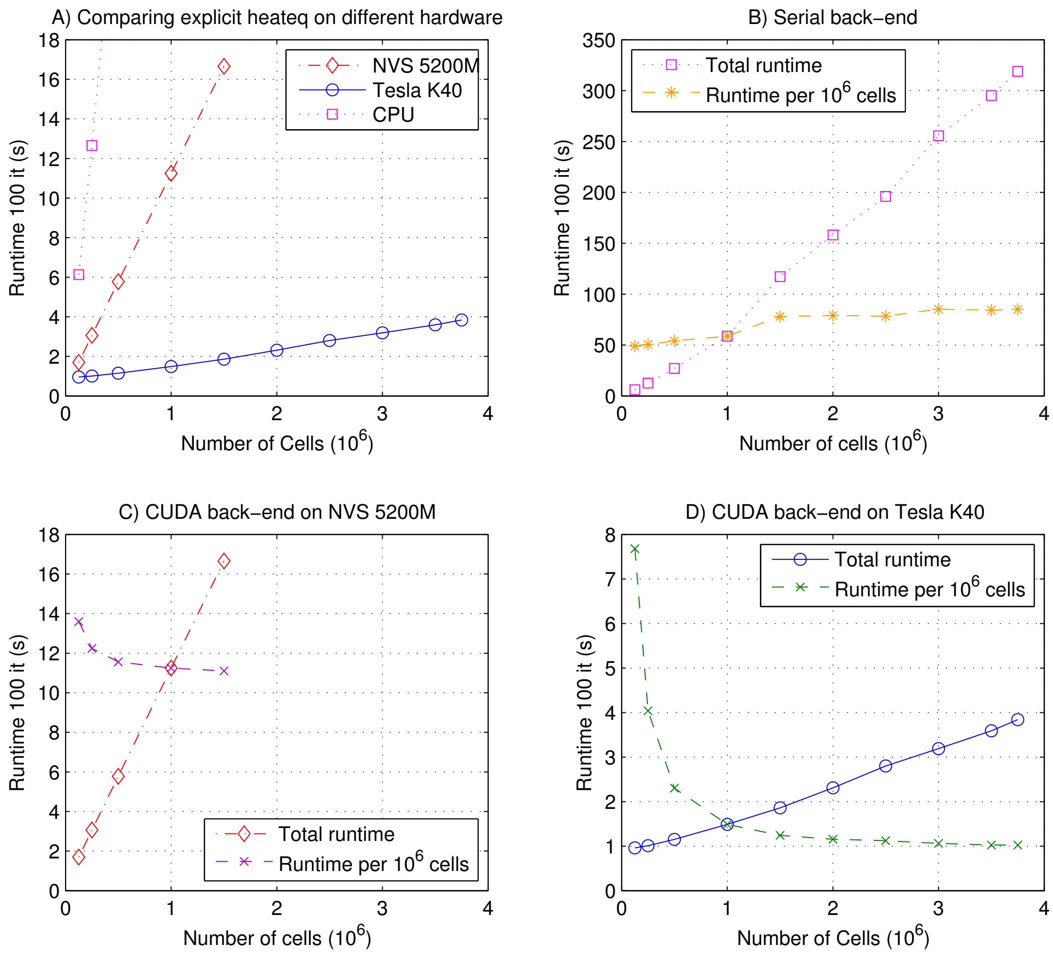


Figure 5.6: Execution times for 100 iterations of the explicit heat equation for various amount of cells on different hardware. We show absolute runtimes in Plot A) where we focus on the two GPUs, and plot B), C) and D) shows relative performance per 1 million cells on each hardware.

do some extra iterations both before we start and after we end the timing. The grid sizes corresponds to the ones given in Table 5.1.

One problem we get when we want to compare implicit methods on different hardware is that the linear solvers available to each hardware are not the same. The computational phase when we use the `NewtonSolve` function can be split in two:

1. Assembling the matrix using AD.
2. Solving the linear system.

If a simulator is slow, we will therefore not immediately know which of the two phases is the slow part.

In order to make the comparison easier for us we run the experiments by three different methods:

(a) CUDA back-end on the Tesla K40 using the default solver (BiCGStab and diagonal preconditioner) from Cusp.

(b) CUDA back-end on the Tesla K40 using the CPU linear solver with input `istl`.

(c) Serial back-end using the CPU linear solver with input `istl`.

The `istl` input to the CPU solver makes use of a conjugate gradient method with an algebraic multi-grid preconditioner. In order to compare the two back-ends we use that the difference between method (b) and (a) represent the overhead from using the CPU solver, and that

$$(d) := (c) - ((b) - (a)) \tag{5.8}$$

therefore gives an artificial measure of using the serial back-end with the *solver load from Cusp*. We will therefore use the artificial method (d) as a comparison with method (a).

The results of the comparison are shown in Figure 5.7. Plot A) shows the total runtimes for the methods (a), (b) and (c), with a focus on (a), and in Plot B) we have changed the focus to cover all timings from (c) as well. We have also added the artificial timings for method (d), where we have subtracted from (c) the difference between (b) and (a). Since (d) is not that much faster than (c), and since case (a) using the Cusp solver is still very fast, it seems like the serial back-end is dominated by the assembly process rather than the linear solver.

In Plots C) and D) we look at the relative processing speed for 1 million cells. Both solvers show a constant relative processing time, where the Tesla reaches peak performance at 1 million cells. We see that both Plot C) and D) indicate that the BiCGStab method from Cusp is of linear computational complexity. The complexity in the matrix assembly phase depends on the number of cells, and the number of of non-zeros in each row, as $\mathcal{O}(kn)$. Since all grid sizes are based on 3-dimensional Cartesian grids, k becomes constant and we get the assembly phase to be $\mathcal{O}(n)$ as well.

Based on large cases of more than 1 million cells, we see that the CUDA back-end requires 21 seconds per 10 iterations of 1 million cells, while the required time for method (d) using the serial back-end is 700 seconds. This corresponds to a speed-up of 33.6.

These two back-ends are implemented with approximately the same amount of optimization when it comes to assembling the matrix for implicit methods, and both of them have huge potentials for faster execution. We will look at the current bottlenecks for the implicit heat equation in Section 6.2.2.

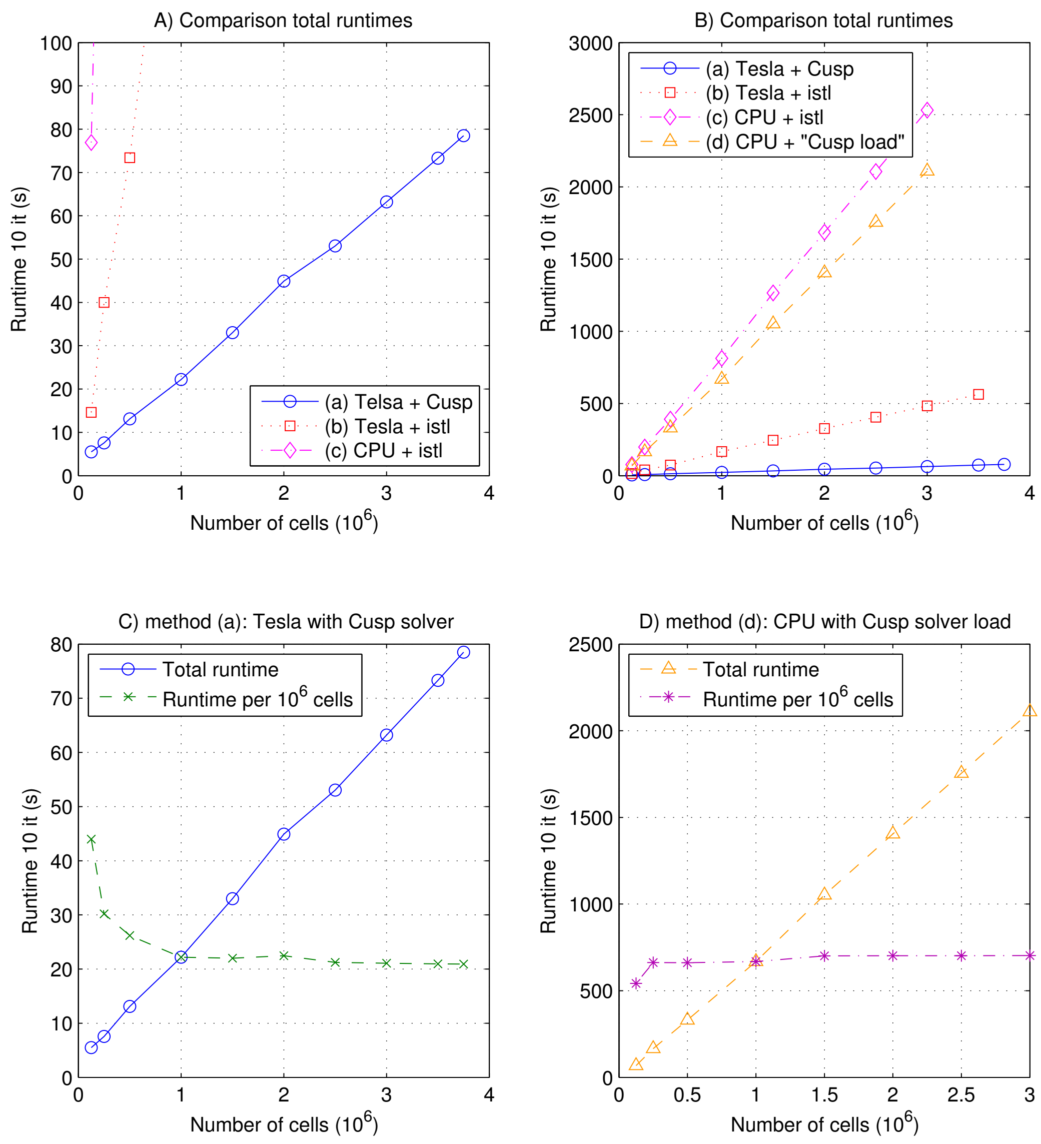


Figure 5.7: Performance testing for 10 iterations of the implicit solution of the heat equation running on different Equelle back-ends. We have compared the serial back-end and the CUDA back-end by creating artificial measurements of the serial back-end using the same time to obtain the linear solution as the CUDA back-end. Plot A) and B) show absolute runtimes, and Plot C) and D) show relative processing speed per 1 million cells.

The Shallow Water Equations

We conduct the same experiments for the shallow water simulator as we did for the explicit heat equation. We use different grid sizes to measure performance of 100 time steps, using some extra iterations before and after the timing similar to before. We use 2D grids as shown in Table 5.3, and make measurements on all three hardware platforms, the Tesla K40 and NVS 5200M GPUs and the CPU.

The results are analysed and shown in Figure 5.8. The first observation we make is that the total runtimes for the Tesla does not follow the same trends as the others for small cases. We therefore add more grid sizes in order to have more data to analyse. In Plot A) we see how the Tesla outperforms both the NVS GPU and the CPU, but not for the smallest cases. In Plot C) we therefore show the results for grids of size less than 175 000. We see that both the serial back-end and the CUDA back-end on the NVS follow approximately straight lines, but the CUDA back-end on the Tesla has a profile more similar to a staircase. The Tesla runs in approximately the same time for cases between 40 000 to 60 000 cells, then has a small increase and stay close to constant again from 80 000 cells and up. The runtimes on the Tesla for cases between 80 000 and 250 000 cells differs only by 10%. We also see in Plot D) that the relative processing speeds are a lot more stable on the NVS, and that the difference between the NVS and the Tesla is relatively low for the small cases.

In plot B) from Figure 5.8 we see how the relative processing speed is dramatically increasing during this period, and even after the total runtimes starts increasing we still get a higher relative performance on the Tesla. When we compare this with the NVS, we see that we achieve peak performance much earlier on the NVS than on the Tesla. After 400 000 cells, the NVS has approximately constant processing time per million cells, as we see in Plot E). Plot F) shows that the relative performance of the serial back-end is decreasing when we increase the number of cells for small cases. This is related to how much data that fits to the CPU's cache.

If we compare the CUDA back-end to the serial back-end, we see that for the smallest cases we do not achieve any huge speed-up. This is likely to be related to the high number of kernel calls (see Section 6.2.3), and we see that when comparing relative processing speed for the largest cases we get a formidable speed-up for this simulator as well.

For the NVS we get a peak performance of 6.25 times the performance of large cases on the CPU. Note also that we are not able to run the largest cases on the NVS as the device run out of memory. For the Tesla however, we get a peak performance 7 times higher as on the NVS, meaning that we get a 43 times speed-up compared to the serial back-end.

Comparison by Optimized CUDA Code

So how good are our results really? As we have pointed out earlier, there are limited information in comparing unoptimized serial code with what we can achieve on the GPU. In comparison it will neither be fair to compare our one-kernel-per-operation simulators to highly tuned and hand-optimized CUDA implementations. But since we have already done the first comparison we should do the second one as well.

A highly efficient shallow water simulator using the numerical method on which we have based our Equelle program, was implemented by Brodtkorb et.al.[10] in 2010. They used a second order approximation in space, while we use a first order approximation of

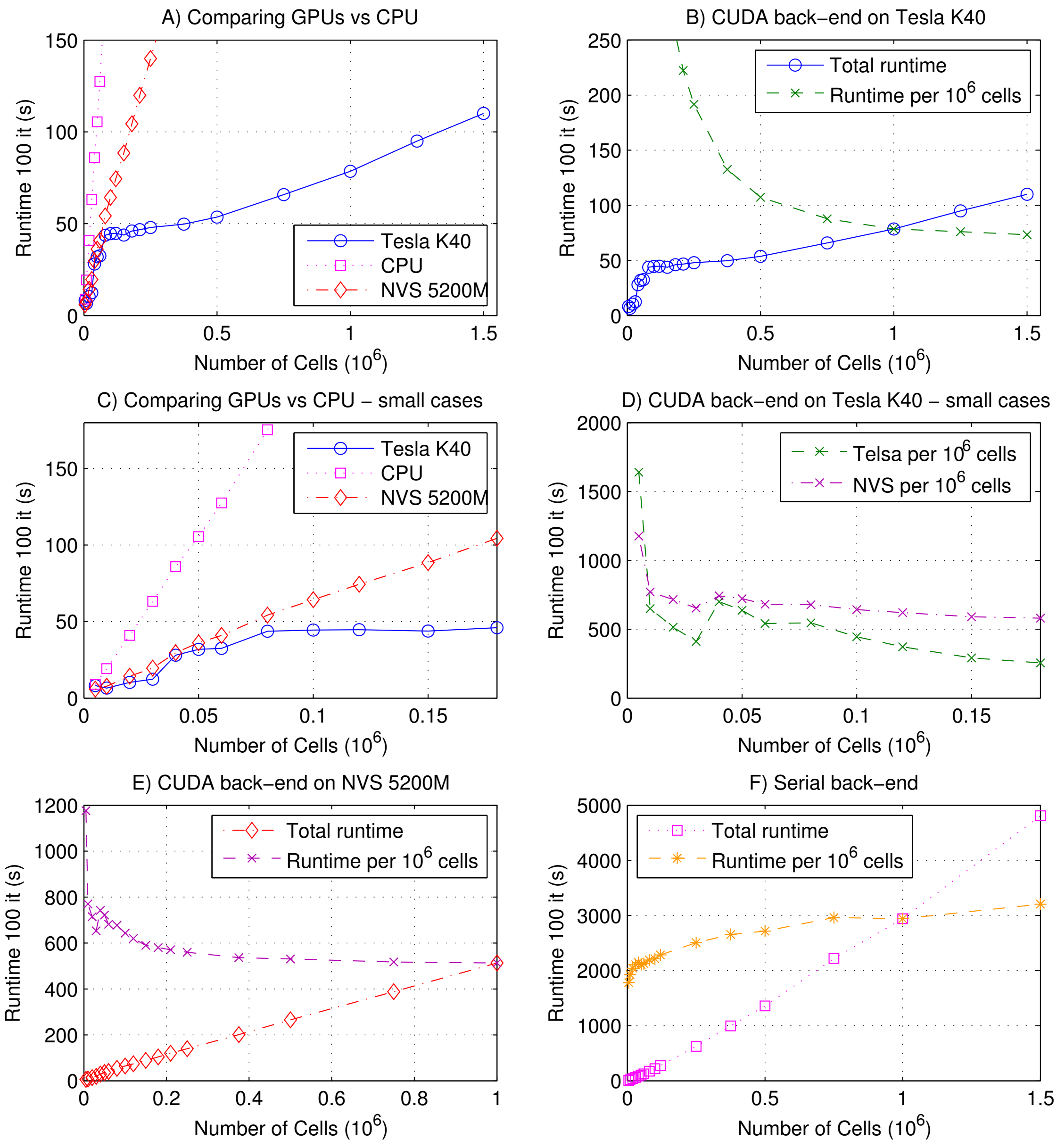


Figure 5.8: Performance testing for 100 iterations of the shallow water simulator for different grid sizes on different hardware. Plot A) gives the trends of the runtimes of all three hardware with focus on the Tesla K40. Since we have different trends on the Tesla for relatively small and large cases, we take a closer look at the small cases on all three hardware in Plot C). Plot B) and Plot E) shows relative processing speed for the Tesla and NVS respectively, where Plot D) shows the same on small grids. Plot F) shows relative speed for the serial back-end.

Table 5.3: The grid sizes used in order to test performance of the shallow water equation.

Num cells (10^6)	0.005	0.01	0.02	0.03	0.04	0.05	0.06	0.08	0.10	0.12
nx	50	100	200	300	200	200	200	200	200	300
ny	100	100	100	100	200	250	300	400	500	400
Num cells (10^6)	0.15	0.18	0.21	0.25	0.375	0.5	0.75	1	1.25	1.5
nx	500	600	300	500	500	500	1000	1000	1000	1500
ny	300	300	700	500	750	1000	750	1000	1250	1000

the same method. The time step is handled by the same Runge-Kutta method in both cases.

They report peak performance of the GPU in terms of megacells (millions of cells) processed per second, and give numbers based on two cases. The first case is a dam break from a wet area onto a dry area, and case two is a radial dam break similar to our test case from Figure 5.4. They show a higher peak performance on the cases with a higher ratio of dry cells, as those cells require less computations by their method. We have used a model on which all cells, dry or not, require the same amount of computations, and we also use a filled basin as test case for the performance tests shown in Figure 5.8. We therefore compare our simulator with their radial dam test case where there are water in all cells of the domain.

The peak performance reported by Brodtkorb et.al using a NVIDIA GeForce GTX 480 is about 150 megacells per second. Our peak performance is obtained from the Tesla which solves one iteration in 0.73 seconds per 1 million cells when using the largest test case. This translates to 1.36 megacells per second, meaning that our Equelle simulator is worse by a factor of 110 compared to the highly tuned CUDA simulator. Note also that the reported numbers are about 4 years older than ours, and that using the tuned simulator on new and more powerful GPUs are likely to give them even higher performance.

In Section 6.2.3 we will take a closer look at some of the reasons for the performance penalty we experience with our shallow water simulator.

Chapter 6

Discussion

6.1 Implemented Optimizations

As briefly mentioned in Section 4.1, we make use of the Nvidia Visual Profiler in the optimization phase of the back-end development. After getting the back-end to the level where it correctly evaluates all functionality of Equelle, we use the profiler to find out which of the functions are most time consuming, and look at how these functions can be evaluated with higher performance. The back-end implementation as described in Chapter 4 are the optimized version, and this section will therefore describe some discarded implementation designs and why those were sub-optimal.

The performance analysis was done with respect to the explicit heat equation simulator. We have looked at both the performance of the initialization phase and compute phase, and therefore chosen a test case using a grid with 555 000 cells structured in a 3D Cartesian grid with dimensions $100 \times 111 \times 50$. Since this is a large case, and the program consists of relatively few operations per cell per time step, we have commented out the line which writes the results to file in order to avoid that process dominating runtime. In order to have a confirmation that we still get correct results, we keep on writing the maximum value of the solution of each time step to the screen. The test case uses 20 time steps.

The test hardware is the laptop GPU, Nvidia NVS 5200M, and we will here go through a four step process that provided significant performance boost to the application. The first prototype used 15.8 seconds on the entire simulation, including parameter file reading, grid construction and the 20 iterations of computation. The reported timings are summarized in Figure 6.1. In Appendix A we have listed the git commits that represents version of the back-end as we will describe here.

Figure 6.2 shows a screen shot of the Nvidia Visual profiler, showing the main features we have used during this process. The screen is split in two parts. The uppermost part shows a time line of the program execution. The line shows what CUDA kernels or GPU activity has been active at all time. For example, we can see how we have very few instances of host to device memory transfers, while we do device to device copying almost all the time. It also seems like we do a lot of copying from the device to host, but as we hover the mouse above each instance, we see that the size of these copies are of the order 8 bytes representing the result from the maximum reduction. The last line we can see in the screen shot shows us how often we do computation on the GPU by launching kernels.

Note that there seems to be no activity between 2 and 7 seconds after the program

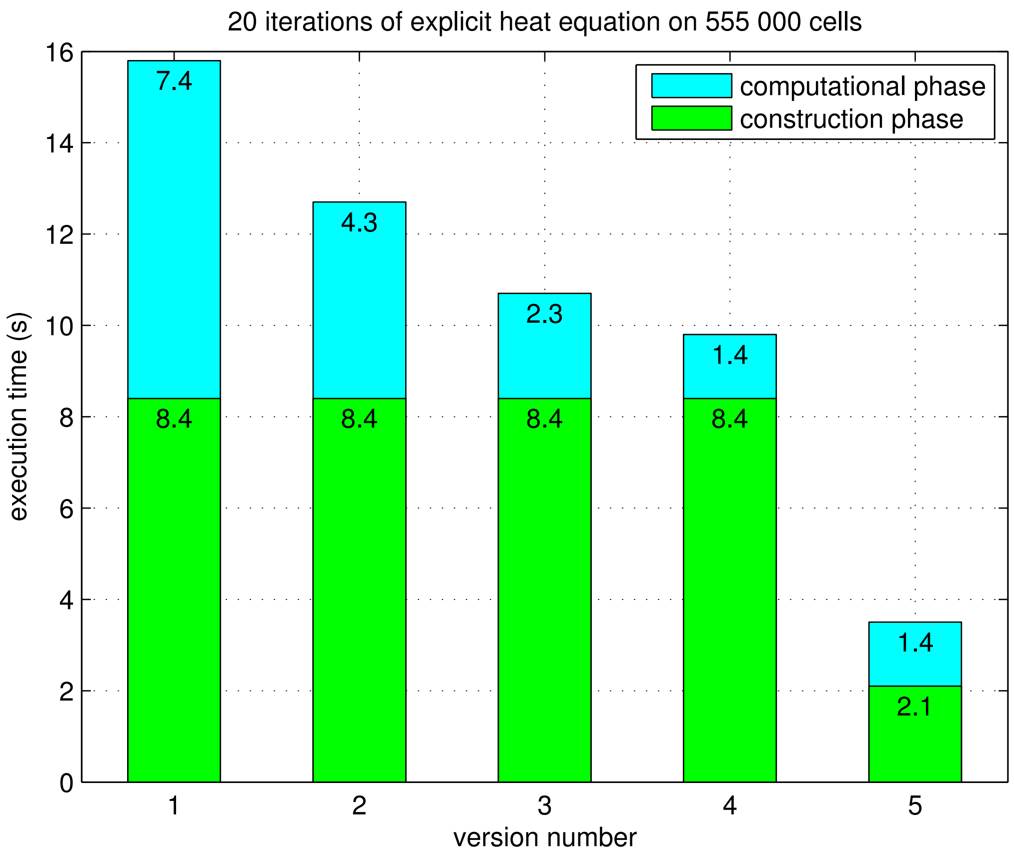
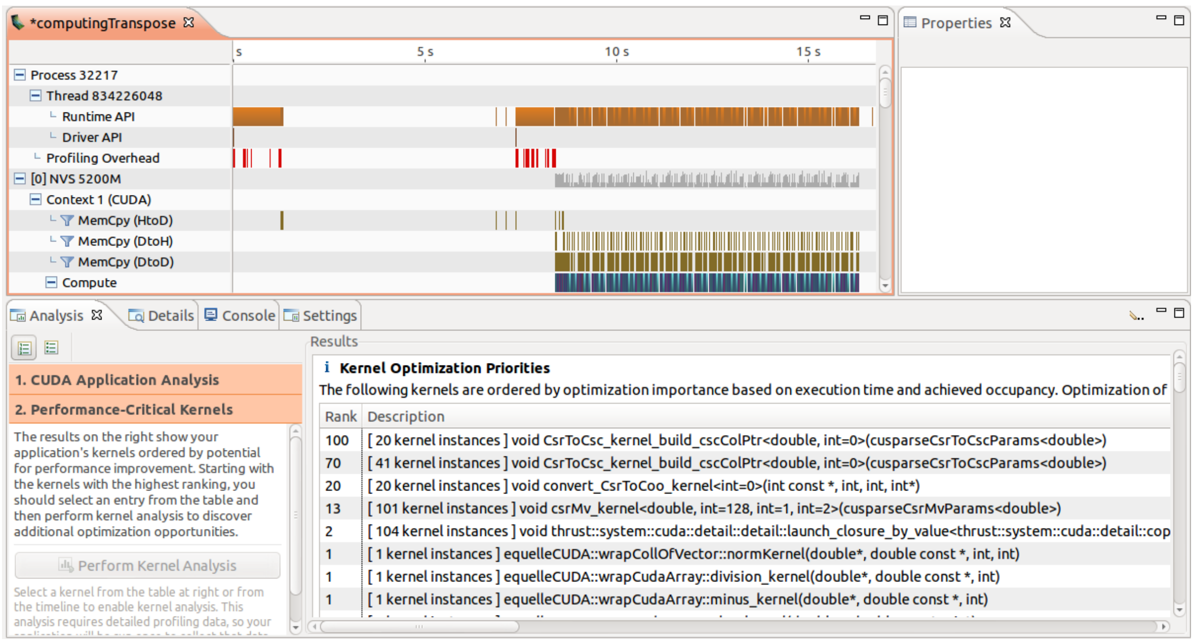


Figure 6.1: Bar graph showing the improvement in simulator execution time as we re-evaluated performance bottlenecks. The numbers on the bars show the amount of time spent in each of the construction and computational phases. Versions are as follows: 1) First prototype before profiling, 2) Removing computation of transpose matrices, 3) Using our own kernels instead of sparse matrix-vector products, 4) Lazy evaluation of grid subsets, 5) Lazy evaluation of `Gradient` and `Divergence` matrix operators.

is started. This is the initialization phase, where the CPU constructs the grid and its neighbourhood relations. This process is not analysed by the profiler because this profiler targets GPU activity only. From the time line it seems like the initialization phase lasts to about 8 seconds, and that our test program therefore spends the same amount of time initializing the simulation as computing the result. Note also that the time line from the profiler is not accurate, as the profiling process requires some overhead. The timings we report here are therefore from running the simulator without profiling. Doing this, we found that the construction phase was 8.4 seconds on the first prototype, as seen in bar 1 in Figure 6.1.

The lower half of Figure 6.2 gives the most valuable information to us regarding which part of our the code we should optimize. The profiler analyse the time spent by kernels and the number of times each of them are called, and give each kernel a rank which reflects the expected trade-of we might get from optimizing the kernel. Kernels with high ranks are expected to have big impact on the performance of the application, while kernels with



The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the kernels with the highest ranking, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.

Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your

Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of

Rank	Description
100	[20 kernel instances] void CsrToCsc_kernel_build_cscColPtr<double, int=0>(cusparseCsrToCscParams<double>)
70	[41 kernel instances] void CsrToCsc_kernel_build_cscColPtr<double, int=0>(cusparseCsrToCscParams<double>)
20	[20 kernel instances] void convert_CsrToCoo_kernel<int=0>(int const *, int, int, int*)
13	[101 kernel instances] void csrMv_kernel<double, int=128, int=1, int=2>(cusparseCsrMvParams<double>)
2	[104 kernel instances] void thrust::system::cuda::detail::detail::launch_closure_by_value<thrust::system::cuda::detail::cop
1	[1 kernel instances] equelleCUDA::wrapCollOfVector::normKernel(double*, double const *, int, int)
1	[1 kernel instances] equelleCUDA::wrapCudaArray::division_kernel(double*, double const *, int)
1	[1 kernel instances] equelleCUDA::wrapCudaArray::minus_kernel(double*, double const *, int)

Figure 6.2: Screenshot from the Nvidia Visual Profiler analysing the first prototype of the complete CUDA back-end used on the explicit heat equation.

low ranks are not significant. It is therefore a straight forward process to find the kernel bottlenecks using this tool.

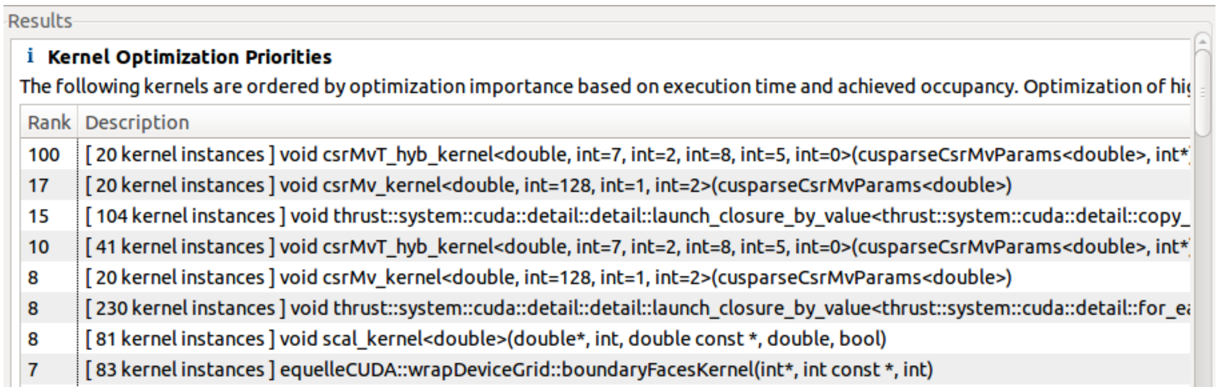
6.1.1 Matrix Transpose

We described in Section 4.3.2 how we construct the matrix representing the `Extend` operator by constructing the `On` operator for the reverse case and taking its transpose. The first implementation of the transpose used the cuSPARSE library for actually computing the transposed matrix with the `cusparseDcsr2csc` conversion function. The function maps a matrix stored in the CSR-format over to CSC-format (compact sparse *column*-major format). By interpreting a matrix stored in CSC-format as a matrix in CSR-format, we get the transposed.

This method produced the correct results, but kernels used to compute the transpose was identified to be a significant performance bottleneck. This can be seen in Figure 6.2, where we see that the three highest ranked kernels are related to the `cusparseDcsr2csc` function. These three kernels are called from inside the cuSPARSE library, but they are all called from this one function.

In order to avoid this bottleneck, we added a private member in the `CudaMatrix` class of the type `cusparseOperation_t`. This is an enumerator defined in the cuSPARSE library which is used to flag if matrices used as input to the library functions should be interpreted as being transposed or not. This allows us to store the non-transpose of all matrices, but make other functions use the transposed of it.

From the cuSPARSE documentation[2] we also read that the `csr2csc` function requires a significant amount of extra storage. By storing a single flag instead we avoid using this temporary memory as well. By recompiling our test simulator we found a decrease in execution time with 3.1 seconds. Since the matrix transpose is not used in the initialization



Results

Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of hi

Rank	Description
100	[20 kernel instances] void csrMvT_hyb_kernel<double, int=7, int=2, int=8, int=5, int=0>(cusparseCsrMvParams<double>, int*
17	[20 kernel instances] void csrMv_kernel<double, int=128, int=1, int=2>(cusparseCsrMvParams<double>)
15	[104 kernel instances] void thrust::system::cuda::detail::detail::launch_closure_by_value<thrust::system::cuda::detail::copy_
10	[41 kernel instances] void csrMvT_hyb_kernel<double, int=7, int=2, int=8, int=5, int=0>(cusparseCsrMvParams<double>, int*
8	[20 kernel instances] void csrMv_kernel<double, int=128, int=1, int=2>(cusparseCsrMvParams<double>)
8	[230 kernel instances] void thrust::system::cuda::detail::detail::launch_closure_by_value<thrust::system::cuda::detail::for_ea
8	[81 kernel instances] void scal_kernel<double>(double*, int, double const *, double, bool)
7	[83 kernel instances] equelleCUDA::wrapDeviceGrid::boundaryFacesKernel(int*, int const *, int)

Figure 6.3: The kernel bottlenecks for the explicit heat equation after redesigning the treatment of transposed matrices.

phase, this increase in performance only took place in the computation phase, which was decreased from about 7.4 to 4.3 seconds, as can be seen in bar 2 in Figure 6.1.

6.1.2 Discarding Sparse Matrix-Vector Products

We ran a new profiler analysis in order to find the new hotspots, and the new Kernel Optimization Priorities are shown in Figure 6.3. In order to understand them we need once again to go into the earlier implementation choices.

When we started writing the CUDA back-end, we focused on explicit solutions only, meaning that we did not need to consider derivatives and AD. We then expanded the back-end to also include AD and added the matrices for `Gradient`, `Divergence`, `On` and `Extend` as described in Sections 4.3.2 and 4.4.1. With these matrices at hand, we could just as well use them for the evaluation of the values also, and not only for the derivatives. This was an attractive alternative to the kernel implementations, as the kernels consisted of several functions and about 150 lines of code for each of the Equelle functions. If all these lines could be replaced with a single multiplication instruction it would make the code more readable, as well as easier to maintain and expand in the future.

From Figure 6.3 we see that this design was not a good idea. Note that 4 out of 5 of the most important kernels to optimize are related to sparse matrix-vector products. This makes it clear that if we are able to remove matrix-vector multiplication from our code altogether, we are likely to experience better performance. We therefore went back to the implementation where the scalar values were computed using our own kernels.

Rerunning our simulator without using the matrices, we achieved another 2 seconds faster execution time, giving a total runtime of 10.7 seconds. As with the matrix transpose, this performance boost was only taken from the computing phase of the simulator, which is an improvement of about 46% in this phase, as can be seen in bar 3 Figure 6.1.

Note that the profiler analysis from Figure 6.3 shows that the single highest ranked kernel is still related to matrix transpose, and even with similar rank values as we saw in the first analysis from Figure 6.2. This could have been interpreted as if removing the explicit computation for the transpose did not provide any improvements. However, the significant improvement of the execution time shows that using transpose flags instead gave a huge impact. The analysis showing that transposed matrix-vector multiplication is still a hotspot only indicates how important it was to change that design in the first place.

6.1.3 Lazy Evaluation of Common Grid Subsets

When we look at the profiler suggestions from Figure 6.3, we see that the two kernels with highest ranks not related to the matrix-vector product are functions from the Thrust library. When we use our own kernels instead of using the matrix operations, the Thrust kernels are still highest ranked by the profiler, with rank values 100 and 54. The third highest ranked kernel then becomes our `boundaryFacesKernel`, which also can be seen in the last line of Figure 6.3, now with rank 43.

The Thrust functions are related to the `remove_if` function used in the process of finding `Boundary` and `Interior` grid subsets. As described in Section 4.3.1, these functions are implemented by initializing all elements with a non-valid integer, updating only the elements that are part of the set we want, and extracting all elements that now contain valid indices. This extraction is what the profiler indicates is our new bottleneck.

It seems hard to find a new and faster method to find these subsets, compared to the previous cases, and we have decided to not change the way we compute these sets. Instead, we look at the option of moving the computation of these sets to the construction and initialization phase of the simulation, then store it and just use this one copy whenever it is needed.

Pre-computing the grid subsets require some extra memory to be used during the entire program, but we have to remember that the entire back-end already is designed without memory minimization in mind. First of all, the data arrays stored in the `DeviceGrid` class contains several arrays which sizes depend on the grid. Second, there is a minimal reuse of variables, since all variables defined by the user is `const` by default. Storing four extra arrays which in total contain `number_of_faces_ + number_of_cells_` integers is therefore not expected to have a dramatic impact on the entire memory consumption.

When we consider storing these grid sets, it is natural to ask "What if we do not need them?" When we look at Figure 6.1, we see that the construction phase is large enough as it is, and we do not want to add more unnecessary work. With this in mind, we decide to do a *lazy evaluation* of the grid subsets. This means that we compute the grid subset the first time it is requested, and then store it for later. Any standard grid subset is therefore computed no more than once, and we never compute sets that are not used in the simulator.

This is done by adding to the `DeviceGrid` class `mutable CollOfIndices` member variables which we are allowed to change even if we use a `const` instance of the class. We also store boolean variables indicating whether the collections are initialized or not, and implement the functions requesting the sets as shown by the example in Listing 6.1. Note that we do not create a copy of the `boundary_faces_` by calling this function, but rather return a constant reference to the variable in the `DeviceGrid` class.

We recompile our test simulator with this updated functionality and experience a drop of another 0.9 seconds in execution time. Since this optimization is also done in the computational phase, we actually notice that this reduces the computational phase by almost 40%. This is quite significant having in mind that most simulations are likely to have more than 20 explicit time steps. This test is shown in bar 4 of Figure 6.1.

Listing 6.1: Lazy evaluation of `BoundaryFaces()` in the `DeviceGrid` class.

```
const CollOfFace& DeviceGrid::boundaryFaces() const {
    if ( boundaryFacesEmpty_ ) {
        createBoundaryFaces_();
    }
    return boundary_faces_;
}
```

6.1.4 Lazy Evaluation of `Gradient` and `Divergence` Matrices

Looking at bar 4 in Figure 6.1, we see that most of the execution time is now spent in the construction phase of the program. Since we are likely to have a huge number of iterations, and since this phase is independent of number of iterations, performance is not critical for this phase. However, we are likely to run small cases during the development of new simulators, or when we implement new functionality to the back-end, and we therefore look at how we can improve this part of the program as well.

The construction phase consists of the following steps:

- Constructing the grid.
- Transferring the grid to the device.
- Construction of host matrices for `Gradient` and `Divergence` operators by the class `Opm::HelperOps`.
- Transferring these matrices from host to device.

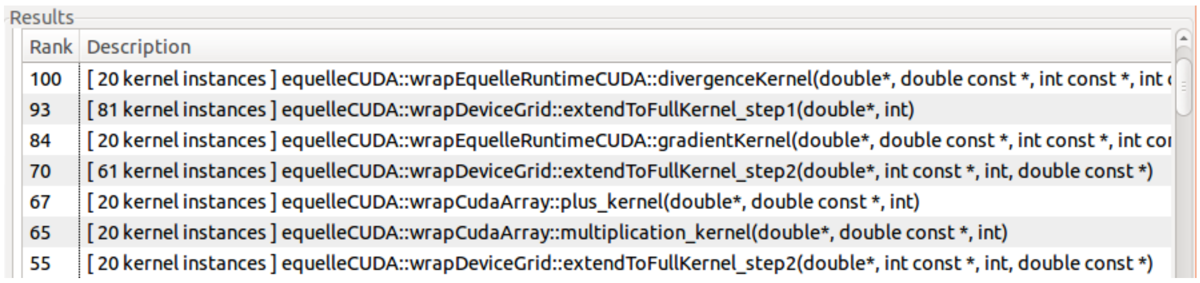
The matrices for the `Gradient` and `Divergence` operators are only used to compute the derivative matrices, and they are therefore never used in explicit methods. Because of this we would prefer not to spend time on constructing matrices we are never going to use. We therefore implement a lazy evaluation of these matrices as well.

Since the process of constructing these two matrices are a two-step procedure, we also implement a two-step lazy evaluation. Taking the `DeviceHelperOps::gradient()` function as example, we use the work flow as described in pseudo code in Listing 6.2.

Listing 6.2: Pseudo code for lazy evaluation of the matrix representation of the `Gradient` operator.

```
const CudaMatrix& DeviceHelperOps::gradient() {
    if (gradient matrix not on device) {
        if (host matrices not created) {
            create host matrices using OPM
        }
        copy gradient matrix from host to device
     }
    return gradient_matrix_;
}
```

Since the test simulator uses an explicit method, it will never construct any matrices for these operators. By running the simulator again, we do the construction phase in only one fourth of the time compared to before, saving us a lot of time during small tests. Note that for implicit methods, we will still have to do the same amount of work in the


Results

Rank	Description
100	[20 kernel instances] equelleCUDA::wrapEquelleRuntimeCUDA::divergenceKernel(double*, double const *, int const *, int
93	[81 kernel instances] equelleCUDA::wrapDeviceGrid::extendToFullKernel_step1(double*, int)
84	[20 kernel instances] equelleCUDA::wrapEquelleRuntimeCUDA::gradientKernel(double*, double const *, int const *, int cor
70	[61 kernel instances] equelleCUDA::wrapDeviceGrid::extendToFullKernel_step2(double*, int const *, int, double const *)
67	[20 kernel instances] equelleCUDA::wrapCudaArray::plus_kernel(double*, double const *, int)
65	[20 kernel instances] equelleCUDA::wrapCudaArray::multiplication_kernel(double*, double const *, int)
55	[20 kernel instances] equelleCUDA::wrapDeviceGrid::extendToFullKernel_step2(double*, int const *, int, double const *)

Figure 6.4: Kernel performance analysis after the 4 steps of optimization.

construction phase, but the construction of the `Gradient` and `Divergence` matrices will then be moved to the first iteration of the computation phase.

By running the profiler again, we get the ranking of our kernels as shown in Figure 6.4. The first thing we note, is that all the kernel bottlenecks at this stage are our own kernels, meaning that we do not suffer from loss of performance by taking advantage of external libraries. Second, all 7 kernels shown in the figure have ranks above 50. Earlier we had the case that one kernel, or a set of kernels related to the same library function, was significantly more important to optimize than others. Now however, it seems like it will not be enough to optimize only one kernel or functionality to get a performance boost, but that we rather have to re-evaluate how we have implemented all of them.

In summary we have achieved a 4 times speed-up by using the profiler's "Kernel Optimization Priorities" analysis to detect what functionality had biggest impact on performance. This performance gain came equally from both the construction phase and computation phase of the simulator, meaning that we have this amount of speed up regardless of the number of iterations.

6.2 Performance Limiting Factors

We have shown in Chapter 4 how the CUDA back-end has been implemented to match the minimally changed compiler front-end, and we have given results showing that the implementation gives correct simulators, along with a performance analysis in Chapter 5. In this section our main focus is to look at which factors are stopping us from achieving even higher performance, targeting the Tesla K40 GPU.

We will go through each of the three simulators used in Chapter 5 and look at what the of performance limiting factors for each of them are. The explicit heat equation has to some extent been analysed in Section 6.1, but we will in Section 6.2.1 look at what are the current bottlenecks. In Section 6.2.2 we will discuss some of the improvement potential related to the automatic differentiation, and we will also look at the profiler's suggestions for the shallow water equations in Section 6.2.3, as that simulator requires significantly more operations per iteration than the heat equation.

6.2.1 Explicit Heat Equation

We use a test case of 3 million cells on a Cartesian grid and run the simulator for 40 timesteps, a program executing in about 3.3 seconds on the Tesla GPU, and where the

2 first seconds are initializing and grid construction. The profiler makes a similar analysis for the kernel optimization potentials as shown in Figure 6.4, which was done on the NVS 5200M GPU, even though the actual rankings contain some small differences. There is still no single kernel which has significantly higher rank than the others.

On a more general note, the profiler inform us that our use of memory is inefficient. This is not directly related to our low reuse of variables or the big amount of temporary variables needed (see Section 6.3.1), but rather because of our irregular memory access pattern. Two of the warnings from the profiler say that the global memory load and store efficiency is low, and that this most likely is because of irregular access patterns or bad alignment.

Since the back-end is designed for handling unstructured grids, we already expect operations that use multiple elements from the same arrays to be unaligned as we use only linear memory. Normal memory optimization strategies for GPUs, such as using shared memory and multidimensional arrays are therefore not possible to implement. In order to see this, we look at the `divergenceKernel` as an example, and use 2D Cartesian grid for simplicity. The kernel code was shown in Listing 4.12. Since a Cartesian grid is only a special case of unstructured grids, the memory access pattern will have the same irregularities as for unstructured grids.

In the 2D Cartesian grid all cells have 4 faces each, and in order to compute the `Divergence` we want to do some processing of the data stored in an input array `flux`. Figure 6.5 shows which operations each thread has to make towards the GPU's global memory. In order to know which elements we need from the `flux` array we read which faces that belong to our cell in `cell_faces_`, and in order to know where to look in the `cell_faces_` array we need to read two elements from `cell_facepos_`. We also need to read one element from `face_cells_` for each `flux` element, in order to know if it should be added or subtracted to the result. This means that each thread is required to read 15 elements from global memory in order to compute one element for the resulting array `div`.

As we see from Figure 6.5, we are guaranteed that the elements in `cell_faces_` are continuously stored in memory, but the elements in `face_cells_` and `flux` will not be. For the case of 3D Cartesian grids, we will need to access 6 faces for each cell, and therefore end up with 21 reads from global memory. We will here give two suggestions on how to limit the number of global reads compared to the amount of computation.

Writing Separate Kernels for Computations on Cartesian Grids

It is hard to avoid irregular memory access when we use unstructured grids. However, it is not unthinkable that we could implement the back-end so that it uses different kernels depending on whether the grid is Cartesian or not. We could set a flag in simulations where the parameter file defines `nx`, `ny` and `nz`, and check this flag each time we call a function which could take short-cuts for Cartesian grids.

Consider again the `Divergence` operator on a 2D Cartesian grid, where we also define all normal vectors to be in positive x or y directions. We can then define a mapping so that the linearly stored values on `AllCells()`, c, can be organized in a $n_x \times n_y$ grid in C, so that

$$C(i,j) = c(i + n_x j) \quad \text{for} \quad i = 0,1,...,n_x - 1,\ j = 0,1,...,n_y - 1. \tag{6.1}$$

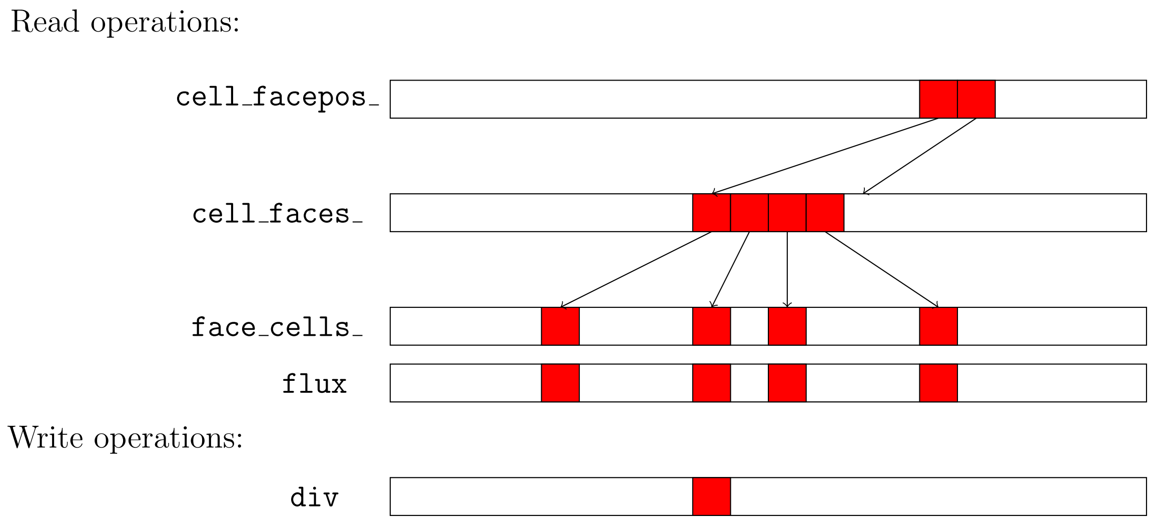


Figure 6.5: The data elements each thread is required to access in global memory in order to compute the `Divergence` on an unstructured grid. This is the model used in the CUDA back-end.

We could create a similar mapping for values f stored linearly on `AllFaces()` as well onto two grids $F_x = (n_x + 1 \times n_y)$ and $F_y = (n_x \times n_y + 1)$, so that

$$F_x(i,j) = f(i + (n_x + 1)j) \quad \text{for} \quad i = 0, 1, ..., n_x,\ j = 0, 1, ..., n_y - 1 \tag{6.2}$$

and

$$F_y(i,j) = f(i + n_x j + (n_y(n_x + 1))) \quad \text{for} \quad i = 0, 1, ..., n_x - 1,\ j = 0, 1, ..., n_y. \tag{6.3}$$

This mapping onto the grid can be seen in Figure 6.6.

We can now formulate the `Divergence` operator on f and store it in u as

$$U(i,j) = F_x(i+1,j) - F_x(i,j) + F_y(i,j+1) - F_y(i,j). \tag{6.4}$$

With this structured interpretation, we could further map the `Divergence` operator nicely to a 2 dimensional block in CUDA, where threads within the same block could cooperate on the process of reading data from the global memory. By calling kernels with blocks of size 8 by 8 we could use them to compute a 7 by 7 domain of divergences.

The idea would be to allocate shared memory according to Figure 6.7, and let each thread with thread ID that matches an element in those domains load this element. We would then have to synchronize the block so that all shared memory elements would be written to before we would try to read them. The result could then be easily computed by Equation (6.4). Listing 6.3 shows an outline of how this CUDA kernel might look like.

Other functions that could benefit from special Cartesian implementations are

- `Gradient`,
- volume of cells as all cells are the same size which we can compute,
- area of faces as all faces with the same orientation are the same size,

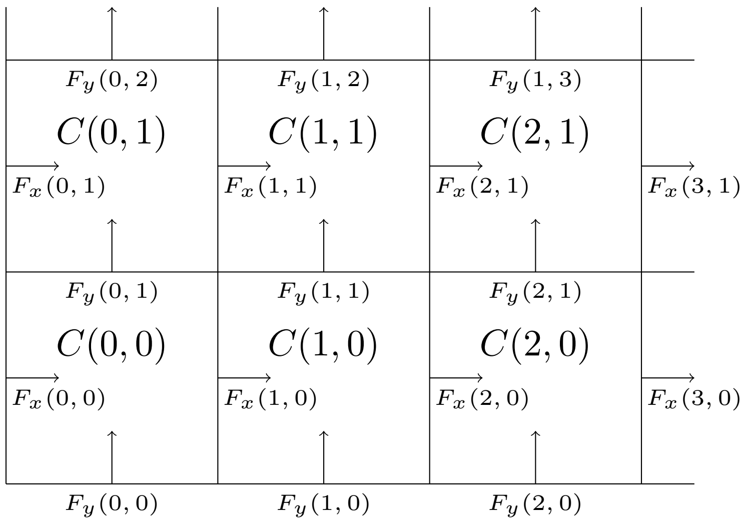


Figure 6.6: Cell and face data interpreted by a 2D Cartesian mapping. In the linear memory model used in the CUDA back-end, elements would be stored as $[C(0,0), C(1,0), ..., C(n_x-1,0), C(0,1), ..., C(n_x-1, n_y-1)]$, and all F_x values would be stored before any F_y.

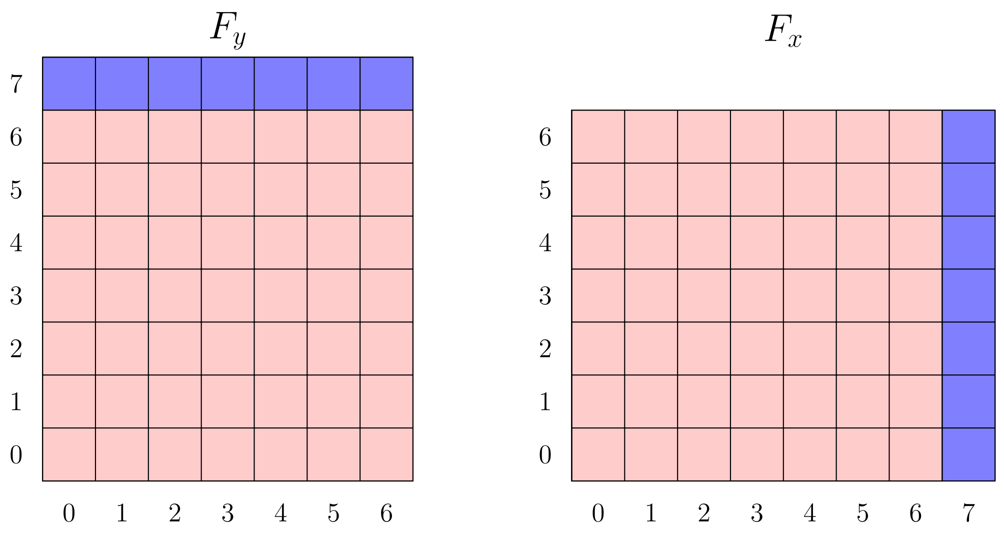


Figure 6.7: In order to compute the `Divergence` of a 7 by 7 domain, we would need face data according to these two domains, where F_x and F_y as in Equation (6.2) and (6.3). The blue elements are read by threads that do not compute anything for the output, but only helps loading data to shared memory for the other threads to use.

- subsets of the grid.

Listing 6.3: Outline for a CUDA kernel for computing the `Divergence` by assuming 2D Cartesian grid.

```
__global__ void cart_DivergenceKernel( double* div ,
                                       const double* flux ,
                                       const int nx,
                                       const int ny)
{   // Assumes block sizes 8 by 8,
    // Find our global index:
    const int i = threadIdx.x + (blockDim.x - 1)*blockIdx.x;
    const int j = threadIdx.y + (blockDim.y - 1)*blockIdx.y;

    // Allocate shared memory:
    __shared__ double Fx[8][7];
    __shared__ double Fy[7][8];

    // Write to shared memory:
    if (i < 8 && j < 7)
        Fx[i][j] = flux[ i + (nx+1)*j ];
    if ( i < 7 && j < 8)
        Fy[i][j] = flux[ i + nx*j + ny*(nx+1) ];
    // Synchronize all threads in block:
    __syncthreads();

    // Calculate divergence
    if ( i < 7 && j < 7 ) {
        div[ i + nx*j ] = Fx[i+1][j] - Fx[i][j] + Fy[i][j+1] - Fy[i][j];
    }
}
```

Kernel Fusion by Changing the Compiler Front-End

Another reason for why the Profiler points out memory usage as an issue with our back-end, is that reading from and writing to global memory is usually a more time consuming task than performing arithmetic operations. Our kernels are mostly very small, and most of the times we read one or two elements from global memory, do one arithmetic operation and write the result back to global memory.

If we were able to combine kernels which operates on the same data, we would likely be able to improve performance by reducing memory operations towards global memory. As we will see in Section 6.3.1 we generate a lot of intermediate results which are written just to be read at once, and then go out of scope. The process of combining multiple kernels into one bigger kernel is called kernel fusion, and we will take a closer look at it later.

6.2.2 Implicit Heat Equation

By running the implicit heat equation simulator through the profiler, we easily get a fast overview of which functionality are limiting factors for most implicit simulators. We apply a 3D Cartesian grid as test case, using 500 000 cells and 10 iterations.

The major kernel bottleneck is the same as the original bottleneck for the explicit case as described in Section 6.1. It is kernels related to the cuSPARSE function `csr2csc` for converting a matrix from CSR- to CSC-format. This may seem surprising at first, since we never call it and only use a flag to mark transposed matrices. The reason it is still used is that the sparse matrix-sparse matrix multiplication function from cuSPARSE, `cusparseDcsrgemm`, is only implemented for non-transposed matrices. When we call the `csrgemm` function with the transpose flag, cuSPARSE calls the `csr2csc` function internally before multiplying the matrices as non-transposed matrices.

After kernels related to the `csr2csc` function, we get highest rank on kernels related to the `csrgemm` sparse matrix-sparse matrix multiplication function. Full matrix-matrix products are operations that maps perfecty to GPUs, but when the matrices are sparse, the non-regular memory access pattern results in bad performance.

The multiplication process also uses a two-step procedure, as described in Section 4.2.3. Each time we multiply two sparse matrices, we first need to find the new non-zero pattern, which gives us the `csrRowPtr` array in the CSR storage format. Since the sparsity patterns we get will not change between iterations, one potential optimization could be a lazy evaluation of the sparsity patterns. That way we could compute the `csrRowPtr` arrays during the first iteration and then get away with only a one-step multiplication procedure for all other iterations.

One optimization that we have already implemented is multiplication with diagonal matrices from the left. This is an operation that is widely used in the CUDA back-end, and can be translated to multiplying the diagonal entry to all elements in the corresponding row in the right-hand-side matrix.

In order to optimize implicit simulators, an effort has to be made to find good ways to treat transposed matrices, or to omit the use of transposed matrices in the back-end all together. The next step would likely be to look at special cases for sparse matrix-sparse matrix multiplications given the sparsity patterns that are typical for Equelle simulators.

6.2.3 Shallow Water Equations

The shallow water simulator differs in many ways from the simulator for the heat equation. The heat equation only consists of a few operations for each iteration, where it calculates the fluxes across all faces using the `Gradient` operator, applying some scaling and then the `Divergence` operator before the result is added to the next time step.

The shallow water Equelle program consists of 400 lines of code, with an array of three `Collection Of Scalar`s used for the unknowns, one for each of the three equations. There are plenty of ternary if operators checking for dry cells, and in general a lot of complex expressions. As we saw in Section 5.4 the simulator does not gain much from using the CUDA back-end as long as the number of cells are small.

By running the shallow water simulator through the profiler, we find that each iteration invokes more than 1000 kernels, where the multiplication kernel alone is called 140 times. In comparison, each iteration of the heat equation requires less than 35 kernels to be launched. It is therefore expected that the shallow water simulator would get a larger performance gain from combining kernels than what the heat equation would get.

Another problem with the huge amount of kernel invocations is that we do a lot of device-to-device copying. The test case we look at is a 100 by 100 grid, and therefore consists of 10 000 cells. The equations are solved for the water depth h, and the velocities

u and v in x and y direction respectively, and we are therefore interested in generating 3 `Collection Of Scalar On AllCells()` variables. The three of them sums up to

$$3 \cdot 10000 \cdot 8\text{bytes} = 240\text{kB} \tag{6.5}$$

of results for each iteration. The total amount of device-to-device copying is more than 300 MB per iteration. This is more than 1000 times more data than needed just for the unknowns. We note that the order of extra data we generate is comparable to the number of kernels launched in each iteration. It is therefore likely that by implementing some optimizations in the front-end for reducing the amount of kernel calls, we will also reduce the amount of data that we move around on the device.

6.3 Future work

Even though we now have multiple back-ends available for Equelle, the language is still in its early development phases. There are plans for extending the language by adding more functionality to it, and even to add new dialects to the language for solving PDEs in terms of node stencils instead of in terms of flux relations as we do now. There is also an MPI back-end under development for deploying the simulators on computer clusters, and a cloud based back-end is also planned. We will in this section focus on the future of the CUDA back-end, rather than the future of the entire language.

The future work for providing better performance for this back-end should be focused in two directions. The first direction is to implement functionality in the compiler front-end for doing optimization on the abstract syntax tree. This would make the intermediate representation more complex, but could potentially give huge performance improvements for both the serial and the CUDA back-ends.

The second direction would be to keep improving the CUDA back-end. We point out some weaknesses with the back-end and some additional techniques or concepts that we have not prioritized to look into yet.

6.3.1 Front-End Optimizations

In the discussion on the intermediate representation in Section 3.2.1, we mentioned that in order to produce the intermediate code used for our back-end, we needed to sacrifice back-end specific optimizations that could have been implemented in the front-end. We will here give some suggestions on concepts that should be optimized in order to get higher performance.

Consider an arithmetic expression

$$d = (a + b)c + ab \tag{6.6}$$

where all variables are of type `Collection Of Scalars` defined on the same set. When this line is executed by the CUDA back-end, the arithmetic operations are executed by launching the kernels for addition and multiplication one by one, as

```
temp1 = a + b
temp2 = temp1 * c
temp3 = a * b
d = temp2 + temp3
```

Each of the operations will launch kernels where each thread does two reads from global memory, perform the operation and write the result. In total the expression in Equation (6.6) would require 8 reads and 4 writes to global memory, distributed on 4 distinct kernel calls.

The better way to handle the expression in Equation (6.6) would be to make the front-end write a CUDA kernel to take a, b and c as input and d as output, where each thread would do the entire computation

```
d[i] = (a[i] + b[i])*c[i] + a[i]*b[i];
```

This operation requires only 3 reads and 1 write, and would therefore be more efficient. The front-end would also have to generate a wrapper function for setting up and launching the kernel, and a suggestion for a generated code is shown in Listing 6.4. The expression is generated inside a kernel `kernelX` where `X` can be a number giving unique function names. This kernel is called from the function `wrapKernelX`, which first create the resulting `CollOfScalar` which will be of the same size as `a`. The wrapper is called from the main generated function, in which the current front-end generates all code. Also have in mind that the compiler would only let operations between variables defined on the same set be accepted, so all variables would be of the same size.

Listing 6.4: Suggested generated code with optimizing CUDA front-end.

```
// Inside main generated code:
...
const CollOfScalar d = wrapKernelX(a, b, c);
...

// Stand alone functions
CollOfScalar wrapKernelX( const CollOfScalar& a, const CollOfScalar& b,
    const CollOfScalar& c) {
     CollOfScalar out(a.size());
     kernelSetup s(a.size());
     kernelX<<<s.grid, s.block>>>(out.data(), a.data(), b.data(),
         c.data(), a.size());
     return out;
}

__global__ void wrapKernelX( double* out, const double* a, const
    double* b, const double* c, const int size)
{
     const int i = myID();
     if ( i < size ) {
          out[i] = (a[i] + b[i])*c[i] + a[i]*b[i];
     }
}
```

Note that this type of code generation would also be attractive to do on multiple lines of code in cases where all variables operate in a one-to-one fashion without any need to globally synchronize all threads. Implementing such analysis in the front-end is however not straight-forward.

A different technique used to solve this problem on CPUs is using expression templates. This technique overload operators to call template functions which builds an expression tree in order to loop through the data structures only once even for complex expressions.

The technique is used by Eigen[19], among others, on which the serial back-end is built. It does not require any other intermediate code than the one we already use. An alternative to the above front-end changes could therefore be to look at using expression templates with CUDA, even though it is only able to help us one by one line and not for a block of code.

6.3.2 Back-End Improvements

Even though we have shown good performance of the CUDA back-end considering the limitations of the intermediate code, there are still parts of the library that will need more work. The parts we will mention here will mostly be unrelated to the front-end improvements that we have discussed previously.

The CUDA back-end uses a given **number of threads** per block for any kernel and on any GPU. The reason we have chosen 512 threads per block is that this is a block size which all GPUs with double precision floating-point operations support. Most new GPUs, included the two GPUs we have used for testing in this thesis, allow up to 1024 threads in each block, but choosing that many threads would limit the amount of GPUs that could run the back-end.

It is unknown if 512 threads per block is an optimal choice, and investigating this should be on the to-do list for the future development. Additionally, it could be interesting to make the choice of block size depend on the size of the grid used in each execution, and have some kind of automated optimization related to this.

A technique we have not used in the back-end is the use of **streams** to make the GPU execute two or more different tasks in parallel. In order to use streams we need to identify tasks, or in this case Equelle functions, that do not depend on each other, and which therefore can be evaluated independently. An example of such cases is the `Output` function which writes a `Collection Of Scalar` to screen or file. We could use a separate stream for copying the given data from the device to host, and immediately continue on doing computations in the main stream.

There is one Equelle function that is not supported in the CUDA back-end, and that is `NewtonSolveSystem`. The reason for why implicit methods of **systems of PDEs** are harder to solve than for a single equation is that we get multiple primary variables. Consider the shallow water equations, where h, u and v are our unknown. Variables used in the numerical methods would depend on these three variables, and a variable $F = F(h, u, v)$ would get derivatives associated by all three of them, as $\nabla_h F$, $\nabla_u F$ and $\nabla_v F$. Before we could solve a linear system we would have to combine these Jacobian matrices in order to construct the matrix that would be used by the solver. Implementing this functionality has not been prioritized in this thesis.

The class `LinearSolver` for **solving linear systems** has been implemented to cover a minimum of what has been needed. As seen in Table 4.1 we have not implemented all the combinations of solver and preconditioners we have mentioned. This class should therefore be expanded in order to become more robust and flexible. It would perhaps not be necessary to start developing more linear solvers for CUDA from scratch, but rather keeping an eye out for new methods offered by Cusp in the future.

The next step after having a well performing CUDA back-end is to combine the CUDA back-end with the MPI back-end in order to execute Equelle simulators on **multi-GPU** environments. This would increase the amount of total global memory, as well as computational resources, and would therefore help us run even larger simulations.

Chapter 7

Conclusion

In this master thesis we have developed a CUDA back-end for the Equelle compiler. This has been done without changing the way the Equelle front-end generates the intermediate C++ code, and the back-end has therefore been implemented as a CUDA/C++ library. This is according to the idea that new Equelle back-ends should be implemented as libraries with similar user interfaces matching the intermediate code.

We have shown correctness of the implemented back-end by using three different simulators written in Equelle. Two heat equation simulators using explicit and implicit methods, and a simulator for the shallow water equations which was implemented in the Specialization Project[20]. In order to solve the linear system arising in implicit methods, we have used operator overloading to implement automatic differentiation of first order successfully for the GPU. We have also shown that we do not require Cartesian grids, but are able to run the simulators on unstructured grids as well.

Performance has been measured by giving different grid sizes as input to the three simulators, and we experienced a significant speed-up by running the Equelle simulators on a Tesla K40 GPU compared to using the serial back-end. For the largest cases we obtained a 83 times speed-up. However, when comparing the Equelle program to a highly optimized shallow water simulator written in CUDA, our code runs 110 times slower. We therefore see that at the current stage, our CUDA back-end is a good alternative to the serial back-end, but it is not yet ready to compete against hand-tuned CUDA code.

In order to find the performance limiting factors, we have used a profiler to identify bottlenecks and to look at memory usage. We have shown how we have optimized and redesigned parts of our code, resulting in a 4 times more efficient simulator.

The main problem with the CUDA back-end however, seems to be in the design of the intermediate code, resulting in one CUDA kernel call for each operation used in the Equelle source code. The same design is also causing a huge amount of internal memory usage on the GPU, which should be decreased in order to achieve higher performance. We therefore suggest that the future development should be focused on implementing optimizations in the front-end, and generate CUDA kernels directly instead of the current C++ code that calls a library for each operation.

In summary we have managed to provide the first prototype of a CUDA back-end for Equelle, based on the existing front-end and intermediate code design. We have seen that we are able to achieve performance far better than what is available on the CPU, and this back-end is therefore an attractive alternative to the serial back-end when it comes to implementing more test simulators in Equelle.

Appendix A

Build Instructions

We will here go through the main required steps for downloading the Equelle compiler and its back-ends, and running the simulators and tests shown in this thesis. The first thing to do is to visit the Equelle project's Github pages [16], where the wiki found at

`https://github.com/sintefmath/equelle/wiki`

states the requirements needed for compiling the front-end and the serial back-end. The wiki is written for users on the operating system 12.04. The compiler has to be downloaded as source code and compiled by the user.

Since the build instructions are clearly written on the Github pages, we will not go through every step here. The most important steps are the following:

- Install the necessary programs and dependencies for the serial back-end.
- Clone the repository from Github, using

  ```
  git clone https://github.com/sintefmath/equelle.git
  ```

- Do the out-of-source build as recommended, using CMake to generate the make file.

In order to build the CUDA back-end, follow the instructions found on

`https://github.com/sintefmath/equelle/wiki/CUDA-backend`

First of all, the user needs a CUDA enabled GPU, and to install CUDA as described on Nvidia's webpages[1]. The Thrust and cuSPARSE libraries used by the CUDA back-end are included in the CUDA software package, but Cusp has to be downloaded separately, and is available through Github as well[2].

In order to set up CMake correctly with the CUDA compiler `nvcc`, CMake version 2.8.9 is required. This is because the CMake script `FindCUDA.cmake` was updated by that version, and we have provided the CMake script on Github (`equelle/extras/FindCUDA.cmake`) as it may be hard to find the correct subversion of CMake.

Create a separate out-of-source folder, and build the CUDA back-end by the following command:

[1]https://developer.nvidia.com/cuda-downloads

[2]`git clone https://github.com/cusplibrary/cusplibrary.git`

```
cmake -DEQUELLE_BUILD_CUDA=ON ../equelle
make
```

The CUDA back-end should then be ready for use.

In order to run the unit tests to ensure correctness of the library, run

```
make test
```

Building a simulator

The simulators used in this thesis can be found under

```
equelle/examples/simulators/
```

with sub-folders containing Equelle source code, Matlab pre- and post-processing scripts and CMake scripts for the second compile stage. An Equelle program is compiled in-source as follows, by assuming we are in a sub-folder of `simulators/`,

```
$ {path_to_build}/compiler/ec -i mySim.equelle --backend=cuda > mySim.cpp
cmake -DEquelle_DIR={path_to_build} .
make
```

The first line represents the front-end, and writes the C++ intermediate code to `mySim.cpp`, while the two others are for compiling and linking the C++ program to the back-end.

Final Version

The Equelle language is in continuous development, and it is likely that there will be made changes in the source code we have discussed in this thesis. This can be optimizations, restructuring, expansion of functionality etc. In order to be able to download the version that we have described through this thesis, we have made a git tag at the final commit done through this thesis work.

The code at the time of completion of this thesis can be downloaded in two way. The first way is to follow the steps above by cloning the git repository, then checkout the following tag:

```
git checkout tags/version_20140609
```

All files will now be in the same state as they were on 9th of June 2014. Note that in order to use the compiler from this state, the software needs to be recompiled by

```
cd {path_to_build}
make
```

The second way is to visit the Equelle Github page[3], press "release" in the bar above the directory contents, and download `version_20140609` as a `.zip` or `.tar.gz`.

[3]https://github.com/sintefmath/equelle

Table A.1: The git commit hash-keys representing each optimization step for explicit heat equation. Version number is as follows: 1) First prototype before profiling, 2) Remove computations of transpose matrices, 3) Using our own kernels instead of sparse matrix-vector products, 4) Lazy evaluation of grid subsets, 5) Lazy evaluation of `Gradient` and `Divergence` matrix operators.

Version number	Commit hash-key
1	be557bfc64f0aff1e2a730f44f93e223eded6c0e
2	07d40eea08ab1e4245c77806dd81629782209913
3	3c1f1cf9857e6bbdf43056a5c6b8d4c8a3a72fa1
4	fdec65787409b1e89d3668f3ef79a78b301a36e2
5	5702e388577cb7a1db9edc57c5ca9b103c095c3c

Reproducibility

All performance tests and numerical results presented in this thesis can be reproduced by using the code from the final commit and the tag described above. The exception is the step-by-step optimization procedure done for the explicit heat equation.

In order to build and run the different steps in the optimization procedure we went through in Section 6.1, it is possible to go back in the git commit history and retrieve the code at the state of each test. The following command turns the code into the state of a specific commit:

```
git checkout <commit>
```

where `<commit>` is the hash-key representing the commit we want. It is often enough to only use the first 6-8 symbols of the hash-key in order to checkout the correct commit.

Table A.1 lists the commit hash-keys associated to the optimization steps we went through in Section 6.1, where the version number corresponds to the ones given in Figure 6.1.

Documentation

We have added documentation readable by the documentation generator Doxygen. In order to generate the documentation run the following commands, assuming you are standing in the base folder, `equelle`.

```
$ cd backends/cuda
$ doxygen doxyconfig
```

This should generate a new folder named `docs`, containing two more sub-folders, `html` and `latex`. In order to browse the documentation in a html reader, open the file `docs/html/index.html` in a web browser. Enter the tab *Namespaces* and *equelleCUDA* to get to the list of classes and other namespaces discussed in this thesis.

In order to create a pdf document with the entire documentation for the CUDA back-end, do the following:

```
$ cd docs/latex/
$ make
```

This should create the file `refman.pdf` which contains the entire documentation.

Appendix B

Test Cases

We will use this appendix chapter to contain both numerical methods of the test cases, as well as Equelle code.

B.1 The Heat Equation

In this thesis we have used the heat equation as an example, a validator of the correctness of the CUDA back-end, and for performance measuring. We will therefore briefly give an overview of the numerical methods the Equelle simulators are based on. We will refer to the Equelle code shown in Listing B.1, which shows implicit formulation of the problem.

The heat equation is stated as

$$\frac{\partial u}{\partial t} - k\nabla^2 u = 0, \tag{B.1}$$

where $u = u(\mathbf{x}, t)$. By using $\mathbf{f}(u) = -k\nabla u$, we recognise the standard form we used in Equation (2.1). By following the same steps as in Section 2.2.1 and letting U^n be the approximation of $u(\mathbf{x}, t_n)$ we get

$$U^{n+1} = U^n + \frac{\Delta t}{|\Omega|} k F_{\partial\Omega} \tag{B.2}$$

where the numerical flux $F_{\partial\Omega}$ is

$$F_{\partial\Omega} \approx \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \int_{\partial\Omega} \nabla u \cdot \mathbf{n} \,\mathrm{d}\gamma\,\mathrm{d}t. \tag{B.3}$$

Recall that for an implicit method, $F_{\partial\Omega}$ is calculated by using the new unknown solution, $u(\mathbf{x}, t_{n+1})$. For the explicit solution $u(\mathbf{x}, t_n)$ is used to compute $F_{\partial\Omega}$.

The first thing we do, is to assume constant flux in the time interval (t_n, t_{n+1}), and to model ∇u by a first order finite difference method. Let the cell be surrounded of K faces, and $\mathbf{n}_i$ be the unit normal of face i, $i = 1, ..., K$. The gradient across face i becomes

$$\nabla u_i = \frac{\texttt{Gradient}(u_i)}{|\texttt{Centroid}(\texttt{SecondCell}(i)) - \texttt{Centroid}(\texttt{FirstCell}(i))|} \tag{B.4}$$

The integral across the cell boundary interprets to the sum of the weighted gradients across the faces. Since a positive gradient is increasing from `FirstCell` to `SecondCell`, we do not know if this corresponds to positive or negative x, y or z direction. The `Divergence` function uses the same definition of positive direction, and because of that we get the dot product with the outward normal vector into the equation for free. Meaning that

$$F_{\partial\Omega} \approx \texttt{Divergence}\left(\nabla u \cdot |\texttt{InteriorFaces()}|\right) \tag{B.5}$$

where the |`InteriorFaces()`| is the area of each corresponding face. Observe that the $|\cdot|$ statements are independent of u, and can therefore be computed once at the beginning of the program. This, multiplied by k, is stored in the `itrans` variable at line 21. Equation (B.5) is represented in the Equelle program as `computeInteriorFlux` starting at line 33.

The simulator is implemented to use Dirichlet boundary conditions, where a user defined subset of `BoundaryFaces()`, called `dirichlet_boundary`, has constant user provided values `dirichlet_val`. We have to compute fluxes from these values, and therefore compute Equation (B.4) based on the values and centroids on the `dirichlet_boundary` and the values and centroids on the cells attached to these faces. The definition of positive direction that comes for free with the `Gradient` function has to be computed manually (`bf_sign` and `dir_sign` in lines 26 and 30), and the resulting function is `computeBoundaryFlux` starting at line 38.

For finding an implicit solution by the `NewtonSolve`, we require a function that calculates a residual function taking a solution guess as input. This function is defined in the for loop across the lines 66-69. Note how this function uses two more variables, `u0` and `dt`, from the above scope and call `computeResidual` defined on line 49. The residual used here is Equation (B.2) written as

$$r = U^{n+1} - U^{n} - \frac{\Delta t}{|\Omega|} k F_{\partial\Omega}. \tag{B.6}$$

The function `NewtonSolve` on line 71 returns the input to the `computeResidualLocal` such that the function becomes zero.

Explicit Solution

In order to get an explicit simulator, we need to modify the Equelle program in Listing B.1 slightly. We can omit the call to `NewtonSolve`, and instead put a modified version of the `computeResidual` function inside the for loop. Instead of finding the residual we find the next solution at once using Equation (B.2) directly.

The explicit formulation also requires sufficiently small time steps in order to give us stable solutions. Let Δx, Δy and Δz denote the size in each direction of the cells in the grid, and Δt be the time step. The Courant-Friedrichs-Lewy (CFL) condition states that in order to get a stable explicit solution of Equation (5.1), we need to choose Δt such that

$$\frac{1}{2} > \max\left\{\frac{k\Delta t}{\Delta x^2}, \frac{k\Delta t}{\Delta y^2}, \frac{k\Delta t}{\Delta z^2}\right\} \tag{B.7}$$

is satisfied. For unstructured grids, it might not be as easy to find a measurement to match Δx, Δy and Δz, and some conservative educated guess has to be made.

The Equelle Code

Listing B.1 shows the Equelle implementation of the above method, while Listing B.3 shows the intermediate code for the CUDA back-end produced by the Equelle front-end. Listing B.2 shows the changes needed in Listing B.1 in order to get the explicit formulation.

Listing B.1: Implicit finite volume method for solving the heat equation written in Equelle.

```
# Heat conduction with no boundary conditions or source terms.

# Physics that requires specification
k : Scalar = InputScalarWithDefault("k", 0.3) # Heat diffusion constant.

# u_initial should be given initial values (in user input)
u_initial : Collection Of Scalar On AllCells()
u_initial = InputCollectionOfScalar("u_initial", AllCells())

# Trying to add support for Dirichlet boundaries
dirichlet_boundary : Collection Of Face Subset Of (BoundaryFaces())
dirichlet_boundary = InputDomainSubsetOf("dirichlet_boundary", BoundaryFaces())
dirichlet_val : Collection Of Scalar On dirichlet_boundary
dirichlet_val = InputCollectionOfScalar("dirichlet_val", dirichlet_boundary)

# Compute interior transmissibilities.
vol = |AllCells()|                        # Deduced type: Collection Of Scalar On AllCells()
interior_faces = InteriorFaces()          # Deduced type: Collection Of Face
first = FirstCell(interior_faces)         # Deduced type: Collection Of Cell On interior_faces
second = SecondCell(interior_faces)       # Deduced type: Same as for 'first'.
itrans : Collection Of Scalar On interior_faces = k * |interior_faces| / |Centroid(first) - Centroid(second)|

# Compute boundary transmissibilities.
bf = BoundaryFaces()
bf_cells = IsEmpty(FirstCell(bf)) ? SecondCell(bf) : FirstCell(bf)
bf_sign = IsEmpty(FirstCell(bf)) ? (-1 Extend bf) : (1 Extend bf)
btrans = k * |bf| / |Centroid(bf) - Centroid(bf_cells)|

# Compute quantities needed for boundary conditions.
dir_sign = bf_sign On dirichlet_boundary

# Compute flux for interior faces.
computeInteriorFlux : Function(u : Collection Of Scalar On AllCells()) -> Collection Of Scalar On InteriorFaces()
computeInteriorFlux(u) = {
    -> -itrans * Gradient(u)
}

# Compute flux for boundary faces.
computeBoundaryFlux : Function(u : Collection Of Scalar On AllCells()) -> Collection Of Scalar On BoundaryFaces()
computeBoundaryFlux(u) = {
    # Compute flux at Dirichlet boundaries.
    u_dirbdycells = u On (bf_cells On dirichlet_boundary)
    dir_fluxes = (btrans On dirichlet_boundary) * dir_sign * (u_dirbdycells - dirichlet_val)
    # Extending with zero away from Dirichlet boundaries (i.e. assuming no-flow elsewhere).
    -> dir_fluxes Extend BoundaryFaces()
}

# Compute the residual for the heat equation.
computeResidual : Function(u : Collection Of Scalar On AllCells(), u0: Collection Of Scalar On AllCells(), dt : Scalar) -> Collection Of Scalar On AllCells()
computeResidual(u, u0, dt) = {
    ifluxes = computeInteriorFlux(u)
    bfluxes = computeBoundaryFlux(u)
    # Extend both ifluxes and bfluxes to AllFaces() and add to get all fluxes.
    fluxes = (ifluxes Extend AllFaces()) + (bfluxes Extend AllFaces())
    # Deduced type: Collection Of Scalar On AllCells()
    residual = u - u0 + (dt / vol) * Divergence(fluxes)
    -> residual
}

timesteps : Sequence Of Scalar
timesteps = InputSequenceOfScalar("timesteps")
u0 : Mutable Collection Of Scalar On AllCells()
u0 = u_initial

For dt In timesteps {
    computeResidualLocal : Function(u : Collection Of Scalar On AllCells()) ->
        Collection Of Scalar On AllCells()
    computeResidualLocal(u) = {
        -> computeResidual(u, u0, dt)
    }
    u_guess = u0
    u = NewtonSolve(computeResidualLocal, u_guess)
    Output("u", u)
    Output("maximum of u", MaxReduce(u))
    u0 = u
}
```

Listing B.2: Explicit finite volume method for solving the heat equation written in Equelle. Note that this listing starts at line 48, as the first part is identical to the implicit version.

```
48 ##
49 ## Above code is omitted as it is identical with the
50 ## implicit program.
51 ##
52
53 expU : Mutable Collection Of Scalar On AllCells()
54 expU = u0
55 For dt In timesteps
56 {
57     ifluxes = computeInteriorFlux(expU)
58     bfluxes = computeBoundaryFlux(expU)
59     # Extend both ifluxes and bfluxes to AllFaces() and add to get all fluxes.
60     fluxes = (ifluxes Extend AllFaces()) + (bfluxes Extend AllFaces())
61     # Deduced type: Collection Of Scalar On AllCells()
62     expU = expU - (dt / vol) * Divergence(fluxes)
63     Output("expU", expU)
64     Output("maximum of u", MaxReduce(expU))
65 }
66 Output("expU", expU)
```

Intermediate Code

The intermediate code for the implicit heat equation simulator follows in Listing B.3.

Listing B.3: Intermediate code for the implicit heat equation produced by the Equelle front-end for the CUDA back-end

```

// This program was created by the Equelle compiler from SINTEF.

#include <opm/core/utility/parameters/ParameterGroup.hpp>
#include <opm/core/utility/ErrorMacros.hpp>
#include <opm/core/grid.h>
#include <opm/core/grid/GridManager.hpp>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <cmath>
```

```
#include <array>

#include "EquelleRuntimeCUDA.hpp"

void ensureRequirements(const EquelleRuntimeCUDA& er);
void equelleGeneratedCode(equelleCUDA::EquelleRuntimeCUDA& er);

#ifndef EQUELLE_NO_MAIN
int main(int argc, char** argv)
{
    // Get user parameters.
    Opm::parameter::ParameterGroup param(argc, argv, false);

    // Create the Equelle runtime.
    equelleCUDA::EquelleRuntimeCUDA er(param);
    equelleGeneratedCode(er);
    return 0;
}
#endif // EQUELLE_NO_MAIN

void equelleGeneratedCode(equelleCUDA::EquelleRuntimeCUDA& er) {
    using namespace equelleCUDA;
    ensureRequirements(er);

    // ============= Generated code starts here ================

    const Scalar k = er.inputScalarWithDefault("k", double(0.3));
    const CollOfScalar u_initial = er.inputCollectionOfScalar("u_initial",
        er.allCells());
    const CollOfFace dirichlet_boundary =
        er.inputDomainSubsetOf("dirichlet_boundary", er.boundaryFaces());
    const CollOfScalar dirichlet_val = er.inputCollectionOfScalar("dirichlet_val",
        dirichlet_boundary);
    const CollOfScalar vol = er.norm(er.allCells());
    const CollOfFace interior_faces = er.interiorFaces();
    const CollOfCell first = er.firstCell(interior_faces);
    const CollOfCell second = er.secondCell(interior_faces);
    const CollOfScalar itrans = (k * (er.norm(interior_faces) /
        er.norm((er.centroid(first) - er.centroid(second)))));
    const CollOfFace bf = er.boundaryFaces();
    const CollOfCell bf_cells = er.trinaryIf(er.isEmpty(er.firstCell(bf)),
        er.secondCell(bf), er.firstCell(bf));
    const CollOfScalar bf_sign = er.trinaryIf(er.isEmpty(er.firstCell(bf)),
        er.operatorExtend(-double(1), bf), er.operatorExtend(double(1), bf));
    const CollOfScalar btrans = (k * (er.norm(bf) / er.norm((er.centroid(bf) -
        er.centroid(bf_cells)))));
    const CollOfScalar dir_sign = er.operatorOn(bf_sign, er.boundaryFaces(),
        dirichlet_boundary);
    std::function<CollOfScalar(const CollOfScalar&)> computeInteriorFlux = [&](const
        CollOfScalar& u) -> CollOfScalar {
        return (-itrans * er.gradient(u));
    };
    std::function<CollOfScalar(const CollOfScalar&)> computeBoundaryFlux = [&](const
        CollOfScalar& u) -> CollOfScalar {
        const CollOfScalar u_dirbdycells = er.operatorOn(u, er.allCells(),
            er.operatorOn(bf_cells, er.boundaryFaces(), dirichlet_boundary));
        const CollOfScalar dir_fluxes = ((er.operatorOn(btrans, er.boundaryFaces(),
            dirichlet_boundary) * dir_sign) * (u_dirbdycells - dirichlet_val));
        return er.operatorExtend(dir_fluxes, dirichlet_boundary, er.boundaryFaces());
    };
    std::function<CollOfScalar(const CollOfScalar&, const CollOfScalar&, const
        Scalar&)> computeResidual = [&](const CollOfScalar& u, const CollOfScalar&
        u0, const Scalar& dt) -> CollOfScalar {
        const CollOfScalar ifluxes = computeInteriorFlux(u);
        const CollOfScalar bfluxes = computeBoundaryFlux(u);
        const CollOfScalar fluxes = (er.operatorExtend(ifluxes, er.interiorFaces(),
            er.allFaces()) + er.operatorExtend(bfluxes, er.boundaryFaces(),
            er.allFaces()));
        const CollOfScalar residual = ((u - u0) + ((dt / vol) *
            er.divergence(fluxes)));
        return residual;
    };
    const SeqOfScalar timesteps = er.inputSequenceOfScalar("timesteps");
    CollOfScalar u0;
    u0 = u_initial;
    for (const Scalar& dt : timesteps) {
        std::function<CollOfScalar(const CollOfScalar&)> computeResidualLocal =
            [&](const CollOfScalar& u) -> CollOfScalar {
            return computeResidual(u, u0, dt);
        };
        const CollOfScalar u_guess = u0;
        const CollOfScalar u = er.newtonSolve(computeResidualLocal, u_guess);
        er.output("u", u);
        er.output("maximum of u", er.maxReduce(u));
        u0 = u;
    }

    // ============ Generated code ends here ================

}

void ensureRequirements(const EquelleRuntimeCUDA& er)
{
    (void)er;
}
```

B.2 The Shallow Water Equations

We will here give a short presentation of the numerical scheme on which the Equelle simulator for the shallow water equations is written. The simulator was written as part of the Specialization Project[20], and a more thorough presentation of the method can be found there. The method is based on the second order scheme presented by Kurganov and Levy [22] but simplified to a first order scheme suitable for Equelle.

The equations model the water depth h and the water velocities u and v in x and y direction respectively, when the free surface is only under the influence of gravity g. We use source terms to account for bottom topography B. The equations are stated as

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix}. \tag{B.8}$$

The numerical method is based on considering the conserved variables $q = [q_1, q_2, q_3]^T$, where $q_1 = \omega := h + B$, $q_2 = hu$ and $q_3 = hv$. The variable ω becomes the surface elevation. These variables are inserted in Equation (B.8), and written in vector form

$$q_t + f(q)_x + g(q)_y = h_B(q). \tag{B.9}$$

We consider a general domain $\Omega \in \mathbb{R}^2$ discretised in a regular recangular grid with cells $\Omega_{j,k}$ defined by

$$\Omega_{j,k} = \{(x, y) \in \mathbb{R}^2, x_{j-1/2} \leq x \leq x_{j+1/2}, y_{k-1/2} \leq y \leq y_{k+1/2}\}. \tag{B.10}$$

We use Δx and Δy as width and height for the cells respectively.

The method is written as in Section 2.2.1, where Equation (B.9) is integrated over $\Omega_{j,k}$ while the divergence theorem is applied on the flux terms. This leads to

$$\frac{\partial}{\partial t}\int_{\Omega_{j,k}} q \, d\Omega_{j,k} + \int_{\partial\Omega_{j,k}} [f \; g] \cdot \mathbf{n} \, d\gamma = \int_{\Omega_{j,k}} h_B \, d\Omega_{j,k}. \tag{B.11}$$

Using the rectangular domain, we can express the line integral on the cell boundary as

$$\begin{aligned}\int_{\partial\Omega_{j,k}} [f\ g]\,\mathrm{d}\gamma = &\int_{y_{k-1/2}}^{y_{k+1/2}} \left[f(q(x_{j+1/2},y)) - f(q(x_{j-1/2},y))\right]\,\mathrm{d}y \\ &+ \int_{x_{j-1/2}}^{x_{j+1/2}} \left[g(q(x,y_{k+1/2})) - g(q(x,y_{k-1/2}))\right]\,\mathrm{d}x.\end{aligned} \tag{B.12}$$

Approximate values for q, h_B, and the flux terms are then defined as

$$Q_{j,k} \approx \frac{1}{\Delta x \Delta y}\int_{\Omega_{j,k}} q(x,y)\,\mathrm{d}\Omega, \tag{B.13}$$

$$F_{j\pm1/2,k} \approx \frac{1}{\Delta y}\int_{y_{k-1/2}}^{y_{k+1/2}} f(q(x_{j\pm1/2},y))\,\mathrm{d}y, \tag{B.14}$$

$$G_{j,k\pm1/2} \approx \frac{1}{\Delta x}\int_{x_{j-1/2}}^{x_{j+1/2}} g(q(x,y_{k\pm1/2}))\,\mathrm{d}x, \tag{B.15}$$

$$S_{j,k} \approx \frac{1}{\Delta x \Delta y}\int_{\Omega_{j,k}} h_B(x,y)\,\mathrm{d}\Omega. \tag{B.16}$$

Equation (B.11) then becomes

$$\frac{\partial}{\partial t}Q_{j,k} + \frac{F_{j+1/2,k} - F_{j-1/2,k}}{\Delta x} + \frac{G_{j,k+1/2} - G_{j,k-1/2}}{\Delta y} = S_{j,k} \tag{B.17}$$

We then use a central-upwind scheme using the following numerical fluxes

$$F_{j+1/2,k} = \frac{a^+_{j+1/2,k} f(Q_{j,k}) - a^-_{j+1/2,k} f(Q_{j+1,k})}{a^+_{j+1/2,k} - a^-_{j+1/2,k}} + \frac{a^+_{j+1/2,k} a^-_{j+1/2,k}}{a^+_{j+1/2,k} - a^-_{j+1/2,k}}\left(Q_{j+1,k} - Q_{j,k}\right) \tag{B.18}$$

$$G_{j,k+1/2} = \frac{b^+_{j,k+1/2} g(Q_{j,k}) - b^-_{j,k+1/2} g(Q_{j,k+1})}{b^+_{j,k+1/2} - b^-_{j,k+1/2}} + \frac{b^+_{j,k+1/2} b^-_{j,k+1/2}}{b^+_{j,k+1/2} - b^-_{j,k+1/2}}\left(Q_{j,k+1} - Q_{j,k}\right) \tag{B.19}$$

where

$$a^+_{j+1/2,k} = \max\left\{\lambda_3\left(\frac{\partial f}{\partial q}(Q_{j+1,k})\right), \lambda_3\left(\frac{\partial f}{\partial q}(Q_{j,k})\right), 0\right\} \tag{B.20}$$

$$a^-_{j+1/2,k} = \min\left\{\lambda_1\left(\frac{\partial f}{\partial q}(Q_{j+1,k})\right), \lambda_1\left(\frac{\partial f}{\partial q}(Q_{j,k})\right), 0\right\} \tag{B.21}$$

$$b^+_{j,k+1/2} = \max\left\{\lambda_3\left(\frac{\partial g}{\partial q}(Q_{j,k+1})\right), \lambda_3\left(\frac{\partial g}{\partial q}(Q_{j,k})\right), 0\right\} \tag{B.22}$$

$$b^-_{j,k+1/2} = \min\left\{\lambda_1\left(\frac{\partial g}{\partial q}(Q_{j,k+1})\right), \lambda_1\left(\frac{\partial g}{\partial q}(Q_{j,k})\right), 0\right\} \tag{B.23}$$

Here, $\lambda(\cdot)$ denotes the eigenvalues of the Jacobian matrices of their argument, such that $\lambda_1 \leq \lambda_2 \leq \lambda_3$. The eigenvalues we need are

$$\lambda_1\left(\frac{\partial f}{\partial q}\right) = u - \sqrt{g(\omega - B)}, \qquad \lambda_3\left(\frac{\partial f}{\partial q}\right) = u + \sqrt{g(\omega - B)}, \tag{B.24}$$

$$\lambda_1\left(\frac{\partial g}{\partial q}\right) = v - \sqrt{g(\omega - B)}, \qquad \lambda_3\left(\frac{\partial g}{\partial q}\right) = v + \sqrt{g(\omega - B)}. \tag{B.25}$$

The source term is treated as $S_{j,k} = [0, S^{(2)}_{j,k}, S^{(3)}_{j,k}]^T$, where

$$\begin{aligned} S^{(2)}_{j,k} &= g\frac{B(x_{j+1/2}, y_k) - B(x_{j-1/2}, y_k)}{\Delta x} \cdot \frac{(\omega_{j,k} - B(x_{j+1/2}, y_k)) + (\omega_{j,k} - B(x_{j-1/2}, y_k))}{2}, \\ S^{(3)}_{j,k} &= g\frac{B(x_j, y_{k+1/2}) - B(x_j, y_{k-1/2})}{\Delta y} \cdot \frac{(\omega_{j,k} - B(x_j, y_{k+1/2})) + (\omega_{j,k} - B(x_j, y_{k-1/2}))}{2}. \end{aligned} \tag{B.26}$$

It should be said that we read the bottom topography by its values on the faces of the grid. The surface elevation $\omega = h + B$ is then calculated by using the mean value of each cell's face values.

We now have all terms for Equation (B.17) such that it is in semi-discrete form. We solve by the time derivative by using a second-order stability preserving Runge-Kutta method as used in [8]. Denote time step n of the solution in cell $\Omega_{j,k}$ as $Q^n_{j,k}$. We reorganize Equation (B.17) and denote the new right hand side as $R(Q)_{j,k}$ as

$$\frac{\partial}{\partial t}Q_{j,k} = -\frac{F_{j+1/2,k} - F_{j-1/2,k}}{\Delta x} - \frac{G_{j,k+1/2} - G_{j,k-1/2}}{\Delta y} + S_{j,k} =: R(Q)_{j,k}. \tag{B.27}$$

The Runge-Kutta method is then given as

$$\begin{aligned} Q^*_{j,k} &= Q^n_{j,k} + \Delta t R(Q^n)_{j,k} \\ Q^{n+1}_{j,k} &= \frac{1}{2}Q^n_{j,k} + \frac{1}{2}\left[Q^*_{j,k} + \Delta t R(Q^*)_{j,k}\right], \end{aligned} \tag{B.28}$$

which solves the shallow water equations in cell $\Omega_{j,k}$ for time step $n + 1$.

Bibliography

[1] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Visited March 24th 2014.

[2] cuSPARSE User Guide. http://docs.nvidia.com/cuda/cusparse/index.html. Visited April 29th 2014.

[3] Nvidia Tesla GPU Accelerators. http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf. Visited June 2nd 2014.

[4] Thrust QuickStart Guide. http://docs.nvidia.com/cuda/thrust/index.html. Visited April 29th 2014.

[5] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.

[6] Nathan Bell and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0.

[7] A. R. Brodtkorb. A MATLAB Interface for the GPU. Master's thesis, University of Oslo, 2008.

[8] A. R. Brodtkorb. *Scientific Computing on Heterogeneous Architectures*. PhD thesis, University of Oslo, 2010.

[9] A. R. Brodtkorb and T. R. Hagen. A Comparison of Three Commodity-Level Parallel Architectures: Multi-core CPU, Cell BE and GPU. *Mathematical Methods for Curves and Surfaces : 7th International Conference*, 2008.

[10] 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.

[11] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M .Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. *Proceedings of the ACM international conference on Object oriented programming systems languages and application*, 2010.

[12] A.J.C. de Saint-Venant. Théorie du mouvement non-permanent des eaux, avec application aux crues des riviéres et á l'introduction des marées dans leur lit. *C. R. Acad. Sci Paris*, pages 147–154, 1871.

[13] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[14] D. A. Di Pietro, J-M. Gratien, and C. Prud'Homme. A Domain-Specific Embedded Language in C++ for Lowest-Order Discretizations of Diffusive Problems on General Meshes. *BIT Numerical Mathematics*, 2012.

[15] Equelle. www.equelle.org.

[16] Equelle. https://github.com/sintefmath/equelle.

[17] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *Transactions on Computing*, C-21(9), 1972.

[18] A. Griewank and A. Walther. *Evaluating Derivatives - Principles and Techniques of Algorithmic Differnetiation*. SIAM, 2nd edition, 2008.

[19] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[20] H. H. Holm. Domain-Specific Languages for Numerical PDEs. Technical report, NTNU - Norwegian University of Science and Technology, 2013.

[21] P. Kogge, K. Bergman, S. Borkar, D. Campbell andW. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA IPTO, Tech. Rep.*, 2008.

[22] A. Kurganov and D. Levy. Central-Upwind Schemes for the Saint-Venant System. *Mathematical Modelling and Numerical Analysis*, 36:397–425, 2002.

[23] E. S. Larsen and D McAllister. Fast Matrix Multiplis using Graphics Hardware. *Supercomputing*, 2001.

[24] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2004.

[25] P. S. Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011.

[26] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *SIGGRAPH*, 2012.

[27] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *PLDI*, 2013.

[28] L. B. Rall. *Automatic Differentiation: Techniques and Applications*. Springer-Verlag, 1981.

[29] A. F. Rasmussen. Equelle Reference Manual. https://github.com/sintefmath/equelle/tree/master/doc/.

[30] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Siam, 2nd edition, 2003.

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

[32] The Open Porous Media Initiative. http://opm-project.org/.

[33] H. A. van der Vorst. Bi-CGStab: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. and Stat. Comput.*, 13(2), 1992.

[34] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26-36, 2000.