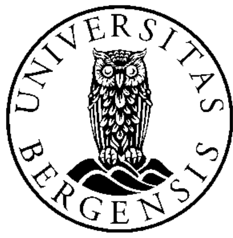

Høgskulen
på Vestlandet

Naive and adapted utilization of the GPU for general purpose computing

Alexander Mjøs

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

September 2017

Preface

This is a master thesis in the joint master programme in software engineering at Western Norway University of Applied Sciences (earlier Bergen University Collage) and University of Bergen from January 2016 to September 2017. The thesis is performed at the Engineering computing research group with Jon Eivind Vatne and Talal Rahman as supervisors. We assume that the reader has a basic understanding for computer science and mathematics, however, the mathematics for the Shallow Water Equation in Chapter 4 is considered difficult for a computer scientist. The recent flood in the USA and tsunamis in Greenland, Greece and Japan indicates that simulating The Shallow Water Equations is highly relevant today.

Bergen, 2017-09-11

Alexander Mjøs

Acknowledgment

I would first like to thank my thesis advisors, Associate Professor Jon Eivind Vatne and Talal Rahman of the Department of Computing, Mathematics and Physics at Western University of Applied Sciences. Their offices were always open whenever I ran into a trouble spot or had a question about my research or writing. They consistently allowed this paper to be my own work, but steered me in the right direction whenever they thought I needed it.

Finally, I must express my very profound gratitude to my parents and to my partner for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

A.M.

Summary and Conclusions

In this thesis we will see that adaptive utilization of the graphical processing unit (GPU) has much better performance than naive utilization, but it takes longer time to implement and requires more knowledge about GPU-programming. To choose which strategy to use for utilizing the GPU depends on how we weigh implementation time against performance. If we want to get results fast and the size of the problem is not too large (or too small where CPU is more suitable), the naive utilization of the GPU is probably the best choice. If we want high performance, use large dataset or want a more sophisticated solution in terms of graphics and functionality, the adapted utilization is probably the best choice. This conclusion is based on implementation of numerical solutions for two partial differential equations (PDEs), The Heat Equation and The Shallow Water Equations. These solutions were implemented to utilize the power of the GPU to do the actual approximation of the equations. The power of the GPU can be utilized in different ways. In this thesis we want to implement the numerical solutions to utilize the GPU in both a naive way and an adaptive way. In the naive way, we use standard libraries where we can apply code written for the central processing unit (CPU) to the GPU without any knowledge about how it is executed. In the adaptive way, we have to specify every little detail about setting up the connection and communication with the GPU, how memory is used on the GPU and how to write code on the GPU. For each implementation, we measure running time with different problem sizes to see how each solution performed.

Contents

Preface i

Acknowledgment ii

Summary and Conclusions iii

1 Introduction **1**

1.1 Background 1

1.2 Previous related work 1

1.3 Goal of thesis 2

1.4 Research questions 2

1.5 Objectives 2

1.6 Limitations 3

1.7 Approach 3

1.8 Structure of the report 3

2 Technologies **5**

2.1 MATLAB 5

2.2 MATLAB Parallel Computing Toolbox 5

2.3 C++ 6

2.4 OpenCL 6

2.5 OpenGL 6

2.6 Betelgeuse 7

2.7 Desktop 7

3 The Heat Equation **8**

3.1 Overview 8
3.2 1D heat equation 8
3.3 Stepping up to 2D 11
3.4 Naive implementation 14
3.4.1 MATLAB CPU code 14
3.4.2 Converting to GPU code 16
3.5 Adaptive implementation 17
3.5.1 C++ code 18
3.5.2 OpenCL code 19
3.6 Complexity 22
3.7 Comparison of implementations 22
3.7.1 MATLAB 22
3.7.2 C++/OpenCL/OpenGL 22
3.8 Efficiency 23
3.8.1 Efficiency of computation, MATLAB 23
3.8.2 Efficiency of computation, C++/OpenCL 26
3.8.3 Efficiency of visualization, MATLAB 27
3.8.4 Efficiency of visualization, C++/OpenCL/OpenGL 29
3.9 Analysis 30

4 Shallow Water **32**
4.1 Overview 32
4.2 Discretization 33
4.3 Boundary conditions 36
4.3.1 Reflective boundaries 37
4.3.2 Periodic boundaries 37
4.3.3 Free boundaries 37
4.4 Initial values 38
4.5 Naive implementation 39
4.5.1 CPU implementation 39

4.5.2 GPU implementation 44
4.6 Adaptive implementation 45
4.6.1 C++ 45
4.6.2 OpenCL 49
4.6.3 OpenGL 51
4.7 Numerical instability 54
4.8 Complexity 57
4.9 Efficiency, computations 57
4.9.1 MATLAB CPU 57
4.9.2 MATLAB GPU 58
4.9.3 C++/OpenCL 59
4.10 Efficiency, visualization 60
4.10.1 MATLAB CPU 60
4.10.2 MATLAB GPU 61
4.10.3 C++/OpenCL/OpenGL 61
4.11 Analysis 64
4.12 Quality and possibilities 64
4.12.1 MATLAB 65
4.12.2 OpenGL 68

5 Summary **72**
5.1 Summary and Conclusions 72
5.2 Recommendations for Further Work 76

A Acronyms **77**

B Additional Information **79**
B.1 Chapter 3 - The Heat Equation 79
B.1.1 Computations 79
B.1.2 Visualization 86
B.2 Chapter 4 - The Shallow Water Equations 92
B.2.1 Computations 92

B.2.2 Visualization 98

Bibliography **104**

Chapter 1

Introduction

1.1 Background

The increase in computational power of Graphics Processing Units (GPUs) in recent years allows much better simulations in computer graphics, for instance for modeling fluid motion (e.g. water, smoke, fire). There are standard libraries for utilizing this power called naive utilization which require little or no knowledge about programming on GPUs, but to fully exploit the GPU for a given task, an implementation adapted to the task will probably perform better.

1.2 Previous related work

We have not succeeded to find any previous related work which compare naive and adapted utilization of the GPU. However, there has been done research on comparing GPU versus CPU on The Shallow Water Equations. Visual simulation of shallow-water waves, a paper published by SINTEF [1] they claim the following:
Modern numerical schemes for such models are inherently parallel in the sense that very little global communication is needed in the computational domain to advance the solution forward in time. Therefore, this application can readily exploit the parallel architecture of modern GPUs. We have seen in practical computations that moving from a serial CPU-based implementation to an implementation on a modern GPU decreases the runtime by more than one order of magnitude. (The runtime of the full dambreak simulation was reduced from 2 h to 5 min). To achieve

the same speedup using CPUs, one would have to resort to a cluster of twenty or more processing nodes [1].

1.3 Goal of thesis

The goal of this thesis is to compare running time for various implementations of the same algorithms. A naive implementation using standard packages and libraries will be quicker to develop, but the end result of an adapted and more sophisticated implementation will perform better. We want to weigh the extra effort for using an adapted implementation against the possible payoff in terms of running time and visualization.

1.4 Research questions

In this thesis we will try to answer the following research questions:

- Will an adaptive implementation of fluid simulation be faster than a naive implementation? And if so, how much faster will it be?
- In what situations is doing an adaptive implementation worth the increased effort?

1.5 Objectives

In order to reach the goal of this thesis, we will look at two partial differential equations (PDE). The main objectives of this Master's project is to study these PDEs in the following order:

1. Develop one naive and one adaptive solution for the 1D heat equations.
2. Develop one naive and one adaptive solution for the 2D heat equations.
3. Develop one naive and one adaptive solution for the Shallow water equations in isolated box environment.

1.6 Limitations

In this thesis we limit our research to apply numerical approximation of partial differential equations (PDE) running on a single standalone desktop graphics card. We are also limited to testing on NVIDIA graphics card. Graphics card from other vendors, integrated graphics or multiple GPU are not represented. Naive utilization on the GPU are in this thesis limited to MATLAB and MATLAB parallel computing toolbox, and adapted utilization of the GPU are limited to C++ together with OpenCL and OpenGL.

1.7 Approach

The Partial Differential Equations (PDE) will be solved first using MATLAB utilizing the Central Processing Unit (CPU) with small data to verify the result. Next, the program will be scaled and benchmarked and optimized to ensure that the running time is good enough for a fair comparison and then added support for running on the GPU. The same algorithm will then be implemented in C++ together with OpenCL. At last, OpenGL will be integrated in the C++ solution. With this approach, we start with small data and compare the result to ensure correctness.

1.8 Structure of the report

The rest of the report is organized as follow:
Chapter 2 gives an introduction to the technologies used in the thesis.
Chapter 3 starts with an introduction to the heat equations in one and two dimensions before we move on and explain how the heat equation is discretized and implemented in the naive way and the adapted way. At the end of the chapter we take a look at the complexity and the running times for the different implementations.
In Chapter 4 there is an overview over the Shallow water equations starting with the equations itself before we take a look at the discretization, initial values and different boundary conditions. Next, we look at the naive and adapted implementation and discuss numerical instability and the complexity. At the end of the chapter we look at the running time for the different implementations and analyze what we have observed in the chapter.

Chapter 5 is the final chapter which is summary and conclusion for the thesis along with recommendations for further work.
Appendix A is a list of acronyms used in the thesis.
Appendix B contains all the running times and graphs for the implementations in Chapter 3 and 4.

Chapter 2

Technologies

This chapter gives an overview over technologies used in the thesis.

2.1 MATLAB

MATLAB (**mat**rix **lab**oratory) is a programming environment built around the MATLAB scripting language. The MATLAB platform is optimized for solving engineering and scientific problems. The matrix-based MATLAB language is a natural way to express computational mathematics. Built-in graphics make it easy to visualize and gain insights from data [2]. MATLAB is chosen because it is easy and it takes less time to implement code versus other popular programming language like Java, C or C#. This is because you don't need classes, all the code is written in script or functions for larger chunks of code. MATLAB has also the benefit of being a loosely typed programming language which means that you don't need to specify which type a variable is, for instance int, double or string.

2.2 MATLAB Parallel Computing Toolbox

MATLAB Parallel Computing Toolbox is used for utilizing the power of the GPU in a naive way. This means that the same code that runs on the CPU can be executed on the GPU by applying this toolbox. Mathworks, the company behind MATLAB describes the MATLAB Parallel Computing Toolbox as following:

Parallel Computing Toolbox lets you solve computationally and data-intensive problems using multi-core processors, GPUs, and computer clusters [3]. High-level constructs—parallel for-loops, special array types, and parallelized numerical algorithms—let you parallelize MATLAB applications without CUDA or MPI programming [3]. Parallel Computing Toolbox provides GPUArray, a special array type with several associated functions that lets you perform computations on CUDA-enabled NVIDIA GPUs directly from MATLAB [4]. Using MATLAB for GPU computing lets you accelerate your applications with GPUs more easily than by using C or Fortran [5]. With the MATLAB language you can take advantage of the CUDA GPU computing technology without having to learn the intricacies of GPU architectures or low-level GPU computing libraries [5].

2.3 C++

C++ is a high level, general-purpose object oriented programming language. It is cross platform and can therefore run on various hardware and platforms.

2.4 OpenCL

Open Computing Language (OpenCL) is an industry standard framework for programming computers composed of a combination of CPUs, GPUs, and other processors [6]. According to OpenCL Programming Guide [6], these so-called heterogeneous systems have become an important class of platforms, and OpenCL is the first industry standard that directly addresses their needs. OpenCL is cross platform and can run on various devices such as workstations, laptops, smart phones, FPGA and other hardware with support for OpenCL. No other parallel standard has such a wide reach, and that is one of the reasons why OpenCL is so important [6].

2.5 OpenGL

Open Graphics Library (OpenGL) is a cross platform Application Programming Interface (API) for developing 2D and 3D application. Together with OpenGL, freeglut is used for windows

managing and to manage keyboard input.

2.6 Betelgeuse

Server at campus with NVIDIA Tesla K40c graphics card and with remote desktop access to perform computations on a high performance GPU. This server does not provide OpenGL support, so it is only for general purpose computing on graphics processing units (GPGPU).

2.7 Desktop

Desktop computer used for visualization and GPGPU. HP workstation with NVDIA GTX760 graphics card and Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz processor. This was used for all benchmarks because of both OpenCL and OpenGL support.

Chapter 3

The Heat Equation

3.1 Overview

The heat equation is a partial differential equation (PDE) which describes the change of temperature over time. It is derived from Newton's law of cooling, which states that the rate of change of the temperature of an object is directly proportional to the difference in temperature between the object and its surroundings [7].
This has been an introductory project to my thesis, and the reason why is because it is a simple PDE, but also an important one. By using this approach it is easy to verify the result, both numerically and visually.

3.2 1D heat equation

The heat equation in normalized form for one dimension

$$u_t(x,t) = u_{xx}(x,t), \quad x \in \mathbf{R}, t > 0 \tag{3.1}$$

arises in the model of temperature evolution in uniform materials [8]. The independent variables are time, t, and one space variable, x. Here $u(x,t)$ describes the temperature at time t and position x. u_t denotes the partial derivative with respect to time, and u_{xx} the double partial derivative with respect to position. As an example, we can envision a rectangular steel plate

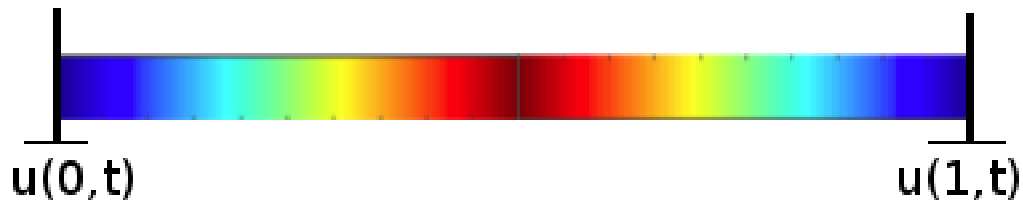

Figure 3.1: Heat equation illustrated with red as warm and blue as cold.

with fixed length of 1 as showed in Figure 3.1. We imagine that no heat leaves the plate, and the boundary conditions at the ends of the plates are that the temperatures are set to zero:

$$u(0,t) = u(1,t) = 0, \quad t > 0 \tag{3.2}$$

This method for setting the boundary conditions is called Dirichlet-type boundary, where a specific value is set on the boundaries [8]. For other applications, there are also other standardized methods for setting boundary conditions (see e.g. Tveito and Winther).
Initially, the function $f(x)$ describes the temperature at $t = 0$.

$$u(x,0) = f(x), \quad x \in (0,1) \tag{3.3}$$

In order to solve an approximation of this PDE on a computer, we derive a finite difference scheme. For the Heat Equation, an explicit scheme is used because the values in each time step can be calculated independently of each other. This makes explicit schemes suitable for parallel computations. We use the following approximation for time and space:
Forward differences for time approximation

$$u_t(x,t) = \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} + O(\Delta t) \tag{3.4}$$

and central differences for space approximation

$$u_{xx}(x,t) = \frac{u(x-\Delta x,t) - 2u(x,t) + u(x+\Delta x,t)}{\Delta x^2} + O(\Delta x^2). \tag{3.5}$$

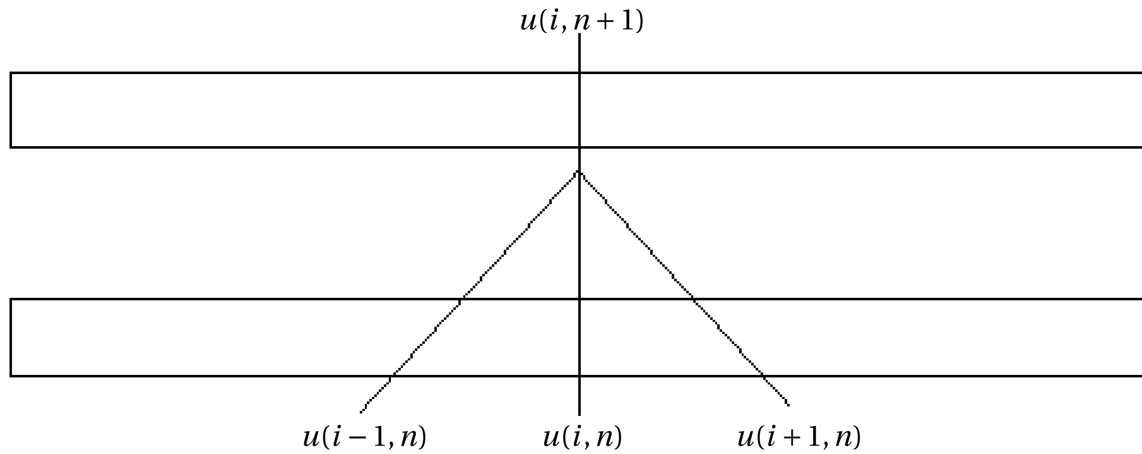

Figure 3.2: Illustrates computation for next step in the Heat equation.

which is known as FTCS (**F**orward differences for **t**ime and **c**entral differences for **s**pace) [9].
From 3.1, we get

$$\frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} \approx \frac{u(x-\Delta x,t)-2u(x,t)+u(x+\Delta x,t)}{\Delta x^2} \tag{3.6}$$

and

$$u(x,t+\Delta t) \approx u(x,t)+\Delta t\frac{u(x-\Delta x,t)-2u(x,t)+u(x+\Delta x,t)}{\Delta x^2}. \tag{3.7}$$

For $f(x)$ we choose the sinus function as initial value to get numerical results.

$$f(x) = \sin(\pi \cdot x). \tag{3.8}$$

Now, all we need is to decide Δt and Δx. The accuracy of the approximation is increased by lowering Δt [8], but it will require more computing power. We need $\Delta t \leqslant \frac{1}{2}\Delta x^2$ to get good approximations, see e.g. [8].
Since we can only do an approximation in fixed position on the computer, we define time steps and space steps as

$$u(i,n) \approx u((i-1)\cdot\Delta x,(n-1)\cdot\Delta t). \tag{3.9}$$

The equation from (3.7) can directly be translated into MATLAB code with two simple loops. As we can see from Figure 3.2, all we need to compute the temperature at the next time step in position x is the temperature at the current time step in position x together with the temperature Δx to the left and Δx to the right of position x. Here, "deltaX" is $\frac{1}{steps}$, which means that u(i-1,n)

in the code is the same as $u(x-\Delta x, t)$.

```
u = zeros(steps+1, timeSteps);
xx = 0:deltaX:1;
for i = 1 : steps+1
    u(i,1) = sin(pi*xx(i)); % Initial value
end
for n = 1:timeSteps-1
    for i = 2 : steps
        u(i,n+1) = u(i,n)+deltaT*(u(i-1,n)-2*u(i,n) + \
            u(i+1,n)) / (deltaX^2);
    end
    u(1, n) = 0;                    % Boundary conditions
    u(steps+1, n) = 0;
end
```

3.3 Stepping up to 2D

The 2D heat equation adds another independent space variable, y, and an extra part for the equation on the right hand side. For this problem, we can envision a square with leading dimension d, with fixed temperature at the edges of the square as boundary conditions.

$$u_t(x, y, t) = u_{xx}(x, y, t) + u_{yy}(x, y, t), \quad x, y \in \mathbf{R}, t > 0 \tag{3.10}$$

Central differences for y-direction is approximated in the same way as for the x-direction.

$$u_{yy}(x, y, t+\Delta t) = \frac{u(x, y-\Delta y, t) - 2u(x, y, t) + u(x, y+\Delta y, t)}{\Delta y^2} + O(\Delta y^2) \tag{3.11}$$

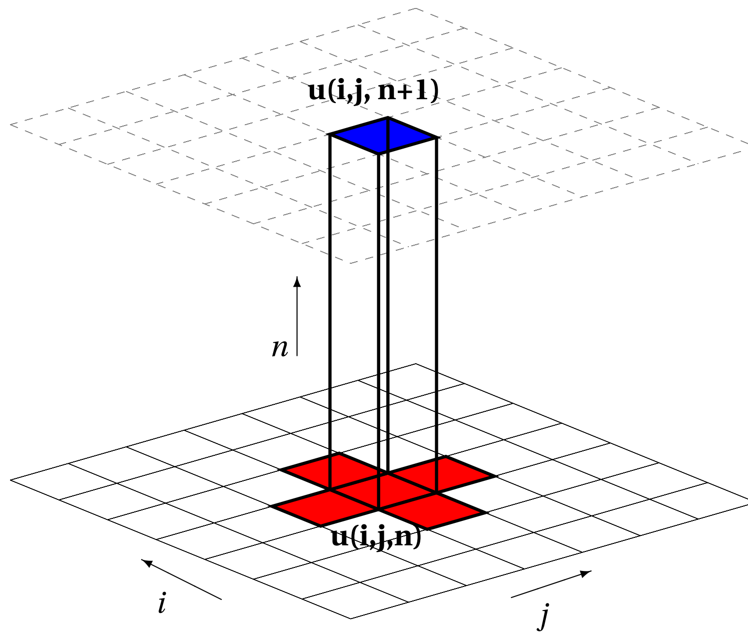

Figure 3.3: Illustrates which data is needed to calculate $u_t(x, y, t+\Delta t)$.

We choose boundary conditions with 0 temperature at the edges of the square.

$$
\begin{aligned}
u(x,0,t) &= 0, \\
u(d,0,t) &= 0, \\
u(0,y,t) &= 0, \\
u(d,y,t) &= 0, \\
x, y \in (0,d), t &> 0.
\end{aligned}
$$

For better visualization, we have added an extra boundary to the problem which is not a part of the original equation. This extra boundary is that we set the center of the square to 100°c shown in Figure 3.4.

$$u(d/2, d/2, t) = 100,$$

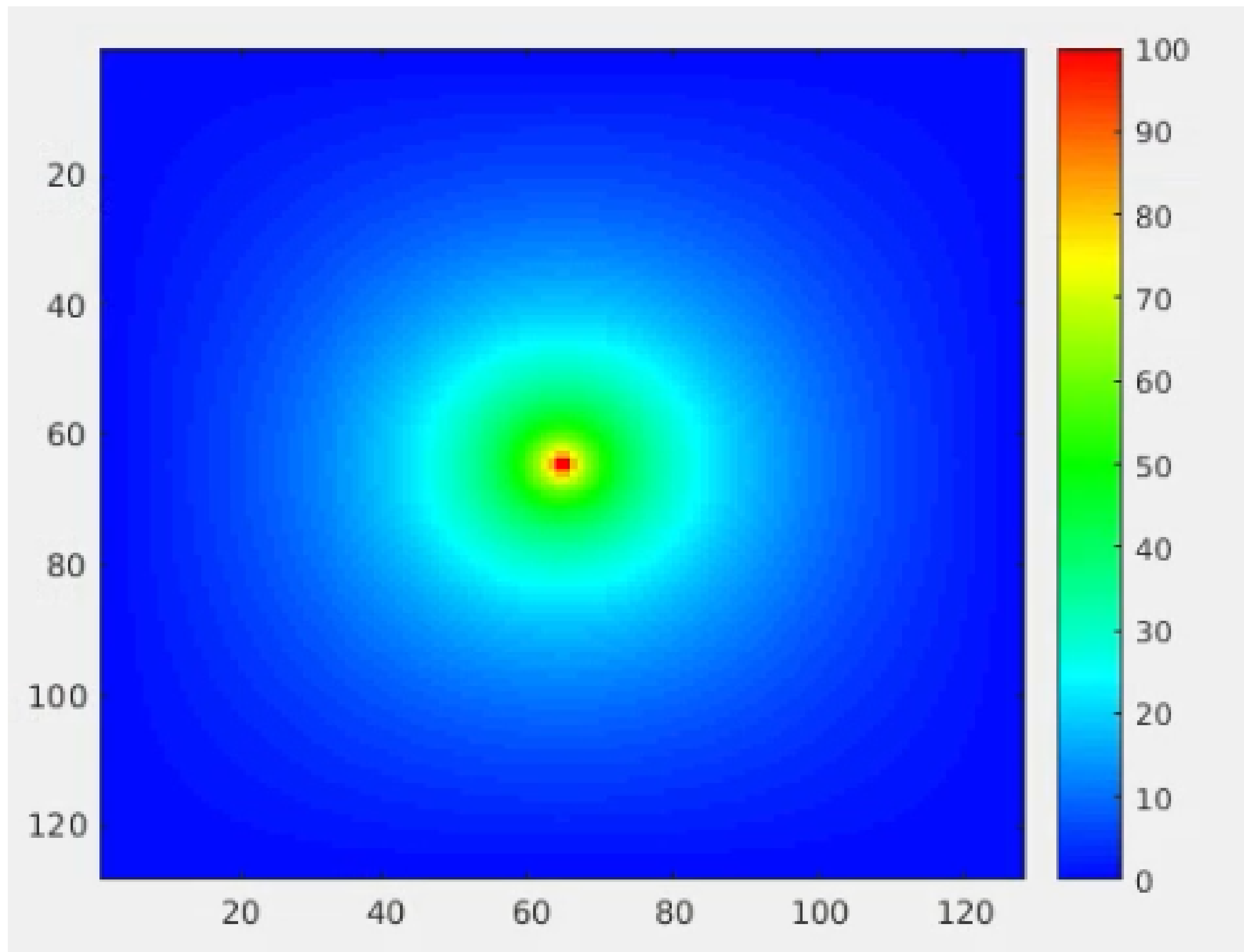

Figure 3.4: Heat Equation output from MATLAB.

3.4 Naive implementation

For the naive implementation, MATLAB is used. To get as fair comparison between naive and adaptive implementation as possible, it is important to understand how MATLAB works. MATLAB stands for MATrix LABoratory, and it is optimized for doing operations on matrices. If you implement an algorithm using loops (for, while or even worse nested loops), the code will run very slowly and you will get poor performance. It is faster to create a vector, which is basically a one dimensional matrix, and perform the calculations on this vector. This technique is called vectorization, and the computer performs operations on the vector instead of each element. Pre-allocation of memory is also important for performance, otherwise MATLAB has to reallocate new memory blocks for each iteration. Also, according to Mathworks [10], writing code in a function is faster than writing the same code as a script. With this in mind, the MATLAB code should be as good as possible to have a fair comparison versus the adaptive implementation.

3.4.1 MATLAB CPU code

This is straight forward to implement if you have the optimization from previous section in mind. In the code in Figure 3.4.1, "bound" is a function to set the boundary conditions. This is done for each time step in the algorithm. "imagesc" is used to create an image from an matrix, and the command "set" is used to update the image data. Whenever "drawnow" is called, MATLAB is refreshing the image and "limitrate nocallbacks" limits the number of updates to 20 frames per second and skips updates if the renderer is busy, according to MATLABs API. MATLABs built in profiler is used to run and analyze the code to ensure that the code is as efficient as possible. In this project, we specified the mapping for the color scheme. MATLAB has built in color maps to choose from, but we specified it because we want it to be equal to the color map used in the adaptive implementation.

```
function [] = heatSquare
    % Heat Equation for 2D square
    deltaX = 0.2;
    deltaY = deltaX;
    deltaT = 0.01;
    timeSteps = 10000;
    N = 512; % Steps in one direction
    Uold = zeros(N);
    Unew = zeros(N);
    Uold = bound(Uold, N); % Boundary conditions
    map = createColormap(); % Specify the color scheme
    colormap(map); % Apply the color scheme
    p = imagesc(Uold); % Creates an image of the heat matrix
    i = 2:N-1;
    j = 2:N-1;
    for n = 1:timeSteps
        Uxx = (Uold(i,j-1) - 2 * Uold(i,j) + Uold(i,j+1))/deltaX^2;
        Uyy = (Uold(i-1,j) - 2 * Uold(i,j) + Uold(i+1,j))/deltaY^2;
        Unew(i,j) = Uold(i,j) +  deltaT * (Uxx + Uyy);
        Unew = bound(Unew,N); % Setting boundary conditions
        Uold = Unew;
        set(p,'cData', Unew); % Update image
        drawnow limitrate nocallbacks  % Render updated image
    end
    end
```

Figure 3.5: The naive implementation of The Heat Equation.

3.4.2 Converting to GPU code

As mention in Section 1.1, the naive implementation uses standard libraries for utilizing the power of the GPU. For MATLAB, this library is called MATLAB Parallel Computing Toolbox. To utilize this, we only need to change the declaration of the matrices from

```
Unew = zeros(N);
```

to

```
Unew = gpuArray(zeros(N));
```

and similarly for the "Uold" matrix, and all operation on these matrices will be performed on the GPU without having to specify how. To fetch the result back to the CPU requires only to use "gather("arrayname");"

```
set(p, 'cData', gather(Unew));
```

The problem arises when we want to visualize the data. MATLAB has no support for interoperability between the graphics and GPGPU. This means that for each time step, MATLAB will execute in this order:

1. Compute on the GPU
2. Send data from GPU to CPU
3. CPU renders the data

MATLAB has an option to use OpenGL to render, but it does not allow us to visualize data which is already on the GPU. This makes visualization on MATLAB using GPU less efficient that using only CPU due to the data transfer for small and medium size problems. For computational comparison, this is not a problem, but it might be for the visual part. We will for that part try to figure out if there is any effort in making the naive implementation a bit more adaptive, or if its just better to do it all adaptively.

The first attempt is to do all computations on the GPU, and gather the results on CPU for animation. Animation is not a fair comparison versus real time rendering, but it is interesting to see if it can be done. To do so, the GPU has to store all the data. In the implementation in Section 3.4.1, the GPU stored the matrix representation of the square which is $I \cdot J$ numbers, where I is

the number of space steps in x direction and J is the number of space steps in y direction. Now the GPU also needs to store data for all time steps, resulting in $I \cdot J \cdot T$ numbers, where T is the number of time steps. This is only suitable for either small squares or few time steps. After a few benchmarks with increasing problem size, the conclusion was that this solution was inexpedient.

With this in mind, it was clear that the problem needed to be divided into smaller problems, and combined at the CPU. When the size of the problem is so large that the GPU cannot store all data, time steps can be divided in two time periods. The first period is first computed and saved on the CPU before the second period is computed on the GPU and combined with the first period on the CPU. We refer to this model of computations as GPU with block computations. This worked better than the previous attempt. If the GPU cannot store all the data, it will be given smaller time periods to compute. Unfortunately, this solution did not scale very well as the CPU also went out of memory when the size of the problem increased.

With this experience, we know now that the most efficient implementation in MATLAB is the most naive one. The outer for loop is needed because for each time step, we need the values calculated from the previous time step in addition to applying boundary conditions. Therefore, we cannot vectorize the outer for loop and this has a major drawback on the performance.

3.5 Adaptive implementation

The adaptive implementation is implemented in C++ and use OpenCL to utilize the GPU. In the adaptive implementation, the programmer has to specify everything, staring with discovering platforms and selecting a platform. A platform is an OpenCL implementation, for instance AMD OpenCL, NVIDIA OpenCL or Intel OpenCL. After selecting the platform, the programmer must select a device from the platform and a create context from this. A device is for instance a CPU or GPU. Selecting one platform together with one or more devices is called a context. Next, the code which runs on the GPU, called kernel code written in OpenCL C, must be written. The kernel code is read and compiled during runtime. A command queue is used to communicate with the GPU. The programmer has to create the command queue and attach a kernel to it to be able to run code on the GPU. All allocation on the CPU and GPU needs to be specified, and how

many work-groups and work-items to use in the calculation. In OpenCL, a work item is one of many parallel executions of a kernel. A collection of work items is called a work group which share memory. The programmer also has to consider how the memory is accessed, because the performance is better if the accessed data is already in the cache. Swapping cache costs a lot of time.

3.5.1 C++ code

In the adaptive implementation, C++ acts as a host. This means that C++ is not performing any computations or visualization. C++ is responsible for setting up data structures, initializing data, sending to and receiving data from the GPU. C++ is also responsible for queuing jobs to the GPU. In short terms, the program runs the following order:

```
Setup and initialize data structures;
Setup OpenGL;
Setup OpenCL;
while Program still running do
    Tell GPU to compute next time step (OpenCL);
    Tell GPU to change texture (OpenCL);
    Tell GPU to render the new texture (OpenGL);
end
```

The case presented in Section 4.6.1 contains more details.

3.5.2 OpenCL code

OpenCL is responsible for the computational part. OpenCL uses OpenCL C as programming language, and it is very similar to C. One of the largest difference is that OpenCL C does not support two-dimensional arrays. Arrays in OpenCL C must be allocated in one dimension, which makes indexing different. This means that an array which would be allocated in C++ as A[n][m] need to be allocated as A[n*m]. Loops can be done in OpenCL, but the idea is that each work-item is replacing the loops. Each work-item has a global identifier, and this is used to index the array. For example:

```
int idx = get_global_id(0);
int idy = get_global_id(1);
int index = idx+dim*idy;
```

get global id returns the global id for each work-item, and the parameter is 0 for x-direction, 1 for y-direction and 2 for z-direction. For this problem we only use two dimensions. It is important to notice that "idx*dim+idy" where "dim" is the leading dimension, gives the exact same result as "idx+dim*idy", but with worse performance due to memory swapping. With this in mind, the kernel to compute one step of the heat equation and update the boundaries is:

```
__kernel void heatEquation_kernel(__global float *Uold, \
        __global float *Unew,const int dim){
        int idx = get_global_id(0);
        int idy = get_global_id(1);
        float deltaT = 0.01;
        float dx2 = 0.2*0.2;
        float dy2 = 0.2*0.2;
        int ind = idx+dim*idy;
        int left = ind-1;
        int right = ind+1;
        int up = ind-dim;
        int down = ind+dim;
        int mid = dim*dim/2 + dim/2;
        // If we are not on the boundary, update values
        if(idx != 0 && idy != 0 && idx != dim-1 && \
        idy != dim-1 && ind != mid){
                float uxx = (Uold[left] - 2 * Uold[ind] + Uold[right])/dx2;
                float uyy = (Uold[up] - 2 * Uold[ind] + Uold[down])/dy2;
                Unew[ind] = Uold[ind] + deltaT * (uxx+uyy);
        }
        else if(ind == mid){
                Unew[ind] = 100.0; // Center at 100 degrees
        }
        else if(ind == mid-1){
                Unew[ind]= 100.0;
        }
        else if(ind == mid-dim){
                Unew[ind]= 100.0;
        }
        else if(ind == mid-dim-1){
                Unew[ind]= 100.0;
        }
        else{
                Unew[ind] = 0.0;  // Boundary condition
        }
}
```

To update the texture according to the new values, we call another kernel which computes the color with respect to the temperature value.

```
__kernel void texture_kernel(__write_only image2d_t texture, \
        int dim, __global float *Unew)
{
        int2 coord = { get_global_id(0), get_global_id(1) };
        int ind = coord.x+dim*coord.y;
        __private float r, g, b;
        if(Unew[ind] >= 75){
                r = 1.0f;
                g = 0+((100-Unew[ind])*0.04);
                b = 0.0f;
        }
        else if(Unew[ind] >= 50){
                r = 2-((100-Unew[ind])*0.04);
                g = 1.0f;
                b = 0.0f;
        }
        else if(Unew[ind] >= 25){
                r = 0.0f;
                g = 1.0f;
                b = -2 + ((100-Unew[ind])*0.04);
        }
        else{
                r = 0.0f;
                g = 4 - ((100-Unew[ind])*0.04);
                b = 1.0f;
        }
        float4 col = {r, g, b, 1.0f}; // red, green, blue, alpha
        write_imagef(texture, coord, col); // Update texture
}
```

3.6 Complexity

In each time step, we need to calculate Uxx, Uyy and combine to a new matrix of U as shown in the code in Section 3.4.1. To calculate Uxx, we require one addition, one subtraction, one multiplication and one division. These four operations are done for $\forall(i,j)$ where $i = [1,2,\ldots,I]$ and $j = [1,2,\ldots,J]$ resulting in $4 \cdot I \cdot J$ operations. For Uyy, it is the same amount of operations, and for Unew it is two additions and one multiplication. In total there are $11 \cdot I \cdot J$ operations in each time step. This gives us the complexity of $\mathcal{O}(I \cdot J)$ per time step, and a total complexity of $\mathcal{O}(I \cdot J \cdot T)$ where T is the number of time steps. The $I \cdot J$ computations can be processed in parallel, but for each time step in N, there is a need to set boundary conditions and update graphics.

3.7 Comparison of implementations

3.7.1 MATLAB

The effort of changing the implementation from CPU to GPU in MATLAB is very small. As mentioned in 3.4.2, the only change of code is initialization of the arrays, and a gather to send data from GPU back to CPU after computations. This is done in minutes if the CPU implementation is implemented in such way that there is no dependency in each time step.

3.7.2 C++/OpenCL/OpenGL

Implementing in C++ and OpenCL + OpenGL requires more knowledge by the programmer, and the implementation time varies from a day up to a week, depending on code quality, refactoring of the code and optimization. Knowledge about memory and implementation quality are more important in the adaptive implementation e.g. because the efficiency of the program is better if the data is already in the cache. For instance, the running time of this implementation went about ten times faster just by changing the indexing of the arrays.

3.8 Efficiency

To be able to compare the different implementations, we want to measure the efficiency of each implementation in terms of running time for the problems with different problem sizes. We also want to see how the algorithm scales, and to measure the scalability for the algorithm we increase the problem, but keep the amount of work per work item the same. This is a fair comparison since MATLAB does not allow us to control how many work items are used on the GPU.

3.8.1 Efficiency of computation, MATLAB

We begin with leading dimension of 64 which gives us a total number of elements of $64 \cdot 64 = 4096$. We scale the problem linear in terms of total number of elements to compute, so the next leading dimensions will be $\sqrt{4096 \cdot 2} = 91$, $\sqrt{4096 \cdot 3} = 111$ until the leading dimension is 448. For each dimension, the program run from $t = 1$ to $t = 10000$ and calculate minimum, maximum and average running time.

CPU implementation

As a reference, we begin to look at the running time of MATLAB CPU implementation to ensure that the GPU implementation is not limited due to the lack of interoperability support. As we can see in Figure 3.6, using the CPU is not suitable when the size of the problem is increasing.

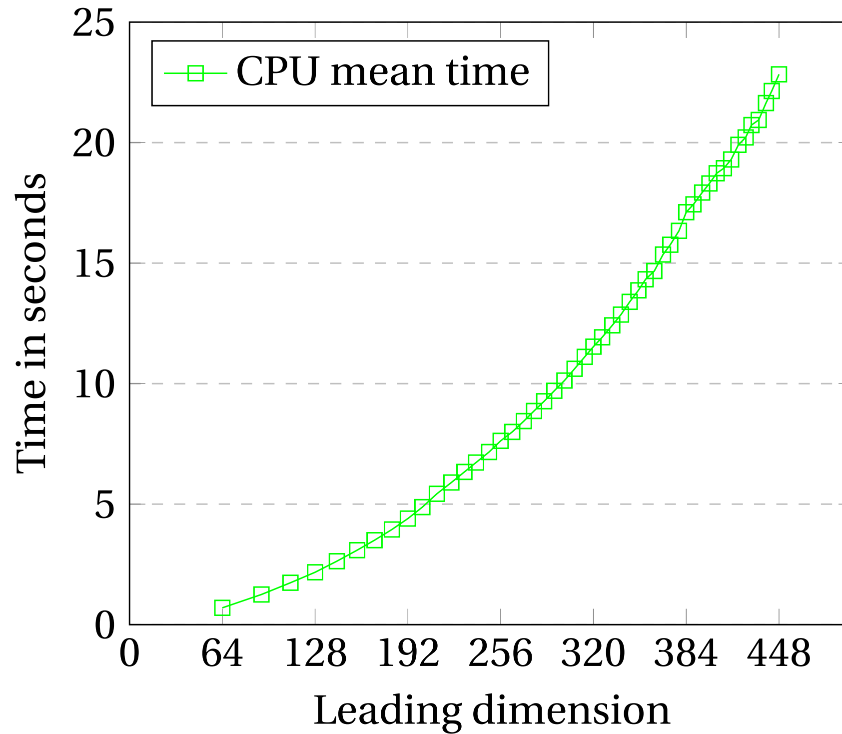

Figure 3.6: CPU running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.1.

Naive GPU implementation

This is the most naive implementation. For small problem size, it runs slowly compared to the other implementation, but it scales much better. There is almost no increased time when increasing the size of *LD* from 64 to 448.

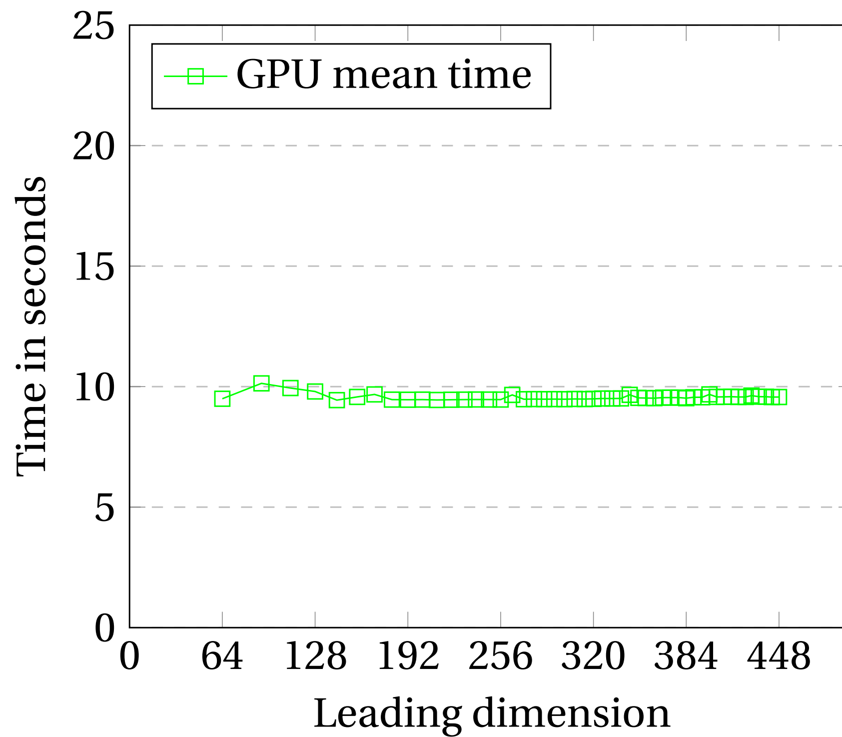

Figure 3.7: GPU running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.2.

3.8.2 Efficiency of computation, C++/OpenCL

OpenCL has the advantage of holding the memory on the GPU during the whole computation, and only getting instructions from C++. This means that the CPU sends the GPU memory objects to the GPU once, and after that the CPU only queue jobs to the GPU. The only time the GPU has to send data back to the CPU is when the programmer specifies it. We can see from the table below that this affects the running time in a positive way. To benchmark, the Hayai [11] library is used. The program computes the first 10000 time steps and the program is timed 100 times, just like we did with the MATLAB implementation. As we can see in Figure 3.8, the overhead of using the GPU is less than in MATLAB and it is much faster than the MATLAB implementation in Figure 3.7

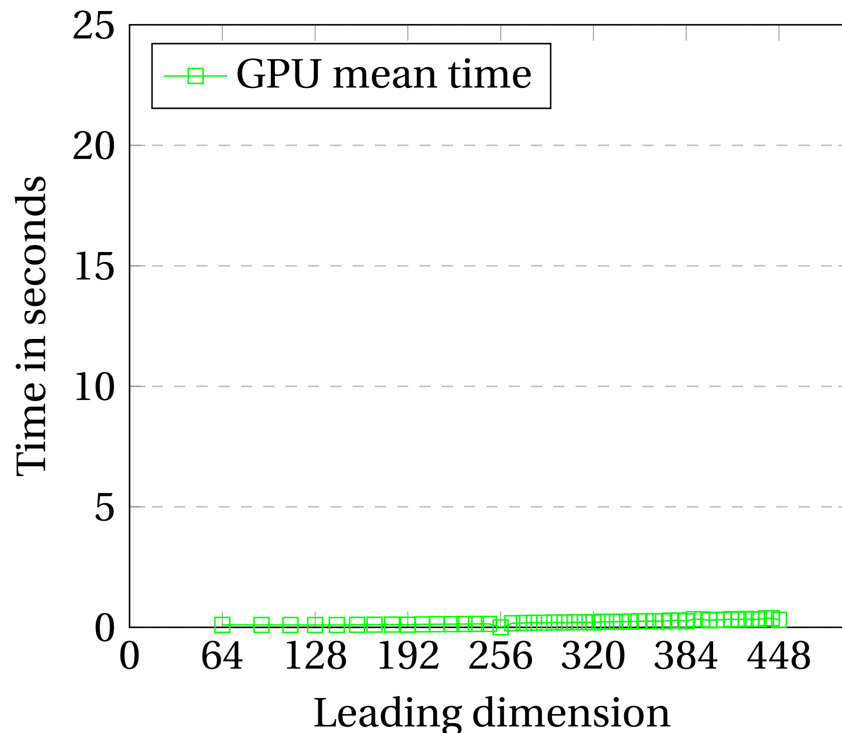

Figure 3.8: GPU running time in seconds, C++/OpenCL. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.3.

3.8.3 Efficiency of visualization, MATLAB

In this section, we will look at the running time for the naive implementation for rendering the heat equation in real time. We use the CPU implementation again as a reference implementation to ensure that using GPU in MATLAB is beneficial.

CPU implementation

The graph in Figure 3.9 shows CPU mean running time. It is easy to see that the CPU implementation scales badly when the size of the problem increases.

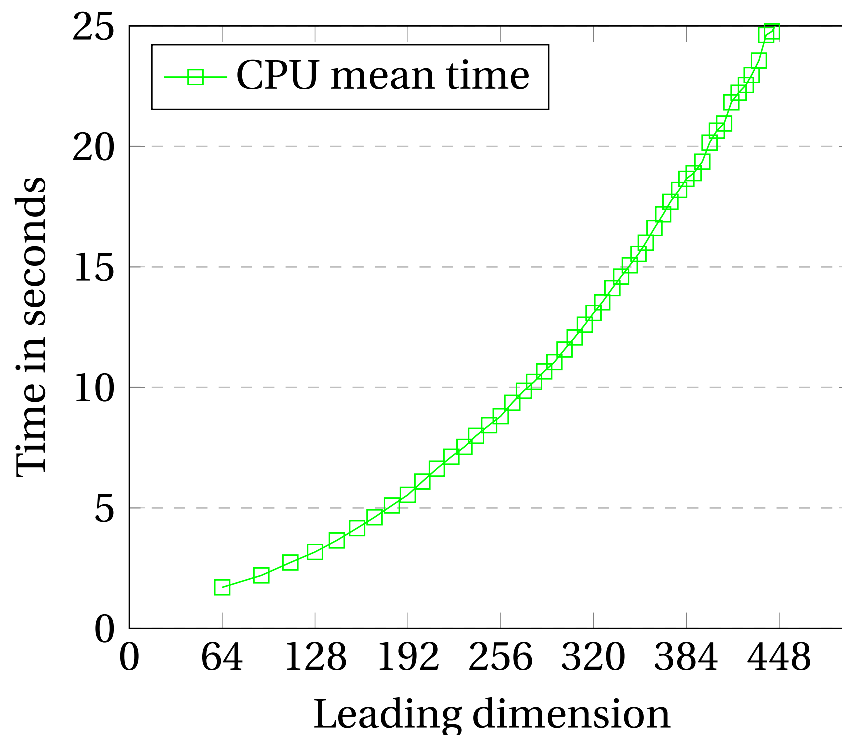

Figure 3.9: CPU with visualization running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.4.

Naive GPU implementation

The naive GPU implementation is as expected slower than the CPU implementation for small problem sizes, but it scales much better. From the running time in Figure 3.10 we can see that MATLAB suffers from lack of interoperability between graphics and computation on the GPU and this affects the efficiency. As we can see in Figure 3.10, the increase in running time starts earlier and the increase is greater than for computation only, seen in Figure 3.7.

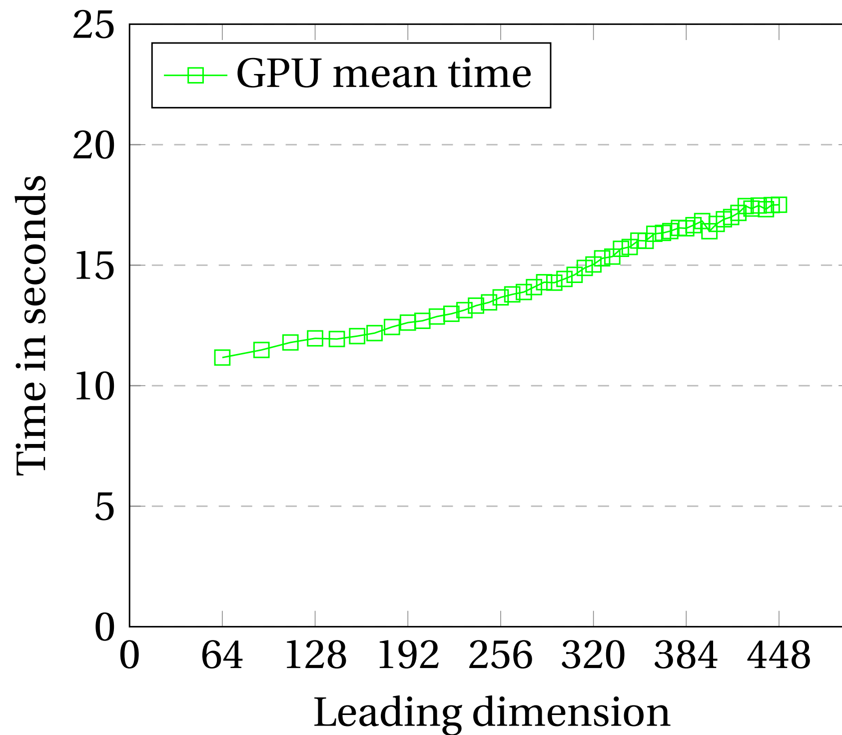

Figure 3.10: GPU running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.5.

3.8.4 Efficiency of visualization, C++/OpenCL/OpenGL

The graph in Figure 3.11 shows that the running time has increased with a few seconds, which is the overhead of using OpenGL. The scaling is similar to Figure 3.8 with only computations, but with some increased running time due to writing new values to the texture and render it for each time step.

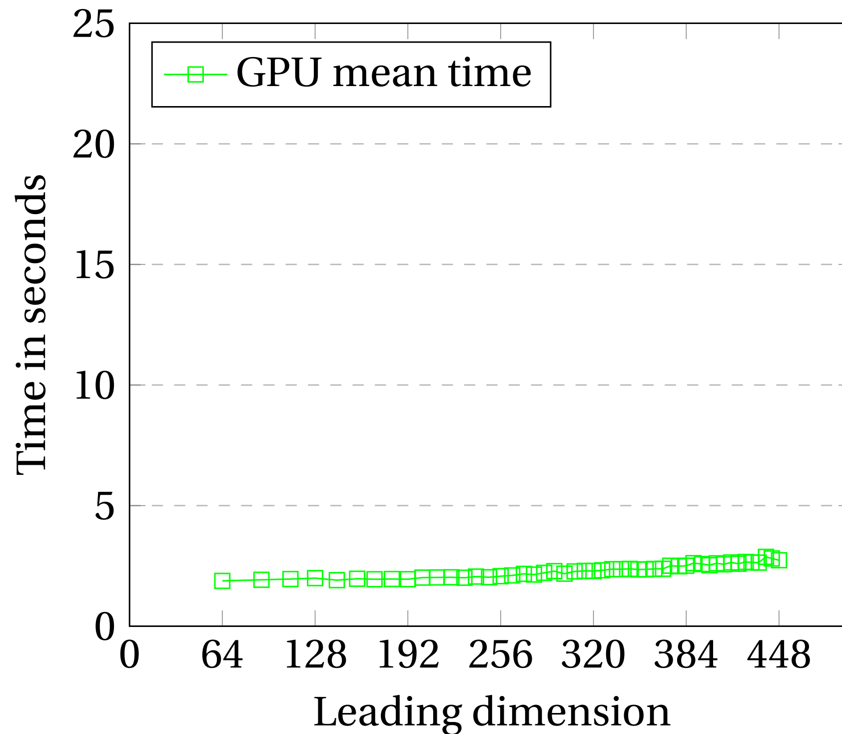

Figure 3.11: GPU running time in seconds, C++/OpenCL/OpenGL. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.6.

3.9 Analysis

What we have learned in this chapter is that there is an overhead just to setup the GPU for computations. The overhead was largest for MATLAB which uses about ten seconds no matter how small the problem is. This is not just the GPU overhead, but also the slow for-loop which is for the time steps. The adapted implementation has a smaller overhead and is therefore much faster, but it takes longer time to implement.

One can imagine that since OpenCL can launch millions of work-items, this algorithm would run completely in parallel since each computation can be done independently of other. That is true, but if we look at Figure 3.10 and Figure 3.11, the running time is increasing as the size of the problem is increasing. There are multiple reasons for this. First of all, we need to look at the OpenCL platform model. One platform has multiple compute units or streaming multiprocessors as NVIDIA calls them, each consisting of a number of stream processors depending on the hardware [12]. On NVIDIA hardware the multiprocessor will execute 32 threads at once (which they call a "warp group"), if the work group contains more threads than this they will be serialized, which has obvious implications on the consistency of local memory [12]. When the amount of work is increasing, the work groups are queued for execution and have to wait for available multiprocessor. The second reason is that there is more to synchronize. The command queue is in-order, which means that the kernels is executing in order and have to wait for the previous kernel to finish.

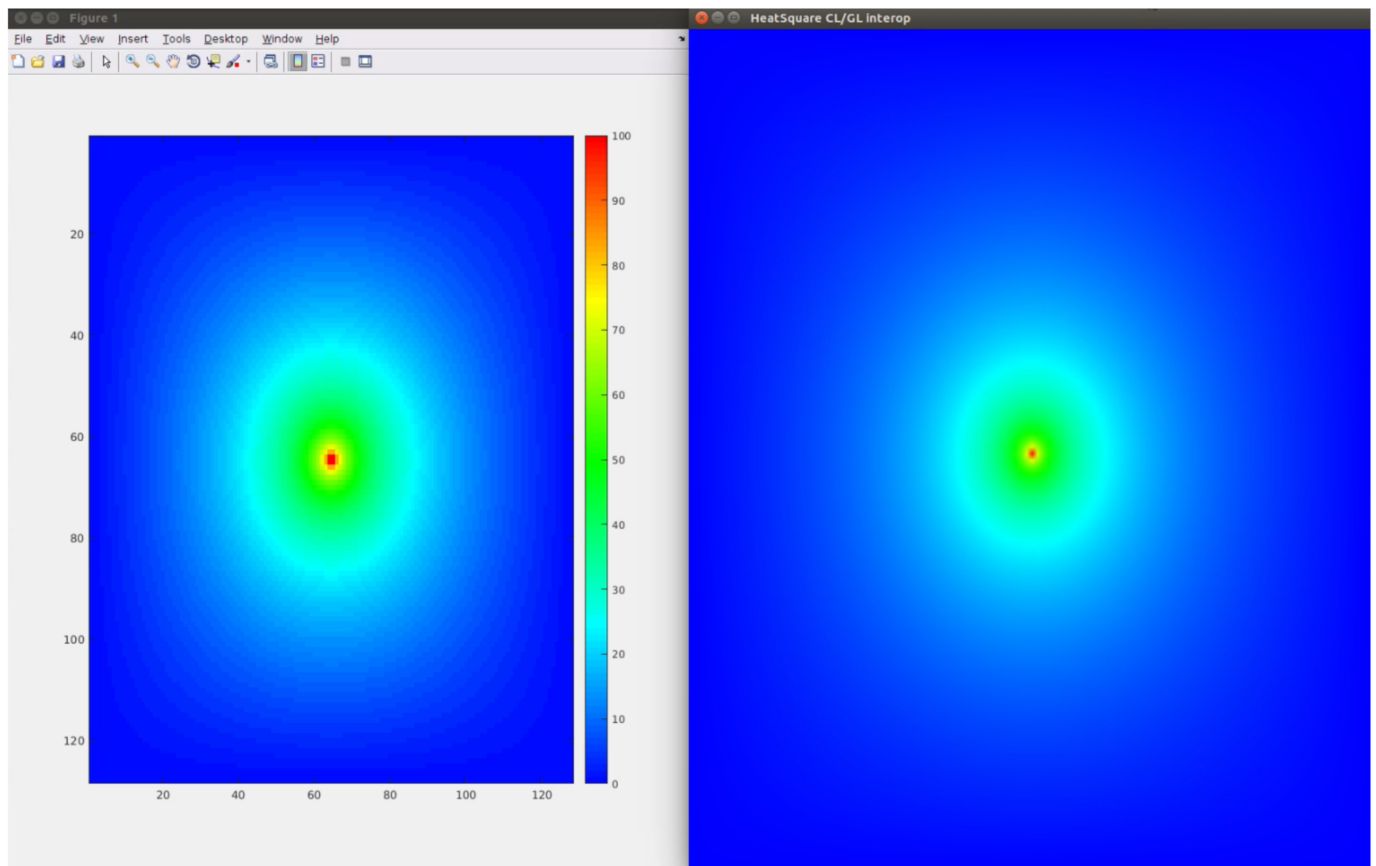

Figure 3.12: Graphical output from MATLAB and C++/OpenGL. MATLAB on the left side and C++/OpenGL on the right side.

Chapter 4

Shallow Water

4.1 Overview

The Shallow Water equations are a set of partial differential equations which is a special case of water simulation. In the Shallow Water equations, we assume that the water is shallow with respect to the wave length. This means that we can ignore vertical variations in the velocity field and just keep track of the depth-averaged horizontal velocities $u(x, y)$ and $v(x, y)$ [13].

It is not only restricted to simulating water, but it can also be used to simulate avalanches and other incompressible fluids. In this project we will focus on the Shallow Shallow Water equations applied on water in a fixed square. The partial differential equations are:

$$\frac{\partial h}{\partial t} + \frac{\partial uh}{\partial x} + \frac{\partial vh}{\partial y} = 0 \tag{4.1}$$

$$\frac{\partial (uh)}{\partial t} + \frac{\partial (u^2 h + \frac{1}{2} g h^2)}{\partial x} + \frac{\partial (uvh)}{\partial y} = 0 \tag{4.2}$$

$$\frac{\partial (vh)}{\partial t} + \frac{\partial (uvh)}{\partial x} + \frac{\partial (v^2 h + \frac{1}{2} g h^2)}{\partial y} = 0 \tag{4.3}$$

The independent variable are time, t, and two space variables x and y. The dependent variables are the height h and the velocities u and v. Acceleration due to gravity is denoted g, which is $9.81 m/s^2$.

By introducing three vectors, H, F(H) and G(H),

$$H = \begin{pmatrix} h \\ uh \\ vh \end{pmatrix} \tag{4.4}$$

$$F(H) = \begin{pmatrix} uh \\ u^2h + \frac{1}{2}gh^2 \\ uvh \end{pmatrix} \tag{4.5}$$

$$G(H) = \begin{pmatrix} vh \\ uvh \\ v^2h + \frac{1}{2}gh^2 \end{pmatrix}, \tag{4.6}$$

we can write the equations in a compact form.

$$\frac{\partial H}{\partial t} + \frac{\partial F(H)}{\partial x} + \frac{\partial G(H)}{\partial y} = 0. \tag{4.7}$$

4.2 Discretization

In order to solve an approximation of this PDE on a computer, we need to come up with a numerical solution. With the previously introduced vectors, the Shallow Water equations are an instance of a hyperbolic conservation law [14]. Since the equation is a hyperbolic partial differential equation, it can be solved with the Lax-Wendroff [15] method which is based on finite differences and is second order accurate in both time and space. We introduce a regular square finite difference grid with a vector-valued solution centered in the grid cells, denoted center point as shown in Figure 4.1 [14]. At the beginning of each time step, the variables represent the solution at the center of the finite difference grid, and we use those variables to calculate the half time step. This is the values of H at $H^{n+\frac{1}{2}}$ as shown in Figure 4.2 [14]. H represent the height matrix and U and V represent the velocity matrices in u and v direction.

The first step of Lax-Wendroff is:

$$f_{i+1/2}^{n+1/2} = \frac{f_i^n + f_{i+1}^n}{2} - \frac{\Delta t}{2 \cdot \Delta x} \cdot (g_{i+1}^n - g_i^n) \tag{4.8}$$

and the final step of is:

$$f_i^{n+1} = f_i^n - \frac{\Delta t}{\Delta x}(g_{i+1/2}^{n+1/2} - g_{i-1/2}^{n+1/2}) \tag{4.9}$$

To apply this on the Shallow Water equation for the first partial derivative, we just change f to H, and g(f) to F(H). Since H a is two dimensional grid, we need to calculate steps in both x and y direction. First step in x-direction:

$$H_{i+1/2}^{n+1/2} = \frac{H_i^n + H_{i+1}^n}{2} - \frac{\Delta t}{2 \cdot \Delta x} \cdot (U_{i+1}^n - U_i^n) \tag{4.10}$$

First step in y-direction:

$$H_{j+1/2}^{n+1/2} = \frac{H_j^n + H_{j+1}^n}{2} - \frac{\Delta t}{2 \cdot \Delta y} \cdot (V_{j+1}^n - V_j^n) \tag{4.11}$$

and the last step:

$$H_{i,j}^{n+1} = H_{i,j}^n - \frac{\Delta t}{\Delta x}(U_{i+1/2}^{n+1/2} - U_{i-1/2}^{n+1/2}) - \frac{\Delta t}{\Delta y}(V_{i+1/2}^{n+1/2} - V_{i-1/2}^{n+1/2}) \tag{4.12}$$

The half time steps for U and V, and last step for U and V are calculated in the same manner.

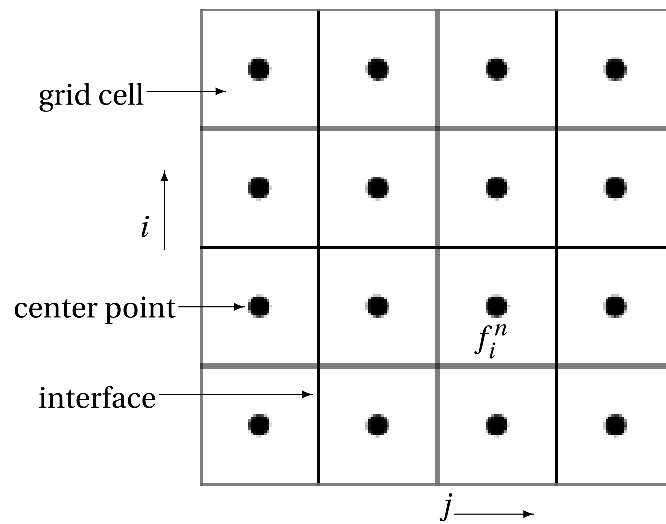

Figure 4.1: At the beginning of a time step, the variables represent the solution at the centers of the finite difference grid [14].

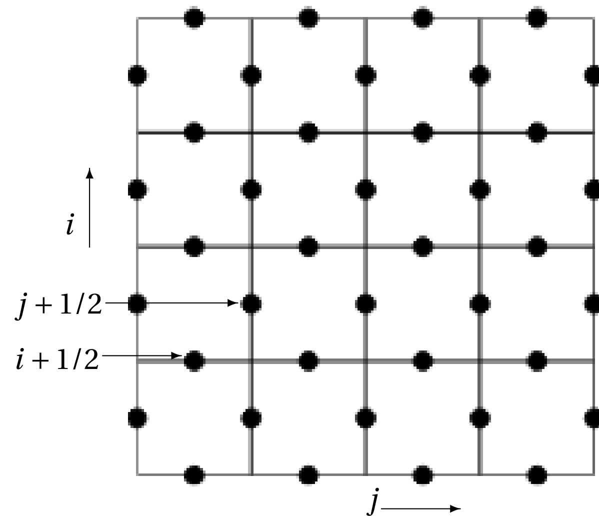

Figure 4.2: The first stage computes values that represent the solution at the midpoints of the interface between the grid cells in the finite difference grid [14].

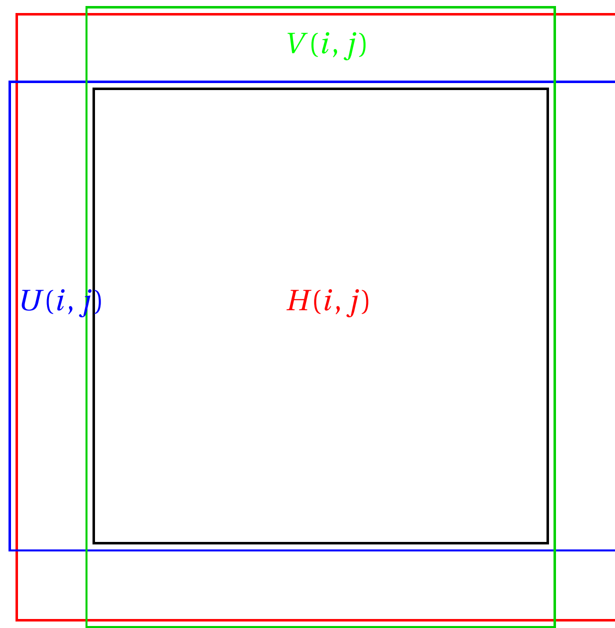

Figure 4.3: Illustration of the ghost cells. Black square represent the finite difference grid, the red square represent the height matrix and the red and green square represent velocity matrices in u and v direction.

4.3 Boundary conditions

Since we calculate the equation for a fixed space, we need to apply boundary conditions to define how we calculate the solution at the points lying on the edge of the water area. This is because finite differences uses values from the neighboring cells, and the outermost cell rows and columns do not have neighbors outside the computed region. In this project, ghost cells are applied around the finite difference grid and we copy values into the ghost cell according to boundary conditions given. As seen in Figure 4.3, the black square is the finite difference grid. The red square is the height matrix including the ghost cells. The blue square is the velocity matrix in u direction with ghost cells and the green square is the velocity matrix in v direction with ghost cells. We have used three different boundary conditions:

4.3.1 Reflective boundaries

With reflective boundaries, all waves which hit the boundary are returned in the opposite direction with negative velocity.

$$H_{i,1} = H_{i,2} \quad H_{i,I+2} = H_{i,I+1} \quad H_{1,i} = H_{2,i} \quad H_{I+2,1} = H_{I+1,2} \quad i \in [1,2,\dots,I] \tag{4.13}$$

$$U_{i,1} = U_{i,2} \quad U_{i,I+2} = U_{i,I+1} \quad U_{1,i} = -U_{2,i} \quad U_{I+2,1} = -U_{I+1,2} \quad i \in [1,2,\dots,I] \tag{4.14}$$

$$V_{i,1} = -V_{i,2} \quad V_{i,I+2} = -V_{i,I+1} \quad V_{1,i} = V_{2,i} \quad V_{I+2,1} = V_{I+1,2} \quad i \in [1,2,\dots,I] \tag{4.15}$$

4.3.2 Periodic boundaries

With periodic boundaries, all waves which hit the boundaries will continue at the opposite side of the grid. You can envision that left and right side, and top and bottom side of the grid are connected like a torus. For the periodic boundaries, there is no boundary to be "set" for each iteration and there is no ghost cell. In each iteration of the approximation of the equations, grid cells in the leftmost column using grids cells in the rightmost column. Grid cells in the rightmost column are using grid cells in the leftmost column. The same goes for rows on the top and bottom.

4.3.3 Free boundaries

With free boundaries, all waves which hit the boundaries will just be copied out to the ghost cells and fade.

$$H_{i,1} = H_{i,2} \quad H_{i,I+2} = H_{i,I+1} \quad H_{1,i} = H_{2,i} \quad H_{I+2,1} = H_{I+1,2} \quad i \in [1,2,\dots,I] \tag{4.16}$$

$$U_{i,1} = U_{i,2} \quad U_{i,I+2} = U_{i,I+1} \quad U_{1,i} = U_{2,i} \quad U_{I+2,1} = U_{I+1,2} \quad i \in [1,2,\dots,I] \tag{4.17}$$

$$V_{i,1} = V_{i,2} \quad V_{i,I+2} = V_{i,I+1} \quad V_{1,i} = V_{2,i} \quad V_{I+2,1} = V_{I+1,2} \quad i \in [1,2,\dots,I] \tag{4.18}$$

4.4 Initial values

As initial values, the velocity in U and V direction is set to zero. For the height matrix, H, the initial values are set by the Gaussian function:

$$y = ce^{-10(x^2+y^2)} \tag{4.19}$$

where c is height constant and x and y are coordinates in the grid. This results in a raised wave in the center of the finite difference grid as showed in Figure 4.4. These initial values are used because it makes it easy to verify the result and correctness of the equations and boundary conditions.

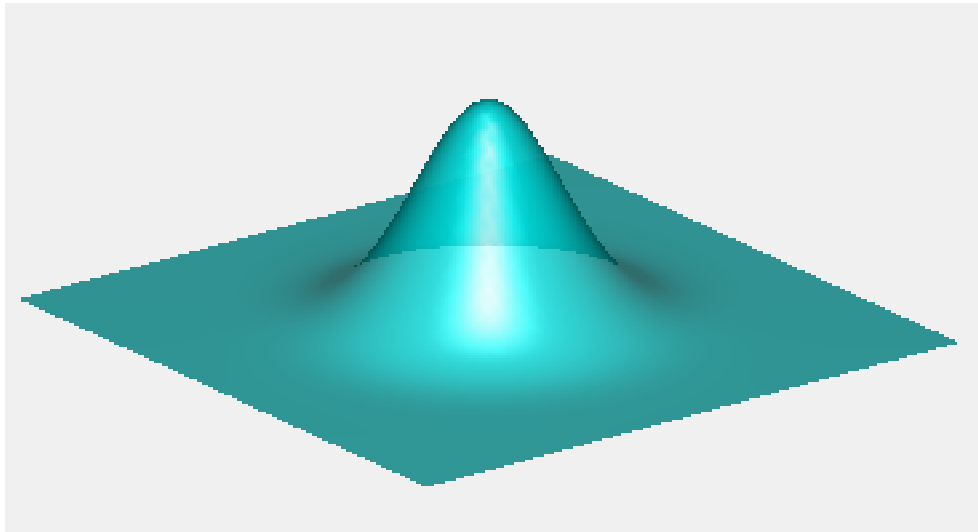

Figure 4.4: Gaussian function for initial values

4.5 Naive implementation

For the naive implementation, MATLAB is used. On the CPU, it is straight forward.

4.5.1 CPU implementation

First, set up variables, matrices and set the initial values.

Here, RaiseWater is a the function from Equation (4.19). The values from Lax-Wendroff half step

```
gravity = 9.81;
deltaX = 0.3;    deltaY = 0.3;    deltaT = 0.01;
grid = 64; % Space steps in each direction
steps = 5000; % Time steps
H=ones(grid+2);      Hx=zeros(grid+1);    Hy=zeros(grid+1);
U=zeros(grid+2);     Ux=zeros(grid+1);    Uy=zeros(grid+1);
V=zeros(grid+2);     Vx=zeros(grid+1);    Vy=zeros(grid+1);
H = RaiseWater(H, 3.5, 0); % For raising water in a circle.
```

are stored in separate matrices called Hx for x-direction and Hy for y-direction and similarly for U and V.

```
function H = RaiseWater(H, height, offset)
    str = size(H);
    resolution = 2 / (str(1)-1);
    [x,y] = ndgrid(-1:resolution:1);
    wave = height * exp(-10 * (x.^2 + y.^2));
    waveSize = size(wave,1);
    i = 1+offset:waveSize+offset;
    j = 1+offset:waveSize+offset;
    u = 1:waveSize;
    v = 1:waveSize;
    ii = 1+mod(i-1,str(1));
    jj = 1+mod(j-1,str(1));
    H(ii,jj) = H(ii,jj) + wave(u,v);
end
```

The main loop of the program is done in three steps:

1. Set boundary conditions
2. Compute half step
3. Compute Final step

```
for k = 1:steps
    % Boundary conditions
    [H, U, V] = ReflectiveBoundary(H,U,V,grid);
    % Halfstep
    [ Hx, Ux, Vx, Hy, Uy, Vy ] = HalfStep( H, U, V, 1, grid, deltaX, ...
        deltaY, deltaT, gravity );
    % Update values for hight and velocity
    [ H, U, V ] = FinalStep( H, Hx, Hy, U, Ux, Uy, V, Vx, Vy, 2, grid, ...
        deltaX, deltaY, deltaT, gravity);
end
```

where the boundaries are from Section 4.3.1

```
function [ H, U, V ] = ReflectiveBoundary( H, U, V, grid )
    H(:,1) = H(:,2);  U(:,1) = U(:,2);  V(:,1) = -V(:,2);
    H(:,grid+2) = H(:,grid+1); U(:,grid+2) = U(:,grid+1); ...
        V(:,grid+2) = -V(:,grid+1);
    H(1,:) = H(2,:);  U(1,:) = -U(2,:);  V(1,:) = V(2,:);
    H(grid+2,:) = H(grid+1,:); U(grid+2,:) = -U(grid+1,:);  ...
        V(grid+2,:) = V(grid+1,:);
end
```

Figure 4.5: The code in MATLAB for calculating the Lax-Wendroffs first half time step

```
function [ Hx, Ux, Vx, Hy, Uy, Vy ] = HalfStep( H, U, V, runFrom, runTo, ...
        deltaX, deltaY, deltaT, gravity )
    % X-direction
    i = runFrom:runTo+1;
    j = runFrom:runTo;
    Hx(i,j) = (H(i+1, j+1) + H(i,j+1)) / 2 - deltaT/(2*deltaX)* ...
        (U(i+1,j+1)-U(i,j+1));
    Ux(i,j) = (U(i+1, j+1) + U(i,j+1)) / 2 - deltaT/(2*deltaX)*(...
       (U(i+1,j+1).^2./H(i+1,j+1) + gravity/2*H(i+1,j+1).^2) - ...
       (U(i,j+1).^2./H(i,j+1) + gravity/2*H(i,j+1).^2));
    Vx(i,j) = (V(i+1, j+1) + V(i,j+1)) / 2 - deltaT/(2*deltaX)*(...
       (U(i+1,j+1).*V(i+1,j+1)./H(i+1,j+1)) - (U(i,j+1).*V(i,j+1)./H(i,j+1)));

    % Y-direction
    i = runFrom:runTo;
    j = runFrom:runTo+1;
    Hy(i,j) = (H(i+1, j+1) + H(i+1,j)) / 2 - deltaT/(2*deltaY)* ...
        (V(i+1,j+1)-V(i+1,j));
    Uy(i,j) = (U(i+1, j+1) + U(i+1,j)) / 2 - deltaT/(2*deltaY)*(...
        (U(i+1,j+1).*V(i+1,j+1)./H(i+1,j+1)) - ...
        (U(i,j+1).*V(i,j+1)./H(i,j+1)));
    Vy(i,j) = (V(i+1, j+1) + V(i+1,j)) / 2 - deltaT/(2*deltaY)*(...
        (V(i+1,j+1).^2./H(i+1,j+1) + gravity/2*H(i+1,j+1).^2) - ...
        (V(i+1,j).^2./H(i+1,j) + gravity/2*H(i+1,j).^2));
end
```

The half step seen Figure 4.5 are calculated according to Lax-Wendroffs first step from Section 4.8 and the final step seen in Figure 4.6 are calculated according to Lax-Wendroffs final step from Section 4.9. In the two last function there are two variables "runFrom" and "runTo". These makes it possible to only calculate certain areas of the finite grid and exclude dry spots where there is no water or the water level is lower than the terrain.

Figure 4.6: The code in MATLAB for calculating the Lax-Wendroffs final step

```
function [ H, U, V ] = FinalStep( H, Hx, Hy, U, Ux, Uy, V, Vx, Vy, ...
        runFrom, runTo, deltaX, deltaY, deltaT, gravity )
    i = runFrom:runTo+1;
    j = runFrom:runTo+1;
    H(i,j) = H(i,j) - (deltaT/deltaX)*(Ux(i,j-1)-Ux(i-1,j-1)) - ...
        (deltaT/deltaY)*(Vy(i-1,j)-Vy(i-1,j-1));
    U(i,j) = U(i,j) - (deltaT/deltaX)*(...
        (Ux(i,j-1).^2./Hx(i,j-1) + gravity/2*Hx(i,j-1).^2) - ...
        (Ux(i-1,j-1).^2./Hx(i-1,j-1) + gravity/2*Hx(i-1,j-1).^2)) ...
        - (deltaT/deltaY)*((Vy(i-1,j).*Uy(i-1,j)./Hy(i-1,j)) - ...
        (Vy(i-1,j-1).*Uy(i-1,j-1)./Hy(i-1,j-1)));
    V(i,j) = V(i,j) - (deltaT/deltaX)*( (Ux(i,j-1).*Vx(i,j-1)./Hx(i,j-1)) ...
        - (Ux(i-1,j-1).*Vx(i-1,j-1)./Hx(i-1,j-1))) - (deltaT/deltaY)* ...
        ((Vy(i-1,j).^2./Hy(i-1,j) + gravity/2 * Hy(i-1,j).^2) ...
        - (Vy(i-1,j-1).^2./Hy(i-1,j-1) + gravity/2*Hy(i-1,j-1).^2));
end
```

To apply graphics, we use "surfplot" from MATLABs API and define colors and lightning to make the waves look more realistic. More details about the graphical part is explained in Section 4.12.1.

```
surfplot = surf(H);
surfplot.FaceColor = 'cyan';
surfplot.EdgeColor = 'none';
camlight headlight;
lighting phong
alpha(surfplot,0.8)
set(gca,'zlim',[0 8])
axis off
```

For each iteration we need to update the surfplot.

```
    surfplot.ZData = H;
    drawnow limitrate nocallbacks
```

In the updates, MATLAB update the plot whenever "drawnow" is called. By using "limitrate nocallbacks", the rendering is none-blocking and the overall performance is increased as discussed in Section 3.4.1.

4.5.2 GPU implementation

As mentioned in Section 1.1, the naive implementation uses standard libraries for utilizing the power of the GPU. For MATLAB, this library is called MATLAB Parallel Computing Toolbox. To utilize this, we only need to change the declaration of the matrices from normal array to gpuArray. From the CPU implementation, we had:

```
H=ones(grid+2);      Hx=zeros(grid+1);    Hy=zeros(grid+1);
U=zeros(grid+2);     Ux=zeros(grid+1);    Uy=zeros(grid+1);
V=zeros(grid+2);     Vx=zeros(grid+1);    Vy=zeros(grid+1);
```

To utilize the GPU, we only need to change to:

```
H=gpuArray(ones(grid+2));
Hx=gpuArray(zeros(grid+1));    Hy=gpuArray(zeros(grid+1));
U=gpuArray(zeros(grid+2));
Ux=gpuArray(zeros(grid+1));    Uy=gpuArray(zeros(grid+1));
V=gpuArray(zeros(grid+2));
Vx=gpuArray(zeros(grid+1));    Vy=gpuArray(zeros(grid+1));
```

On the CPU, we updated the plot by:

```
surfplot.ZData = H;
drawnow limitrate nocallbacks
```

On the GPU however, we need to gather the information stored on the GPU before we can render.

```
surfplot.ZData = gather(H);
drawnow limitrate nocallbacks
```

4.6 Adaptive implementation

The adaptive implementation is implemented in C++ using OpenCL to utilize the GPU for computations, and using OpenGL to visualize.

4.6.1 C++

C++ is responsible for creating variables, data structures and sending data and instructions to OpenCL and OpenGL. To be able to use OpenCL/OpenGL interoperability, we need to create a shared context between OpenCL and OpenGL. This is platform dependent which means that the programmer has to specify which platform the code is written for. This code is designed to run on both Windows and Linux machines and the code is separated by "ifdef _WIN32" and "elif defined(_GNUC _)". If the code is compiled on a Windows machine, the code in "elif defined(_GNUC _)" is ignored by the compiler, and if the code is compiled on a Linux machine the code in "ifdef _WIN32" is ignored.

```
cl::Context createSharedContext(){
        std::cout << "Creating␣shared␣context" << std::endl;
        cl_int errNum;
        std::vector<cl::Platform> platformList;
        cl::Platform::get(&platformList);
        if (platformList.size() == 0){
                std::cerr << "Failed␣to␣find␣any␣OpenCL␣platforms." ...
                << std::endl;
                return EXIT_FAILURE;
        }
        cl_context_properties contextProperties[] = {

#ifdef _WIN32
        CL_CONTEXT_PLATFORM,
    (cl_context_properties)(cl_context_properties)(platformList[0])(),
    CL_GL_CONTEXT_KHR,
    (cl_context_properties)wglGetCurrentContext(),
    CL_WGL_HDC_KHR,
    (cl_context_properties)wglGetCurrentDC(),
#elif defined( __GNUC__)
        CL_CONTEXT_PLATFORM, (cl_context_properties)(platformList[0])(),
    CL_GL_CONTEXT_KHR, (cl_context_properties)glXGetCurrentContext(),
    CL_GLX_DISPLAY_KHR, (cl_context_properties)glXGetCurrentDisplay(),
#endif
                        0 };
        cl::Context context(CL_DEVICE_TYPE_GPU, contextProperties, ..
        NULL, NULL, &errNum);
        if (errNum != CL_SUCCESS){
                std::cout << "Could␣not␣create␣GPU␣context.." ...
                << std::endl;
        }
        std::cout << "Context␣created.␣" << std::endl;
        return context;
}
```

After creating the context, we need to compile the OpenCL kernel code during runtime.

```
cl::Program createProgram(cl::Context context, ...
        std::vector<cl::Device> devices, const char *filename) {
        std::ifstream file(filename);
        std::string code(std::istreambuf_iterator<char>(file), ...
        (std::istreambuf_iterator<char>()));
        cl::Program::Sources source(1, std::make_pair(code.c_str(), ...
        code.length() + 1));
        cl::Program program(context, source);
        program.build(devices);
        return program;
}
```

To be able to communicate with OpenCL, we create a command queue which we queue kernel jobs to. This is an in-order command queue, which means that the first job needs to finish before the second job is executed. By using in-order queue, we are ensured that the Lax-Wendroff half step is computed before the last step of Lax-Wendroff without using synchronization in the algorithm itself.

```
commandQueue = cl::CommandQueue(context, devices[0], 0);
    cl::Program program = createProgram(context, devices, "main.cl");
    halfStepKernel = cl::Kernel(program, "halfStep_kernel");
    finalStepKernel = cl::Kernel(program, "finalStep_kernel");
    boundaryKernel = cl::Kernel(program, "boundary_kernel");
    vboKernel = cl::Kernel(program, "vbo_kernel");
```

Because we want to do the computations on OpenCL and not the CPU, the H,U,V matrices along with the Hx,Hy,Ux,Uy,Vx and Vy matrices need to be sent to the GPU. This is accomplished with:

```
cl::Buffer HBuffer = cl::Buffer(context, CL_MEM_READ_WRITE | ...
        CL_MEM_COPY_HOST_PTR, totalSize2 * sizeof(float), (void*)&H[0]);
```

and similarly for U,V,Hx,Hy,Ux,Uy,Vx and Vy. For each kernel, we need to manually set the kernel arguments.

```
        halfStepKernel.setArg(0, HBuffer);
        halfStepKernel.setArg(1, HxBuffer);
        halfStepKernel.setArg(2, HyBuffer);
        halfStepKernel.setArg(3, UBuffer);
        halfStepKernel.setArg(4, UxBuffer);
        halfStepKernel.setArg(5, UyBuffer);
        halfStepKernel.setArg(6, VBuffer);
        halfStepKernel.setArg(7, VxBuffer);
        halfStepKernel.setArg(8, VyBuffer);
        halfStepKernel.setArg(9, grid + 1);
```

To launch to kernel on OpenCL, we use the command queue to queue jobs to the kernel.

```
commandQueue.enqueueNDRangeKernel(halfStepKernel, cl::NullRange, global, ...
        cl::NullRange); // Half-step
```

4.6.2 OpenCL

The logic to calculate next time step for the Shallow Water equations is implemented in OpenCL kernels. OpenCL kernels are written in OpenCL C, which is similar to the C programming language. As mentioned in Chapter 3, there is no support for multidimensional arrays in OpenCL, so the matrices are allocated in one row. To find the correct index in the array, we use the global identifier for each work-item. The work-items are organized in a two dimensional grid, which makes index calculation simpler. As we can see in the code below, there are two index calculations, and two dimensions. This is because the H,U and V arrays have ghost cells of different size as seen in Figure 4.3. Variables are created in the kernel because it will then be stored in the local memory of a work-item which is faster than the global memory.

```
// Index calculations
int idx = get_global_id(0);
int idy = get_global_id(1);

int ind = idx+dim*idy;
int dim2 = dim+1;
int ind2 = idx+dim2*idy;

int i1j1 = ind2+dim2+1;
int ij1 = ind2+1;
int i1j = ind2+dim2;

// Variables
float gravity = 9.81f;
float deltaX = 0.3f;
float deltaY = 0.3f;
float deltaT = 0.01f;
```

For the computation of the first step, we use the i1j1, ij1 and i1j to find the neighboring cells. Here, "i1j1" denotes i+1 and j+1, so H[i1j1] in code is the same as $H^n_{i+1,j+1}$. and similarly for final step and boundary conditions.

```
if(idx < dim-1){
    Hx[ind] = (H[i1j1] + H[ij1]) / 2.0 - deltaT/(2*deltaX) * ...
        ( U[i1j1]-U[ij1] );
    Ux[ind] = (U[i1j1] + U[ij1]) / 2.0 - deltaT/(2*deltaX) * ...
        ( (pow(U[i1j1],2)/H[i1j1] + gravity/2*pow(H[i1j1],2)) - ...
        ( pow(U[ij1],2)/H[ij1] + gravity/2*pow(H[ij1],2) ) );
    Vx[ind] = (V[i1j1] + V[ij1]) / 2.0 - deltaT/(2*deltaX) * ...
        ( U[i1j1]*V[i1j1]/H[i1j1]  - U[ij1]*V[ij1]/H[ij1] );
        }
if(idy < dim-1){
    Hy[ind] = (H[i1j1] + H[i1j]) / 2.0 - deltaT/(2*deltaY) * ...
              (V[i1j1]-V[i1j]);
        Uy[ind] = (U[i1j1] + U[i1j]) / 2.0 - deltaT/(2*deltaY) * ...
              ( (U[i1j1]*V[i1j1]/H[ij1]) - (U[ij1]*V[ij1]/H[ij1])    );
        Vy[ind] = (V[i1j1] + V[i1j]) / 2.0 - deltaT/(2*deltaY) * ...
              ( (pow(V[i1j1],2)/H[i1j1] + gravity/2*pow(H[i1j1],2)) - ...
        ( pow(V[i1j],2)/H[i1j] + gravity/2*pow(H[i1j],2) ) );
        }
```

4.6.3 OpenGL

OpenGL is responsible for visualizing the result in real time, using glut and glew library. Unlike MATLAB, where the rendering is done with one single line, we have to specify every little detail in OpenGL. First we need to create a vertex buffer object, more known as VBO. This is all the vertices in the mesh with position x,y,z and w. x and y use the indexing of i and j in the height matrix H as position with spacing of ten pixels as seen in the code snipped below.

```
int offset = 0;
for(int i = 0; i < vboGrid; i++){
        offset = i*vboGrid;
        for(int j = 0; j < vboGrid; j++){
                vbo_data[offset+j].x = j*10;
                vbo_data[offset+j].y = 10;
                vbo_data[offset+j].z = i*10;
                vbo_data[offset+j].w = 1;
        }
}
```

The downside to vertex buffers comes when we use many of the same vertices over and over again [16]. For example, a height map can be broken down into a series of triangle strips [16]. Since each neighbor strip shares one row of vertices, we will end up repeating a lot of vertices with a vertex buffer [16] as seen in Figure 4.7. From Figure 4.7 we can see that each middle row is sent twice to the GPU. This will cost in terms of memory and efficiency when the size of the problem is increasing. A workaround for this is to use an index buffer object, more known as IBO. This is an array containing only the order to draw the VBO. Now, if we look at the code above, the VBO looks more like in Figure 4.8. Now it only contains the position of each vertex in ascending order, and not in the order of drawing as in Figure 4.7. It is now the IBO who decides the drawing order. If we want to draw the VBO as a triangle strip, the order of the IBO must be 1, 6, 2, 7, 3, 8, 4, 9, 5 and so on. When we repeat the middle row of vertices, we only repeat the number, instead of repeating the entire block of data [16]. When we are creating a grid containing multiple triangle strips, like we do when we are creating a mesh for water, we need to add an extra vertex at the end of each strip and one extra vertex at the beginning of the next strip. This is known as a degenerate triangle, which is a triangle that has no area, and when the GPU encounters such triangles, it will simply skip them [16] as shown in Figure 4.9.

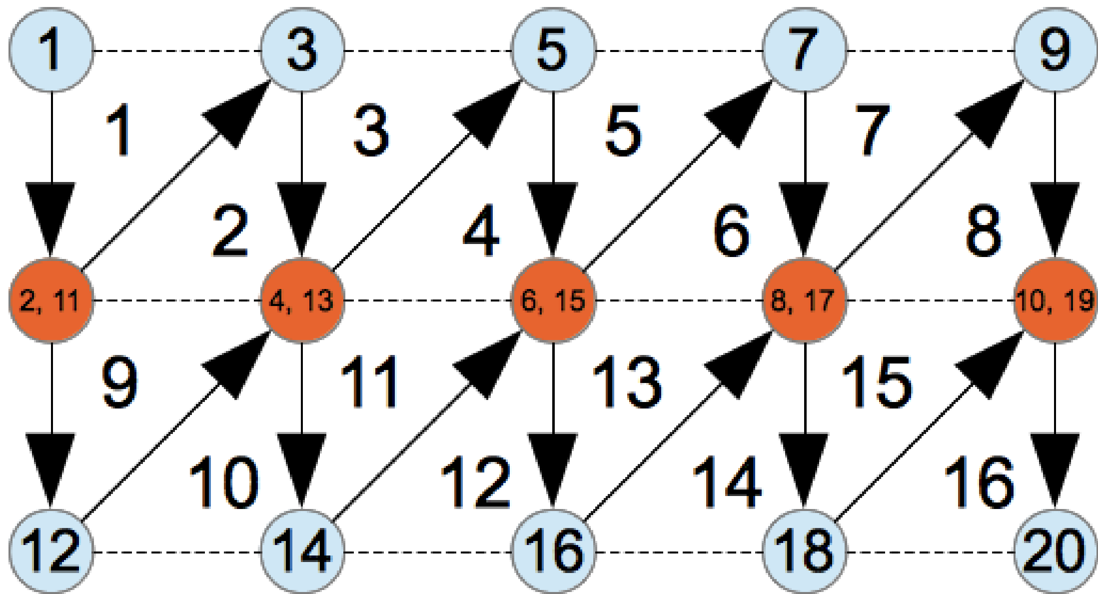

Figure 4.7: An example of VBO drawn as a triangle strip, from [16]

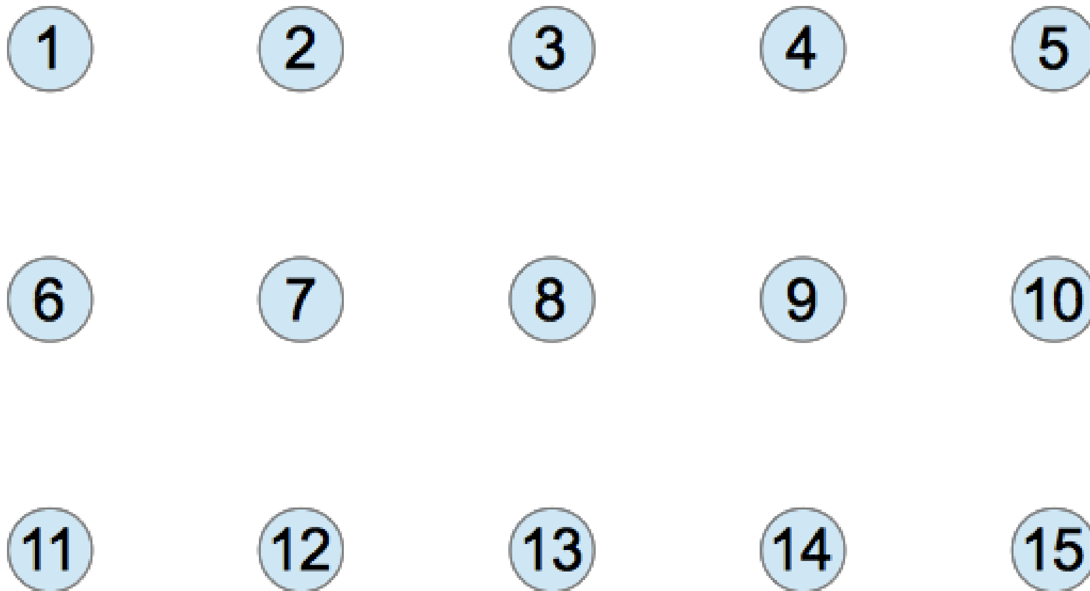

Figure 4.8: VBO without any edges, from [16]

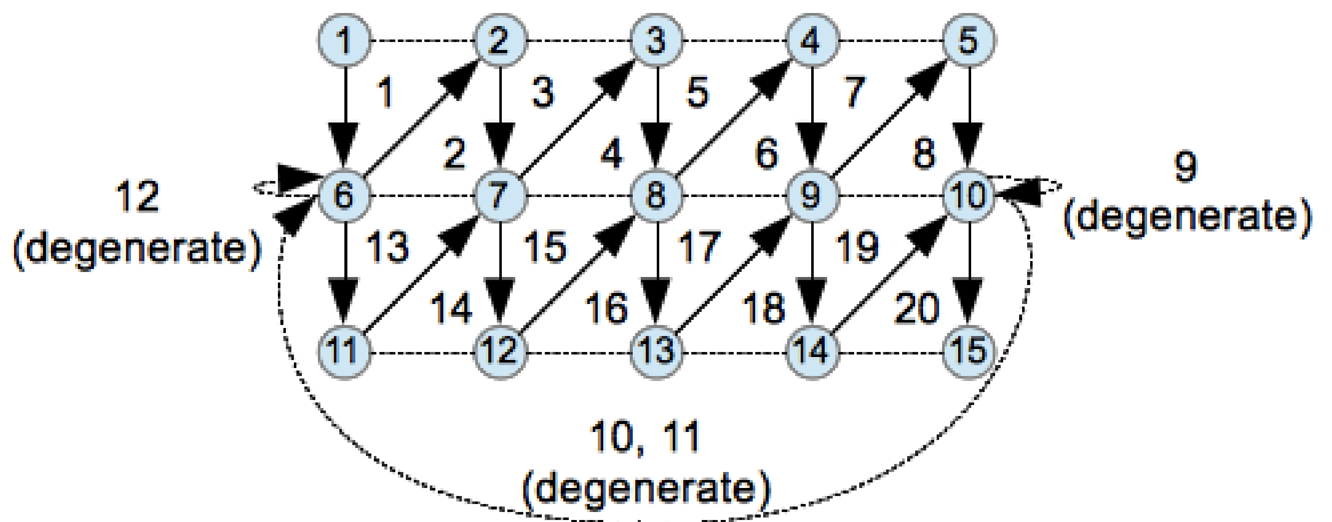

Figure 4.9: Example of degeneration when starting on a new row with triangle strip, from [16]

4.7 Numerical instability

Numerical approximation using explicit schemes requires selecting Δt, Δx and Δy to avoid instability. The CFL condition, named after Courant, Friedrichs and Lewy, defines the conditions as

$$CFL = \frac{u_x \Delta t}{\Delta x} + \frac{u_y \Delta t}{\Delta y} \leqslant C_{max} \tag{4.20}$$

where C_{max} is usually 1 for explicit schemes [17]. In this project, we experience numerical instability by using a combination of a large wave as initial values and reflective boundaries. By applying the Gaussian function from Section 4.4 on a fraction of the finite difference grid, we eliminate the problem with numerical instability, but the resolution of the wave is lower and the wave has four higher points as seen in Figure 4.10. If we apply the Gaussian function on the whole finite difference grid, the resolution is better and the wave looks more realistic (see Figure 4.11) but we get numerical instability when large waves are meeting at the boundaries as seen in Figure 4.12. By studying the numerical values right before the instability occurs, we see that the velocity is building up rapidly and lowering the Δt did not help in this case. The equations have certain limitation in terms of assumptions, such as ignoring the vertical variations in the velocity fields as mentioned in Section 4.1. This could be the reason why we experience numerical instability. Further investigation from the mathematical point of view is considered beyond the scope of this thesis.

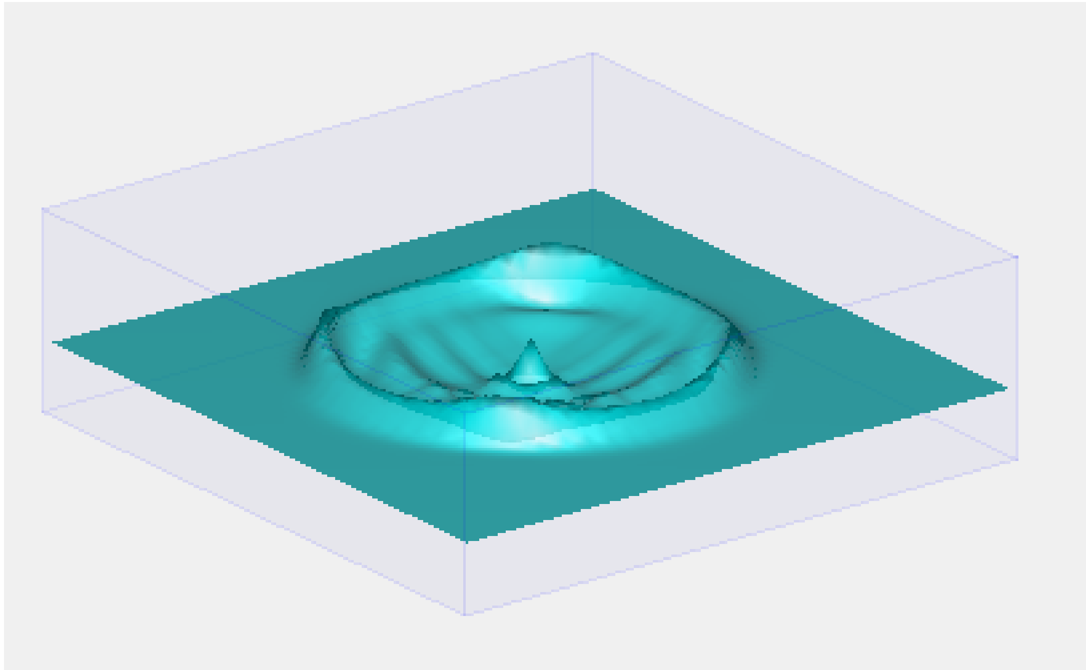

Figure 4.10: Gaussian initial values applied on 32x32 middle point of the finite difference grid with a total of 64x64 grid points.

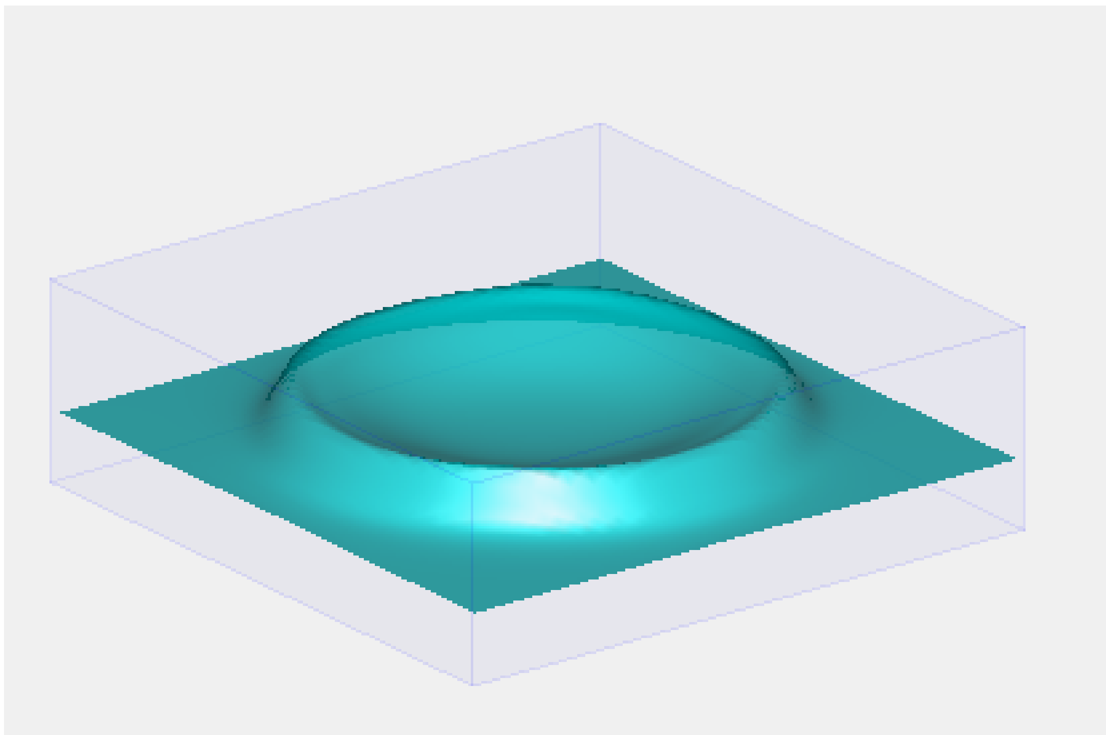

Figure 4.11: Gaussian initial values applied on the whole finite difference grid.

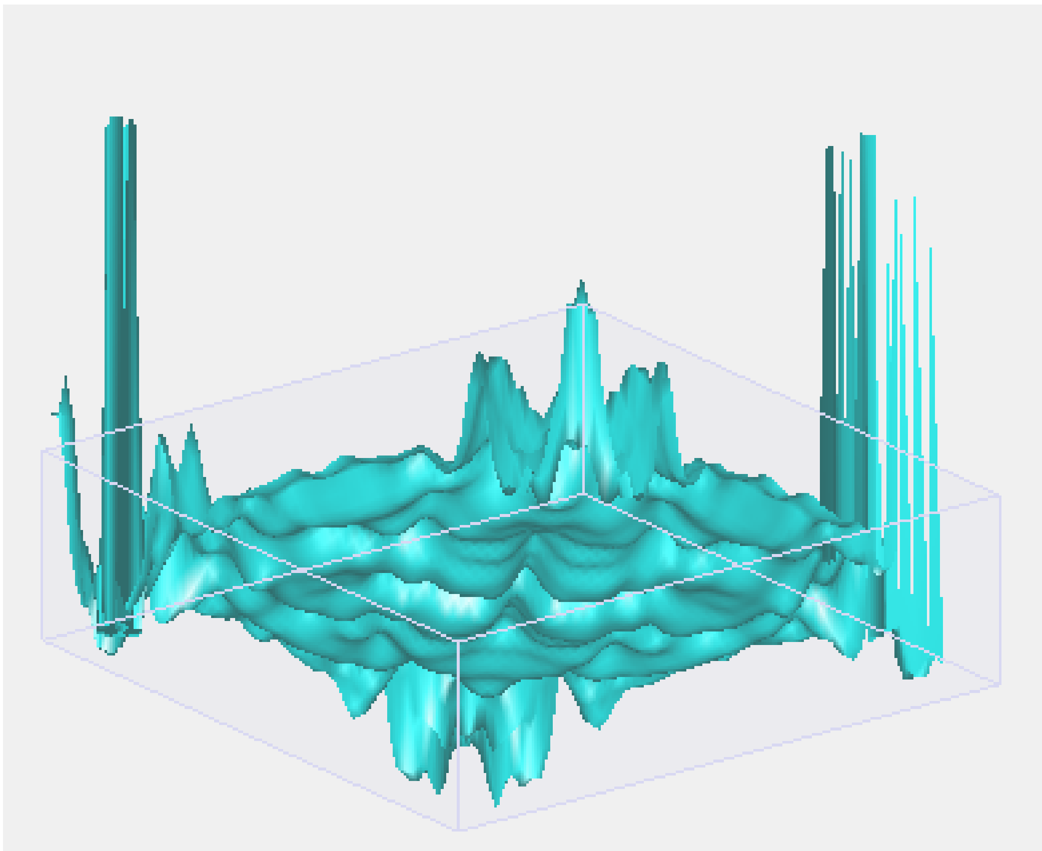

Figure 4.12: An example of numerical instability after 700 time steps and $\Delta t = 0.01$.

4.8 Complexity

In each time step, we need to calculate new values for the height, and velocities in u and v direction. This must be done for each space step. For the Lax-Wendroff half step, it is ten additions, twelve subtractions, twenty multiplications, twenty-four divisions and eight power functions. For the Lax-Wendroff final step, it is four additions, eleven subtractions, fourteen multiplications, eighteen divisions and eight power functions. In total 129 operations steps for $\forall(i,j)$ where $i = [1, 2, \ldots, I]$ and $j = [1, 2, \ldots, J]$. This gives us the complexity of $\mathcal{O}(I \cdot J)$ per time step, and a total complexity of $\mathcal{O}(I \cdot J \cdot T)$ where T is the number of time steps. This is the same complexity as for the Heat Equation, but the Big-O notation hides the constant factor which is 129 in Shallow Water equations versus 11 in the Heat Equation seen in Section 3.6. The $I \cdot J$ computations cannot be completely computed in parallel as we could in the Heat Equation. This is because the Lax-Wendroff half step need to be computed before the final step can be computed, and then we can update graphics and set boundary conditions. In the C++ implementation there is also some extra work to be done related to manually update the VBO, but it doesn't affect the complexity in the Big-O notation.

4.9 Efficiency, computations

To measure the scalability for the algorithm we increase the problem, but keep the amount of work per work item the same. This is a fair comparison since MATLAB does not allow us to control how many work items used on the GPU. We begin with leading dimension of 64 which gives us a total number of elements of $64 \cdot 64 = 4096$. We scale the problem linear in terms of total number of elements to compute, so the next leading dimensions will be $\sqrt{4096 \cdot 2} = 91$, $\sqrt{4096 \cdot 3} = 111$ until the leading dimension is 448. For each dimension, the program runs from $t = 1$ to $t = 1000$ and calculates minimum, maximum and average running time.

4.9.1 MATLAB CPU

As a reference, we begin to look at the running of MATLAB CPU implementation. If we look at Figure 4.13 we see that it scales very badly and the CPU is not suitable when the problem size

increases. In Section 4.8 we estimated the complexity to polynomial per time step and the graph in Figure 4.13 proves that claim. This is as expected since the CPU is only using one core.

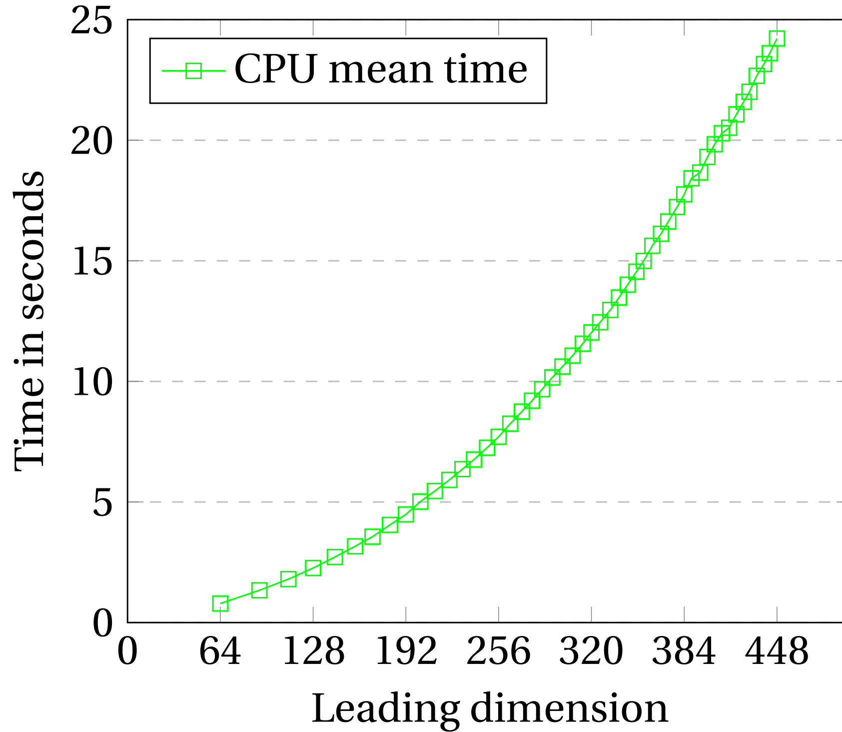

Figure 4.13: CPU running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.7.

4.9.2 MATLAB GPU

The GPU implementation runs the Lax-Wendroff half step and final step in parallel for each time step. However, the outermost loop which iterate over each time step runs sequentially on the CPU because we need to ensure that the previous time step is calculated before we move on to the next time step. In addition to that, we also need to set boundary conditions. For calculating the first 1000 times steps of the Shallow Water Equations, it takes just above seven second no matter how small the problem is. This is expected due to the GPU overhead and slow performance MATLAB has when performing loops. As we can see in the graph in Figure 4.14, there is almost no increased running time when we increase the size of the problem. In Section 4.8 we claimed that the complexity was $\mathcal{O}(I \cdot J)$ per time step, and a total complexity of $\mathcal{O}(I \cdot J \cdot T)$. Next, we claimed that the $I \cdot J$ computation could be done completely in parallel except for the

update of boundary conditions and updating of graphics. The graph in Figure 4.14 proves that claim. The small increase in running time is because as the size of the problem increase, there will be more synchronization and the GPU has to queue parts of the computations since the GPU has a limit of threads it can execute in parallel.

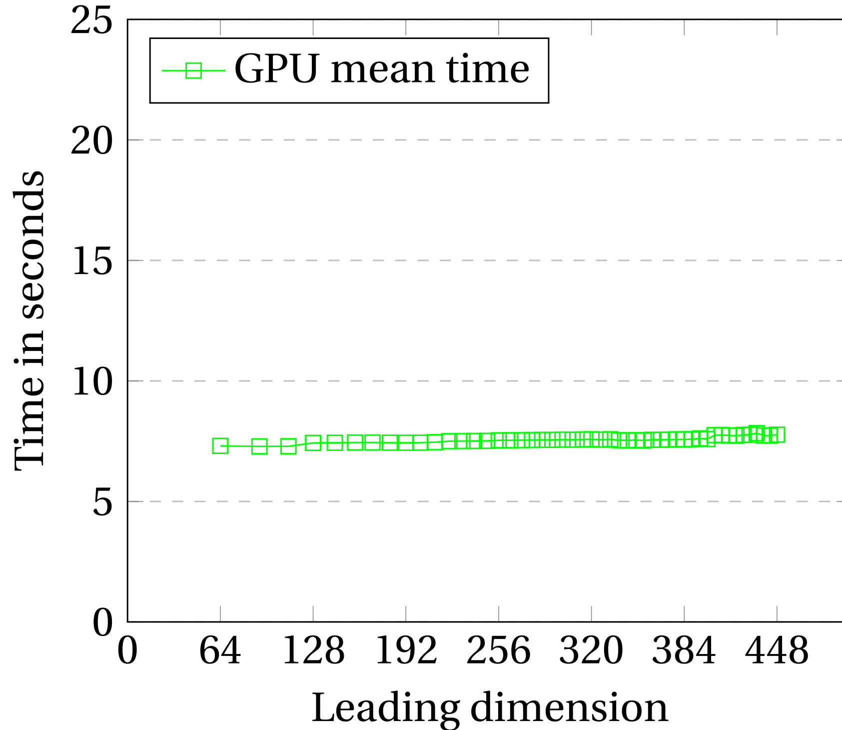

Figure 4.14: GPU running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.8.

4.9.3 C++/OpenCL

The GPU implementation using C++ and OpenCL has less overhead for utilizing the GPU than MATLAB and has much better performance as we can see in Figure 4.15. The complexity is the same as for the MATLAB implementation, $\mathcal{O}(I \cdot J)$ for each time step, as mention in Section 4.8. As we can see from Figure 4.15, there is almost no increase in runtime when we increase the size of the problem.

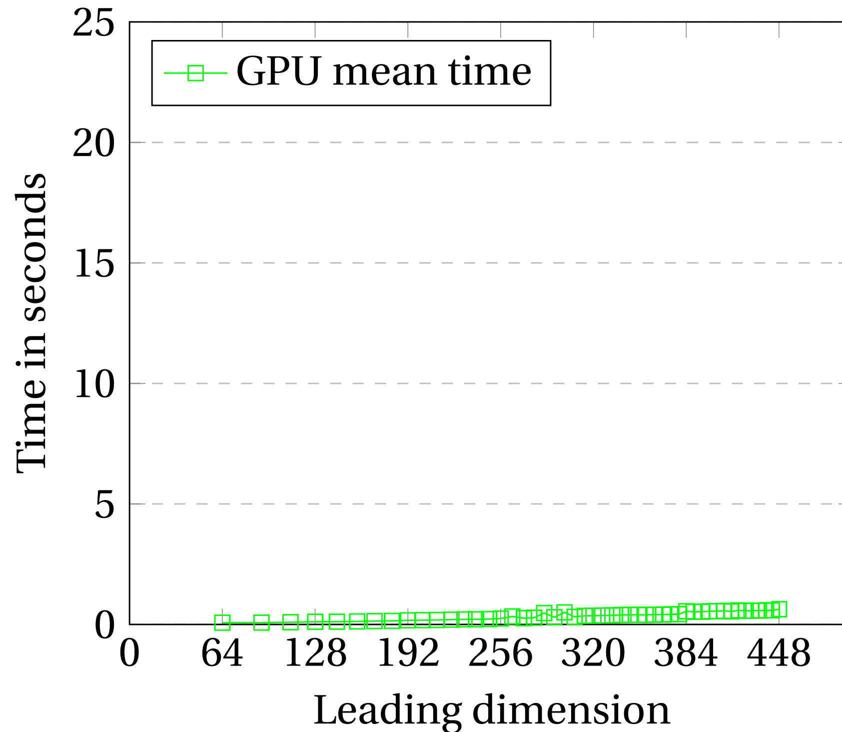

Figure 4.15: GPU running time in seconds, C++/OpenCL. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.9.

4.10 Efficiency, visualization

In this section, we will look at the total running time of the naive and adaptive implementation. The total running time is the running time for approximation of the Shallow Water Equation as we did in Section 4.9, together with visualization in real time. The benchmark is executed in the same manner as for computations only, explained in Section 4.9.

4.10.1 MATLAB CPU

As a reference, we begin to measure the running time for the CPU. By comparing the graph in Figure 4.16 with the graph in Figure 4.13 which is the same implementation without visualization, we get an impression of how much visualization is affecting the running time.

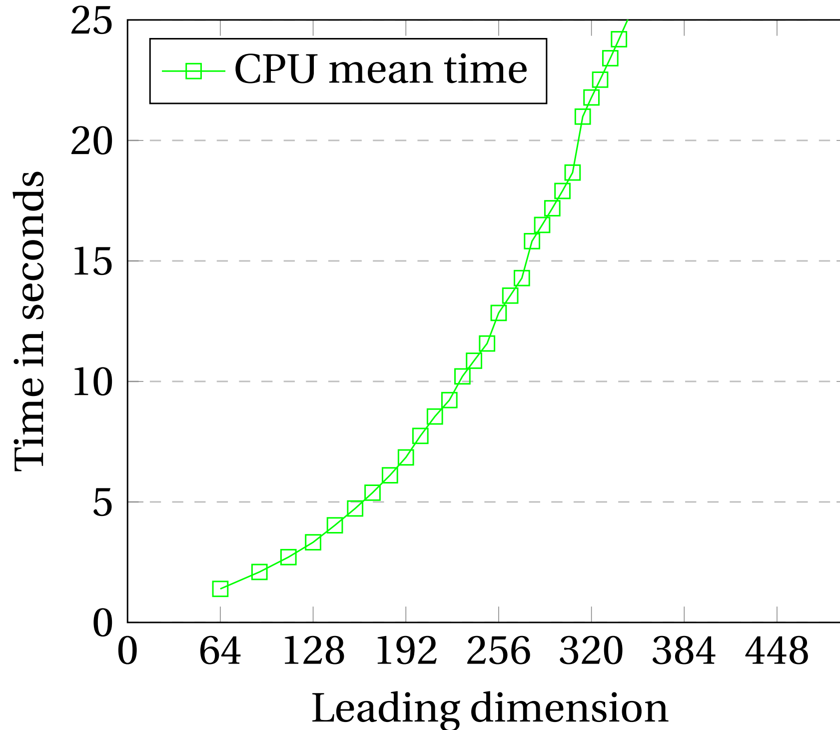

Figure 4.16: CPU with visualization, running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.10.

4.10.2 MATLAB GPU

If we compare the graph in Figure 4.17 with the Figure 4.14, we see that the running time with visualization is slower than computations only. It is also increases faster than for only computations. This is because MATLAB lacks support for interoperability between the data computed on the GPU and the data rendered on the GPU. For each time step, the GPU needs to send data to the CPU to render, instead of rendering it directly from the GPU like OpenCL and OpenGL does. This means that when the size of the problem increases, the amount of data to send for each time step is increased and therefore the communication time is increased.

4.10.3 C++/OpenCL/OpenGL

The adaptive implementation has less overhead to visualize, but there is still some overhead. If we compare Figure 4.18 we see that the running time is greater than in Figure 4.15 which is without visualization. This increased running time is because we need to create VBO and IBO

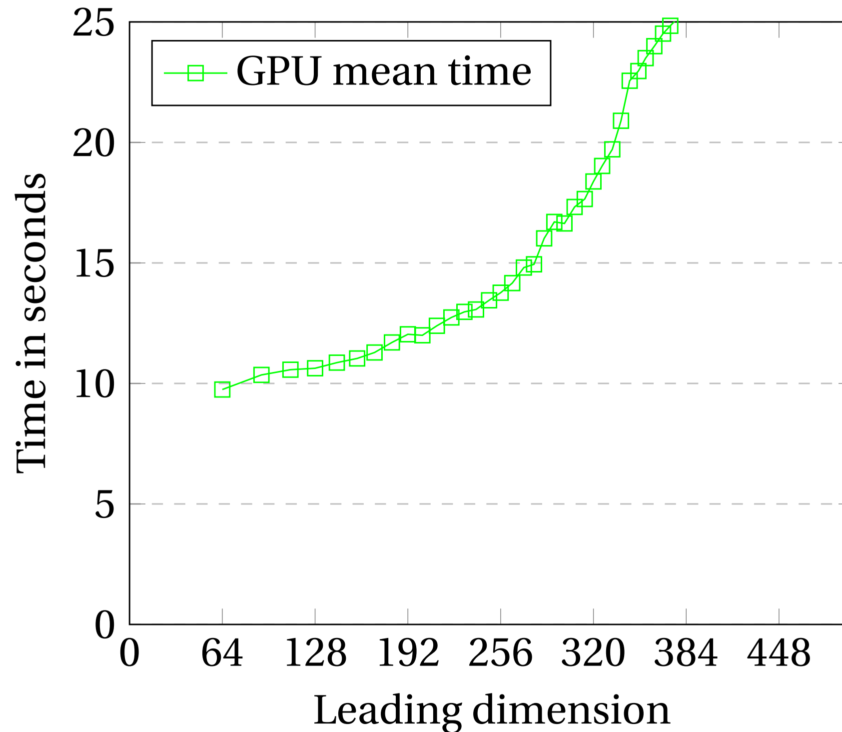

Figure 4.17: GPU with visualization running time in seconds, MATLAB. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.11.

for the mesh, we need to update the VBO according to the height calculated in the Lax-Wendroff method and OpenGL need to draw to the screen. If we add texture and water depth, there will be another few milliseconds added to the time.

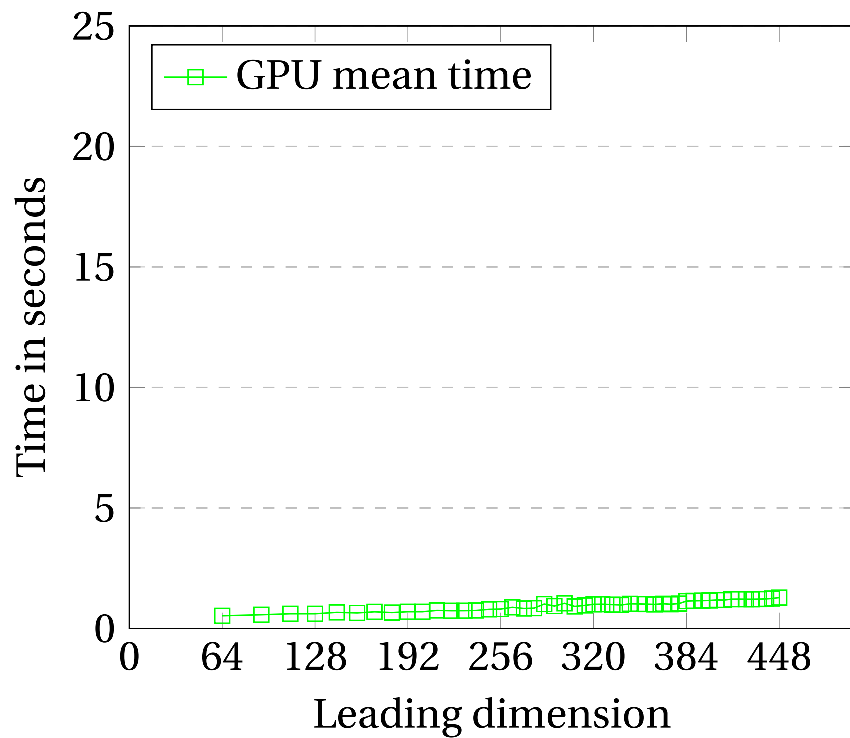

Figure 4.18: GPU with OpenGL running time in seconds, C++/OpenCL/OpenGL. Table overview for the data behind this graph along with minimum and maximum running time can be seen in Appendix B.12.

4.11 Analysis

What we have learned from this chapter is that the performance for utilizing the GPU is better in C++ together with OpenCL than using MATLAB together with MATLAB Parallel Computing Toolbox. There are two main reasons for this. First, for loops are slow in MATLAB compared to C++. MATLAB is optimized for performing operations on matrices and vectors, but in this algorithm we need to use one for loop for the time steps as described in Section 4.9.2. The other reason, is that we have no control over what is executed on the GPU by using MATLAB parallel computing toolbox. We basically write code to run on the CPU and say "run this on the GPU", without having any knowledge on how it is done. Since this is a general library which will work on any problem, it is not optimized for the given task. The OpenCL implementation on the other hand, is designed for this specific algorithm and is optimized for best performance. As an example, we claimed in Section 3.5.2 that different ordering of indexing gave different efficiency. In the last benchmark where the leading dimension was equal to 448, the average running time with OpenCL was about 0.63 seconds. If we were to change the order of indexing, the running time for this benchmark increased to about 4.5 seconds. This is because of cache swapping. In the fastest indexing, we are reading along the cache line, but in the slowest indexing we are swapping cache. We have full control over this in the OpenCL implementation, but we have no control over how this is done in the MATLAB implementation.

4.12 Quality and possibilities

Until now we have been looking at the running time and implementation of the naive and adaptive utilization of the GPU for calculating an approximation of the Shallow Water Equations. In this section, we will look at the naive and adaptive implementation for the graphical point of view. What we want to investigate is how easy is it to change from a console application with only text output to an application with graphical user interface. And, what are the possibilities if you put an extra effort into it.

4.12.1 MATLAB

Visualizing data in MATLAB is very easy. The only required line of code is

```
surfplot = surf(H)
```

where "surf" is the built in method for rendering a surface plot from the data stored in "H" which is our height matrix. For updating the surface plot, there is only required two additional lines of code:

```
surfplot.ZData = H; % or gather(H) if H is gpuArray.
drawnow
```

With only these three lines of code, you get the result as shown in Figure 4.19. If we want to make the visualization more realistic, the surf function has some variables that can be changed such as camera light, lightning, color of the edge and color of the faces. In our project, we changed the value of these variables so the model would look more like water.

```
surfplot.FaceColor = 'cyan';
surfplot.EdgeColor = 'none';
camlight headlight;
lighting phong
alpha(surfplot,0.8) & Set transparency
```

By this approach, the result looks like in Figure 4.20. Further, we can apply a texture to the surface with only four lines. By setting the variable FaceColor to texturemap, it will use the data in CData as face color.

```
texture = imread('water.jpg'); % Load the texture image
surfplot.FaceColor = 'texturemap';
surfplot.CData = texture;
surfplot.EdgeColor = 'none';
```

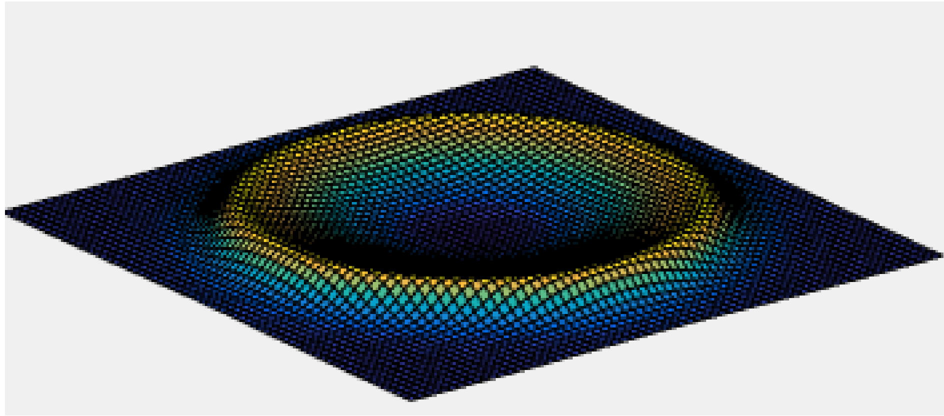

Figure 4.19: The simplest form for visualization in MATLAB.

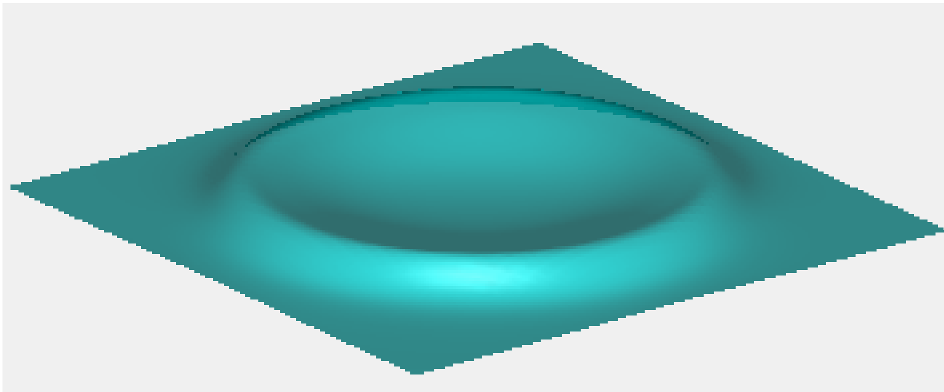

Figure 4.20: Visualization in MATLAB with cyan face color and none edge color.

The result will look like in Figure 4.21. If we want to improve it even more, we can create a box around the water area to make it look like the water is moving inside a box. This is done by creating 6 squares with the MATLABs commando called "fill3". The final result is shown in Figure 4.22.

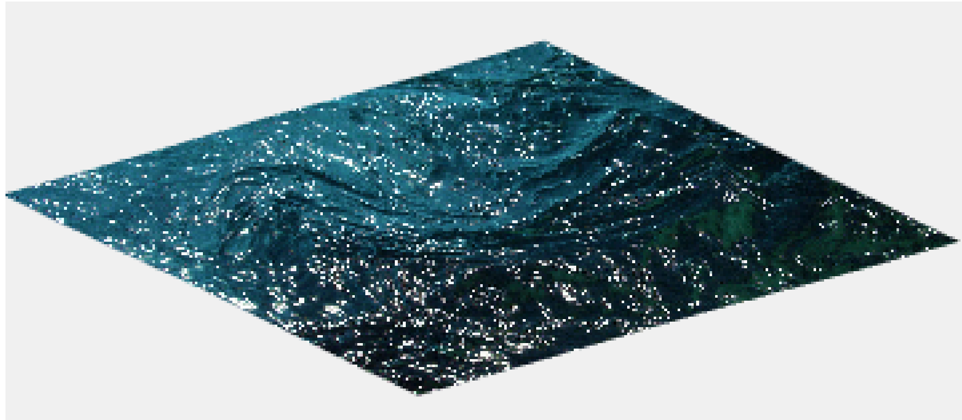

Figure 4.21: Visualization in MATLAB with texture.

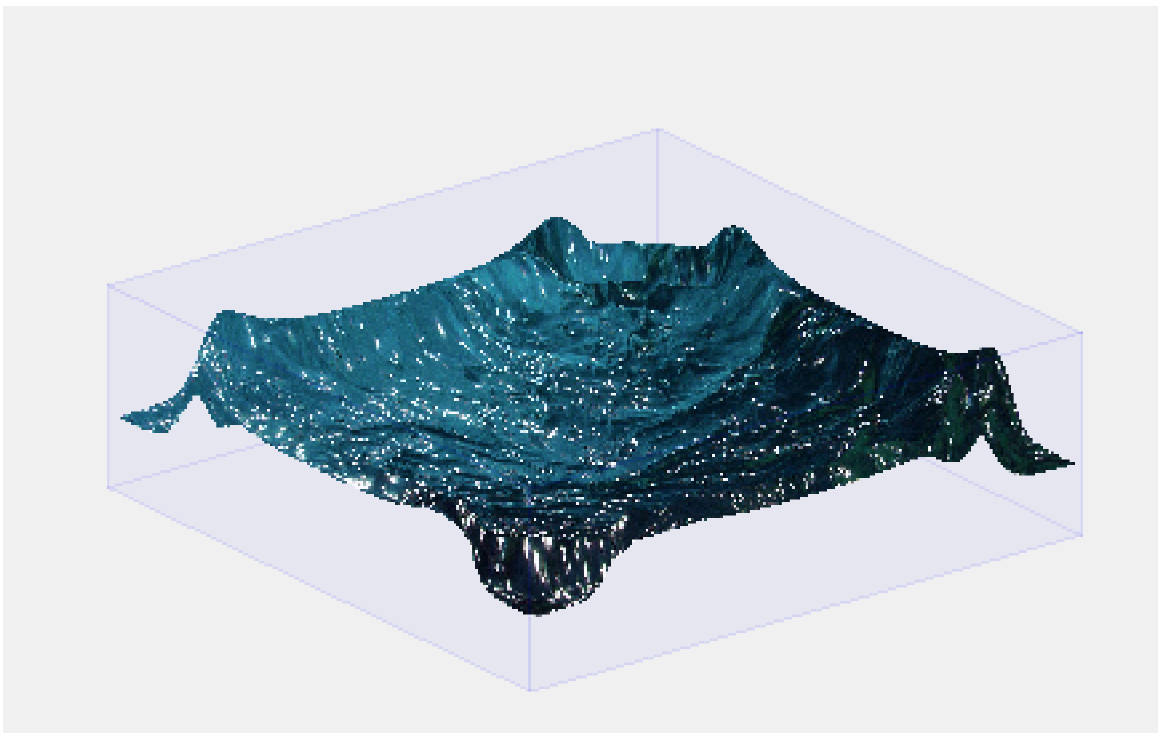

Figure 4.22: Visualization in MATLAB with texture inside a box.

4.12.2 OpenGL

Visualizing data in C++ together with OpenGL is more complicated than in MATLAB. First, we need to create a VBO as explained in Section 4.6.3. We only need the VBO to render the data, but as mentioned in Section 4.6.3, we will use less data if we also create an IBO. With VBO and IBO implemented, we can visualize with only these few lines.

```
glColor4f(r,g,b,a); // Red, green, blue and alpha
glPolygonMode(GL_FRONT_AND_BACK, GL_LINES);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer(4, GL_FLOAT,0,NULL);
glDrawElements(GL_TRIANGLE_STRIP, ((grid+2)*2+1)*2+((grid+2)-3)* \
        ((grid+2)*2+2), GL_UNSIGNED_INT, (void*)0);
```

With this minimal implementation, we will get a graphical representation like we see in Figure 4.23. For an even more realistic look, we can add water depth to the water. We were not able to do this in MATLAB, but in OpenGL we created triangle strip around the whole finite difference grid and then changed the height of the top of the triangle trip according to the wave height on the boundaries. The result is shown in Figure 4.24. If we want to improve the visualization, we can apply a texture to the model. In order to do this in C++/OpenGL, we need to load the image we want to use as texture, convert the image to RGBA format, generate texture from the converted image and apply correct texture wrapping so the texture fits our model. Next, we can apply a cube around the water area to make it look like the water is moving inside a cube as we did in the MATLAB implementation. This can be achieved by creating eight vertex, one for each corner, and creating a cube by OpenGL quads as shown in the code below followed by creating VBO and IBO for the quad in the same manner as for the waves. For more details, see Section 4.6.3. We can also add ground to the model by creating a triangle fan consisting of four vertices. A triangle fan is an OpenGL primitive to draw triangles in a fan shape where all triangles share one vertex. With these features, the models now look as shown in Figure 4.25. To improve the model even further, there is more opportunities in OpenGL such as shaders which we did not have time to implement due to limited time. With a shader we could have implemented

```
        float4 v1 = {0.0f, 0.0f, 0.0f, 1.0f};
        float4 v2 = {dim, 0.0f, 0.0f, 1.0f};
        float4 v3 = {dim, 0.0f, dim, 1.0f};
        float4 v4 = {0.0f, 0.0f, dim, 1.0f};
        float4 v5 = {0.0f, cubeHeight, 0.0f, 1.0f};
        float4 v6 = {dim, cubeHeight, 0.0f, 1.0f};
        float4 v7 = {dim, cubeHeight, dim, 1.0f};
        float4 v8 = {0.0f, cubeHeight, dim, 1.0f};
        float4 cube_data[quads] = {
                        v1, v2, v3, v4,
                        v5, v6, v7, v8,
                        v2, v3, v7, v6,
                        v3, v7, v8, v4,
                        v4, v8, v5, v1,
                        v1, v2, v6, v5
        };
```

light sources, light direction and reflection to make the water look very realistic. We could also implement rotation, camera and real time interaction to create new waves. The possibilities when using C++ and OpenGL for rendering is more or less endless. In the naive implementation on the other hand, we are limited to functionality provided by MATLAB.

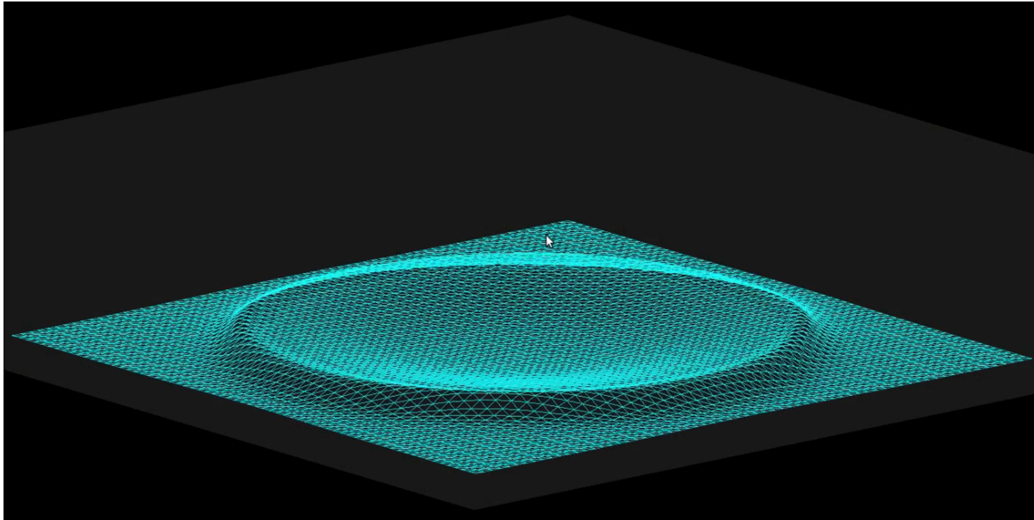

Figure 4.23: The simplest form for visualization with OpenGL where only the triangles strips are drawn with a color.

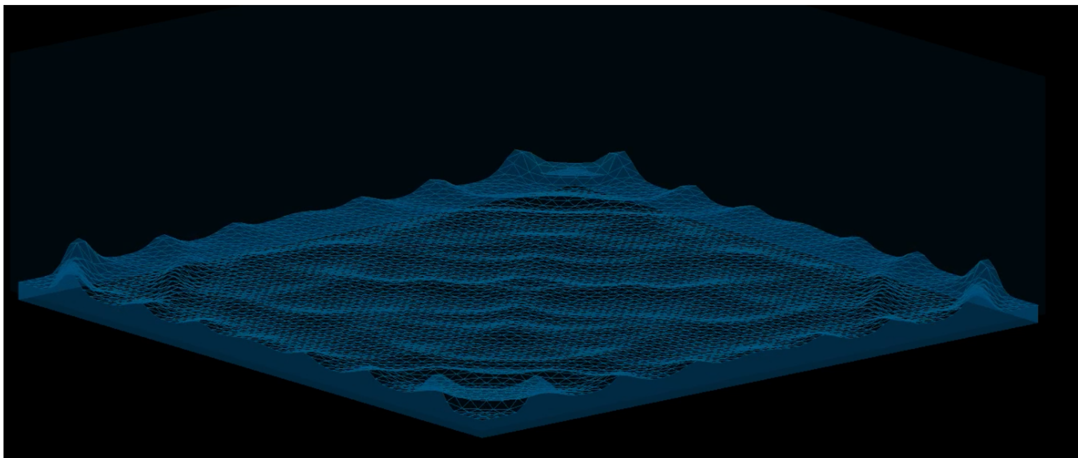

Figure 4.24: OpenGL mesh with water depth.

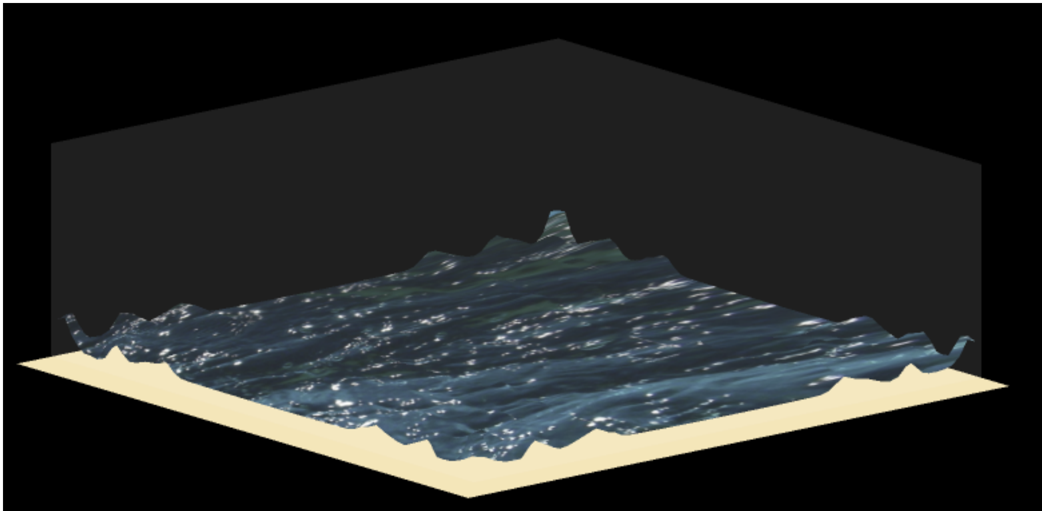

Figure 4.25: Illustration of OpenGL using texture and ground inside a box.

Chapter 5

Summary and Recommendations for Further Work

5.1 Summary and Conclusions

Utilizing the power of the GPU gives the opportunity to calculate large computationally intensive problems faster than only using the CPU due to its highly parallel structure and a greater number of cores. MATLAB provides us with a toolbox for executing code on the GPU without writing actual GPU code, allowing us to easily utilize the power of the GPU. On the over hand, the Khronos Group provides us with OpenCL which is a framework for writing programs to execute on, among other things, the graphics card.

In order to compare the naive utilization provided by MATLAB's parallel computing toolbox with the adaptive utilization by using OpenCL, we selected two partial differential equations (PDEs), The Heat Equation (Chapter 3) and The Shallow Water Equations (Chapter 4). In Chapter 3 and 4 we see that adaptive utilization of the GPU has much better performance than naive utilization, whether it is only for computations or both for computations and visualization. In the Analysis in Section 4.11 we concluded with three main reasons for the naive implementation being slower than the adapted implementation. First, for loops are slow in MATLAB compared to C++. MATLAB is optimized for performing operations on matrices and vectors, but in this algorithm we need to use one for loop for the time steps as described in Section 4.9.2. The other reason, is that we have no control over what is executed on the GPU by using MATLAB parallel

computing toolbox. We basically write code to run on the CPU and say "run this on the GPU", without having any knowledge on how its done. Since this is a general library which will work on any problem, it is not optimized for the given task. The OpenCL implementation on the other hand, is designed for this specific algorithm and is optimized for best performance. As an example, we claimed in Section 3.5.2 that different ordering of indexing gave different efficiency. In the last benchmark where the leading dimension was equal to 448, the average running time with OpenCL was about 0.63 seconds. If we were to change the order of indexing, the running time for this benchmark increased to about 4.5 seconds. This is because of cache swapping. In the fastest indexing, we are reading along the cache line, but in the slowest indexing we are swapping cache. We have full control over this in the OpenCL implementation, but we have no control how this is done in the MATLAB implementation. The last reason is that MATLAB has no support for interoperability between the GPU memory used for computations and the GPU memory used for visualization as described in Section 3.4.2. On the other hand, implementing the naive implementation in MATLAB takes less time and requires less knowledge about GPU-programming and programming in general. As we can see in Section 4.5, utilizing the GPU in MATLAB only requires a few lines of code, but the utilizing of GPU using C++ and OpenCL requires a lot of code as seen in Section 4.6.
Visualizing the results from the PDEs is very simple in MATLAB. From Section 4.12.1, we can see that creating a mesh and visualizing it requires only a few lines. Adding custom colors at edges and faces, adding lightning or texture is also done in a few additional lines. In OpenGL, the mesh and everything else needed for visualization needs to be specified in code by the programmer, see Chapter 4.12.2 for all the details. However, the adapted implementation has better performance, and more possibilities such as shaders where we can implement real time rendering with interactions.
In Chapter 1, we wanted to answer the following questions:

- Will an adaptive implementation of fluid simulation be faster than a naive implementation? And if so, how much faster will it be?
- In what situations is doing an adaptive implementation worth the increased effort?

For the first question, the answer is yes, the adaptive implementation of fluid simulation is

faster than the naive implementation. How much faster depends on the size of the problem , how computationally intensive the problem is and how complex the graphics are. Usually some place between ten and twenty times as fast, as we can see in Section 4.9 and Section 4.10. For the second question, the answer is not so clear. To choose which strategy to use for utilizing the GPU depends on how we weigh implementation time against performance. If we want to get results fast and the size of the problem is not too large (or too small where CPU is more suitable), the naive utilization of the GPU is probably the best choice. If we want high performance, use large datasets or want a more sophisticated solution in terms of graphics and functionality, the adapted utilization is probably the best choice.

If we look at the paper from SINTEF [1], they got a speedup between 9.53 and 30.6 depending on scheme and size of the problem. As we can see from Figures 5.1 and 5.2, our implementation of The Shallow Water Equations performs similarly for the adapted implementation which is also used in [1]. However, it is worth mentioning that we use different hardware and different numerical schemes.

LD	MATLAB CPU	Naive GPU	Adapted GPU	Speedup Naive	Speedup Adapted
64	0.790825	7.307220	0.073840	0.108	10.709
128	2.261987	7.430279	0.111032	0.304	13.299
192	4.485078	7.434853	0.170077	0.603	26.370
256	7.705122	7.536320	0.245257	1.022	31.416
320	12.029904	7.577522	0.361643	1.587	33.264
384	17.759003	7.573304	0.545613	2.344	32.548
448	23.956670	7.771071	0.632402	3.082	37.882

Figure 5.1: Table shows speedup for the naive and adapted implementation applied at The Shallow Water Equations without visualization, compared to the CPU

LD	MATLAB CPU	Naive GPU	Adapted GPU	Speedup Naive	Speedup Adapted
64	1.393095	9.747331	0.527130	0.142	2.642
128	3.326229	10.630329	0.610865	0.312	5.445
192	6.847627	12.039569	0.694933	0.568	9.853
256	12.842405	13.765078	0.806527	0.932	15.923
320	21.781746	18.379540	1.004061	1.185	21,693
384	34.758954	26.369500	1.137292	1.318	30.562
448	56.912076	39.173698	1.281585	1.452	44.407

Figure 5.2: Table shows speedup for the naive and adapted implementation applied at The Shallow Water Equations with visualization, compared to the CPU

5.2 Recommendations for Further Work

The adaptive implementation is not using shaders. The end result will probably look better with vertex and fragment shaders. By using shaders we have more opportunities such as navigation, rotation and better graphics. For instance a water shader for the Shallow Water Equation and a glass shader for the box around will strengthen the conclusion of the adaptive implementation having more opportunities.
MATLAB is probably utilizing the GPU in the most naive way, and C++ together with OpenCL is probably the most adaptive way to utilize the power of the GPU. It would be interesting to compare with other languages in-between, such as Java, Python or C#.
The Shallow Water Equation is solved using the Lax-Wendroff method which is based on finite differences. Solving the equations using finite volume or finite element which is based on integration, has in theory smaller errors, but is slower. It would be interesting to see how finite element or finite volume is compared to finite differences.

Appendix A

Acronyms

API Application programming interface

CFL Courant, Freidrichs and Lewy

CPU Central processing unit

CUDA Compute unified device architecture

FPGA Field-programmable gate array

FTCS Forward differences for time and central differences for space

GPGPU General-purpose computing on graphics processing units

GPU Graphics processing unit

IBO Index buffer object

LD Leading dimension

MATLAB Matrix laboratory

MPI Message passing interface

OpenCL Open computing language

OpenGL Open graphics library

PDE Partial differential equation

VBO Vertex buffer object

Appendix B

Additional Information

All benchmark results are listed here.

B.1 Chapter 3 - The Heat Equation

B.1.1 Computations

MATLAB CPU

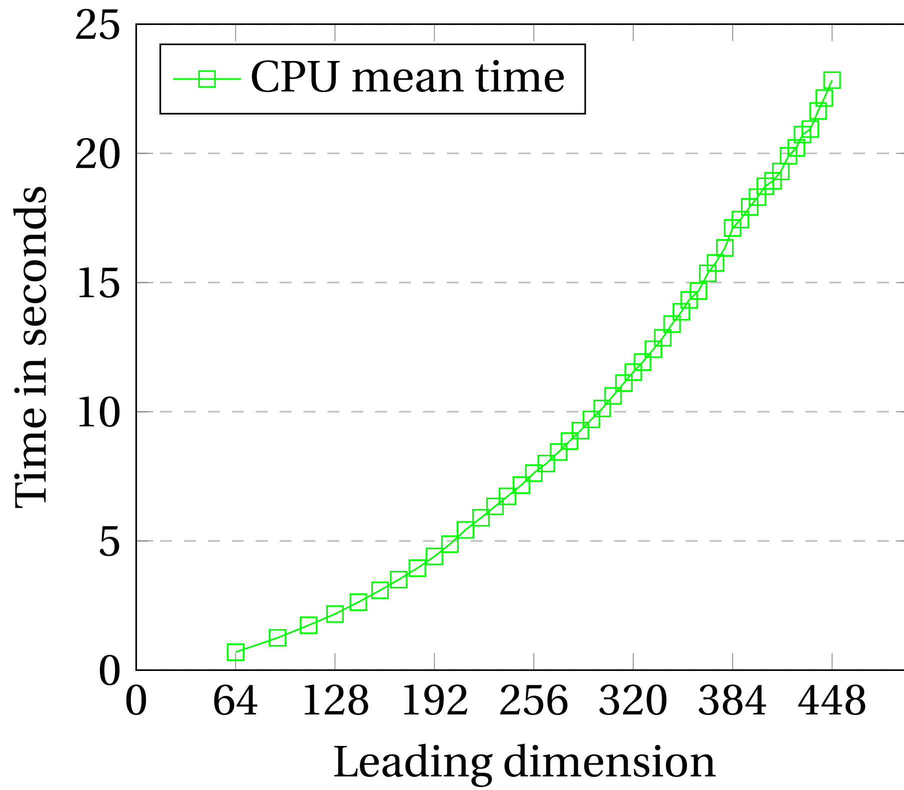

Figure B.1: CPU running time in seconds, MATLAB

LD	Min	Max	Average
64	0.687876	0.906292	0.697881
91	1.232159	1.413995	1.251367
111	1.716736	1.880310	1.738099
128	2.139152	2.409077	2.173245
143	2.576562	2.913656	2.632191
157	3.037950	3.294313	3.089903
169	3.449664	3.682595	3.503841
181	3.884133	4.147580	3.948325
192	4.332266	4.651995	4.400938
202	4.795285	5.112785	4.876478
212	5.325148	5.687581	5.429426
222	5.779985	6.108086	5.897436
231	6.202566	6.506916	6.336404
239	6.594280	6.877742	6.725653
248	7.030091	7.379891	7.162602
256	7.460261	7.811220	7.626614
264	7.873873	8.178861	7.996135
272	8.286943	8.677318	8.446167
279	8.673826	9.058606	8.865754
286	9.074101	9.466889	9.269473
293	9.517744	10.382262	9.703187
300	9.959053	10.358181	10.128068
307	10.431155	10.840984	10.616438
314	10.905756	11.312168	11.111190

LD	Min	Max	Average
320	11.345290	11.772564	11.534710
326	11.728291	12.165450	11.922314
333	12.230083	12.914299	12.418683
339	12.625015	13.107550	12.864578
345	13.156274	13.607175	13.393305
351	13.667685	14.202874	13.871387
356	14.093860	14.654388	14.330524
362	14.444694	14.978068	14.672812
368	15.144922	15.594408	15.357187
373	15.458007	20.512565	15.762641
379	16.063787	16.579764	16.339950
384	16.803930	17.384855	17.113330
389	17.219419	17.736175	17.438377
395	17.431749	18.228851	17.928098
400	17.866152	18.755340	18.304785
405	18.407309	19.075292	18.728136
410	18.410355	19.320577	18.938974
415	18.917833	19.626847	19.299717
420	19.587051	20.207581	19.908749
425	19.882440	20.570784	20.214973
429	20.275744	21.133199	20.728259
434	20.528407	21.547992	20.939826
439	21.190352	22.092520	21.640842
443	21.662584	22.614652	22.141162
448	22.412392	23.216990	22.833517

MATLAB GPU

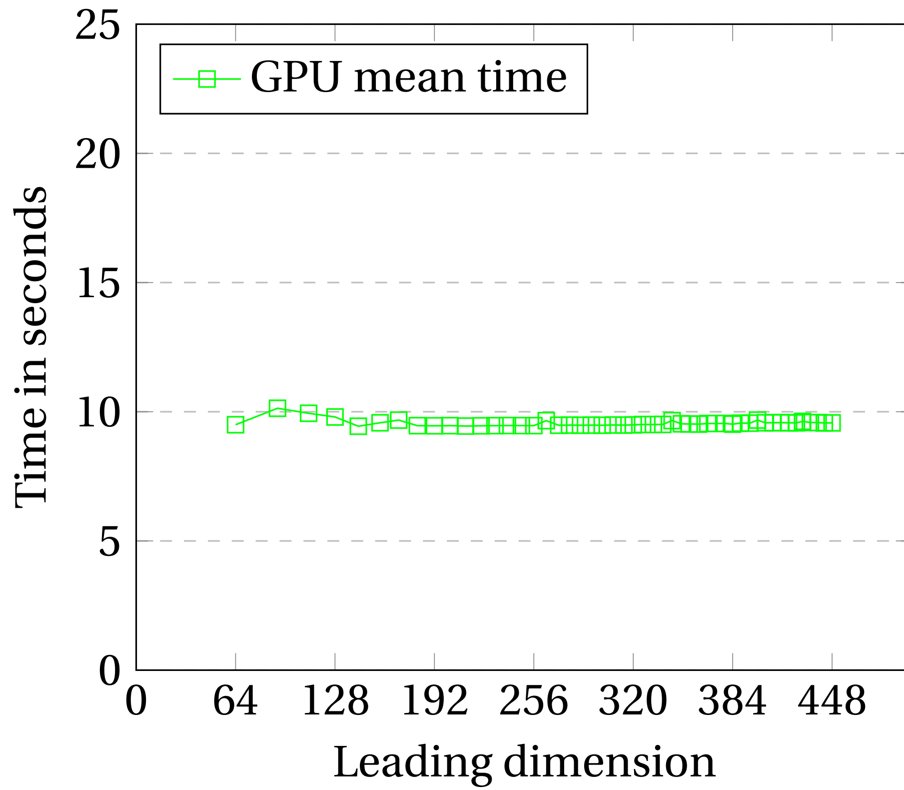

Figure B.2: GPU running time in seconds, MATLAB

LD	Min	Max	Average
64	9.168378	11.112637	9.497189
91	9.201793	11.278203	10.135158
111	9.263353	10.687552	9.941215
128	9.398645	10.791539	9.797757
143	9.410197	9.483061	9.439549
157	9.413935	10.783292	9.572820
169	9.418577	11.406564	9.676754
181	9.435633	10.133362	9.461210
192	9.426423	9.511720	9.456522
202	9.429681	9.539958	9.461757
212	9.410715	9.518909	9.446407
222	9.429266	9.560585	9.456314
231	9.431518	9.502966	9.461864
239	9.433730	9.524663	9.465912
248	9.429027	9.611642	9.463916
256	9.431378	9.558531	9.462818
264	9.438955	11.257232	9.655795
272	9.438450	9.599019	9.477305
279	9.443306	9.615896	9.482346
286	9.429282	9.562545	9.478632
293	9.417308	9.601949	9.481534
300	9.423775	9.577232	9.477486
307	9.426099	9.596461	9.487675
314	9.425215	9.559201	9.483138

LD	Min	Max	Average
320	9.440267	9.574373	9.492099
326	9.448876	9.713091	9.507183
333	9.455866	9.635001	9.506195
339	9.460980	9.574048	9.510173
345	9.453393	9.975270	9.659012
351	9.484721	9.635289	9.533070
356	9.465908	9.620872	9.526303
362	9.455898	9.603753	9.520987
368	9.489026	9.624160	9.544058
373	9.482587	9.611723	9.544782
379	9.490866	9.621078	9.545982
384	9.459508	9.631878	9.520325
389	9.510087	9.651078	9.558204
395	9.518561	9.630701	9.558826
400	9.511700	10.019711	9.677754
405	9.511849	9.683312	9.572264
410	9.507672	9.651183	9.574972
415	9.523127	9.655702	9.581555
420	9.512057	9.677436	9.570621
425	9.519142	9.672046	9.571163
429	9.525132	9.950608	9.627690
434	9.528928	9.700273	9.583672
439	9.526036	9.674899	9.580348
443	9.515020	9.649683	9.566199
448	9.507587	9.652887	9.569150

C++/OpenCL

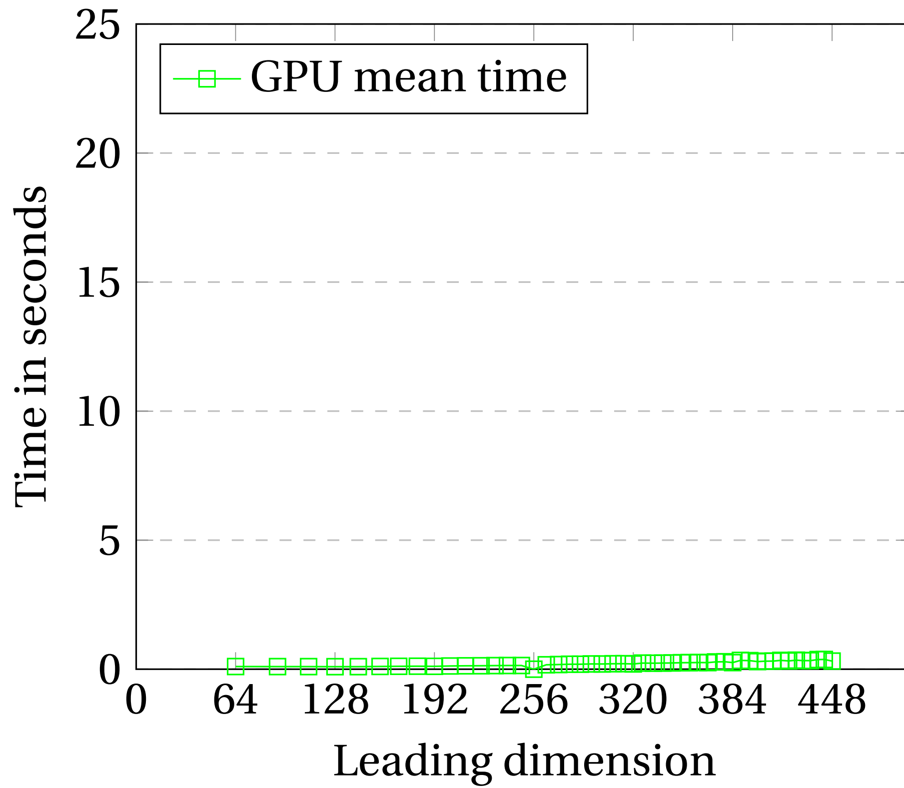

Figure B.3: GPU running time in seconds, C++/OpenCL

LD	Min	Max	Average
64	0.099981	0.210980	0.103016
91	0.098113	0.151312	0.96563
111	0.096563	0.122483	0.100141
128	0.095524	0.110083	0.097359
143	0.095941	0.123678	0.098303
157	0.104700	0.116328	0.106596
169	0.110334	0.134888	0.112336
181	0.110234	0.137815	0.115905
192	0.108714	0.122957	0.111317
202	0.121503	0.164831	0.124240
212	0.128683	0.171282	0.130941
222	0.131142	0.159846	0.133328
231	0.136206	0.149404	0.138958
239	0.139750	0.184004	0.142429
248	0.140717	0.173624	0.142958
256	0.140561	0.168655	0.143167
264	0.168334	0.183440	0.170863
272	0.177856	0.235374	0.180172
279	0.189811	0.221701	0.192464
286	0.190652	0.225642	0.192834
293	0.201304	0.238277	0.203561
300	0.199536	0.234204	0.201737
307	0.207148	0.276963	0.210908
314	0.208390	0.278242	0.212714

LD	Min	Max	Average
320	0.198299	0.232667	0.203053
326	0.231978	0.262841	0.236307
333	0.229453	0.266112	0.232488
339	0.233850	0.273308	0.236935
345	0.239010	0.277902	0.241874
351	0.243015	0.283016	0.251635
356	0.251570	0.269992	0.253235
362	0.254941	0.298686	0.258325
368	0.252061	0.297162	0.254943
373	0.277752	0.325364	0.282181
379	0.283104	0.360959	0.288120
384	0.252135	0.291157	0.255498
389	0.337955	0.385383	0.340925
395	0.321435	0.374038	0.331808
400	0.290393	0.336347	0.293963
405	0.308006	0.353503	0.311155
410	0.308576	0.354621	0.313592
415	0.337998	0.389944	0.342187
420	0.315309	0.363134	0.320586
425	0.346065	0.393194	0.349472
429	0.329116	0.390644	0.333310
434	0.328384	0.376267	0.331439
439	0.368945	0.439969	0.373795
443	0.373370	0.418972	0.377495
448	0.310175	0.352188	0.314576

B.1.2 Visualization

MATLAB CPU

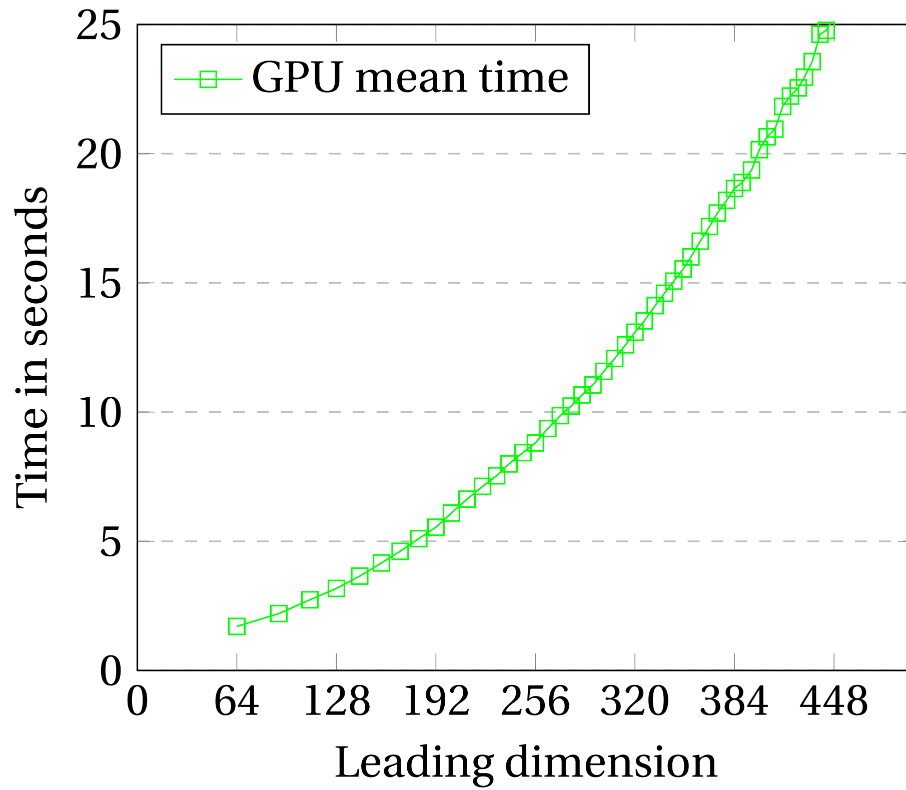

Figure B.4: CPU with visualization running time in seconds, MATLAB

LD	Min	Max	Average
64	1.680095	1.860101	1.704836
91	2.172241	2.307463	2.201107
111	2.689591	2.879469	2.736130
128	3.113623	3.412568	3.173873
143	3.579883	3.793399	3.652112
157	4.064342	4.321199	4.161491
169	4.512637	4.776922	4.611622
181	4.914774	5.299418	5.101754
192	5.387673	5.790275	5.543992
202	5.978273	6.356681	6.094159
212	6.461813	6.835345	6.629179
222	6.959969	7.370410	7.124403
231	7.340688	7.671703	7.533767
239	7.848844	8.132027	7.994585
248	8.207012	8.612991	8.433106
256	8.641801	8.968628	8.805169
264	9.153217	9.618209	9.364258
272	9.647241	10.023264	9.862065
279	9.975746	10.441675	10.230146
286	10.424740	10.813807	10.665084
293	10.888408	11.307610	11.050144
300	11.311404	11.787168	11.573141
307	11.808519	12.295627	12.074088
314	12.378496	12.796458	12.603523

LD	Min	Max	Average
320	12.811696	13.315188	13.088973
326	13.317138	13.721672	13.529384
333	13.849536	14.408056	14.120140
339	14.284632	14.845198	14.598017
345	14.798841	15.313897	15.065979
351	15.245544	15.788919	15.539104
356	15.704320	16.354232	16.003932
362	16.208663	16.953705	16.612970
368	16.957361	17.520848	17.182394
373	17.347997	18.087725	17.701993
379	17.761639	18.477622	18.195006
384	18.300165	19.071753	18.653093
389	18.545919	19.221040	18.887779
395	19.042168	19.760407	19.367010
400	19.728414	20.741642	20.155373
405	20.317939	21.041440	20.653323
410	20.533534	21.288415	20.950250
415	21.240455	22.352553	21.827725
420	21.767289	22.563744	22.229421
425	22.075928	22.874724	22.546990
429	22.342121	23.510568	22.960512
434	23.230426	24.497664	23.562031
439	23.651877	25.766525	24.621850
443	24.255450	25.145482	24.771824
448	24.805994	26.361463	25.276930

MATLAB GPU

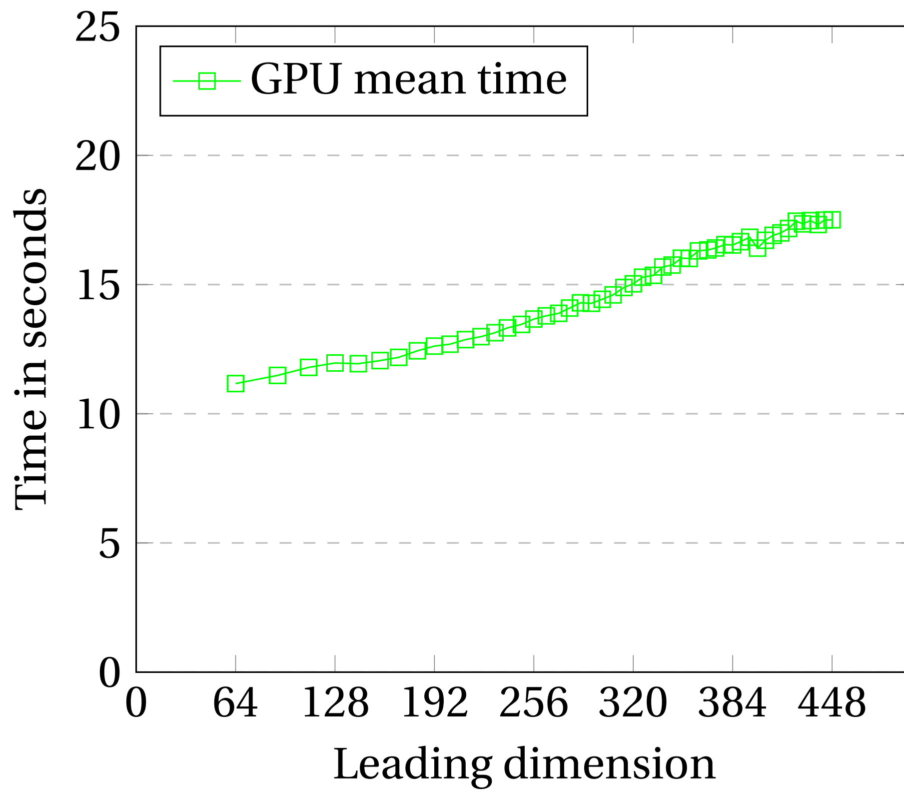

Figure B.5: GPU running time in seconds, MATLAB

LD	Min	Max	Average
64	10.957753	11.395120	11.167792
91	11.171155	11.903500	11.482932
111	11.597856	12.085091	11.793726
128	11.751246	12.448680	11.963746
143	11.744750	12.209192	11.937279
157	11.965880	12.189344	12.057303
169	12.035747	12.300323	12.182088
181	12.265261	12.898050	12.435678
192	12.422780	12.909836	12.617912
202	12.551325	13.062094	12.692028
212	12.749703	13.042949	12.869047
222	12.683414	13.395357	12.983951
231	12.947869	13.517898	13.136493
239	13.107646	13.773229	13.322293
248	13.307352	14.025498	13.457161
256	13.451788	14.155873	13.664992
264	13.515980	14.207783	13.788742
272	13.709230	14.057867	13.886638
279	13.927480	14.447089	14.090276
286	14.076704	14.587779	14.293196
293	14.018541	14.560006	14.274448
300	14.160864	14.583412	14.429150
307	14.411120	14.735364	14.600250
314	14.581036	15.200902	14.886609

LD	Min	Max	Average
320	14.791554	15.131971	15.028989
326	14.832443	15.469440	15.284568
333	15.138476	15.520897	15.356255
339	15.603361	16.040788	15.679269
345	15.627603	15.932622	15.755016
351	15.832493	16.130649	16.020110
356	15.952460	16.107467	16.009689
362	16.262301	16.454254	16.302486
368	16.232359	16.565140	16.339747
373	16.189166	16.705049	16.414802
379	16.422366	16.637742	16.546094
384	16.387091	16.704184	16.531557
389	16.418553	17.016065	16.660863
395	16.423268	17.039509	16.830746
400	16.240581	16.746847	16.409703
405	16.259382	16.964330	16.712538
410	16.791284	17.060006	16.905004
415	16.689074	17.207563	16.997841
420	17.057652	17.525382	17.168803
425	17.326521	17.641975	17.453297
429	17.236112	17.480448	17.348304
434	17.387958	17.842849	17.474797
439	16.923701	17.659438	17.323850
443	17.179448	17.837805	17.494022
448	17.266711	17.677188	17.507362

C++/OpenCL/OpenGL

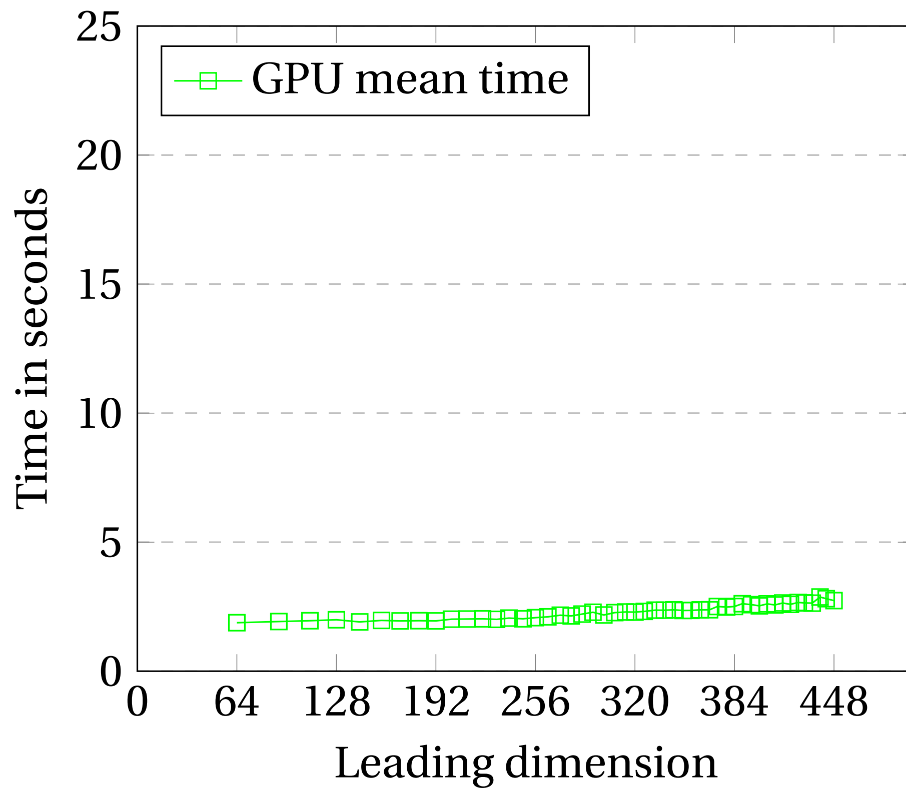

Figure B.6: GPU running time in seconds, C++/OpenCL/OpenGL

LD	Min	Max	Average
64	1.848960	1.911900	1.878977
91	1.851202	2.059207	1.924244
111	1.855998	2.067342	1.955536
128	1.951365	2.051701	1.992943
143	1.852308	2.006516	1.909131
157	1.849290	2.114415	1.967812
169	1.886387	2.061686	1.947584
181	1.884370	2.099298	1.956635
192	1.868516	2.018703	1.948769
202	1.953896	2.087390	2.015788
212	1.954736	2.085877	2.021144
222	1.951586	2.272688	2.028186
231	1.951336	2.127184	2.009297
239	1.987067	2.289036	2.057732
248	1.980000	2.123816	2.029942
256	2.018136	2.171681	2.074438
264	2.068322	2.135143	2.106128
272	2.111942	2.235029	2.170038
279	2.084901	2.241678	2.141609
286	2.134884	2.280205	2.212329
293	2.201513	2.329288	2.284114
300	2.130277	2.240338	2.177518
307	2.201636	2.377544	2.270477
314	2.151617	2.669350	2.293954

LD	Min	Max	Average
320	2.234477	2.528283	2.289417
326	2.251782	2.443672	2.317059
333	2.268557	2.451199	2.365521
339	2.268810	2.502238	2.362940
345	2.301761	2.418351	2.376977
351	2.285152	2.408036	2.352830
356	2.306361	2.473463	2.356175
362	2.343081	2.416953	2.375263
368	2.318470	2.440710	2.380729
373	2.392620	2.681269	2.506911
379	2.434911	2.526926	2.485108
384	2.417764	2.688472	2.508011
389	2.585187	2.635289	2.614389
395	2.521313	2.618471	2.578274
400	2.468638	2.599853	2.532711
405	2.534737	2.789959	2.608265
410	2.504008	2.611525	2.567596
415	2.568535	2.811574	2.642444
420	2.552209	2.665199	2.590283
425	2.605834	2.714257	2.664489
429	2.598276	2.681955	2.645576
434	2.589238	2.688489	2.636451
439	2.702037	3.035608	2.874139
443	2.723424	3.036392	2.809630
448	2.653682	2.884327	2.735545

B.2 Chapter 4 - The Shallow Water Equations

B.2.1 Computations

MATLAB CPU

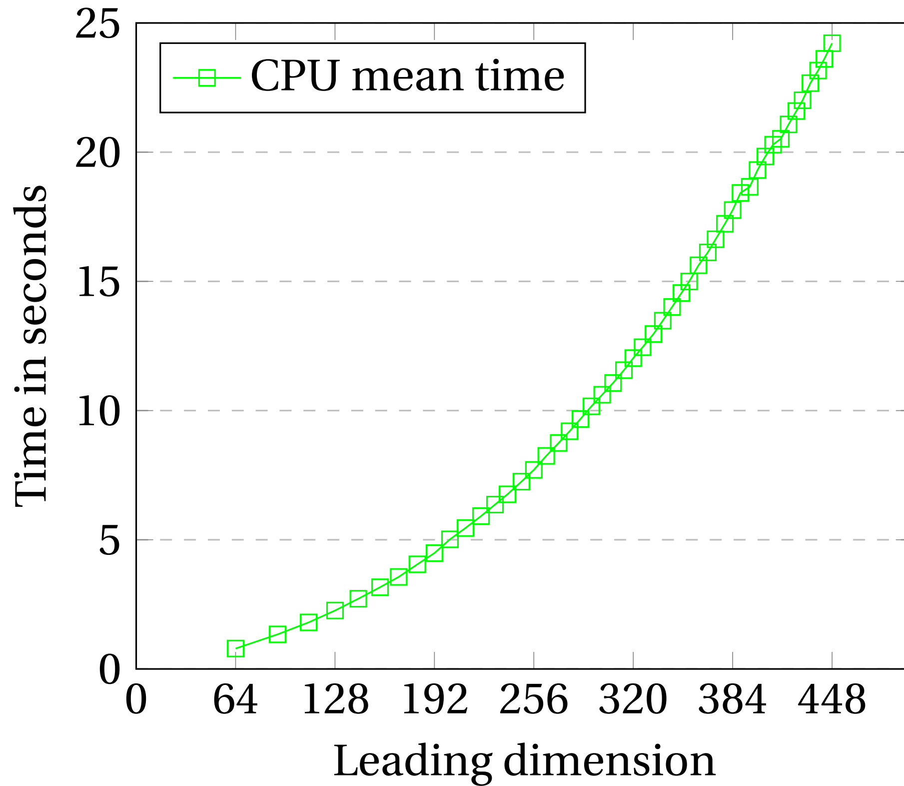

Figure B.7: CPU running time in seconds, MATLAB.

LD	Min	Max	Average
64	0.778130	1.107047	0.790825
91	1.332367	1.466516	1.338651
111	1.780873	2.128833	1.802114
128	2.224564	2.417692	2.261987
143	2.683396	2.870114	2.718367
157	3.125868	3.308465	3.163450
169	3.515143	3.745452	3.562941
181	3.989791	4.346371	4.050245
192	4.427457	4.748058	4.485078
202	4.942034	5.228336	5.016982
212	5.396830	5.694408	5.456509
222	5.851350	6.065590	5.916996
231	6.286891	6.512108	6.362316
239	6.664724	7.029963	6.759603
248	7.181043	7.395416	7.249842
256	7.619462	7.913216	7.705122
264	8.139884	8.408580	8.247762
272	8.658253	8.953282	8.748243
279	9.109247	9.650796	9.200068
286	9.594840	9.881662	9.674216
293	10.060891	10.378906	10.163672
300	10.516169	10.810666	10.613114
307	11.004180	11.261381	11.068762
314	11.492241	11.762505	11.563839

LD	Min	Max	Average
320	11.911257	12.309029	12.029904
326	12.357118	12.668882	12.452101
333	12.882743	13.125817	12.962349
339	13.394780	13.772387	13.478613
345	13.942815	14.238900	14.010976
351	14.466833	14.787966	14.549361
356	14.887884	15.220189	14.995922
362	15.479939	15.993804	15.623428
368	15.920765	16.471934	16.121683
373	16.410583	17.332362	16.629638
379	16.937582	17.551588	17.230387
384	17.518163	18.113825	17.759003
389	18.043548	20.013538	18.422206
395	18.536497	18.871420	18.661151
400	19.065936	19.596983	19.309138
405	19.682766	20.000557	19.833364
410	20.070482	20.469260	20.291101
415	20.295840	20.940479	20.519905
420	20.850760	21.475782	21.075135
425	21.373488	22.010929	21.591573
429	21.863033	22.262053	22.008739
434	22.424601	23.116170	22.671270
439	22.938088	23.482657	23.159982
443	23.363140	23.965524	23.611054
448	23.956670	24.648032	24.217239

MATLAB GPU

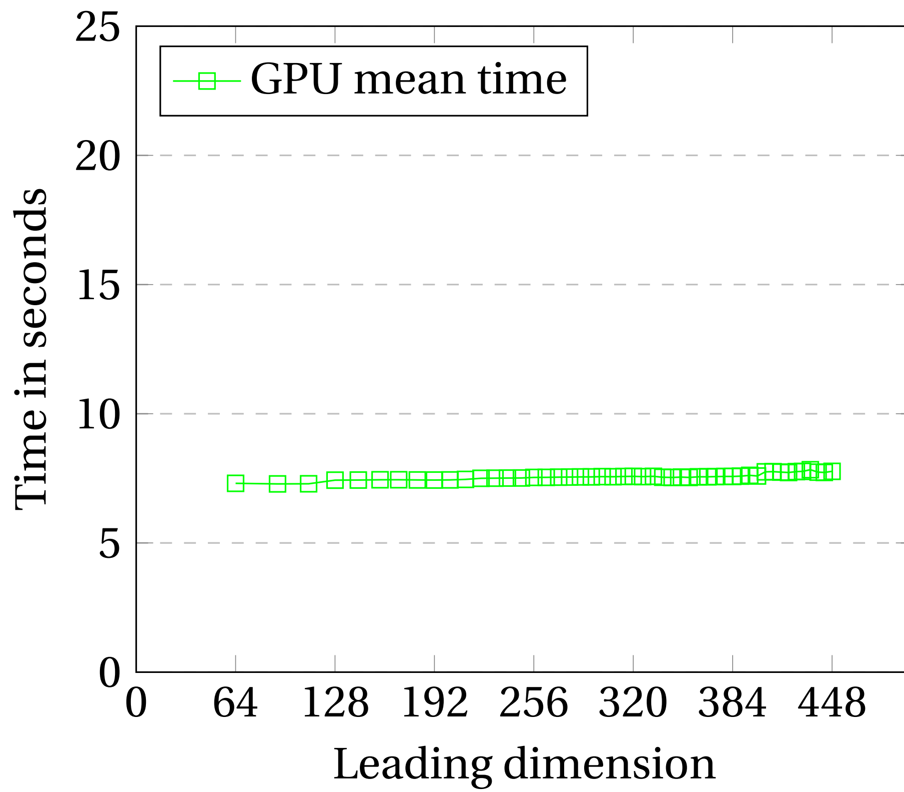

Figure B.8: GPU running time in seconds, MATLAB.

LD	Min	Max	Average
64	7.184199	7.654528	7.307220
91	7.255161	7.559784	7.285074
111	7.264392	7.616637	7.289568
128	7.405377	7.479791	7.430279
143	7.409890	7.469318	7.434790
157	7.411395	7.624724	7.444933
169	7.418935	7.522653	7.446091
181	7.402367	7.694282	7.438566
192	7.406766	7.518185	7.434853
202	7.415011	7.481751	7.439769
212	7.413329	7.657595	7.459755
222	7.476585	7.541621	7.501925
231	7.476278	7.808083	7.506655
239	7.483085	7.561626	7.511498
248	7.483096	7.731924	7.514677
256	7.510910	7.575409	7.536320
264	7.514091	7.575964	7.538723
272	7.514767	7.843379	7.546114
279	7.524174	7.619789	7.550153
286	7.530020	7.810013	7.555705
293	7.531740	7.640495	7.556731
300	7.537505	7.875876	7.567096
307	7.538198	7.673839	7.562505
314	7.545375	7.635869	7.572918

LD	Min	Max	Average
320	7.548049	7.730265	7.577522
326	7.477048	7.699533	7.566417
333	7.547004	7.976490	7.578386
339	7.489021	7.600293	7.542682
345	7.488469	7.602309	7.535341
351	7.480136	8.558686	7.547873
356	7.491829	7.581253	7.535696
362	7.515456	7.804460	7.559878
368	7.507284	7.676598	7.559791
373	7.519887	7.802303	7.567793
379	7.525636	7.646957	7.574727
384	7.516321	7.668465	7.573304
389	7.528946	7.949652	7.585079
395	7.532358	7.973824	7.615670
400	7.533530	7.881495	7.591379
405	7.644478	7.980695	7.752156
410	7.644865	8.017080	7.758736
415	7.648371	7.930022	7.741497
420	7.635782	7.955201	7.726036
425	7.635507	8.004952	7.760457
429	7.675991	8.004472	7.769792
434	7.688534	8.065139	7.833564
439	7.603207	7.917859	7.740006
443	7.607625	8.047330	7.729477
448	7.623932	8.248465	7.771071

C++/OpenCL

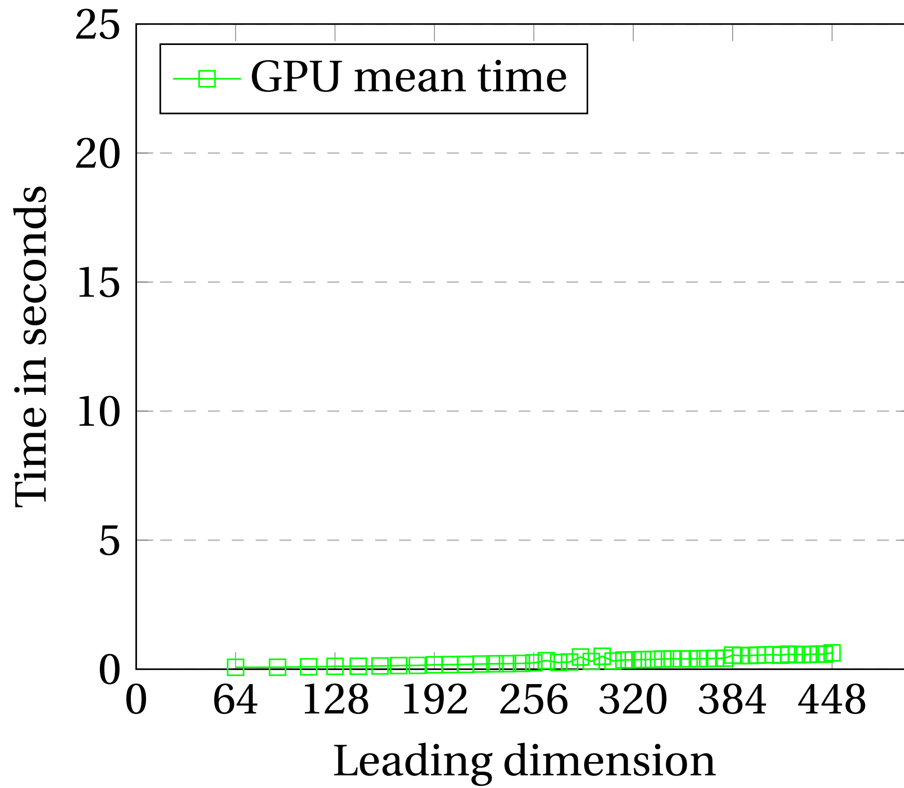

Figure B.9: GPU running time in seconds, C++/OpenCL.

LD	**Min**	**Max**	**Average**
64	0.068321	0.224610	0.073840
91	0.073545	0.103960	0.076463
111	0.089249	0.126606	0.092443
128	0.105579	0.152182	0.111032
143	0.113652	0.133337	0.115011
157	0.121111	0.174755	0.125369
169	0.137693	0.166563	0.140402
181	0.144374	0.207624	0.147297
192	0.166066	0.237189	0.170077
202	0.175591	0.218561	0.179213
212	0.181523	0.261578	0.185504
222	0.188198	0.248239	0.200436
231	0.204382	0.294567	0.209806
239	0.208714	0.244695	0.213344
248	0.216526	0.283684	0.224487
256	0.240191	0.345487	0.245257
264	0.326474	0.414752	0.332479
272	0.255584	0.373058	0.260662
279	0.276649	0.326067	0.279247
286	0.455214	0.601675	0.472862
293	0.292645	0.392574	0.299235
300	0.475992	0.641345	0.501837
307	0.302921	0.401833	0.316571
314	0.340996	0.472146	0.356854

LD	**Min**	**Max**	**Average**
320	0.347343	0.459001	0.361643
326	0.354450	0.483505	0.373078
333	0.370563	0.488477	0.379143
339	0.377441	0.512027	0.394439
345	0.382729	0.511632	0.400372
351	0.385106	0.534357	0.399228
356	0.382185	0.512728	0.397008
362	0.389391	0.523131	0.406773
368	0.395297	0.533109	0.409452
373	0.406995	0.533250	0.418806
379	0.413426	0.545250	0.426809
384	0.509788	0.685803	0.545613
389	0.507781	0.682775	0.524851
395	0.515321	0.687736	0.530350
400	0.531210	0.709982	0.546354
405	0.538205	0.707975	0.552266
410	0.535815	0.724052	0.561574
415	0.531136	0.695556	0.541804
420	0.568211	0.738814	0.583757
425	0.554058	0.735340	0.565289
429	0.562537	0.643157	0.571181
434	0.566649	0.734755	0.575534
439	0.564059	0.761454	0.585594
443	0.572699	0.751011	0.584848
448	0.626331	0.697559	0.632402

B.2.2 Visualization

MATLAB CPU

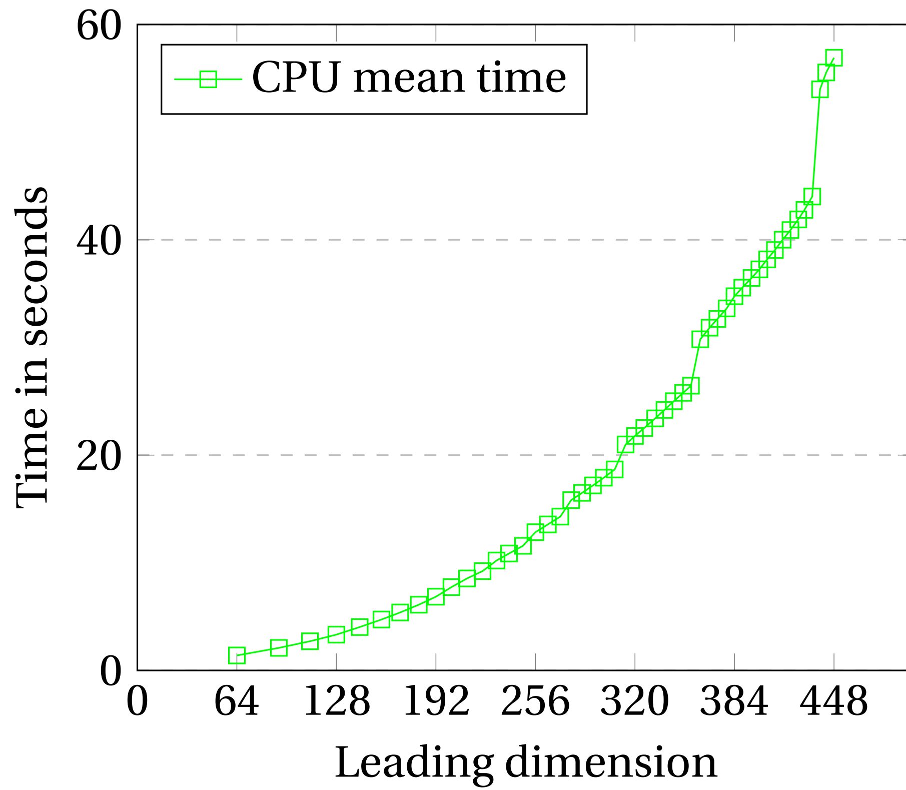

Figure B.10: CPU with visualization, running time in seconds, MATLAB.

LD	Min	Max	Average
64	1.335061	5.551656	1.393095
91	2.059203	2.384044	2.094870
111	2.666055	2.929040	2.710284
128	3.284303	3.525783	3.326229
143	3.956192	4.675671	4.030659
157	4.567473	5.136848	4.728639
169	5.180400	5.948921	5.384193
181	5.944574	6.488905	6.105519
192	6.682940	7.443890	6.847627
202	7.594822	8.123192	7.740363
212	8.284534	9.192152	8.544382
222	9.127187	9.464084	9.225845
231	10.047919	10.402637	10.209284
239	10.791998	11.363699	10.859179
248	11.496085	12.017110	11.573605
256	12.768654	12.922933	12.842405
264	13.487431	14.011421	13.560738
272	14.178479	14.988012	14.287715
279	15.707450	15.973749	15.820234
286	16.424308	16.668248	16.493152
293	17.112787	17.357266	17.185048
300	17.825680	18.243973	17.903073
307	18.569533	18.890063	18.666208
314	20.641934	21.106545	20.988059

LD	Min	Max	Average
320	21.714669	21.915459	21.781746
326	22.450351	22.733415	22.520214
333	23.335884	23.943360	23.410719
339	24.119054	24.407945	24.198795
345	24.887616	25.475520	25.008992
351	25.559353	26.220943	25.782655
356	26.323232	26.890109	26.446052
362	30.299964	30.991049	30.755873
368	31.669812	32.967661	31.833660
373	32.564586	32.747794	32.646392
379	33.490708	34.145028	33.624560
384	34.364655	38.081559	34.758954
389	35.241639	39.788391	35.551721
395	36.302708	36.679290	36.455020
400	37.101184	37.458290	37.251396
405	38.039318	38.526840	38.177932
410	38.846206	39.798984	39.051216
415	39.825114	40.485344	40.006993
420	40.761633	41.265067	40.905295
425	41.683525	42.460012	41.891088
429	42.577942	43.684912	42.776585
434	43.606624	45.191631	44.016994
439	51.150878	54.801706	53.971232
443	54.319331	62.218141	55.544594
448	55.994089	58.456251	56.912076

MATLAB GPU

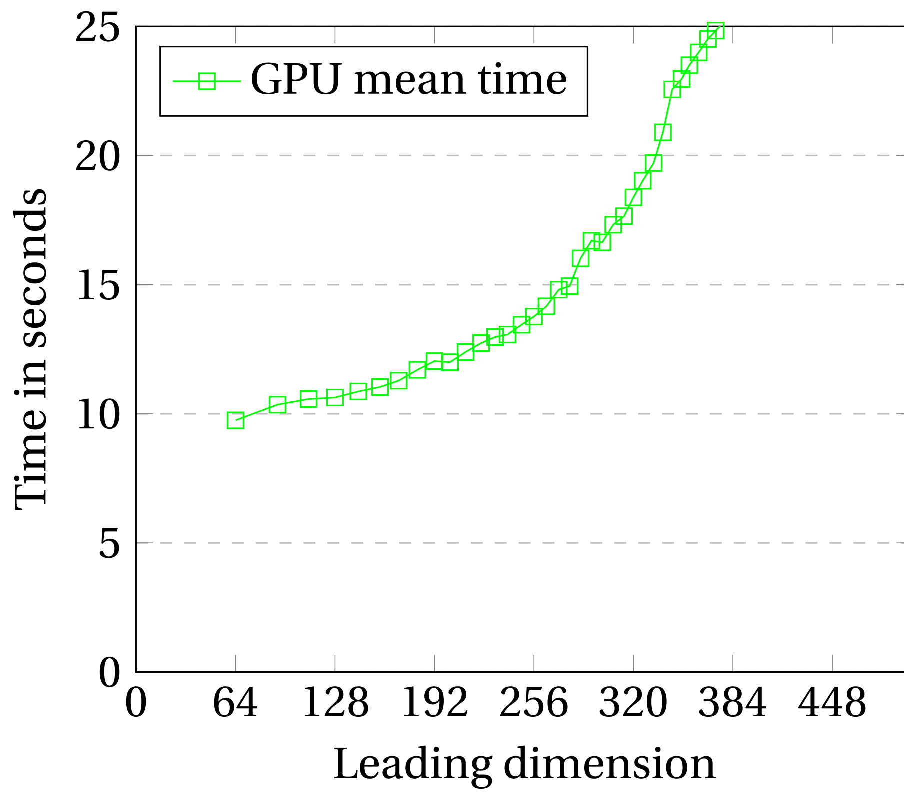

Figure B.11: GPU with visualization running time in seconds, MATLAB.

LD	Min	Max	Average
64	9.366068	27.855200	9.747331
91	10.252561	10.543490	10.353984
111	10.237013	10.866974	10.570609
128	10.486574	10.965731	10.630329
143	10.677048	12.319513	10.863271
157	10.712813	11.700444	11.036896
169	11.011185	11.939842	11.283984
181	11.537218	12.113618	11.700960
192	11.794371	12.524096	12.039569
202	11.682493	12.435800	11.998001
212	12.174902	12.744294	12.392160
222	12.388421	13.235307	12.737755
231	12.695563	14.032732	12.970305
239	12.753252	13.888819	13.068994
248	13.068113	14.340461	13.449333
256	13.324146	14.561511	13.765078
264	13.696692	15.865949	14.162959
272	13.769877	17.497530	14.807790
279	14.102712	16.615652	14.943342
286	15.065783	17.755520	16.018958
293	16.611182	18.203696	16.701843
300	16.199477	17.623333	16.637984
307	16.498538	18.825287	17.322797
314	16.664242	19.464152	17.652078

LD	Min	Max	Average
320	17.930168	20.186461	18.379540
326	17.659845	21.147248	19.023679
333	18.058938	21.895865	19.712272
339	18.759289	22.774243	20.898968
345	20.571394	23.802391	22.559510
351	20.672758	23.479213	22.961425
356	21.700648	24.496719	23.497995
362	22.788903	24.712430	23.993322
368	24.026742	24.944199	24.514916
373	24.446291	25.605481	24.839346
379	24.756246	28.790444	25.213759
384	25.875947	29.584282	26.369500
389	28.218738	31.367794	28.807505
395	30.525199	34.108599	30.888040
400	26.741483	30.873322	29.991909
405	27.343147	31.099556	30.510912
410	28.932982	32.009990	31.256930
415	30.776836	32.807493	31.772518
420	31.902437	34.625742	32.660568
425	32.629699	34.929514	33.146076
429	33.409164	37.314325	34.233974
434	34.490621	39.369535	35.900714
439	34.414335	39.218898	37.303467
443	36.500295	39.631842	38.321354
448	38.516505	42.622988	39.173698

C++/OpenCL/OpenGL

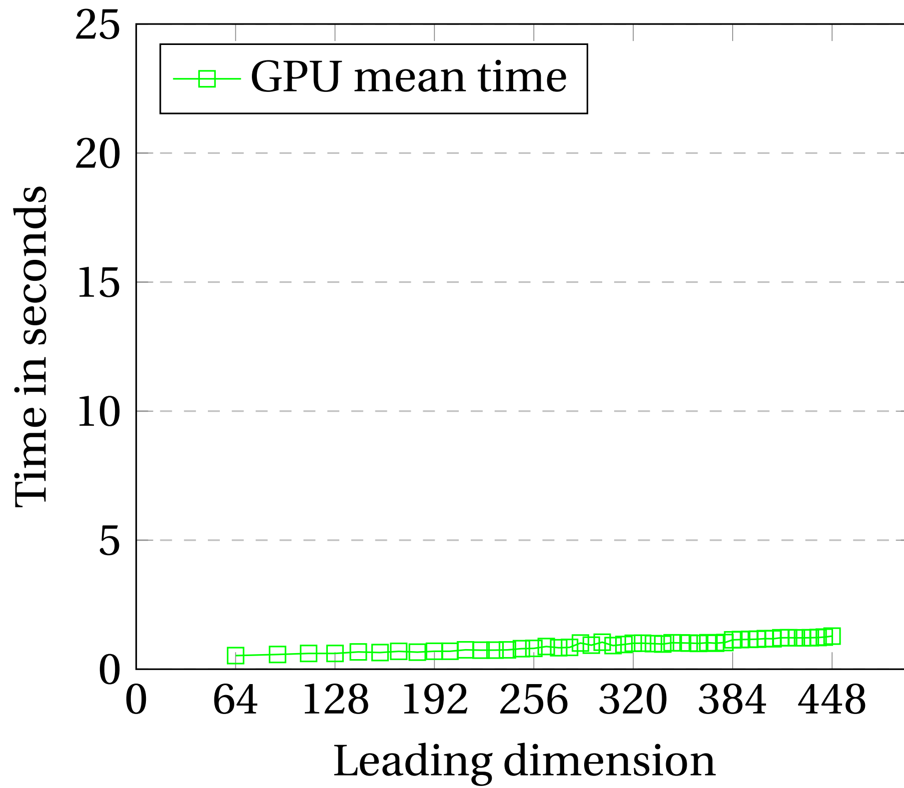

Figure B.12: GPU with OpenGL running time in seconds, C++/OpenCL.

LD	Min	Max	Average
64	0.473112	0.596742	0.527130
91	0.534834	0.639292	0.570572
111	0.566018	0.680853	0.610865
128	0.586048	0.724054	0.610865
143	0.613356	0.730011	0.668926
157	0.605579	0.685515	0.645380
169	0.638522	0.779519	0.691671
181	0.624302	0.696841	0.660336
192	0.654573	0.748904	0.694933
202	0.691834	0.764736	0.694933
212	0.692622	0.830818	0.749972
222	0.696875	0.801789	0.737392
231	0.714456	0.786970	0.739089
239	0.714500	0.714500	0.748375
248	0.737841	0.870364	0.795217
256	0.752003	0.891436	0.806527
264	0.852004	0.939306	0.879200
272	0.784559	0.936779	0.831193
279	0.784509	0.938019	0.845002
286	0.966590	1.079775	1.013793
293	0.852042	1.022068	0.930949
300	1.009267	1.109091	1.045531
307	0.876529	0.976719	0.915698
314	0.898227	1.033912	0.958248

LD	Min	Max	Average
320	0.933251	1.088057	1.004061
326	0.930580	1.059367	1.007188
333	0.931155	1.072763	0.990044
339	0.948367	1.036726	0.974549
345	0.969390	1.103185	1.023461
351	0.947980	1.071339	1.018466
356	0.974580	1.044574	1.011945
362	0.984785	1.039568	0.999281
368	0.984569	1.053967	1.025786
373	0.983004	1.047509	1.006671
379	1.004038	1.102988	1.033593
384	1.110542	1.228218	1.137292
389	1.067318	1.354411	1.148145
395	1.094592	1.243744	1.155428
400	1.097179	1.242446	1.163389
405	1.129033	1.279172	1.184281
410	1.103762	1.226109	1.178646
415	1.165487	1.287098	1.216770
420	1.149067	1.315953	1.219367
425	1.180362	1.287986	1.215562
429	1.167286	1.289659	1.214132
434	1.182806	1.284544	1.219320
439	1.152224	1.314669	1.227097
443	1.199415	1.297347	1.244632
448	1.207210	1.403806	1.281585

Bibliography

[1] K.-A. Lie J.R. Natvig M. Ofstad Henriksen T.R. Hagen *, J.M. Hjelmervik. Visual simulation of shallow-water waves, 2006.

[2] The MathWorks inc. The language of technical computing. https://se.mathworks.com/products/matlab.html.

[3] The MathWorks inc. Perform parallel computations on multicore computers, gpus, and computer clusters. https://se.mathworks.com/products/parallel-computing.html.

[4] The MathWorks inc. Speeding up matlab computations with gpus. https://se.mathworks.com/products/parallel-computing/features.html.

[5] The MathWorks inc. Perform matlab computations on cuda gpus. https://se.mathworks.com/discovery/matlab-gpu.html.

[6] Mattson Fung Ginsburg Munshi, Gaster. *OpenCL Programming Guide*. Pearson Education, 2011.

[7] John Daintith, editor. *A Dictionary of Physics*. OUP Oxford, 6nd edition, 2009.

[8] Tveito and Winther. *Introduction to Partial Differential Equations, A computational approach*. Springer, 2nd edition, 2008.

[9] Richard Pletcher John Tannehill, Dale Anderson. *Computational Fluid Mechanics and Heat Transfer*. Taylor and Francis, 2st edition, 1997.

[10] The MathWorks inc. Techniques to improve performance. https://se.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html.

[11] Nick Bruun. Hayai. https://github.com/nickbruun/hayai.

[12] Schlachter Tompson. An introduction to the opencl programming model. http://cims.nyu.edu/~schlacht/OpenCLModel.pdf.

[13] Robert Bridson. *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press, 2nd edition, 2015.

[14] Cleve Moler. *Experiments with MATLAB*. MathWorks, inc, 1st edition, 2011.

[15] John Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. SIAM: Society for Industrial and Applied Mathematics, 2nd edition, 2014.

[16] Kevin Brothaler. *OpenGL ES 2 for Android. A Quick-Start Guide*. Pragmatic Bookshelf, 1st edition, 2013.

[17] E. Zauderer. *Partial differential equations of applied mathematics*. Wiley interscience, 3rd edition, 2006.