DELFT UNIVERSITY OF TECHNOLOGY

Suitability of Shallow Water Solving Methods for GPU Acceleration

FLORIS BUWALDA

February 18, 2020

In fulfillment of the requirements to obtain the degree of Master of Science in Applied Mathematics, specialized in Computational Science and Engineering

at the Delft University of Technology, Numerical Analysis group, faculty of Electrical Engineering, Mathematics and Computer Science

to be defended publicly on Wednesday February 26, 2020 at 2:00 PM.

Student number:	4241290	
Thesis committee:	Prof. dr. ir. C. Vuik,	TU Delft, supervisor
	Dr. E. de Goede,	Deltares, supervisor
	Prof. dr. ir. H. X. Lin,	TU Delft
Additional Supervisor:	Ir. M. Pronk	Deltares, supervisor

Contents

Abstract

In the past 15 years the field of general purpose computing on graphics processing units, or GPUs, has become well developed and the practice is becoming more mainstream. Computational demands of simulation software are continuously increasing. As such for many applications traditionally computed on the central processing unit the question arises of whether moving to GPU computing is a possible cost effective way of meeting these demands.
The fundamental nature of GPU architecture that makes it so cost effective at doing bulk computation also poses restrictions on which applications are suitable for it.
The shallow water equations are a simplified form of the Navier-Stokes equations and describe water levels and flow currents in suitably shallow water such as rivers, estuaries and the North sea. The main research goal of this thesis project was to determine whether the shallow water equations are suitable for implementation on a GPU. Two options exist, the equations may be solved with either an explicit or implicit time integration method.

First, a literature study was conducted to familiarize with the tools required to build explicit and implicit shallow water models on a GPU. Then both an explicit and implicit shallow water solver were developed first in the MATLAB programming language and later in CUDA C++ on both CPU and GPU.

The main findings are that both explicit and implicit methods are well suited for GPU implementation. Both methods proved to be compatible with a wetting and drying mechanism of numerical cells. The Cuda C++ implementation was in the order of 10 times as fast as a MATLAB implementation for both CPU and GPU. For the benchmark cases tested, the Cuda C++ GPU implementation was in the order of 50 times faster than the equivalent multithreaded CPU implementation.
The implicit implementation was benchmarked using the conjugate gradient method to solve the linear system. Various preconditioners were tested and a Repeated Red Black preconditioner was found to be the most effective. The computation time of the RRB preconditioned implicit method was compared with the explicit method and it was found that the two methods reached parity in computation time when the implicit time step was taken roughly 50 times as large as the explicit time step. For implicit time steps smaller than that the explicit method was faster and when the implicit time step was larger the implicit method was faster.
For the benchmark cases tested, the implicit method using a time step 50 times larger than the explicit method was found to be less accurate and less stable than the explicit method. The conclusion is that for cases similar to the benchmark cases an explicit method is the fastest, most stable and most accurate method and thus the preferred choice.

INTRODUCTION

In the Netherlands, water has always been an important element to consider, as large parts of the country are land claimed from the sea through human engineering.
Already in the 1950s the Directorate-General for Public Works and Water Management, Rijkswaterstaat in Dutch, started the Delta Works project for the closure of the Eastern Scheldt Barrier [1]. In this project numerical methods were applied to predict the impact of this storm surge barrier on water levels and flow currents.
Deltares continues this tradition as an independent institute for applied research in the field of water and subsurface. Deltares' software system Delft3D is state-of-the-art and is applied worldwide for the computation of water-related processes like water levels, water quality processes and waves [2].

Delft3D has been developed for calculations on a computer's central processing unit, or CPU. Since the advent of the 21st century, great developments have been made with doing calculations on a computer's graphics processing unit, or GPU. The GPU is generally more efficient and faster at parallel computations, but has other limitations.

This thesis project focuses on the shallow water equations, which describe water levels and flow currents. Two main methods exist for simulation, explicit and implicit time stepping methods. It is well established that an implicit method is preferred when doing shallow water computations on a CPU. The main focus of research in the project is first to implement both an explicit and implicit method on the GPU, and determine which method is preferred under which circumstances. Therefore the main research question is:

"Which numerical method is best suited for solving the shallow water equations on a GPU in terms of model accuracy, robustness and speed?"

A second item of research is 'drying and flooding' of areas of water. For example, in the Wadden Sea a lot of drying and flooding of shallow areas occurs because of the changing tides. For a shallow water model this mechanic needs to be treated specially, and it is important that this is done efficiently. A secondary research goal is thus to develop an efficient flooding and drying mechanism in both the implicit and explicit implementation.

First, a literature study has been conducted to familiarize with the tools required to build the explicit and implicit implementation. The Shallow Water equations are described in chapter 1, followed by a review of differential equation discretization methods in chapter 2. Then, different time integration methods are discussed in chapter 3 and the inner workings of a GPU in chapter 4. Finally methods for solving an implicit linear system are described in chapter 5.

The implementation and results are structured chronologically, with the initial implementation composing chapters 6 and 7, the second improved implementation chapter 8 and 9, and the implicit implementation in chapter 10.
Finally a summary and conclusions are presented in chapter 11.

This master thesis project was carried out at the TU Delft Department of Computer Science in collaboration with Deltares.

RESEARCH QUESTIONS

As mentioned in the introduction, the main research question to be answered in the coarse of the project is:

“Which numerical method is best suited for solving the shallow water equations on a GPU in terms of model accuracy, robustness and speed?”

Additionally, a number of subquestions have been formulated:

1. What are the tradeoffs involved in solving the shallow water equations on a GPU using explicit methods compared to implicit?
2. How does the performance of existing software packages compare to a self-built solver?
3. What are the tradeoffs involved in solving the shallow water equations on a GPU in 32-bit floating point precision compared to 64 bit and 16 bit?
4. Which method or solver library is best suited for integration into Deltares’ existing FORTRAN based solvers?

The answers to the main research question and subquestion will be presented in chapter 11.

1

THE SHALLOW WATER EQUATIONS

1.1. INTRODUCTION

The SWE are a set of equations that describe fluid flow on a domain which has a much larger length scale than depth scale. This also means that the applicability of the shallow water equations is not necessarily restricted to bodies of water that are actually shallow. The equations were first derived in one dimensional form by Adhémar Jean Claude Barré de Saint-Venant in 1871 who also was the first to derive the Navier-Stokes equations [3]. The Navier-Stokes equations are the full set of equations describing viscous fluid flow. These equations are hard to solve due to their inherent non-linearity and complexity. The shallow water equations can be derived from the Navier-Stokes equations as a special case where the complexity is reduced by averaging over the depth, hence the shallowness condition. This makes them a very popular set of equations for use in simulation where the shallowness condition holds.

1.2. DERIVATION

1.2.1. THE NAVIER-STOKES EQUATIONS

In order to derive the shallow water equations we will first start by stating the Cauchy momentum equation in convective form [4]:

$$\rho \frac{\partial \mathbf{u}}{\partial t} = -\nabla p + \nabla \cdot \tau + \rho g \tag{1.1}$$

Where $\mathbf{u}$ is a 3-dimensional flow velocity vector, ρ the fluid density, p the pressure, τ the deviatoric stress tensor and g the vector of body forces acting on the fluid.

Since we have conservation of mass, we can derive from the continuity equation [4] that $\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{u})$. Substituting this into 1.1 we obtain:

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot \left(\rho \mathbf{u}\mathbf{u}^T\right) = -\nabla p + \nabla \cdot \tau + \rho g \tag{1.2}$$

It is known that water is only slightly compressible. However, when the shallowness assumption holds the pressures involved are small so it can be assumed incompressible, which means the density is constant. 1.2 and the continuity equation then become:

$$\frac{\partial(\mathbf{u})}{\partial t} + \nabla\cdot\left(\mathbf{u}\mathbf{u}^T\right) = \frac{1}{\rho}\left(-\nabla p + \nabla\cdot\tau\right) + g \tag{1.3}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{1.4}$$

These are the incompressible Navier-Stokes equations in conservation form.

1.2.2. Boundary conditions

For illustration a 3 dimensional representation of the shallow water domain is given in figure 1.1.

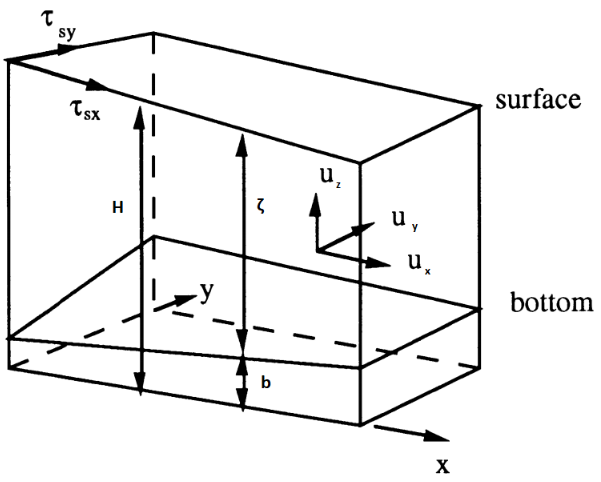


Figure 1.1: A schematic of the shallow water domain. $u_{x,y,z}$ are the flow velocities in their respective directions, source: [5]

The bottom and free surface boundaries are important to consider first for the derivation of the shallow water equations. After that the domain boundary conditions will be discussed.

Bottom boundary

At the bottom $z = -b(x,y)$ we have the following conditions:

1. No slip condition: $\mathbf{u}(x,y,-b(x,y)) = 0$
2. The flux through the bottom is 0: $u_x\frac{\partial b}{\partial x} + u_y\frac{\partial b}{\partial y} + u_z = 0$
3. The bottom shear stress equals $\tau_{bx} = \tau_{xx}\frac{\partial b}{\partial x} + \tau_{xy}\frac{\partial b}{\partial y} + \tau_{xz}$ similarly for y

FREE SURFACE

At the time dependent free surface $z = \zeta(x, y, t)$ we have the following conditions:

1. The flux through the surface is 0: $\frac{\partial \zeta}{\partial t} + u_x \frac{\partial \zeta}{\partial x} + u_y \frac{\partial \zeta}{\partial y} - u_z = 0$
2. The top shear stress equals $\tau_{\zeta x} = -\tau_{xx} \frac{\partial \zeta}{\partial x} - \tau_{xy} \frac{\partial \zeta}{\partial y} + \tau_{xz}$ similarly for y
3. The pressure defined as $p = p - p_0 = 0$ with p_0 the atmospheric pressure.

1.2.3. PRESSURE APPROXIMATION

If we consider at the z component of equation 1.3 it can be assumed that all terms except the pressure can be neglected when compared to the gravitational acceleration, so the equation can be reduced to $\frac{\partial p}{\partial z} = \rho_0 g$.
After integrating we find $p = \rho_0 g(\zeta - z)$, which is simply the hydrostatic pressure.
This also produces the other terms of the gradient of p: $\frac{\partial p}{\partial x, y} = \rho_0 g \frac{\partial \zeta}{\partial x, y}$.

1.2.4. DEPTH AVERAGING

By assuming the density was constant we have essentially eliminated the z dependency of the pressure in equation 1.3. This suggest that we can also approximate the velocities setting them to be their average when integrated over depth. We denote this average as $\bar{u}_{x,y} = \frac{1}{H} \int_{-b}^{\zeta} u_{x,y} dz$ Integrating the the continuity equation of 1.3 we apply the Leibniz integral rule and our boundary conditions to obtain:

$$\int_{-b}^{\zeta} \nabla \cdot \mathbf{u} dz = \nabla \int_{-b}^{\zeta} \mathbf{u} dz - \mathbf{u}(z = \eta) \nabla \eta + \mathbf{u}(z = b) \nabla b = \frac{\partial H}{\partial t} + \frac{\partial}{\partial x}(H\bar{u}_x) + \frac{\partial}{\partial y}\left(H\bar{u}_y\right) = 0 \quad (1.5)$$

Likewise we can also integrate 1.3 and apply the shear stress boundary conditions to obtain:

$$\begin{aligned}
\frac{\partial}{\partial t}(H\bar{u}_x) + \frac{\partial}{\partial x}\left(H\bar{u}_x^2\right) + \frac{\partial}{\partial y}(H\bar{u}_x\bar{u}_y) &= -gH\frac{\partial \zeta}{\partial x} + \frac{1}{\rho_0}\left[\tau_{\zeta x} - \tau_{bx} + \frac{\partial}{\partial x}\bar{\tau}_{xx} + \frac{\partial}{\partial y}\bar{\tau}_{xy}\right] \\
\frac{\partial}{\partial t}\left(H\bar{u}_y\right) + \frac{\partial}{\partial x}(H\bar{u}_x\bar{u}_y) + \frac{\partial}{\partial y}\left(H\bar{u}_y^2\right) &= -gH\frac{\partial \zeta}{\partial y} + \frac{1}{\rho_0}\left[\tau_{\zeta y} - \tau_{by} + \frac{\partial}{\partial x}\bar{\tau}_{xy} + \frac{\partial}{\partial y}\bar{\tau}_{yy}\right] \quad (1.6)
\end{aligned}$$

Now finally we can expand the derivatives on the left-hand side using the chain rule, simplify using 1.5 and then divide by H to obtain:

$$\begin{aligned}
\frac{\partial \bar{u}_x}{\partial t} + \frac{\partial \bar{u}_x}{\partial x}\bar{u}_x + \frac{\partial \bar{u}_x}{\partial y}\bar{u}_y &= -g\frac{\partial \zeta}{\partial x} + \frac{1}{\rho_0 H}\left[\tau_{\zeta x} - \tau_{bx} + \frac{\partial}{\partial x}\bar{\tau}_{xx} + \frac{\partial}{\partial y}\bar{\tau}_{xy}\right] \\
\frac{\partial \bar{u}_x}{\partial t} + \frac{\partial \bar{u}_y}{\partial x}\bar{u}_x + \frac{\partial \bar{u}_y}{\partial y}\bar{u}_y &= -g\frac{\partial \zeta}{\partial y} + \frac{1}{\rho_0 H}\left[\tau_{\zeta y} - \tau_{by} + \frac{\partial}{\partial x}\bar{\tau}_{xy} + \frac{\partial}{\partial y}\bar{\tau}_{yy}\right] \quad (1.7)
\end{aligned}$$

1.5 together with 1.6 are what are called the 2D shallow water equations. The terms that are still undetermined are the surface and bottom stress terms and the stress derivatives on the right-hand side.

Finally, it is known [6] that the divergence of the deviatoric stress equals the viscosity multiplied by the Laplacian of the velocity for incompressible flow.
The surface stress can often be neglected and the bottom stress can be modeled [6] as $\frac{\tau_{bx}}{\rho_0} = \frac{g u_x ||\mathbf{u}||}{C^2 H}$, where C is the Chézy coefficient [7]. Combining this with 1.6 and 1.5 we obtain:

$$\begin{aligned}
\frac{\partial H}{\partial t} + \frac{\partial}{\partial x}(H\bar{u}_x) + \frac{\partial}{\partial y}\left(H\bar{u}_y\right) &= 0 \\
\frac{\partial \bar{u}_x}{\partial t} + \frac{\partial \bar{u}_x}{\partial x}\bar{u}_x + \frac{\partial \bar{u}_x}{\partial y}\bar{u}_y &= -g\frac{\partial \zeta}{\partial x} - \frac{g u_x ||\mathbf{u}||}{C^2 H^2} + \nu\left(\frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2}\right) \\
\frac{\partial \bar{u}_x}{\partial t} + \frac{\partial \bar{u}_y}{\partial x}\bar{u}_x + \frac{\partial \bar{u}_y}{\partial y}\bar{u}_y &= -g\frac{\partial \zeta}{\partial y} - \frac{g u_y ||\mathbf{u}||}{C^2 H^2} + \nu\left(\frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2}\right)
\end{aligned} \tag{1.8}$$

We now have a workable form of the shallow water equations.

1.3. LINEARISED SYSTEM

Non-linear systems are difficult to solve. If global system behaviour is an acceptable result it can often be more practical to linearise the system and obtain an approximate solution instead. In equation 1.8 we can identify a number of non-linear terms. On the left-hand side, we have the product of velocities and their spatial derivatives, and on the right-hand side we have the bottom friction and the viscosity terms.

In order to linearize these equations it is suggested [5] that we consider a steady uniform flow that is perturbed. This means that $\mathbf{u} = (u_x, u_y) = \mathbf{U} + \mathbf{u}'$ and $\zeta = Z + \zeta'$
The viscosity term is linear but its second order derivatives make it complex to solve and its influence is known to be small so it is neglected.

The bottom friction is approximated by a constant C that is proportional to the unperturbed bottom friction coefficient. The linear approximation to the bottom friction also neglects the acceleration terms which are assumed to be small for almost steady uniform flow. This breaks down in the case of tidal flow for example, as in that case the acceleration terms become quite significant.

Inserting this into 1.8 and after canceling some terms and neglecting the higher order terms we obtain our linear approximation:

$$\begin{aligned}\frac{\partial H}{\partial t}+\frac{\partial H}{\partial x}U_x+\frac{\partial H}{\partial y}U_y+Z\left(\frac{\partial u_x}{\partial x}+\frac{\partial u_y}{\partial y}\right)&=0\\ \frac{\partial u_x}{\partial t}+\frac{\partial u_x}{\partial x}U_x+\frac{\partial u_x}{\partial y}U_y&=-g\frac{\partial H}{\partial x}-cu_x\\ \frac{\partial u_x}{\partial t}+\frac{\partial u_y}{\partial x}U_x+\frac{\partial u_y}{\partial y}U_y&=-g\frac{\partial H}{\partial y}-cu_y\end{aligned}\tag{1.9}$$

Where we have omitted the depth averaging bars and perturbation accents for readability.

Since this is a linear system of equations it can be conveniently written in vector-matrix form to aid the discretization process:

$$\frac{\partial \mathbf{u}}{\partial t}=A\frac{\partial \mathbf{u}}{\partial x}+B\frac{\partial \mathbf{u}}{\partial y}+C\mathbf{u}\tag{1.10}$$

Where

$$\mathbf{u}=\begin{bmatrix}u_x\\u_y\\H\end{bmatrix}\quad A=\begin{bmatrix}U_x&0&g\\0&U_x&0\\Z&0&U_x\end{bmatrix}\quad B=\begin{bmatrix}U_y&0&0\\0&U_y&g\\0&Z&U_y\end{bmatrix}\quad C=\begin{bmatrix}c&0&0\\0&c&0\\0&0&0\end{bmatrix}\tag{1.11}$$

1.4. WELL POSEDNESS

A set of differential equations cannot have a unique solution if it is not supplied by initial conditions and boundary conditions. We call a problem well-posed if:

1. A solution exists
2. The solution is unique
3. The solution its behaviour changes continuously with changing initial and boundary conditions. This is often interpreted as the principle that small perturbations to initial or boundary conditions do not have a large impact on the solution.

1.4.1. DOMAIN BOUNDARIES

For a two dimensional shallow water domain there exist two possible domain boundaries, open and closed. A closed boundary permits no flux in the direction normal to the boundary. An open boundary is an artificial boundary through which flow moves unhindered. An example domain with closed boundaries would be when simulating an entire lake, surrounded by land on all sides. Open boundaries would occur when simulating part of a river, where the open boundary would be at the points where the river enters and leaves the domain.

One problem with an open boundary is that imposing a boundary condition in order to guarantee well posedness might lead to wave reflection at the artificial boundary. It can be shown [8] that if the Sommerfeld radiation condition is perfectly satisfied it guarantees no wave reflection. In practice this works only in an ideal case. However properties can be determined that minimize reflection, which is why Sommerfeld radiation conditions are also called weakly reflective boundary conditions.

1.4.2. HYPERBOLIC SYSTEM

According to Courant & Hilbert [9], the shallow water equations are a hyperbolic system of equations if we omit the viscosity term. When the viscosity term is neglected, it is known that the solutions of the linearized SWE are wave-like solutions called Gravity waves.

The system can be written in terms of characteristics which represent the behaviour of the solutions over time. Hyperbolic systems have characteristic solutions and at any point of the boundary of the region it is necessary to specify as many boundary conditions as there are characteristic planes entering the region [5].

The characteristic wave speed can be shown to be related to the long wave speed $\sqrt{gH}$: $u+\sqrt{gH}$ and $u-\sqrt{gH}$.
If $|u| < \sqrt{gH}$ we call the flow subcritical which is the most common scenario. In this scenario when there is a positive flow into the domain there are also two characteristics entering, which requires two boundary conditions.
When there is a negative flow into the domain only a single characteristic enters and we require a single boundary condition for the problem to be well posed.

1.4.3. PARABOLIC SYSTEM

If the viscosity is taken into account the system is parabolic. This means that the system can no longer be described by a set of characteristics. Oliger & Sundström [10] used an energy conservation argument to conclude which additional boundary conditions need to be imposed. For a closed boundary, it is necessary to specify either a no-slip boundary which means the tangential velocity is 0, or a free-slip boundary which implies the shear stress at that boundary is 0. On an open boundary Sundström proposed one should require zero shear stress when water flows through the boundary out of the domain. When water flows into the domain, the flux through the boundary is required to remain constant. [5] notes that the physical significance of the two requirements on an open boundary is not clear.

1.4.4. INITIAL CONDITIONS

If one imagines the simulation space as a plane in the (x, y, t) space, the moment $t = 0$ is a boundary of the region. Thus the principle of characteristics that a boundary condition is necessary for every characteristic entering the region holds. Every single characteristic enters the "region" through this boundary which means that all three possible boundary conditions need to be specified, the initial x velocity, y velocity and water level H.

2

Discretization

2.1. Introduction

In chapter 1 we derived the shallow water equations. Before they can be solved however, the problem needs to be discretized, which means dividing the domain of computation into gridpoints on which function values are evaluated.
There exist three main approaches to discretization: the finite differences method, the finite volumes method and the finite elements method.

2.2. Finite differences

A differential equation involves (partial) derivatives, which are defined as some continuous limit of a function. If we wish to solve a differential equation on a computer, we cannot take a continuous limit because a computer itself operates in discrete terms. Therefore in order to state our problem in a computer language the derivatives must be approximated in a discrete way. If we approximate these derivatives using Taylor polynomials we call this the method of finite differences. Taylor's theorem states that if a function is k times differentiable at a point a we can approximate the function in a neighborhood of a as:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2} + \ldots + \frac{f^{(k)}(a)(x-a)^k}{k!} + \ldots \tag{2.1}$$

Now suppose we wish to approximate a first order derivative on a numerical grid with grid distance h. If we set $x = a+h$ we obtain $f(a+h) = f(a) + f'(a)h + O(h^2)$. Rearranging for the derivative produces:

$$f'(a) = \frac{f(a+h) - f(a)}{h} + O(h) \tag{2.2}$$

Which means that we can approximate the value of the derivative in point a using the function value in point a and a neighbor function value with an error of order h.

An important thing to note is that the form defined in 2.2 is called forward difference, as the function value in the positive neighboring spatial direction is used to approximate the derivative. Other options are central difference where the information is used from both neighbors, or backwards difference where information is used from the negative spatial direction.

The order of the error is quite large, the same order as the grid distance. This means in order to obtain an accurate estimate of the derivatives a very fine grid must be used for computation, which may take a lot of computing time. Various different methods have been developed that provide higher order accuracy at the cost of computational intensity.

One remark regarding finite differences is that it requires equidistant grid points, which imposes some restrictions on real world applicability. It is technically possible to construct a method based on the Taylor approximation of the derivative on an non-equidistant grid, but in practice this is so cumbersome that often a finite element or finite volume method is chosen.

2.3. FINITE VOLUMES

The principle behind the Finite volumes method is Gauss's theorem, which states that in a 2 dimensional domain Ω for a vector field $F(x, y)$ it holds that:

$$\int\int_{\Omega} \nabla \cdot F d\Omega = \int_{S} F \cdot \mathbf{n} dS \tag{2.3}$$

Where S denotes the boundary of Ω and $\mathbf{n}$ denotes the unit vector normal to the boundary S.

In this way a differential equation involving a divergence term can be solved by invoking the above equivalency to simplify the equation. The boundary integral can be numerically approximated using Newton-Cotes integration [11]. The idea of the method is to partition the domain in a set of control volumes on which the differential equation is solved using this principle.

A big advantage of finite volume schemes is that they are conservative: The fluxes are approximated at the boundaries and whatever flows out of one control volume enters the next. Thus if the differential equation that is solved using the method represents a conserved variable such as energy, mass or momentum, the scheme will also guarantee conservation for the numerical approximation which is very desirable.

2.4. FINITE ELEMENTS

The finite element method is similar to the finite volumes method in the sense that the problem is often reformulated in a so called "weak formulation" by integrating with a chosen test function and applying 2.3. The major difference however, is that the solution of the differential equation is approximated by a finite linear combination of basis functions. The problem then reduces into calculating the weights of the basis functions.

These basis functions are defined on the edges of the numerical grid and are preferably nearly orthogonal. This is because the differential equation often involves a sum of inner products between those basis functions. Every non zero inner product will then correspond to an entry in the computation matrix and thus if a sparse matrix is desired the basis functions must be nearly orthogonal.

Many different basis functions can be chosen to suit the differential equation and boundary conditions. For example the incompressible SWE's have the incompressiblity condition $\nabla \mathbf{u} = 0$, and it is possible to choose elements in such a way that this condition is automatically satisfied, significantly reducing computational complexity. Like the finite volume method it is also possible to choose your elements such that conserved variables are also conserved by the numerical scheme. It should be noted however that finding these elements is a very complex task and will only be an option for specific problems.

A big advantage of the finite element method is that the grid on which the basis functions exist does not have to be structured. This means that if greater numerical precision is required on a subdomain of computation, the grid can be refined only on that subdomain and remain coarse on the rest of the domain which reduces computational complexity. However the method is inherently more computationally intensive than finite difference or finite volume volumes which makes these methods preferred for homogeneous grids.

2.5. STRUCTURED AND UNSTRUCTURED GRIDS

The methods described in the preceding sections all have different requirements for the discretized grid that represents the domain of computation. The two main grid categories are structured and unstructured grids.

STRUCTURED GRIDS

A structured grid has a constant structure: it contains a number of nodes that have a regular connectivity. This makes it very easy to represent the domain in matrix form: A grid node at index (i, j) is represented by matrix element (i, j) and nodes adjacent in space are also adjacent in memory. Intuitively one would expect that this means that the domain represented is always a rectangle. A problem with a rectangular domain is that the boundaries of your physical problem may not be rectangular. This means that the boundary values on the gridpoints must be inter- or extrapolated which is cumbersome and introduces errors. The matrix structure is such that there is no need for a connectivity matrix: connections are implied by in-matrix adjacency, which is very storage efficient.

Fortunately a method exists to circumvent this problem: boundary fitted coordinates. By reformulating the problem in general curvilinear coordinates the grid can be morphed to fit the physical boundaries [11]. This solves the boundary problem but reformulation of the original problem into curvilinear coördinates is often nontrivial.

An illustration of a curvilinear reformulation is given in figure 2.1.

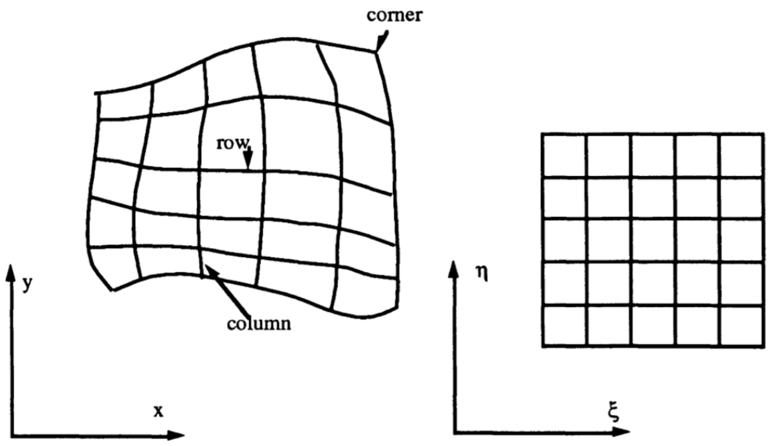


Figure 2.1: An illustration of boundary fitting a structured grid using curvilinear coordinates. Source: [5]

UNSTRUCTURED GRIDS

A structured grid has the requirement that every row and column has a constant number of grid points. An unstructured grid is simply a grid that has no such restriction. This makes unstructured grids very useful for representing complex domains, or domains where greater grid density is required at specific subdomains. An unstructured grid is easier to generate for complex problems but harder to represent and store in computer memory. An example of an unstructured grid is given in figure 2.2.

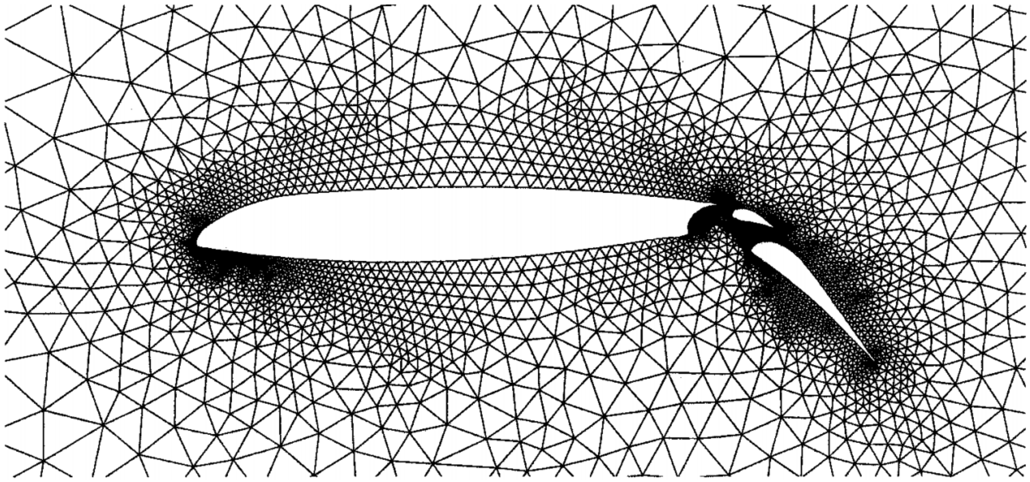


Figure 2.2: An example of an unstructured grid using triangular elements. Source: [12]

Since the domain of computation has a large impact on grid generation and storage complexity, and since grid choice and solving method are closely entwined the optimal choices in these matters are highly problem dependent.

2.6. COLLOCATED AND STAGGERED GRIDS

The linearised shallow water equations 1.9 involve three unknown variables, u_x, u_y and H. In the discretisation process the domain of computation is represented by a finite number of connected grid nodes. When representing a structured grid in computer memory the grid nodes easily map to the matrix elements, and thus it would be very intuitive to define all three variables on these same nodes. This is what we call a collocated grid, where all the variables are defined on the same position. Defining the variables on the same location as the grid nodes is called the vertex-centered approach [11].

It is not necessary however, to define function values at the same location as the grid nodes. If the function values are instead defined in the center of the cells created by the grid nodes, we call this the cell-centered approach. An advantage of the cell-centered approach is that when using the finite volumes method the cell boundaries automatically define the control volumes.

When using the central finite difference approximation of a derivative of a variable, the values from neighboring grid nodes are used to approximate the derivative but not the value on the node itself. This leads to idea that if a derivative of a variable and the variable itself are never used at the same time in the same equation, there is no need for them to be defined on the same node. If a grid is built in this fashion it is called a staggered grid. The advantage here lies in the fact that only a quarter of the total number of variables need to be computed and stored when compared to the original grid.

In the case of the linearized shallow water equations 1.9, the water height and velocities can be staggered in this fashion. Arakawa [13] proposed four different staggered grids. According to [14], the Arakawa C-grid is best suited for the shallow water equations. It is staggered such that the water height H is defined on the grid points, the flow velocity u_x is defined between grid points neighboring in the x-direction and the flow velocity u_y is defined between grid points neighboring in the y-direction. An illustration of the grid is given in 2.3.

The staggering prevents odd-even decoupling leading to numerical solutions perturbed by checkerboarding, and allows for a larger grid size as variable density is reduced.

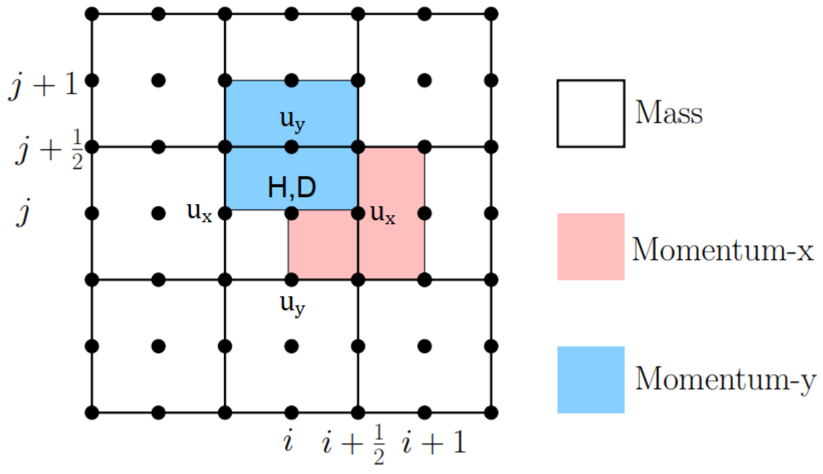


Figure 2.3: An illustration of the Arakawa C-grid. u_x, u_y are the flow velocity in the x, y directions respectively, H is the water depth, D the bathymetry height and i, j are the node indices. The control volumes for the conserved variables are coloured white, pink and blue. Source: [15]

3

Time integration methods

3.1. Introduction

In chapter 2 various discretization methods for solving partial differential equations have been described. However, these methods involve approximating spatial derivatives in order to obtain an approximate numerical solution. The shallow water equations do not only contain spatial derivatives but also temporal derivatives. Taylor's theorem 2.1 can be used in the same way as in chapter 2 to approximate the time derivative:

$$\frac{\partial \phi_n}{\partial t} \approx \frac{\phi_{n+1} - \phi_n}{\Delta t} = F(\phi_{n+\omega}) \tag{3.1}$$

Where is ϕ_n the value of the function ϕ at time t, and ϕ_{n+1} is the value at time $t + \Delta t$ and $F(\phi)$ some function of ϕ that defines the (partial) differential equation, and ω some value between 0 and 1 which exists due to the intermediate value theorem.

There is however one crucial difference between the application of the approximation of the derivative. In the case of a spatial derivative, all function values are known and used to approximate the value of the derivative at a point. In the temporal case the derivative is used to approximate a function value at a later time given the values from the past.

This method is what we call time integration, because the partial differential equation is essentially integrated over a small time step. There exist two different classes of time integration methods, explicit and implicit, which will be covered in more detail in the next sections.

3.2. Explicit time integration

The expression in equation 3.1 is not complete as the function $F(\phi)$ is not yet discretized. In order to approximate the function $F(\phi)$ it seems obvious to take some linear combination of the past and future value:

$$\frac{\phi_{n+1}-\phi_n}{\Delta t}=aF(\phi_{n+1})+(1-a)F(\phi_n) \tag{3.2}$$

Two obvious choices for a exist, which are $a=1$ and $a=0$. If we take $a=0$ then the right-hand side of the equation depends only on the past values of ϕ, and we call the method Explicit. It is then quite easy to reorder the equation to find an expression for ϕ_{n+1}:

$$\phi_{n+1}=\phi_n+\Delta tF(\phi_n) \tag{3.3}$$

Euler [16] was the first to publish this method in 1768. Since it uses information from the present to approximate a function forward in time, it is called the Euler forward method. An advantage of the Euler forward method is that it is very easy to implement. A disadvantage of the method is that it is only numerically stable for small enough time steps.

For explicit time integration the right-hand side of the recurrence relation equation 3.3 is composed of known variables. After discretization of the problem the function $F(\phi)$ if it is a linear function can be expressed as a product of a matrix A and the vector ϕ_n.
The ϕ_n term can be absorbed into A by adding the identity matrix to A. This leads to the following update procedure for explicit time integration

$$\phi_{n+1}=\mathbf{A}\phi_n \tag{3.4}$$

This means that for each timestep a matrix vector product must be calculated. Matrix vector product operations are highly parallelizable because every resulting vector value results from a row-column multiplication that is independent from all other rows. This makes an explicit method an ideal candidate for implementation on a GPU, which will be explained further in chapters 4 and 5.

3.2.1. STABILITY

When numerically time integrating a hyperbolic partial differential equation, it is important to know when the method will converge to a satisfactory solution, i.e. whether it is numerically stable. Many different mathematical definitions of numerical stability exist, but intuitively it means method has a numerical error that is either constant or decreasing in time.

One factor is that when using a Taylor approximation to discretize a PDE as described in chapter 2, only the first order term is taken. This means that an error is made in order of the square of the discretization dimension step size. This error can be seen as something called 'numerical diffusion'. It behaves like diffusion and is introduced as a result of truncating the Taylor expansion.

As explained in chapter 3, explicit methods' stability depends on the size of the time step chosen. Specifically, the time step must satisfy the Courant-Friedrichs-Lewy condition, or CFL condition, who derived it in 1928 [9].

$$C = \frac{\mathbf{u}_x \Delta t}{\Delta x} + \frac{\mathbf{u}_y \Delta t}{dy} \leq C_{max} \tag{3.5}$$

Where C is the Courant number, $\mathbf{u}_x$ and $\mathbf{u}_y$ the characteristic velocity in its respective dimensions, and C_{max} some number that depends on the PDE and the discretization method.

One way of describing the CFL condition is that for an explicit scheme, the speed at which information travels in a single timestep must not exceed the spacing of the grid. Since the Courant number is the ratio of information propagation distance to grid distance it follows that in an Euler forward case the Courant number must be less or equal to 1.

Higher order methods tend to use values from neighbors that are further away, for example the RK3 method has a CFL_{max} number of 3 since it uses spatial information from 3 grid points away [17].

However, the CFL condition is a necessary but not sufficient condition for stability. Usually a method's inherent stability region is decided using the so called test problem. The test problem is defined as

$$y' = \lambda y \tag{3.6}$$

When we apply the Taylor approximation as described in chapter 3 we obtain the following recurrence relation:

$$y_{n+1} = y_n + \Delta t \lambda y_n = (1 + \Delta t \lambda)\, y_n \tag{3.7}$$

It follows that if $|1 + \Delta t \lambda| > 1$ the solution will grow indefinitely over time, which leads to a restriction on the time step based on the value of λ.

Note that this condition means that for positive values of λ, the method is inherently unstable for problems that behave like the test problem. It follows explicit time integration is only a viable option when $\lambda \leq 0$. The stability region of the Euler backwards method is the complement of the region for Euler forward, which means that in such a case an implicit method should be used.

3.3. IMPLICIT TIME INTEGRATION

If $a = 1$ then the right-hand side of 3.2 depends solely on the function value $F(\phi_{n+1})$ and the complexity of finding ϕ_{n+1} is highly dependent on the function F.

This method is also called the Euler backward method and is considered an implicit method as ϕ_{n+1} is defined implicitly.

The big advantage of implicit time integration is that it is unconditionally stable with respect to the size of the time step. However, the numerical error of an implicit method is still dependent on the time step size which also needs to be considered when choosing time step size.

In the case of backwards Euler assuming as before F to be linear the update procedure can be expressed as:

$$(\mathbf{A}+I)\,\phi_{n+1}=\phi_n \tag{3.8}$$

Which is a system of linear equations which needs to be solved which is computationally expensive. Intuitively it appears equation 3.8 can be solved by determining the matrix inverse of $(\mathbf{A}+I)$ and left multiplying it with both sides of the equation. In practice this is never done due to cost and instead a variety of methods can be employed to obtain ϕ_{n+1} from equation 3.8, which will be discussed in chapter 5.
An advantage of an implicit method is that since the method is unconditionally stable often a larger time step can be used offset this. A drawback is that solving a system of linear equations is computationally expensive and not trivial to parallelize.

Chapter 5 describes various solving methods and discusses their suitability for GPU implementation.

3.3.1. MIXED AND SEMI-IMPLICIT METHODS

CRANK-NICHOLSON

In the last two sections we have described the simplest fully explicit or implicit time integration. However both these methods will only produce first order accurate solutions. This of course led to the development of more accurate schemes. For example, Crank and Nicolson [18]found that setting $a = 1/2$ in 3.2 leads to second-order numerical accuracy while preserving the unconditional stability of the Euler backwards method.

SEMI-IMPLICIT EULER

When Euler Forward proves to be unstable one option is to try to improve stability by using the semi-implicit Euler method. The semi-implicit method is a somewhat confusing name as no implicit time integration actually takes place. When time integrating a system of equations explicitly often multiple variables need to be updated every time step. In the case of the Shallow-Water equations 1.9 the water level, x-velocity and y-velocity all need to be updated. In the case of Euler forward all three variables are updated independently using values from the previous timestep.

The idea behind the semi-implicit Euler method updates the variables explicitly in sequential order, where once a variable has been updated the updated expression is used to update the other variables.

Other methods were developed by Runge and Kutta, of which their fourth order method is the most popular, which takes a weighted average of four different increments in order to achieve fourth order accuracy at the cost of additional computation.

ADI

Another interesting method which is very relevant to the shallow water equations is a semi implicit method called the alternating direction implicit method, or ADI. The idea is that for a coupled system of partial differential equations in two spatial directions x and y, a time step is split into two parts where first the x-derivative is calculated explicitly and the y-derivative implicitly, and for the next half time step this is reversed. This results into a tridiagonal system that needs to be solved twice at every time step, which is comparatively computationally cheap.
Aackermann & Pedersen [19] used this method do discretize the SWE and solve the resulting tridiagonal system on a GPU and concluded it was very efficient.

3.4. RUNGE-KUTTA METHODS

Around 1900 Carl Runge and Martin Kutta developed a family of implicit and explicit time integration methods [17]. As mentioned before the Runge-Kutta 4 method has remained very popular to this day. The family of explicit Runge-Kutta methods is given by the following expression:

$$\begin{aligned}
&u_{n+1} = u_n + h\sum_{i=1}^{s} b_i k_i \\
&k_1 = f(t_n, y_n) \\
&k_2 = f(t_n + c_2 h, y_n + h(a_{21}k_1)) \\
&k_3 = f(t_n + c_3 h, y_n + h(a_{31}k_1) + a_{32}k_2) \\
&\vdots \\
&k_s = f\left(t_n + c_s h, y_n + h\left[a_{s1}k_1 + \ldots + a_{s,s-1}k_{s-1}\right]\right)
\end{aligned} \tag{3.9}$$

Where u_n is the solution to the to be solved initial value problem at time $t = t_n$. a_{ij} are the coefficients and b_{ij} and c_{ij} are the weights. The weights and coefficients can be conveniently organised in a so called Butcher tableau, introduced by John C. Butcher 60 years after the RK methods were developed. [17]:

$$\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}$$

Table 3.1: Butcher tableau for weights and coefficients

The question now is which values to pick for the weights and coefficients. For a Runge-Kutta method of the above form to be consistent it is necessary that the sum of coefficients of each row i equals the row-weight c_i.

If this consistency requirement is applied to an RK method with only 1 stage it follows that Euler forward is the only consistent single stage method.

If the formula 3.9 is observed it can be concluded that if all nonzero coefficients lie on the bottom-triangular part of the Butcher tableau the method is explicit, and if they lie on the upper-triangular part the method is implicit.

The Runge-Kutta 4 method for example has the following tableau:

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	1/3	1/3	1/6

The choice of Runge-Kutta method is a trade-off between accuracy and computational intensity.
Butcher [17] shows that in order to obtain accuracy of order p the method must have a number of stages s equal to p for $s \leq 4$ and at least $p+1$ for $s \geq 5$.
This partially explains why the RK4 method is so popular, as it is the highest order method that has a number of stages equal to the order of accuracy. The exact relation between p and s is an open problem.

3.5. PARAREAL

An revolutionary concept in time integration is a method called Parareal [20]. When parallelizing solving a partial differential equation, usually the system of equations is solved in a parallel fashion but sequentially in time. Parareal however, attempts to obtain a higher level by parallelizing the method at the temporal stage. The main idea behind the method is to decompose the time interval over which the initial value problem is integrated into parts that are then assigned each to a parallel processor.

The idea is to have a coarse solving method that is executed serially for all time steps. If speedup is desired then the course method should be chosen in such a way that this serial execution is somewhat accurate and fast. If we denote the coarse method that calculates the solution u at time j given the solution at time $j-1$ by
$u_j = C(u_{j-1}, t_j, t_{j-1})$.

Secondly the solution is iteratively improved in parallel. If we denote the fine solver by $F(u_{j-1}, t_j, t_{j-1})$ and we denote the iteration number by superscript k we obtain the following procedure:

$$u_j^k = C(u_{j-1}^k, t_j, t_{j-1}) + F(u_{j-1}^{k-1}, t_j, t_{j-1}) - C(u_{j-1}^{k-1}, t_j, t_{j-1}) \tag{3.10}$$

It is obvious that if the course method converges, e.g $C(u_{j-1}^k, t_j, t_{j-1}) = C(u_{j-1}^{k-1}, t_j, t_{j-1})$ then the two course terms cancel out and only the fine solver term remains.

3.6. STELLING & DUINMEIJER SCHEME

Stelling & Duinmeijer [21] developed a first order finite difference scheme for the shallow water equations that can be modified for second order accuracy. The scheme is generic allowing for both an explicit and implicit implementation.
They start with the non-conservative two dimensional form given by

$$\frac{\partial \zeta}{\partial t} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} = 0 \tag{3.11}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + g\frac{\partial \zeta}{\partial x} + c_f\frac{u|\mathbf{u}|}{h} = 0 \tag{3.12}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + g\frac{\partial \zeta}{\partial y} + c_f\frac{v|\mathbf{u}|}{h} = 0 \tag{3.13}$$

Where u is the depth averaged flow velocity in the x-direction, v the flow velocity in the y-direction $\mathbf{u}$ the vector containing u and v, ζ the water level above the plane of reference, c_f the bottom friction coefficient, d the depth below the plane of reference and h the total water depth $h = \zeta + d$.

The scheme uses a staggered Arakawa C grid, see figure 2.3, to spatially decouple the values of h and u and v. Discretizing equation 3.11 and noting that the bottom height is time independent leads to

$$\frac{h_{i,j}^{n+1}-h_{i,j}^{n}}{\Delta t}+\frac{h'^{n}_{i+1/2,j}u^{n+\theta}_{i+1/2,j}-h'^{n}_{i-1/2,j}u^{n+\theta}_{i-1/2,j}}{\Delta x}+\frac{h'^{n}_{i,j+1/2}v^{n+\theta}_{i,j+1/2}-h'^{n}_{i,j-1/2}v^{n+\theta}_{i,j-1/2}}{\Delta y}=0 \tag{3.14}$$

Where $u^{n+\theta}=\theta u^{n+1}+(1-\theta)u^{n}$ and
$h'_{i+1/2,j}=h_{i,j}$ if $u_{i+1/2,j}>0$,
$h'_{i+1/2,j}=h_{i+1,j}$ if $u_{i+1/2,j}<0$ and
$h'_{i+1/2,j}=max(\zeta_{i,j},\zeta_{i}i+1,j)+min(d_{i,j},d_{i+1,j})$ if $u_{i+1/2,j}=0$

With rules analogous in the y-direction.

When discretizing equations 3.12 and 3.13 the question is how to approach the non-linear terms, which are the bed friction with a product of u, $|\mathbf{u}|$ and h, and the advection term which is a product of flow velocity and its spatial derivative.

Stelling & Duinmeijer propose two different approximations which can be used depending on which characteristics of the scheme are required. One is a momentum conservative advection approximation, the other an energy head $EH=\frac{u^2}{2g}+\zeta$ conserving approach.

For the momentum conservation the advection terms are approximated using first-order upwinding, which means the flow velocity takes on the values of neighboring points depending on the flow direction. This results in the following expression:

$$\begin{aligned}&\frac{du_{i+1/2,j}}{dt}+\frac{(\bar{q}_u^{x})_{i,j}}{\bar{h}^{x}_{i+1/2,j}}\frac{u_{i+1/2,j}-u_{i-1/2,j}}{\Delta x}+\frac{(\bar{q}_v^{x})_{i,j-1/2}}{\bar{h}^{x}_{i+1/2,j}}\frac{u_{i+1/2,j}-u_{i+1/2,j-1}}{\Delta y}\\&+g\frac{\zeta_{i+1,j}-\zeta_{i,j}}{\Delta x}+c_f\frac{u_{i+1/2,j}||\mathbf{u}_{i+1/2,j}||}{\bar{h}^{x}_{i+1/2,j}}=0\end{aligned} \tag{3.15}$$

Where $\frac{du_{i+1/2,j}}{dt}$ is the time derivative x-velocity u evaluated at grid point $(i+1/2,j)$, $q_u=uh$ and $\bar{h}^{x}_{i+1/2,j}=(h_{i,j}+h_{i+1,j})/2$, with the y equation defined analogously.

The energy-head conserving discretization in the x-direction is given by:

$$\begin{aligned}&\frac{du_{i+1/2,j}}{\Delta t}+\frac{u_{i+1/2,j}+u_{i-1/2,j}}{2}\frac{u_{i+1/2,j}-u_{i-1/2,j}}{\Delta x}+\frac{v_{i+1/2,j-1/2}+v_{i-1/2,j-1/2}}{2}\frac{u_{i+1/2,j}-u_{i+1/2,j-1}}{\Delta y}\\&+g\frac{\zeta_{i+1,j}-\zeta_{i,j}}{\Delta x}+c_f\frac{u_{i+1/2,j}||\mathbf{u}_{i+1/2,j}||}{\bar{h}^{x}_{i+1/2,j}}=0\end{aligned} \tag{3.16}$$

Stelling and Duinmeijer propose the following system of linearized equations based on the θ method that is momentum conservative for $\theta = 0.5$.

$$\frac{h_{i,j}^{n+1} - h_{i,j}^{n}}{\Delta t} + \frac{h'^{n}_{i+1/2,j} u^{n+\theta}_{i+1/2,j} - h'^{n}_{i-1/2,j} u^{n+\theta}_{i-1/2,j}}{\Delta x} + \frac{h'^{n}_{i,j+1/2} v^{n+\theta}_{i,j+1/2} - h'^{n}_{i,j-1/2} v^{n+\theta}_{i,j-1/2}}{\Delta y} = 0 \tag{3.17}$$

$$\frac{u^{n+1}_{i+1/2,j} - u^{n}_{i+1/2,j}}{\Delta t} + u^{n}_{\rightarrow} \frac{u^{n}_{i+1/2,j} - u^{n}_{i-1/2,j}}{\Delta x} + v^{n}_{\uparrow} \frac{u^{n}_{i+1/2,j} - u^{n}_{i+1/2,j-1}}{\Delta y}$$
$$+ g \frac{\zeta^{n+\theta}_{i+1,j} - \zeta^{n+\theta}_{i,j}}{\Delta x} + c_f \frac{u^{n+1}_{i+1/2,j} \left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^{n}_{i+1/2,j}} = 0 \tag{3.18}$$

$$\frac{v^{n+1}_{i,j+1/2} - u^{n}_{i,j+1/2}}{\Delta t} + u^{n}_{\rightarrow} \frac{v^{n}_{i,j+1/2} - v^{n}_{i-1,j+1/2}}{\Delta x} + v^{n}_{\uparrow} \frac{v^{n}_{i,j+1/2} - v^{n}_{i,j-1/2}}{\Delta y}$$
$$+ g \frac{\zeta^{n+\theta}_{i,j+1} - \zeta^{n+\theta}_{i,j}}{\Delta y} + c_f \frac{v^{n+1}_{i,j+1/2} \left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-y})^{n}_{i,j+1/2}} = 0 \tag{3.19}$$

With $(h^{-x})^{n}_{i+1/2,j} = \left(h^{n}_{i,j} + h^{n}_{i+1,j}\right)/2$ and $(h^{-y})^{n}_{i+1/2,j} = \left(h^{n}_{i,j} + h^{n}_{i,j+1}\right)/2$
and $u_{\rightarrow}$ and $v_{\uparrow}$ the convective velocity approximations, which can be either momentum-conservative or energy-conservative.

This system can be represented in matrix form similar to 1.11:
It is proposed that the scheme could be implemented dynamically, switching between the momentum- and energy-conserving algorithm depending on the magnitude of the spatial derivatives.

The system of equations that follows is symmetric and positive definite, which makes the implicit system suitable for the Conjugate Gradient method further discussed in chapter 5.

It is noted that the method can be constructed to be second order accurate by using upwinded second-order approximations instead of first-order in combination with so called 'slope limiters'. Slope limited approximations guarantee non-negative water levels for sufficiently small time steps. A slope limiter is added to the flow velocity terms and is a function of neighboring terms.

It is also important to note that the above approximations are all for positive flow direction. Because upwinding is used which depends on the flow direction, the upwinding terms change too when flow direction is reversed. This makes calculations complex for situations with often reversing flow directions, such as tidal simulations.

4

THE GRAPHICS PROCESSING UNIT (GPU)

4.1. INTRODUCTION

A graphics processing unit, or GPU, is a computer part that is primarily developed, designed and used to generate a stream of output images, computer graphics, to a display device. The most widespread use is to generate the output of a video game. However, in recent years their use for accelerating scientific computations has become an active research topic.

Historically, the field of scientific computing has focused and done most of said computing on the central processing unit, partially because the concept of a central processing unit came first and graphics processing units did not become mainstream until many years later. Early GPUs were designed and used exclusively for video game rendering.

Later it was discovered that the computing capabilities of a GPU could be harnessed for other uses, by rewriting problems and presenting them to the GPU as if it were a video game [22]. It was not until 2007 when Nvidia introduced the CUDA GPU programming framework that GPU computing became more accessible for mainstream use.
Modern GPUs have a large amount of computing cores that when utilized together in parallel provide a great deal of computing power at comparatively low monetary and energy cost. The challenges lie in rewriting programs or algorithms to be suitable for parallel computing and dealing with the other limitations of a GPU.

4.2. GPU STRUCTURE

4.2.1. ARCHITECTURE

As mentioned in the introduction, a GPU contains many cores. In the case of a central processing unit, a program contains a number of threads to be executed which are then mapped to the cores by the operating system. Due to the parallel nature of a GPU this process is a little more complex.

Computer architectures can be classified using Flynn's taxonomy [23]. A classical computer is classified as SISD, Single Instruction Single Datastream, which means a single program is executed on a single dataset sequentially. A GPU is considered a Single Instruction Multiple Datastream, or SIMD device. This means that a single instruction is run multiple times in parallel on different data.

A GPU does not simply contain a number of cores which can execute threads like a CPU. An Nvidia GPU consists of a number of streaming multiprocessors, or SM's, which each contain a number of CUDA cores which can perform floating point operations. Threads are grouped in blocks which are assigned to the SM's, which will explained further in section 4.2.2. Every SM can be considered an SIMD device, as blocks are assigned to the SM's.
As SMs can receive instructions that are not identical within the same program, the SIMD classification does not truly fit a GPU as a whole. Thankfully a new term has been coined: Single Program Multiple Datastreams, or SPMD.

For example, the Nvidia Turing TU102 GPU [24] contains 68 SMs, each with 64 CUDA cores for a total of 4352 cores. The cores have a clockrate of 1350mhz to 2200mhz and can perform 2 floating point operations (flops) per clock cycle.
This results in a total of roughly 12 to 19 Teraflops.
For comparison, an average modern desktop CPU has compute capability in the order of 100 Gigaflops. This means that a perfectly parallelizable program could run around 100 times faster when executed on the GPU.

4.2.2. Blocks & warps

As mentioned before, threads on a GPU are grouped per 32 in warps, which then are grouped together in blocks. This is schematically represented in figure 4.1.

When a program is executed on a GPU, every block in the program is assigned to an SM. If the number of blocks in the program exceeds the number of SMs, they will be executed sequentially. This is schematically represented in figure 4.2.

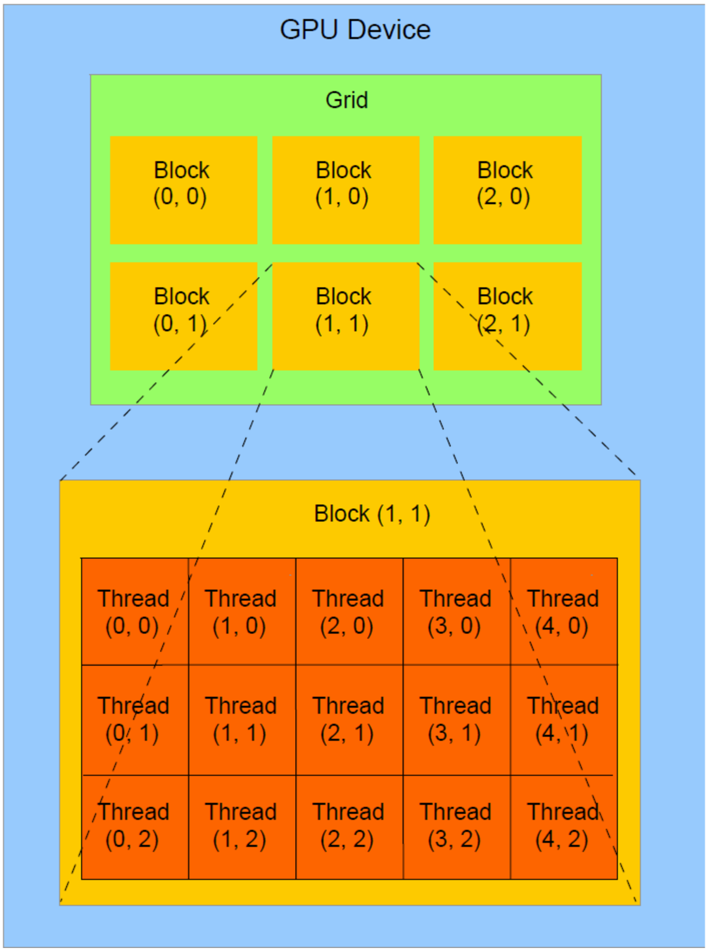


Figure 4.1: A schematic representation of GPU program structure Source: [25]

Because the program is divided into blocks which are then subdivided into warps, it is not self-evident how many blocks and how many warps per block should be chosen. To keep every SM active, there need to be at least as many blocks as there are SMs on the GPU. Since one SM can execute 32 threads, or 1 warp, at the same time, there should be at least 32 threads per block in order to have full GPU utilization. Memory restrictions complicate this a bit further, which are explained in the next section. There can be a maximum of 1024 threads in a single block on modern GPUs, and the maximum number of blocks is $2^{31} - 1$ or around 2 billion for modern GPUS.

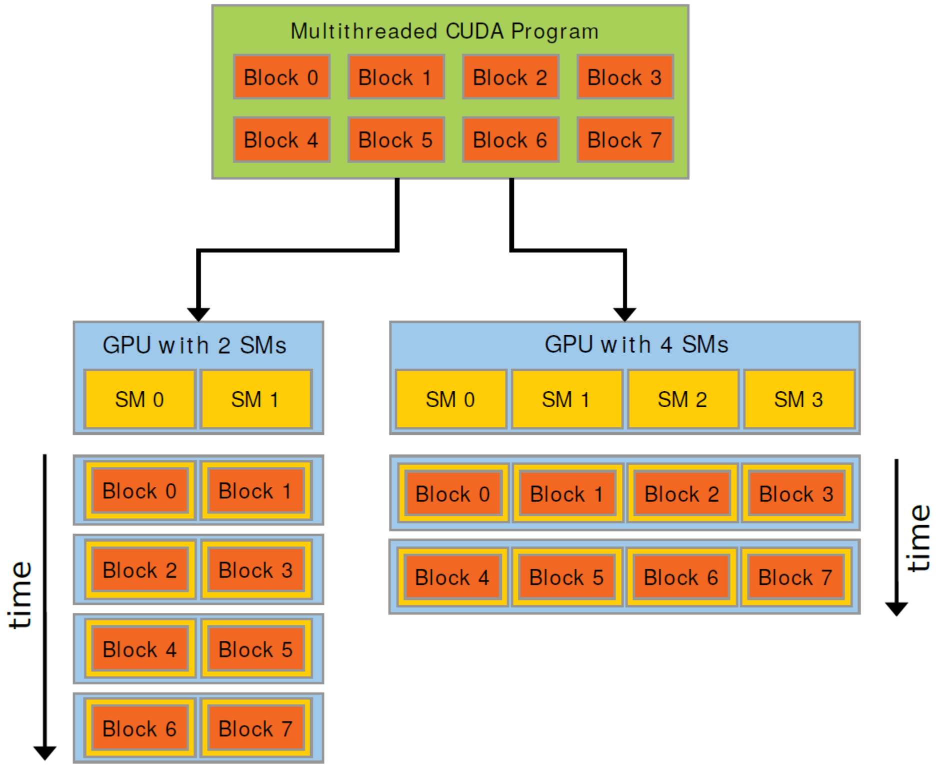


Figure 4.2: A schematic representation of GPU program execution structure Source: [25]

4.2.3. MEMORY

A CPU uses data that is stored in random access memory, or RAM. RAM is faster than storage media such as solid state drives, but it is expensive, has limited capacity and does not retain data when powered off. When running a program on the CPU the only constraint is that you do not exceed the system's RAM capacity.
A GPU's memory structure is more complex. In figure 4.3 the different types of memory a thread has access to is schematically represented.

SHARED MEMORY

Shared memory is arguably the most important memory on a GPU. Shared memory is very fast memory that is accessible to every thread in a block. This for example means that if a matrix matrix product is being done on a GPU, all threads can quickly add their result to the result matrix in the shared memory.

An important aspect to consider when deciding on block count and threads per block when designing a program is the shared memory use. The TU102 GPU has a maximum of 64Kb of shared memory per block. This means that if a block wants to efficiently use

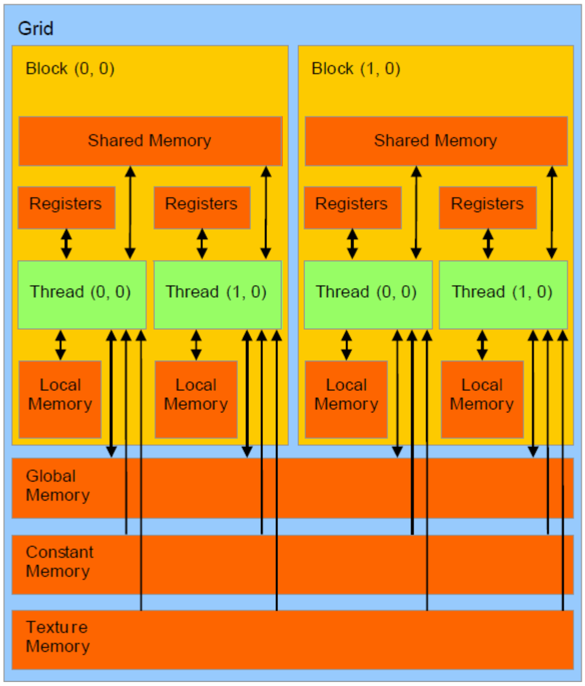


Figure 4.3: A schematic representation of GPU memory structure Source: [25]

shared memory the threads in the block must not occupy a total of more than 64Kb.

GLOBAL MEMORY

The bulk of the memory available within GPU is the global memory. It is generally faster than RAM, but the transfer of data from RAM to Global memory is through the PCI-E bus which has comparatively high latency and low .

REGISTERS

Register memory is extremely fast, but it is only accessible by a single thread and data stored in a register only lasts for the lifetime of the thread. This is usually where memory intensive operations are performed.

LOCAL MEMORY

Local memory is almost identical to registers, except it is off-chip and part of the global memory. The difference is that global memory can be accessed by every thread while local memory is a subsection of global memory that is reserved for a single thread. Because of this the amount of local memory available to a thread is larger than the register memory, but is as slow as global memory.

CONSTANT MEMORY

Constant memory is read-only memory that can only be modified by the host, usually the CPU. It is intended for data that will not change over the course of the program. It is optimized for broadcasting data to multiple threads at the same time which it does faster than global memory.

TEXTURE MEMORY

Texture memory is mainly used for storing video game textures. It is read-only in the same way as constant memory and has high latency, although it is still faster than global memory. However, an advantage of texture memory is that it does not share with global memory, which is beneficial for -limited applications. Texture memory is optimized for spatial locality [26], which means that threads in the same warp access data that is close together in memory will be faster.

Texture memory has some other functions that can be used for free, such as linear interpolation of adjacent data, automatic data normalization on fetch, and automatic boundary handling [26].

4.3. MEMORY BANDWIDTH AND LATENCY

When talking about GPU floating point Teraflops usually the maximum performance is meant, assuming that every GPU core is processing data at the same time. When performance of the execution of actual programs is measured, the throughput is often less than this theoretical maximum. This is because in order for the GPU cores to do calculations they need to be fed instructions and data. Memory bandwidth and latency limitations will often prevent this, and thus code optimization of a GPU program will often mean optimizing memory utilization.

4.3.1. BANDWIDTH

Memory bandwidth is defined as the maximum amount of data that can pass through the memory to the execution units. It is calculated with the following formula: $B = \frac{bw}{8} * mc$. Here B is the in bytes per second, bw is the memory bus width in bits, and mc is the memory clock in Hertz.
For example the TU102 GPU has a 352 bit Memory bus width which is 48 bytes. TU102 memory is GDDR6 with a base memory clock of 14 Ghz for a total bandwidth of 616 Gigabytes per second. If data needs to be sent from CPU RAM to the GPU execution cores this happens through the PCIE-bus. This bus also has a limited bandwidth which needs to be taken into account as well. Modern GPUs still use the PCI Express 3.0 x16 standard released in 2010, which has a maximum of 15,76 Gigabyte per second. Compared to the internal memory of the TU102 GPU this is slower by approximately a factor 40. This means that when doing calculations on a GPU with a very large dataset the limiting factor, also called bottleneck, will be the PCIE interface.

4.3.2. LATENCY

Bandwidth limits the maximum data transfer rate through a memory bus. This figure however is only important when a program is bottlenecked by the available bandwidth. However, even when transferring data and not exceeding the bandwidth capacity there is still a delay between the sending and receiving of the data. This is what is called the data latency, the time it takes for a single byte of data to transfer. Because any memory transfer takes at least as much time as the latency, it is advantageous to send as much data as possible in a single transfer operation. It should be noted that modern GPU devices have a number of techniques to hide latency.

4.3.3. BANDWIDTH LIMITATION EXAMPLE

To illustrate the bandwidth limitations of GPU computing capability, consider a matrix-vector product $b = Ax$ that is to be computed on a GPU in parallel with A an NxN matrix and x an Nx1 matrix.
Doing this calculation sequentially would require N vector-vector products which cost $N * t_1$ seconds for a total of $T_{seq} = N^2 t_1 = O(N^2)$, where t_1 the time it takes for a single flop on the sequential unit.

Doing the same calculation in parallel on P processors would take $T_{par} = \frac{N}{P} * N * t_2 = O(\frac{N^2}{P})$ seconds, as P operations are performed in parallel, where t_2 is the time it takes for a single flop on the parallel machine. Note that the maximum speedup would be achieved when using N parallel processors.

This sounds very appealing until communication time is taken into account, the matrix parts of A and the vector x need to be copied (also called scattering) to the parallel processors, and the result b needs to be copied back (also called gathering). The matrix part is $\frac{N^2}{P}$ copy operations for each processor and the two vectors are each $\frac{N}{P}$ operations.
This results in:
$T_{comm} = P\left[\frac{N^2}{P} + 2\frac{N}{P}\right] t_3 = O(N^2)$ Where t_3 is the time it takes per memory copy operation.
This means that if $\left(\frac{t_2}{P} + t_3\right) << N$ it follows that:
$T_{par_{tot}} = \frac{N^2}{P} t_2 + P\left[\frac{N^2}{P} + 2\frac{N}{P}\right] t_3 = O(N^2)$

Because in this case the sequential and the parallel implementation are of the same order of magnitude, any speedup achieved will be at most a constant factor. Thankfully, various other methods have been developed to work around this communication bottleneck, which will be described further in chapter 5.

The above example is a worst case scenario. One way to circumvent the memory bottleneck is to construct the matrix and vector on the GPU instead of constructing it on the CPU and then scattering it to the GPU.

4.3.4. ROOFLINE MODEL

A Parallel program will often run into some kind of bottleneck, as illustrated in the preceding section, that prevents it from utilizing the maximal computing capabilities of the device it runs on. Since bottlenecks lead to inefficiency, it is important for code writers to know what is the limiting factor. The idea behind the Roofline model is to provide a visual guide on what the limiting factors are. The most simple form of the roofline takes the minimum of two functions [27]:

$$R = min\{\pi, \beta * \frac{W}{Q}\} \tag{4.1}$$

Where R is the roofline, representing the performance bottleneck, π the peak device performance in flops, β the communication bandwidth in bytes, W the program arithmetic intensity in flops and Q the memory usage in bytes per second. An elementary example Roofline model is presented in figure 4.4.

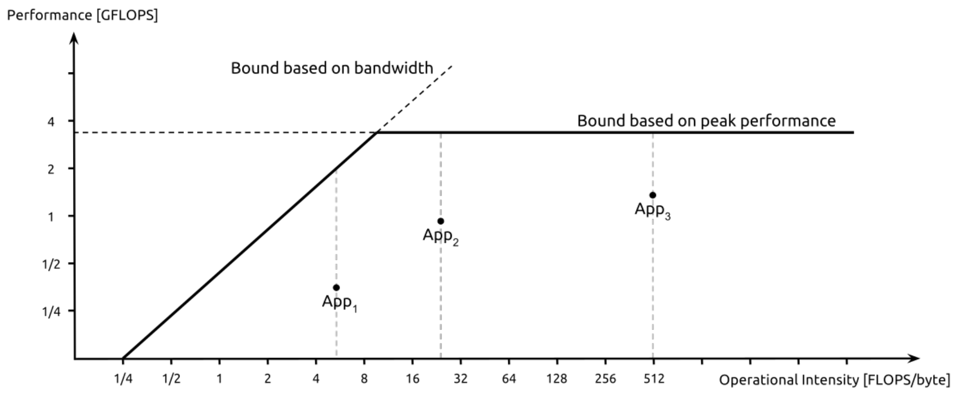


Figure 4.4: A schematic representation of the Roofline model,
maximum program Gflops vs program Operational intensity, where Operational Intensity is $\frac{W}{Q}$ Source: [28]

To increase the accuracy of the model, other limitations can be added to better indicate performance bottlenecks. These include concurrency or cache coherence effects in the memory category, in-core ceilings (lack of parallelism) limiting peak performance or locality walls which limit Operational Intensity [27].

4.4. COMPUTATIONAL PRECISION ON THE GPU

Data and variable values in a computer are often stored as floating point numbers. A floating point number consists of a significant, a base and an exponent. The value of the number is then equal to $F = S * B^E$, where F is the number being represented as a floating point, S the significant, B the base and E the exponent. Computers store data in bits and calculate in binary, and thus they do not have to store the base.

Several standards exist for floating point precision:

- Half precision or FP16: 1 sign bit, 5 exponent bits and 10 significant bits for a total of 16

- Single precision or FP32: 1 sign bit, 8 exponent bits, 23 significant bits for a total of 32

- Double precision or FP64 1 sign bit, 11 exponent bits, 52 significant bits for a total of 64

When using a CPU for computation, double precision calculations are just as fast as single precision. Double precision takes up twice the memory, however, which is something to consider when working with large datasets.

A GPU however is much more specialised. Most video games, which are still the main usecase of a GPU, do not require 64-bit precision. In order to save heat, memory and physical die space GPU CUDA cores were designed to only perform single or half precision flops.
Despite this, every SM has a small amount of FP64 cores. In the case of the TU102 GPU there are 2 FP64 cores per SM compared to 64 FP32 cores. This means that the GPU is able to perform floating point calculations up to 32 times faster when working in single precision, and this is often represented as with the FP64 to FP32 ratio 1:32.
Nvidia GPUs which are not specialized for FP64 computing have ratios between 1:8 and 1:32 [29], with the more modern architectures having the lower ratios.

When using GPUs for high performance scientific computing became more popular it lead to Nvidia developing the Tesla line of GPUs of which the first was the Fermi-based 20 series in 2011. The Tesla line has 1:4 to 1:2 FP64 compute capability and error correcting memory, but they are marketed towards enterprises at enterprise costs. For example a modern Tesla V100 GPU released in 2017 provides 7 Tflop at a release price of roughly 10.000 American dollars [30].

This changed with the release of the GTX Titan which was a consumer card and had an unprecedented 1:3 double precision ratio at a release price of 999 American dollars [31]. It provides 1.882 Tflop of double precision compute power.
The Titan was succeeded by the Titan V which has a 1:2 double precision ratio and provides 7.45 Tflop of double precision compute power. It was released in 2017 at a price of 3000 American dollars and is to this day has the most FP64 performance per dollar for an Nvidia GPU [32]. The Titans lack error correcting memory, however.

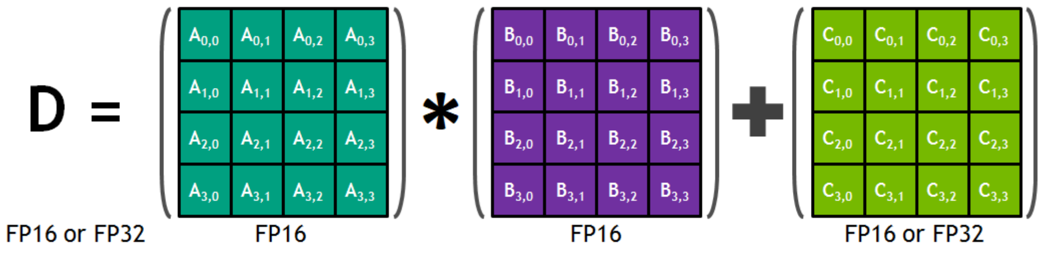
AMD gaming GPUs commonly have ratios between 1:8 to 1:16. Like Nvidia they also released a few consumer GPUs with FP64 ratios of 1:4. These include the Radeon HD7970 with .95 Tflop FP64 at a launch price of 550 USD in 2011, the Radeon R9 280 with 1.05 Tflop FP64 for 300 USD in 2013, and the Radeon VII with 3.36 Tflop for 699 USD in 2019.

AMD also has an enterprise line of double precision GPUs, the Radeon (Fire)Pro line going as far back as 1995. Despite many Radeon Pro GPUs having low FP64 ratios of around 1:16, the Radeon Pro drivers allowed the use of FP32 cores to work together to provide FP64 output at a 1:3 ratio [33].

Despite AMD GPUs providing more double precision flops per dollar, the maturity of the CUDA platform has led Nvidia to be the dominant player in the field of scientific computing [34].

4.5. GPU TENSOR CORES

Nvidia's Volta architecture of which the first GPU was released in 2017 was the first microarchitecture to feature Tensor cores. Tensor cores are a new kind of cores that were specifically designed to be very suitable for artificial intelligence and deep learning related workloads.
A single Tensor core provides a 4x4x4x4 processing array which performs a so called FMA, fused multiply addition, by multiplying two 4x4 matrices and adding the result to a third for 64 floating point operations per cycle. This has been schematically represented in figure 4.5.
The input matrices are FP16 precision, but even if the input is FP16 the accumulator can still be FP32. Because the operation then uses half-precision input to produce a single-precision result, this is also called mixed-precision.

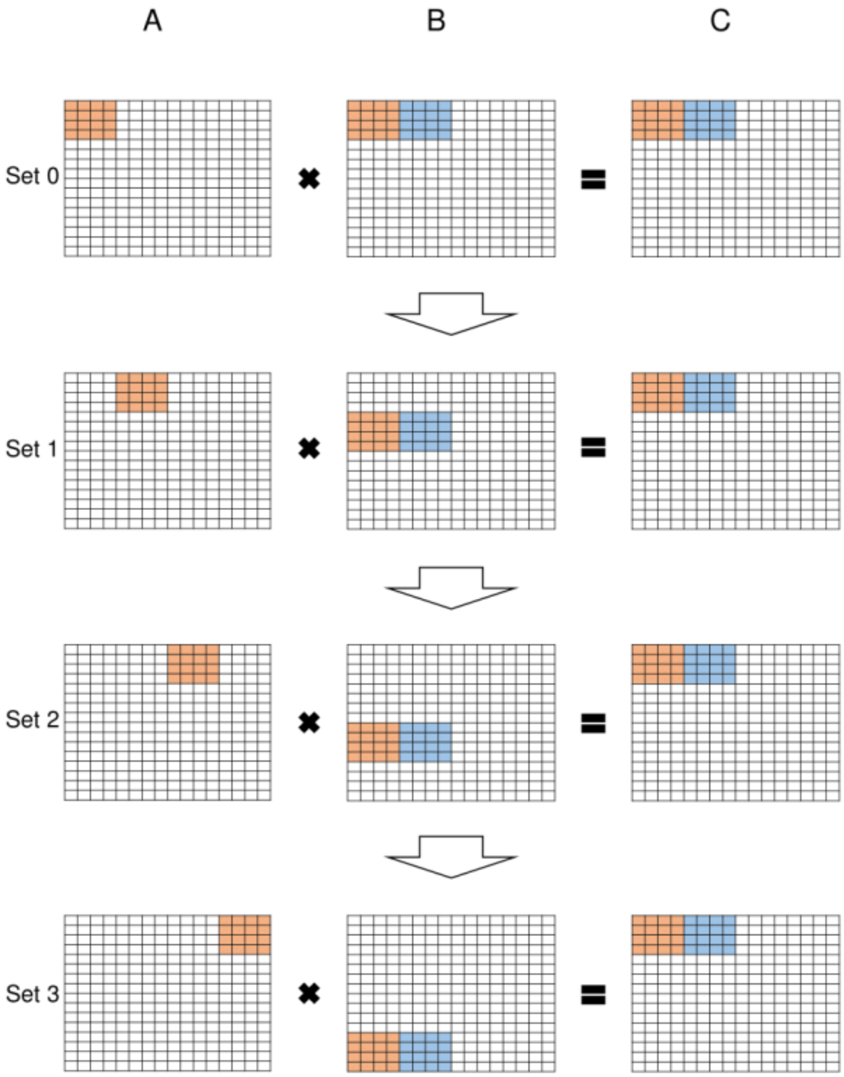


$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} * \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

Figure 4.5: A schematic representation of a Tensor core operation Source: [35]

The TU102 GPU contains 8 Tensor cores per SM, which work together to do a total of 1024 FP16 operations per clock cycle per SM. This concurrency allows the threads within a warp to perform FMA operations on 16x16 matrices every clock cycle. To achieve this the warp of 32 threads is split into 8 cooperative groups which each compute part of the 16x16 matrix in four sequential steps. These four steps for the first top-left group have been schematically represented in figure 4.6.

Tensor cores can be utilized by the CUDA cuBLAS GEMM library. BLAS stands for Basic Linear Algebra Subroutine and cuBLAS contains the fastest GPU basic linear algebra routines.
GEMM stands for General Matrix Multiply. However, in order for the cuBLAS GEMM to utilize Tensor cores, a few restrictions exist because of the 4x4 nature of the basic tensor core operation.
If the GEMM operation is represented as $D = A * B + C$ with A an $ldA * m$, B an $ldB * k$ and C an $ldC * n$ matrix then for the GEMM library to utilize Tensor cores the parameters ldA,ldB,ldC and k must be multiples of 8, and m must be a multiple of 4.
Other rules of matrix multiplication and addition apply too, so $m = ldB$ and $ldA = ldC$ and $k = n$.



Four sets of HMMA instructions complete 4×8 results in matrix C within thread group 0. Different sets use different elements in A and B. The instructions in set 0 execute first, then the instructions in set 1, set 2 and set 3. This way, the 16 HMMA instructions can correctly compute the 4×8 elements in matrix C.

Figure 4.6: A schematic representation of a Tensor core 16x16 operation Source: [36]

4.6. CUDA & OpenCl

There exist two major GPU programming languages: CUDA and OpenCl. CUDA stands for Compute Unified Device Architecture. When using GPUs for general purpose processing gained popularity it was developed by Nvidia and released in 2007. It is a proprietary framework and only compatible with Nvidia GPUs.
OpenCl stands for Open Compute Language which was developed by the Khronos group and released in 2009. It is a more general language for compute devices which include GPUs

CUDA has the advantage of being easier to work with, and also has a variety of tools and profilers have been built by Nvidia to aid development.
OpenCl has the advantage of supporting other GPU brands, with Advanced Micro Devices being the most prominent.

A study done by the TU Delft [37] found that translating a CUDA program into OpenCl reduced performance by up to 30%, but this difference disappeared when the corresponding OpenCl specific optimizations were performed.

A third, less known programming standard was exists called OpenACC, or open accelerators, with the aim to simplify parallel programming on heterogeneous systems, of which the first version was released in 2012 [38]. An advantage of OpenAcc is that it is easier to work with than OpenCl and CUDA but less efficient, according to [39].

4.7. CUDA program structure

In order for a program to be executed on a GPU it must be started from a ’host’, generally the system’s CPU, and also receive the relevant data from the host. In the case of CUDA it has its own compiler called nvcc. It compiles both the host code which is compiled in the C language, and the GPU code which are combined into a single .cu source file.
In the case of CUDA Fortran a program can be compiled using the PGI compiler from The Portland Group [40]. Third party wrappers are available for a variety of languages such as Python, Java, Matlab and OpenCL.

A generic CUDA program structure consists of the following steps:

1. Initialize host program, load CUDA libraries and declare variables
2. Allocate memory on GPU and send data from host
3. Launch kernel on GPU
4. Collect kernel result and send it to the host
5. Process result on host

This structure has been illustrated in figure 4.7:

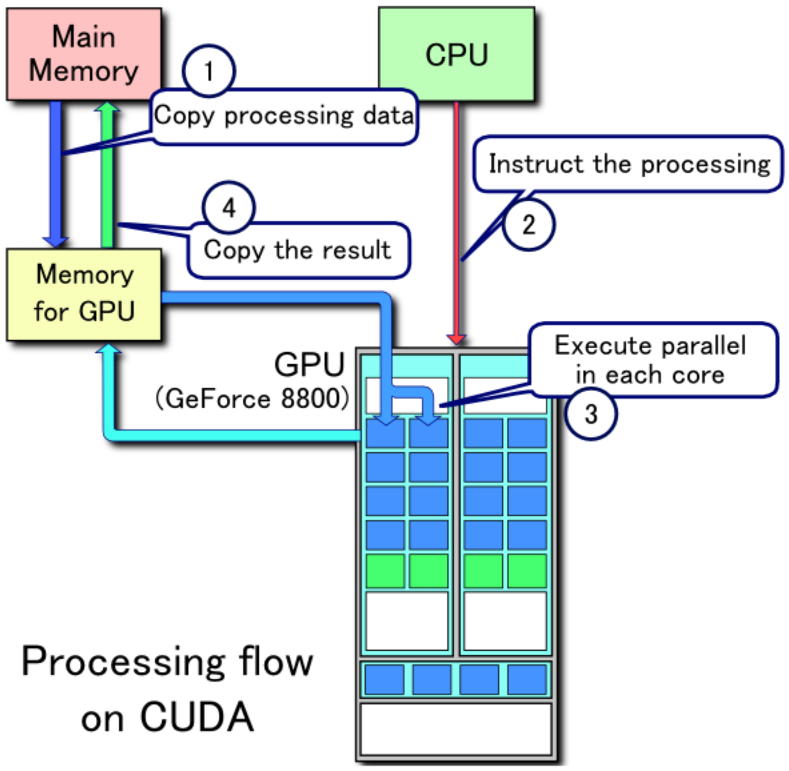


Figure 4.7: A schematic representation of CUDA processing flow Source: [41]

5
Parallel solvers on the GPU

5.1. Introduction

As mentioned in chapter 4, a GPU is a device with an enormous amount of computing power. In chapter 3 was explained that an implicit time integration method requires solving a linear system of equations $Ax = b$ at every time step. The majority of this chapter aims to describe the various methods that exist for efficiently solving systems of linear equations and how the methods can be adapted to be used in parallel on a GPU.

5.2. Matrix structure and storage

As mentioned in chapter 3 time integrating a discretized system of partial differential equations leads to a matrix equation. Often the differential equation is structured on the domain, and can be represented in stencil notation. This means that the matrix will have a band structure, with only a few diagonals filled and everything else zeros. This is also called a sparse matrix.

When storing matrix values into computer memory, it does not make sense to also store the components with value zero. If only the non zero entries of the matrix are stored, the memory footprint is greatly reduced but also calculations are sped up. These two qualities have led to the development of various methods to efficiently store large sparse matrices.

5.2.1. Construction formats

There are two elementary categories. The first are efficient modification systems, which are generally used for constructing sparse matrices such as:

- Coordinate list, a list that contains triples of coordinates and their values.
- Dictionary of keys, a dictionary-structure that maps coordinate pairs to corresponding matrix entry values.

- List of lists, which is a list that stores every column as another list

After constructing the matrix in a construction format it is usually then converted to a more computationally efficient format.

5.2.2. COMPRESSED FORMATS

The second are the Yale and compressed sparse row/column formats. These compressed formats reduce memory footprint without impeding access times or matrix operations [42]

The Yale format stores a matrix A using three one dimensional arrays:

- AA an array of all nonzero entries in row-major order, which means the index loops through the matrix per row.
- IA is an array of integers that contains the index in AA of the first element of the row, followed by the total number of non-zero entries plus one.
- JA contains the column index of each element of IA

Compressed sparse row format, or CSR, is effectively the same as Yale format except JA is stored second and IA is stored third.

Compressed sparse column format, or CSP is 'transposed' CSR. Here IA contains the index in AA of the first element of each column of A, and JA contains the row index of each element of IA.

For example, the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix} \tag{5.1}$$

Is represented in CSR format by:

AA	1	2	3	4	5	6	7	8	9	10	11	12
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

5.2.3. DIAGONAL FORMATS

If a sparse matrix contains only a small amount of non-zero diagonals (for example the ADI method produces a tridiagonal system), even more efficient storage methods can be used to exploit this. The simplest is to only store the diagonals in a "rectified" array as one vector per diagonal. In this case the offset of a row is equal to the column index which makes matrix reconstruction simple.

A slightly more complex method is Modified Sparse Row format, or MSR. The MSR format has just two arrays, AA and JA. The first N elements of AA contain the main diagonal of A. Starting at $N+2$ the array contains all other non-zero elements of A in row-major order. The elements starting at $N+2$ of JA contain the column index of the corresponding elements of AA. The first $N+1$ positions contain the pointer to the beginning of each row in AA and JA.

A third scheme suited for matrices with a diagonal structure is the Ellpack-Itpack format. If the number of diagonals is nd, the scheme stores two Nxnd arrays called COEF and JCOEF. Every row in COEF containing the elements on that row in A, very similar to the trivial storage method. The integer array JCOEF contains the column positions of every entry in COEF.

5.2.4. BLOCK COMPRESSED ROW FORMAT

After discretization of the shallow water equations we have a system of three equations and three unknowns per grid point if a collocated grid is used. In this case every element of A is not a value but instead a diagonal 3x3 matrix. The three vectors are the same as in normal CSR except that the AA array is now not one dimensional but stores the diagonal of each submatrix as a vector.

If the submatrices are dense instead of diagonal, the array becomes three dimensional and the entire submatrix is stored.

5.3. EXPLICIT METHODS

As mentioned in chapter 3, an explicit time integration method means that every timestep a matrix vector multiply needs to be performed. As demonstrated in the example in section 4.3.3 this operation is highly parallelizable but primarily memory-bound. This means that when an explicit method is implemented on a GPU, memory optimizations should be performed to assure data locality and optimize shared memory usage.

5.4. DIRECT SOLUTION METHODS

When solving an $Ax = b$ problem, with A a square matrix and x and b vectors, two classes of solution methods exist: direct solution methods and iterative solution methods. A direct solution method solves the system of equations in a few computationally expensive steps. An iterative solution method uses an iterative process that is computationally light which is repeated until the solution is accurate enough. It is also possible to combine the two methods.

The most simple way of solving a linear system is by means of Gaussian Elimination, also called sweeping. Since for a set of linear equations it is possible to perform linear operations on the equations without changing the solution, the matrix can be reduced to the identity matrix by this method which provides the solution. Performing these linear operations takes in order of N^3 operations, if A is an NxN matrix. Most direct solution methods also rely on Gaussian elimination, but aim to have a computational cost that is smaller than calculating the full matrix inverse.

5.4.1. LU DECOMPOSITION

The idea behind LU decomposition is that if it is possible to write the matrix as

$$A = LU \tag{5.2}$$

Where L is a lower triangular matrix and U an upper triangular matrix.
The system of equations $Ax = b$ can then be solved in two steps by introducing an auxiliary vector w and solving the following two systems:

$$Lw = b \tag{5.3}$$
$$Uu = w \tag{5.4}$$
$$\tag{5.5}$$

Solving these two systems is computationally cheap. The difficulty lies in factorizing A as the product of L and U.

It is first important to check in which case an LU-factorization exists and is unique. The LU-factorization of A can be proven to exist if all principal submatrices are non-singular. A principal submatrix of a matrix consists of the first k rows and columns, for $1 \leq k \leq N$. A non-singular matrix is a matrix that has an inverse, which coincides with having a non-zero determinant.

The LU-factorization can be shown to be unique if either U or L has a main diagonal that consists only of ones.

The simplest way to compute an LU factorization is to perform a sequence of row operations that bring A to upper triangular form through Gaussian elimination. If we represent the matrix A as

$$\begin{bmatrix} l_{11} & \\ \mathbf{l}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} u_{11} & \mathbf{u}_{12} \\ & \mathbf{U}_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & \mathbf{a}_{12} \\ \mathbf{a}_{21} & \mathbf{A}_{22} \end{bmatrix} \tag{5.6}$$

Where a_{11} is the matrix element at index $(1,1)$, $\mathbf{a}_{12}$ the remaining 1x(N-1) first matrix row and $\mathbf{a}_{21}$ the remaining (N-1)x1 first column and $\mathbf{A}_{22}$ the N-1xN-1 trailing matrix after removal of the first row and column.

Because $l_{11} = 1$ we know $u_{11} = a_{11}$ and $u_{12} = a_{12}$ and $l_{21} = a_{21}./a_{11}$. What remains is the trailing matrix update $L_{22}U_{22} = A_{22} - l_{21}u_{21}$.

Observe that this method factorizes a single row-column of the original matrix per step. This method is not well suited for parallel implementation on a GPU as it is inherently sequential: it is only possible to start factorizing the next row-column pair after the trailing matrix update has been performed. This method is also called the right-looking method since it moves through the columns from left to right and updates the trailing matrix on the right side.

It can be shown that using this method the computational cost of solving the linear system is $O\left(\frac{2}{3}N^3\right)$

PIVOTING AND FILL-IN

In the LU factorization algorithm described in the previous section, the column update involves dividing the values on the column by the value on the diagonal of that column in the original matrix A. If the value on the diagonal is very small then the updated column values will become very large, leading to an ill-conditioned matrix which makes the solution unreliable. If the value on the diagonal is 0 the algorithm breaks down. Therefore it is important that the values on the diagonal of A are not too small and of comparable size.

Fortunately which values lie on the diagonal is flexible since for a system of linear equations the order is irrelevant and can be shuffled as desired. The process of swapping rows to make sure the diagonal of the matrix contains desirable values is also called pivoting.

Another reason to use row pivoting is to reduce an effect called fill-in. A big problem that factorization algorithms have is that if A is a sparse matrix with a certain band width, this does not guarantee that L and U have comparable bandwidth. In certain cases it is possible for the matrix L and U to be almost full matrices in their nonempty sections, which is inefficient both computationally and memory wise.

This is why most factorization algorithms also have a so called "preordering" phase, where the order of equations is changed in such a way that predicted fill-in is minimal. This is usually achieved by reordering the matrix A in such a way that the densest rows are in the lowest part of the matrix and the densest columns are in the rightmost part.

PARALLEL SPARSE LU FACTORIZATION ON A GPU BY KAI HE ET AL.

Kai He et al. [43] developed a parallel column-based right-looking LU factorization algorithm designed for the GPU. In order to parallelize the factorization they perform a symbolic analysis to predict non-zero LU factors, after which data dependence between columns can be identified. Every dependency introduces a new graph layer and columns in the same layer are independent and thus can be updated in parallel. This process is represented in figures 5.2 and 5.1.
For example, the first row has a non-zero right looking entry on column 8, which means that column 8 must be factorized after column 1. Subsequently, the second row has a non-zero entry on column 4, which means column 4 must be factorized after column 2. If this process is repeated for every row we obtain the top graph in figure 5.1.

Since the column levels are factorized sequentially it is important to distribute the work among levels as equally as possible. This is to prevent a single level running into a bottleneck while on other levels the majority the GPU is idling. This technique is also called load balancing. This is why in figure 5.1 the levels are redistributed in such a way that the first level contains three columns and the third contains two, instead of four and one. The maximum number of columns per level should be chosen in such a way that the GPU occupancy is maximal but not exceeded.

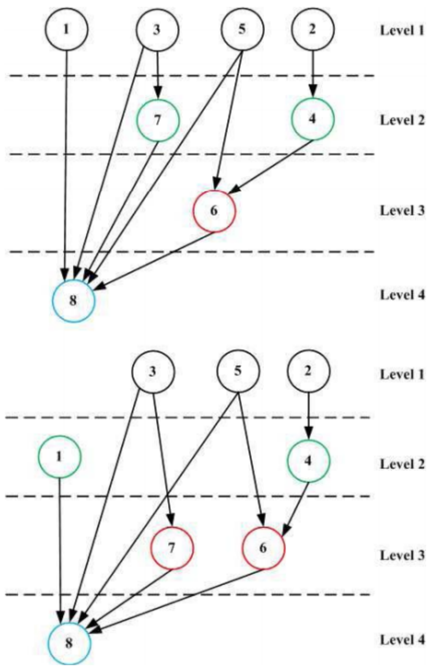


Figure 5.1: Levelization tree of the columns of the right matrix shown in figure 5.2.
The top figure shows naive column leveling, the bottom figure shows equalized column leveling.
Source: [43]

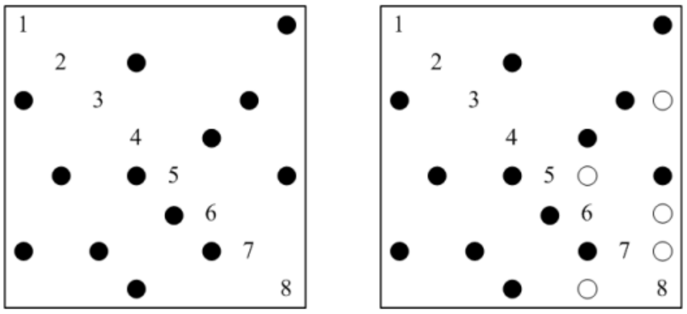


Figure 5.2: Representation of expected fill-in of a simple matrix with 8 rows and columns, where the white entries on the right are the predicted fill-in elements.
Source: [43]

After the preprocessing algorithm then consists of two steps:
First all columns of the L matrix in the current level are computed in parallel.
Then the subcolumns of the trailing matrix which depend on the corresponding columns in the L matrix need to be updated, which can also be done in parallel.
These two steps are repeated for every sequential level that was constructed during preprocessing.

CHOLESKY DECOMPOSITION

If the matrix A is symmetric and positive definite (SPD), the LU-decomposition reduces to its so-called Cholesky decomposition, which means it is possible to write

$$A = CC^T \tag{5.7}$$

Where C is a lower triangular matrix and C^T its transpose.

The fact that only a single lower triangular matrix needs to be computed theoretically cuts memory requirements and the necessary number of flops in half, which makes Cholesky decomposition very attractive. Furthermore, because A is positive definite in this case this guarantees non-zero diagonal elements which means no partial pivoting is needed.

5.5. ITERATIVE SOLUTION METHODS

As mentioned before, the second class of linear system solvers is the iterative solution methods. Instead of computing a solution directly instead an iterative process is used whose result converges to the exact solution. Two main classes of iterative solution methods exists, namely the basic iterative methods and the Krylov subspace methods, which will be covered in their respective subsections.

When solving a system $Ax = b$ using an iterative method, we call the k'th approximation of the solution x^k. If the true solution is x, the solution error at step k is defined as

$$e^k = x - x^k \tag{5.8}$$

The problem however is that knowing the error is equivalent to knowing the true solution. Therefore instead often the residual vector r^k is used as a measure of the error. The residual vector follows from the fact that

$$Ax^k = b + r^k \tag{5.9}$$

and thus

$$r^k = b - Ax^k \tag{5.10}$$

5.5.1. BASIC ITERATIVE METHODS

A basic iterative method is a method that uses a splitting of the matrix A by defining a non-singular matrix M such that $A = M - N$ in order to obtain a recursion relation for the solution approximation in the following way:

$$\begin{aligned} Ax &= b \\ Mx &= Nx + b \\ x &= M^{-1}Nx + M^{-1}b \\ x &= M^{-1}(M - A)x + M^{-1}b \\ x &= x + M^{-1}(b - Ax) \end{aligned} \tag{5.11}$$

Now since $Ax = b$ we have essentially written $x = x$ in a fancy way.
However remember that if we do not take x but instead substitute the approximate solution x^k then $b - Ax^k = r^k$. This suggests the expression can be used to define the recurrence relation:

$$x^{k+1} = x^k + M^{-1}r^k \tag{5.12}$$

Which is the basis for all basic iterative methods. The question now becomes how to define the matrix M. Since the matrix M is inverted it must be the case that it is much easier to invert M than to invert A, otherwise the method provides no advantage.

JACOBI METHOD

The simplest iterative method is the method of Jacobi, named after Carl Gustav Jacob Jacobi (1804-1851) who presumably was the first to propose the method. As mentioned before, the matrix M should be easily invertible. The method of Jacobi consists of choosing M to be the diagonal of the matrix A which we denote as D. In this case inverting M is a matter of simply replacing every non-zero value on the diagonal by its reciprocal value.

Because the Jacobi method involves multiplications with a diagonal matrix it means that all components of the vector x^k are updated independently of each other. This makes the method inherently parallel and thus well suited for GPU implementation.

GAUSS-SEIDEL METHOD

The Gauss-Seidel method, named after Carl Friedrich Gauss (1777-1855) and Philipp Ludwig von Seidel (1821-1896), is another basic iterative method.
Where the Jacobi method chooses the diagonal of A as M matrix, the Gauss-Seidel method instead takes the diagonal and the lower triangular part of A. If we call the strictly lower triangular part of A E and the strictly upper part F and insert these expressions into 5.12 after some reshuffling the Gauss-Seidel recursion relation can be written as

$$x^{k+1} = D^{-1}\left(f - Ex^{k+1} - Fx^k\right) \tag{5.13}$$

This may not look like a good recurrence relation because x^{k+1} exists on both sides of the equals sign. However it is important to note that on the right-hand side x^{k+1} is multiplied by a strictly upper triangular matrix. This means that to calculate the nth component of x^{k+1}, only the values x_1^{k+1} through x_{n-1}^{k+1} are necessary. In other words, the Gauss-Seidel method uses newly calculated components of x^{k+1} as soon as they become available.

This means that the Gauss-Seidel method converges faster than the Jacobi method, but is inherently sequential making it ill-suited for parallel implementation on a GPU.

RED-BLACK ORDERING

As mentioned before, a linear system of equations may be reordered to aid computations. In the case of the Gauss-Seidel method, the structure of the method makes it inherently sequential which makes the method ill-suited for parallel computing. Reordering provides the solution to this problem.

The Gauss-Seidel method is sequential because nodes require the computed values from neighboring nodes in order to do their own computations. If nodes are marked either red or black, with black nodes surrounded by only red nodes and vice versa, a checkerboard configuration is obtained.

The advantage here lies in that red nodes are surrounded by only black nodes thus only require information from black nodes to update their own values. This means that if all red values are known, subsequently all black values can be computed independently and thus in parallel. In the next step, the roles of red and black are reversed.

A red-black ordered matrix problem has the form

$$A = \begin{bmatrix} D_R & C^T \\ C & D_B \end{bmatrix} \tag{5.14}$$

Where D_R is a diagonal block matrix corresponding to the red nodes, D_B a diagonal matrix corresponding to the black nodes and C a matrix representing the connectivity of the nodes. A simple example for a 4x4 tridiagonal system can be seen in figure 5.3

$$\left|\begin{array}{cccc} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{array}\right| \left|\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}\right| = \left|\begin{array}{c} b_1 \\ b_2 \\ b_3 \\ b_4 \end{array}\right| \rightarrow \left|\begin{array}{cccc} 2 & 0 & -1 & 0 \\ 0 & 2 & -1 & -1 \\ -1 & -1 & 2 & 0 \\ 0 & -1 & 0 & 2 \end{array}\right| \left|\begin{array}{c} x_1 \\ x_3 \\ x_2 \\ x_4 \end{array}\right| = \left|\begin{array}{c} b_1 \\ b_3 \\ b_2 \\ b_4 \end{array}\right|$$

Figure 5.3: Example of reordering of a tridiagonal 4x4 matrix in Red-Black format
Source: [44]

Note that if the connectivity of the system of equations is such that nodes are connected not only in the x or y direction but also in the xy directions, as for example with a 9-point stencil, the Red-Black ordering does no longer work and more colors are needed.

5.5.2. CONVERGENCE CRITERIA

In the introduction section to basic iterative methods the error vector at step k e^k was defined as $x - x^k$, the difference between the approximation of the solution and the true solution. The error vector is not very useful during calculations since it cannot be computed without knowing the true solution. However if we use relation 5.12 we can define:

$$\begin{aligned} e^{k+1} &= x - x^{k+1} \\ &= x - x^k - M^{-1} r^k \\ &= e^k - M^{-1} A e^k \\ &= \left(I - M^{-1} A\right) e^k \\ &= \left(I - M^{-1} A\right)^{k+1} e^0 \end{aligned} \tag{5.15}$$

Intuitively this means that if the matrix $\left(I - M^{-1}A\right) = B$ makes the vector it is right-multiplied with smaller, i.e. B has an operator norm $||B|| < 1$ then the error will decrease every time step. This is a vague statement as a definition of the operator norm of B was not given, but there exist a more precise mathematical definition of the convergence criteria:

$$\rho\left(I - M^{-1}A\right) < 1 \iff \lim_{k\to\infty} e^k = 0 \iff \lim_{k\to\infty} x^k = x \tag{5.16}$$

Where the function ρ is the spectral radius of the matrix which is equal to its largest eigenvalue in absolute value.
This spectral radius also determines the convergence speed. If it is close to 1 the convergence will be very slow, while if it is small convergence will be fast.

5.5.3. DAMPING METHODS

From the definition of the Jacobi iteration matrix $M = D$ it follows that the error propagation matrix of the Jacobi method $B = I - D^{-1}A = E + F$. This means that if the diagonal of A is small compared to the upper and lower triangular parts E and F the Jacobi method will not converge. This problem can be solved by introducing a damping parameter ω:

$$x^{k+1} = (1-\omega)\, u^k + \omega x_{JAC}^{k+1} \tag{5.17}$$

Where x_{JAC}^{k+1} is the original value of x^{k+1} calculated with the Jacobi method.

It follows that the new error propagation matrix $B_{Jac} = I - \omega D^{-1} A$. Which means the parameter ω may be used to adjust the convergence rate depending on A.

This same damping strategy can also be used to modify the Gauss-Seidel method, after which it is called the Successive Overrelaxation method, or SOR. Again the new recurrence relation is

$$x^{k+1} = (1-\omega)x^k + \omega x_{GS}^{k+1} \tag{5.18}$$

which written in matrix-vector form is:

$$(D + \omega E)\, x^{k+1} = (1-\omega) D x^k - \omega F u^k + \omega b \tag{5.19}$$

It follows that $M_{SOR(\omega)} = \frac{D}{\omega} + E$.

Another variant is the symmetric SOR method, which consist of a forward and backward step with $M_1 = \frac{D}{\omega} + E$ and $M_2 = \frac{D}{\omega} + F$. The resulting product iteration matrix is

$$M_{SSOR(\omega)} = \frac{1}{\omega(2-\omega)} (D+\omega E) D^{-1} (D+\omega F) \tag{5.20}$$

5.6. CONJUGATE GRADIENT METHOD

The conjugate gradient method is one of the most efficient methods for solving linear systems that it can be applied to. For this reason it is one of the more popular methods used today. This section will describe some of the theory on which the method was built, followed by an analysis of the methods properties.

5.6.1. THE KRYLOV SUBSPACE

The conjugate gradient method falls in the category of the so called the Krylov subspace methods. One of the problems with the basic iterative methods described in section 5.5 is that while they are more suited for large problems than the direct solution methods of 5.4, their convergence speed is linear. As will be shown later, an advantage of Krylov subspace methods is that they will exhibit superlinear convergence behaviour under the right circumstances. A disadvantage is that they are not suitable for all problems, and adapting the problem to suit the method may be nontrivial.

The Krylov subspace is named after Alexei Krylov (1863-1945) who was the first to introduce the concept in 1931. The Krylov subspace appears organically when one examines the recursion relation of basic iterative methods 5.12:

$$\begin{aligned}
x^{k+1} &= x^k + M^{-1}r^k = x^k + M^{-1}\left(b - Ax^k\right) \\
x^{k+2} &= x^{k+1} + M^{-1}\left(b - Ax^{k+1}\right) \\
x^{k+2} &= x^k + M^{-1}r^k + M^{-1}\left(b - A\left(x^k + M^{-1}r^k\right)\right) \\
x^{k+2} &= x^k + M^{-1}r^k + M^{-1}\left(b - Ax^k\right) + M^{-1}AM^{-1}r^k \\
x^{k+2} &= x^k + 2M^{-1}r^k + M^{-1}AM^{-1}r^k \\
x^{k+3} &= x^k + C_1M^{-1}r^k + C_2M^{-1}AM^{-1}r^k + C_3\left(M^{-1}A\right)^2 M^{-1}r^k
\end{aligned} \tag{5.21}$$

Since this also holds for $k=0$ it follows that x^k is some linear combination of powers of $M^{-1}A$ multiplied with $M^{-1}r^0$.
In other words:

$$x^k \in x^0 + span\left\{M^{-1}r^0, M^{-1}AM^{-1}r^0, \dots, \left(M^{-1}A\right)^{k-1} M^{-1}r^0\right\} \tag{5.22}$$

A Krylov subspace of order r generated by an $N \times N$ matrix A and an $N \times 1$ vector b is defined as the span of the images of b under the first r powers of A:

$$K^k(A,b) = span\left\{b, Ab, \dots, A^{k-1}b\right\} \tag{5.23}$$

It follows that the BIM solution at step k is equal to the starting guess x^0 plus some element of the Krylov subspace $K^k\left(M^{-1}A, M^{-1}r^0\right)$.

The fact that the iterative method converges to the true solution x means that this true solution must also lie in this subspace. This means the structure of this subspace may be exploited in order to find the solution in an efficient fashion.

The conjugate gradient method is a method that takes advantage of the Krylov subspace nature of iterative solutions. It was discovered independently many times in the early 20th century, but the first paper on it was published by Stiefel and Hestenes in 1951 [45]. It is a very popular method because it is well suited for sparse linear systems and convergence is superlinear under the right circumstances.

5.6.2. THE METHOD OF GRADIENT DESCENT

The conjugate gradient method is a modification of the method of gradient descent [46]. The method of gradient descent is a very intuitive method. If one imagines himself at night and walking around a hilly landscape with the objective to find the lowest point in the area, traversing in the direction of steepest descent is guaranteed to lead to a local minimum. An illustration of this can be observed in figure 5.4.

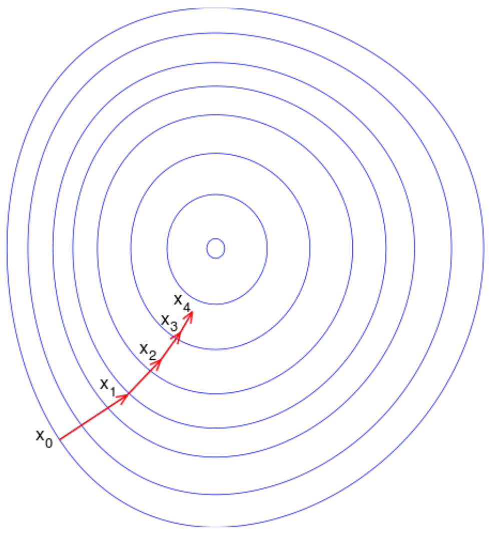


Figure 5.4: Illustration of successive steps of the Gradient Descent method
Source: [47]

The method of gradient descent is based on the same principle. In mathematical terms in the case of the linear problem $Ax = b$ this means the objective of the method is to find a minimum of the function

$$F(\mathbf{x}) = \| A\mathbf{x} - \mathbf{b}\|_2 \tag{5.24}$$

Where the lower case 2 means the L^2 norm or Euclidian norm.

Now the direction of steepest descent in a point x is minus the gradient of the function evaluated at that point:

$$-\nabla F(\mathbf{x}) = -\begin{bmatrix} \frac{\partial F(\mathbf{x})}{\partial x_1} \\ \frac{\partial F(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial F(\mathbf{x})}{\partial x_n} \end{bmatrix} \tag{5.25}$$

Because $F(\mathbf{x})$ is a linear function the gradient can be conveniently written as a matrix vector product:

$$\nabla F(\mathbf{x}) = 2A^T\left(A\mathbf{x} - b\right) \tag{5.26}$$

If we substitute $\mathbf{x} = x^k$ we find

$$\nabla F(x^k) = 2A^T r^k \tag{5.27}$$

This means the direction of steepest descent in a point x^k lies in the direction of the residual in that point left-multiplied by the transpose of the problem matrix A:

$$x^{k+1} = x^k + \alpha A^T r^k \tag{5.28}$$

The question is now what the value of α should be.
For readability, assume $x^0 = 0$ which implies $x^1 = \alpha A^T r^0$.
Then

$$\left\|x - x^1\right\|_2 = \left(x - \alpha A^T r^0\right)^T \left(x - \alpha A^T r^0\right) = x^T x - \alpha\left(r^0\right)^T Ax - \alpha x^T A^T r^0 + \alpha^2\left(A^T r^0\right)^T A^T r^0 \tag{5.29}$$

Using that $Ax = b$ produces:

$$\left\|x - x^1\right\|_2 = x^T x - \alpha\left(r^0\right)^T b - \alpha b^T r^0 + \alpha^2\left(A^T r^0\right)^T A^T r^0 = x^T x - 2\alpha\left(r^0\right)^T b + \alpha^2\left(A^T r^0\right)^T A^T r^0 \tag{5.30}$$

Setting the derivative with respect to α to zero produces

$$\alpha = \frac{\left(r^0\right)^T b}{\left(A^T r^0\right)^T A^T r^0} \tag{5.31}$$

One of the problems with the method of Gradient descent is that it will only converge to a global minimum if ∇F is Lipschitz continuous and F is a convex function.

5.6.3. CONJUGATE DIRECTIONS

One of the problems of the method of Gradient Descent is that it can occur that it often takes steps in the same direction as earlier steps. One way of avoiding this is to make sure that every new step is in a direction that is orthogonal to every previous step direction. Since you are no longer moving in the direction of the gradient this is no longer the method of gradient descent. However it does eventually lead to the conjugate gradient method.

Now one way to implement this orthogonal search direction idea is to use the condition that the new search direction must be orthogonal to the current error direction. Otherwise it could occur that the method would have to move more than once in a particular direction. This means $x^{k+1} = x^k + \alpha_k d_k$ where d_k is the k'th search direction. The orthogonality criterion gives

$$d_k^T e^{k+1} = 0$$
$$d_k^T (e_k + \alpha_k d_k)$$
$$\alpha_k = -\frac{d_k^T e_k}{d_k^T d_k} \tag{5.32}$$
$$\tag{5.33}$$

This is a useless expression as the error vector e_k is unknown. This can be solved by instead using the matrix A-norm for the orthogonality criterion, which can be shown that this inner product and corresponding norms are properly defined when the matrix A is symmetric and positive definite.
It is defined as:

$$\langle \mathbf{y}, \mathbf{y} \rangle_A = \mathbf{y}^T A \mathbf{y} \tag{5.34}$$
$$\|\mathbf{y}\|_A = \sqrt{\langle \mathbf{y}, \mathbf{y} \rangle_A} \tag{5.35}$$

Where $\mathbf{y}$ is some vector of dimension N.

Since the value of α in equation 5.32 is the quotient of two inner products we may use the A inner product instead to ensure A-orthogonality. This produces:

$$\alpha_k = -\frac{d_k^T A e_k}{d_k^T A d_k} = \frac{d_k^T r^k}{d_k^T A d_k} \tag{5.36}$$

Now an α has been found the method is not yet complete. It remains to construct N orthogonal search directions.
A classic method of orthogonalizing a set of vector with respect to an inner product is by the Gram-Schmidt orthogonalisation process.

5.6.4. COMBINING THE METHODS

If the set of vectors to be used as search directions is chosen to be the residuals we call the resulting method the conjugate gradient method.

It uses the search directions of the method of Gradient descent and then proceeds to make them more efficient by orthogonalizing them with respect to the A-norm.

One of the big advantages of the Conjugate gradient method is that since the search directions are orthogonal, and for every search direction the method reduces the component of the residual in that direction to zero, it follows that the method arrives at the solution after at most N iterations.
In practice, floating point errors cause the search directions not to be completely orthogonal and causes an error in the residual as well, which may slow down the method.

5.6.5. CONVERGENCE BEHAVIOUR OF CG

It can be proven [48] that the following relation holds:

$$\left\| x - x^k \right\|_A \leq 2 \left(\frac{\sqrt{C(A)} - 1}{\sqrt{C(A)} + 1} \right)^k \left\| x - x^0 \right\|_A \tag{5.37}$$

Where $C(A)$ is the condition number of the matrix A. The condition number associated with a linear equation $Ax = b$ is defined as the maximum ratio of the relative error in the solution x to the relative error in b. This can be proven to be equal to $\|A\| \left\| A^{-1} \right\|$.

If A is a normal matrix, e.g. A commutes with its conjugate transpose, the condition number equals the ratio of the largest eigenvalue of A and the smallest eigenvalue of A. This criterion is automatically satisfied when A is symmetric and positive definite.
This means that the conditioning of the matirx A is a very important factor in its suitability for the CG algorithm.
Preconditioning modifies the eigenvalues of A, so a good preconditioner will provide a large speedup for the CG method.

The above bound is a linear bound with a constant rate of convergence. When using the CG method in practice, one may observe superlinear convergence behaviour instead.

RITZ VALUES

According to [48] the Ritz values θ_i^k of the matrix A with respect to the Krylov subspace K_k are defined as the eigenvalues of the mapping

$$A_k = \pi_k A|_{K_k} \tag{5.38}$$

where π_k is the orthogonal projection upon K_k and k is the iteration number and i is the index of the value.

Correspondingly, the Ritz vectors y_i^k are the normalized eigenvectors of A_i corresponding to θ_i^k with the property that

$$A y_i^k - \theta_i^k y_i^k \perp K_i \tag{5.39}$$

The normalized residual matrix R_k is then defined as

$$R_k = \left[\frac{\mathbf{r}^i}{\|\mathbf{r}^1\|_2} \quad \dots \quad \frac{\mathbf{r}^{k-1}}{\|\mathbf{r}^{k-1}\|_2} \right] \tag{5.40}$$

The Ritz matrix is then defined as

$$T_k = R_k^T A R_k \tag{5.41}$$

The Ritz matrix can be seen as the projection of A onto the Krylov subspace $K^k\left(A; r^0\right)$

The Ritz values approximate the extreme eigenvalues of A, and correspondingly the Ritz vectors approximate the eigenvectors of A.
Now suppose a Ritz value θ_k and corresponding eigenvector are exactly an eigen value and vector $\mathbf{y}_k$ of A. Then, since we can write u as a linear combination of eigenvectors and its projection upon those vectors we obtain:

$$\mathbf{u} = \sum_{i=1}^{N} \left(\mathbf{u}^T \mathbf{y}_i\right) \mathbf{y}_j \tag{5.42}$$

Because $\mathbf{y}_j$ is contained in the Krylov subspace it must follow that $\mathbf{r}^k$ is perpendicular to it. Thus

$$\left(\mathbf{r}^k\right)^T \mathbf{y}_k = (\mathbf{u} - \mathbf{u}_k)^T A^T \mathbf{y}_k = (\mathbf{u} - \mathbf{u}_k)^T \theta_k \mathbf{y}_k = \theta_k \mathbf{e}_k^T \mathbf{y}_k = 0 \tag{5.43}$$

The conclusion is that the component of the error vector in the direction of the eigenvector $\mathbf{y}_k$ is zero.
Thus it follows that although the condition number of A remains unchanged, the convergence of the conjugate gradient method is now limited not by the condition number of A but by the effective condition number. Where the effective condition number is the condition number with θ_k excluded.

5.6.6. PARALLEL CONJUGATE GRADIENT

The conjugate gradient method consists of calculating inner products and matrix-vector multiplications, which are perfectly parallelizable. However as mentioned in chapter 4 subsection 4.3.3 these operations are memory-bound.

The process finding a preconditioner is well suited for parallel implementation and preconditioning the method will also provide speedup. The process of finding an effective preconditioner using a parallel method will be explored in section 5.8.

5.6.7. KRYLOV SUBSPACE METHODS FOR GENERAL MATRICES

Preconditioned Conjugate Gradient is one of the most efficient methods for solving linear problems where A is symmetric and positive-definite. However many problems will not fit this requirement. Thus various Krylov methods have been developed to accomodate for this.

Many of these methods involve expanding the matrix A in order to make it symmetric. One idea was to precondition the system with the matrix A^T to obtain

$$A^T A x = A^T b \tag{5.44}$$

However a problem here is that A^T is often a very poor preconditioner with respect to the condition number, significantly slowing down this method.

Another option is the so called Bi-CG method. The idea is instead of defining the residual vector of CG to be orthogonal to the Krylov subspace constructed it creates a second subspace to account for the non-symmetry of A.
Where the first subspace was $K^k\left(M^{-1}A, M^{-1}r^0\right)$ this second subspace is of the form $K^k\left(M^{-1}A^T, M^{-1}r^0\right)$ and thus is constructed out of the powers of the transpose of the problem matrix.

The Bi-CG method thus also introduces a second residual that relates to this second subspace, where it tries to make the second residuals A-orthogonal instead of standard orthogonal. It also uses the bi-Lanczos method to orthogonalize the residuals instead of the classical CG orthogonalization method.

One of the problems with Bi-CG is that it is numerically unstable and thus not robust. A modification was developed around 1992 [49] which was called Bi-CGSTAB where STAB stands for stabilized. The main idea is that in the Bi-CG method the residuals do not need to be explicit, and instead defines a modified residual that is multiplied with a polynomial:

$$\tilde{\mathbf{r}}_i = Q_i\left(A\right)\mathbf{r}_i = \left(I-\omega_1 A\right)\dots\left(I-\omega_i A\right)\left(A\right)\mathbf{r}_i \tag{5.45}$$

Which allows for smoother and more stable convergence.
Bi-CG uses short recurrences but is only semi-optimal. It is possible for the method to experience a near-breakdown which may still produce instabilities.

Another method for general matrices is the GMRES algorithm, which stands for Generalized Minimal RESidual. It uses Arnoldi's method for computing an orthonormal basis of the Krylov subspace $K^K\left(A; r^0\right)$ [50]. It is an optimal method in terms of convergence, but has as a drawback that k vectors need to be stored in memory for the k-th iteration.
A second drawback is that the cost of the Gram-Schmidt orthogonalization process, which is part of Arnoldi's method, scales quadratically in the number of iterations. One option to remedy this is to restart GMRES after a chosen number of iterations, but this destroys the superlinear convergence behavior and optimality property.

Thus when preconditioning GMRES often aggressive preconditioners are chosen that aim to greatly limit the number of GMRES iterations needed for convergence but are costly to compute.

5.7. MULTIGRID

Multigrid methods were developed in the late 20th century and have since then become quite popular due to their computational efficiency. Properly implemented Multigrid exhibits a convergence rate that is independent from the number of unknowns in the discretized system, and thus is called an optimal method [51].

The name Multigrid comes from the fact that the method solves an $Ax = b$ problem using multiple grids with different mesh sizes.

Usually the complexity of a problem is expressed in powers of N. For example, a full inversion of the matrix A using Gaussian elimination costs $O(N^3)$ floating point operations, and using an LU decomposition costs $\frac{2}{3}O(N^3)$ operations, where A is an $N \times N$ matrix.

Thus the time it takes to solve linear problems usually scales somewhere between quadratically and cubic in the number of unknowns. It follows that an easy way of reducing computation time is to sacrifice some accuracy by coarsening the grid. Halving the number of unknowns will reduce the problem matrix by a factor 4. However, this often leads to a loss of accuracy that is unacceptable and thus is not feasible.

The multigrid method instead uses this cheaper coarse solution to accelerate the process of finding the solution on the fine grid.
This is done by successively coarsening the grid and then using a basic iterative method to obtain a coarse residual. The next step is solving the problem once grid coarseness is so low that computational cost is negligible, and finally resharpening by interpolation and then correcting the solution with the calculated coarse solution and residuals.
An illustration of the method can be observed in figure 5.5.

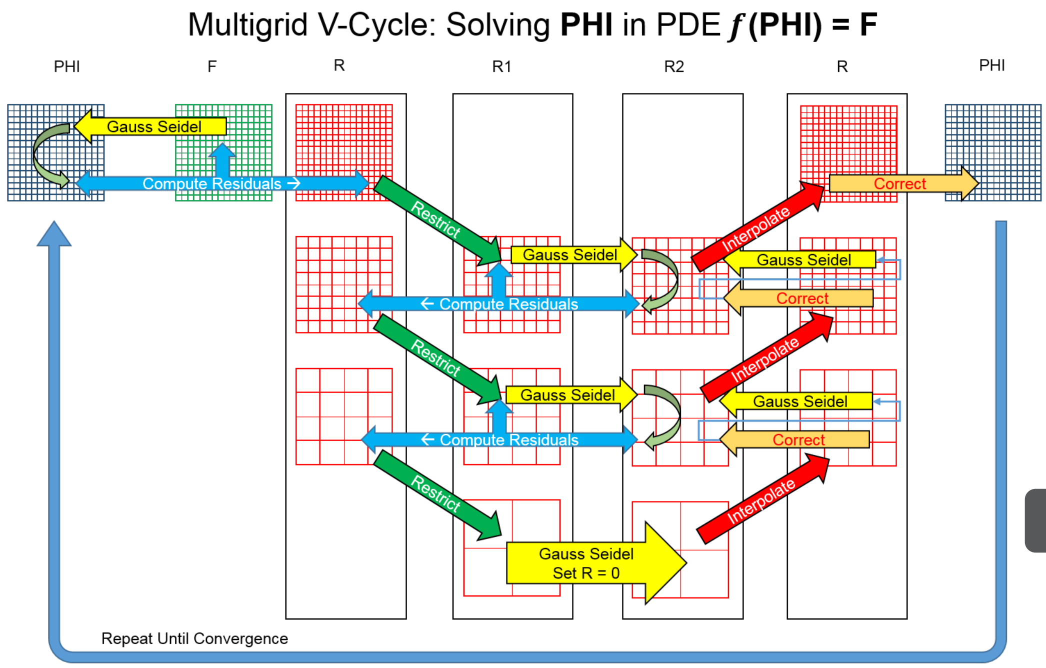


Figure 5.5: Illustration of an iteration of a V-cycle multigrid method
Source: [52]

5.7.1. ALGEBRAIC VS GEOMETRIC MULTIGRID

When discretizing a physical problem there is an obvious choice as how to construct the coarse matrix: it follows naturally from discretizing the problem with half the number of variables as in the original problem. However when all you have is a matrix A without any knowledge of the underlying problem constructing a coarser grid becomes nontrivial.

Choosing a coarsening method that is optimal for the multigrid method based on nothing but the problem matrix A is known as Algebraic multigrid. Because the problems considered in this thesis are all of the geometric type, it has been chosen to not go into the theory behind Algebraic multigrid any further.

5.7.2. ERROR FREQUENCY

One of the key elements in the efficiency of the multigrid method is the fact that it combines a coarse solution with a basic iterative method which complement each other well. Since the problem matrix A can be decomposed into a sum of weighted eigenvectors, this means that any function on the grid is some weighted sum of eigenmodes corresponding to these eigenvectors.
These eigenmodes will be either high frequency or low frequency, where a low frequency

mode corresponds to a small eigenvalue of A and a high frequency mode corresponds to a large eigenvalue of A.

This means that the error vector $x - x^k$ can also be decomposed into these eigenmodes. In other words: the error is a sum of low and high frequency errors.

Solving the system of equations on a coarsened grid will only correct the low frequency errors. This is because when a smooth function is discretized coarsely, the interpolation between nodes will approximate the true function well since it varies slowly. An illustration of this principle can be observed in figure 5.6.

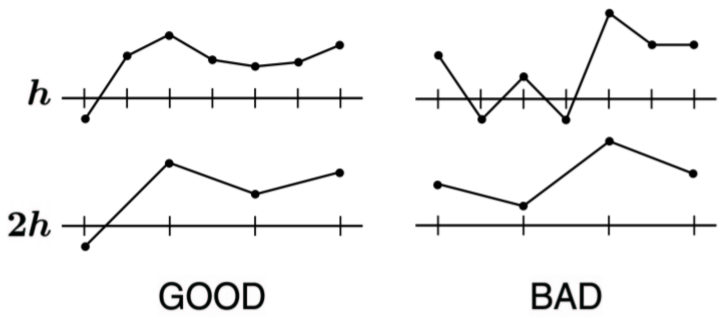


Figure 5.6: Illustration of coarsening error for a smooth function (left) and an oscillatory function (right) when moving from a fine grid with grid distance h to a coarse grid with grid distance $2h$.
Source: [51]

Fortunately, a basic iterative method is very effective at reducing high frequency errors but is slow to correct low frequency errors. This explains the slow convergence rate of unrelaxed Jacobi or Gauss-Seidel.

To illustrate this, if the Jacobi method $M = D$ is substituted into expression 5.15 the error propagation matrix consists of the rescaled strictly lower and upper triangular parts of the matrix A. This means that the error in a certain point i after multiplication with the error propagation matrix will be some linear combination of neighboring values depending on the structure of A.

It follows intuitively that after this repeated error averaging the high frequency errors will reduce quickly, while low frequency errors will be on average the same and thus will damp out slower.

5.7.3. CONVERGENCE BEHAVIOUR

As mentioned before, the multigrid method is an optimal method for problems that are suitable and thus converges in $O(1)$ iterations as it reduces the error by a fixed factor independent of the problem size N. This means that the total number of floating point operations necessary to solve a linear problem depend only on the cost of the iterations. A basic iterative method residual computation and the grid coarsening operations will be of order $O(N)$ when A is sparse enough, which would make the multigrid method an order N method [51].

5.8. PARALLEL PRECONDITIONERS

As mentioned before, many iterative solvers suffer from a lack of robustness. The convergence speed of the methods is highly dependent on the characteristics of the problem matrix which makes them ill suited for wide applications. Fortunately there exists a method called preconditioning that aims to compensate for this.

Preconditioning a linear problem $Ax = b$ means left or right multiplying both sides of the equation with a preconditioner M^{-1}, which produces

$$M^{-1}Ax = M^{-1}b \tag{5.46}$$

or

$$\begin{aligned} AM^{-1}u &= M^{-1}b \\ x &= M^{-1}u \end{aligned} \tag{5.47}$$

The solution of these new systems is exactly the same as the original problem, except the problem matrix is transformed. The characteristics of the preconditioner depend on the solution method that it tries to accelerate. For example the convergence rate of the Conjugate Gradient method described in section 5.6 is bounded by the ratio of the largest and smallest eigenvalues of A. Thus when preconditioning the system for acceleration of CG the aim of M^{-1} is to bring the ratio of the smallest and largest eigenvalues of the preconditioned system closer to 1.

When choosing a preconditioner it is important to keep in mind that its inverse must also be implicitly formulated. Two trivial choices of preconditioner would be the identity matrix I or the problem matrix inverse A^{-1}. In the first case the preconditioned system is exactly the same as the original problem and thus the preconditioning is useless.
In the second case inverting the preconditioner is exactly as hard as the original problem so nothing is gained either.

A good preconditioner will have the property that inverting it and multiplying the system with it saves more time for the iterative method than the cost of computing it.

Note that when preconditioning a Krylov subspace method there is the extra condition that the preconditioned system must remain symmetric and positive definite. In this case the simplest way to make sure of this is to require the preconditioner to be Cholesky decomposable: $M = PP^T$.

In which case the system becomes:

$$\begin{aligned} P^{-1}AP^{-T}u &= P^{-1}b \\ x &= P^{-T}u \end{aligned} \tag{5.48}$$

5.8.1. INCOMPLETE DECOMPOSITION

As mentioned in section 5.4, LU or Cholesky decomposition algorithms are rarely used due to poor scaling with problem size and fill in. The incomplete factorization preconditioner aims to remedy these problems. It involves the system matrix as $A = LU - R$, where L and U are lower- and upper triangular matrices, and R is some factorization residual due to the factorization being incomplete.

The idea is to only factorize the parts of A that do not produce any fill-in, and leave the rest as a residual for the actual solution method to solve. This is also called ILU(0) incomplete factorization.

The accuracy of the incomplete ILU(0) factorization may be insufficient to accelerate the actual solution method, as discarding fill in terms can prove to be quite significant. In this case, it can be chosen to allow some but not all fill in during the factorization.

5.8.2. BASIC ITERATIVE METHODS AS PRECONDITIONERS

Basic iterative methods themselves exhibit slow convergence behaviour as they are generally only effective at reducing high frequency errors. However they are well suited for use as a preconditioner. Because for a basic iterative method the matrix A is split into $A = M - N$, from the third step in equation 5.11 we have $x = M^{-1}(M - A)x + M^{-1}b$ Which we can rewrite to

$$\left(I - M^{-1}(M - A)\right)x = M^{-1}Ax = M^{-1}b \tag{5.49}$$

Thus it follows that the splitting introduced by a basic iterative method also automatically defines a preconditioned system associated with the splitting. Note that if the preconditioner is intended to be used with the preconditioned conjugate gradient method, M must be symmetric and positive-definite.

The simplest preconditioner is the Jacobi preconditioner, where $M^{-1} = D^{-1}$ the inverse of the diagonal of A. This preconditioner is effective at reducing the condition number of the preconditioned system. It is easy and cheap to calculate and it has the efficient property that the diagonal of the preconditioned matrix consists of only ones, which saves N multiplications per matrix vector product.

The Relaxed Gauss-Seidel preconditioner is similar with $M = D - \omega E$. An important thing to mention is that it is also possible to use the Symmetric Gauss-Seidel method, described in section 5.5.3 which is already of the form $M = PP^T$ and thus SPD.

5.8.3. Multigrid as a preconditioner

The multigrid algorithm is theoretically optimal for certain classes of problems when implemented properly. However its performance is highly problem specific and often requires fine-tuning of the smoothing to coarsening ratio. Using Multigrid as a preconditioner instead to obtain a rough solution which is then subsequently improved by a solver may yield convergence rates similar to a full multigrid method while being more robust [53] [54].

In [51] an explicit form is derived for the two-grid preconditioner which is called B_{TG}^{-1} to avoid confusion:

$$B_{TG}^{-1} = \left[M\left(M+M^T-A\right)^{-1}\right)M^T\right]^{-1} + \left(I-M^{-T}A\right)IA_c^{-1}I^T\left(I-AM^{-1}\right) \tag{5.50}$$

Where M is the chosen Basic Iterative Method splitting from $A = M - N$, A_c the once coarsened matrix A with dimensions $\frac{N}{2} \times \frac{N}{2}$ in the case of a 1 dimensional problem and I is the intergrid transfer operator with the property that $A_c = IAI^T$

The full Multigrid preconditioner can then be defined recursively as

$$B_k^{-1} = \left[M\left(M+M^T-A\right)^{-1}\right)M^T\right]^{-1} + \left(I-M^{-T}A\right)I_{k+1}^k B_{k+1}^{-1}\left(I_{k+1}^k\right)^T\left(I-AM^{-1}\right) \tag{5.51}$$

Where at the coarsest level $k = 1$ we take $B_l = A_l$, the matrix A coarsened l times.

According to [55] the multigrid method is very well suited as a preconditioner for the conjugate gradient method as it preserves the symmetry and positive definitiveness of the system. Using multigrid as a preconditioner for CG retains the $O(N)$ convergence rate but is more robust than pure multigrid, making it an attractive preconditioning choice.

5.8.4. Sparse Approximate Inverse preconditioners

The idea behind the sparse approximate inverse preconditioner is its suitability for parallel computing. It involves finding a sparse inverse of the problem matrix A, which is equivalent to finding

$$\min_{M \in P} \|I - MA\| \tag{5.52}$$

Where P is defined as the subset of $N \times N$ matrices that fit some chosen sparsity pattern.

It can be shown [56] that the Frobenius norm is optimal, which is defined as

$$\|A\|_F^2 = \sum_{i,j=1}^{N,N} a_{i,j}^2 = trace\left(A^T A\right) \tag{5.53}$$

It follows that

$$\min_{M \in P} \|I - MA\|_F = \sum_{j=1}^{N} \min_{\mathbf{m}_j \in P_j} \left\|e_j - A\mathbf{m}_j\right\|_2^2 \tag{5.54}$$

Where m_j is the j-th column of M and P_j is the sparsity pattern of column j

This means that finding a sparse approximate inverse equations to solving N least squares problems in parallel, which are cheap as long as P is sparse.

In the paper [57] it is stated that if A is a discrete approximation of a differential operator, it is very likely that the spectral radius of the matrix $G = I - A$ is less than one, and it then follows that

$$(I-G)^{-1} = I + G + G^2 + G^3 + \dots \tag{5.55}$$

It follows that the approximate inverse is a truncated form of this power series.

The inverse of a non-diagonal matrix is generally dense, and thus a sparse approximate inverse can never be a perfect matrix inverse. Despite this, a sparse approximate inverse is a good preconditioner if the significant elements of A^{-1} is well approximated by the chosen sparsity pattern P, but in practice it is hard to predict this which makes an optimal choice of P a matter of trial and error.

5.8.5. Polynomial preconditioners

The idea of a polynomial precondtioner is to accelerate a Krylov subspace method by first finding a polynomial that encloses the eigenvalue spectrum of A. The aim is to make sure the zeroes of the polynomial lie close to the matrix eigenvalues in the complex plane, so that the preconditioning will make these eigenvalues small. Once such a polynomial is found either the Chebychev or Richardson's iterative method [58] is used to construct the preconditioner. In practice this is done by starting with the standard Krylov subspace method until the Ritz values can be computed
Then the Ritz values from the first stage are used to compute parameters for the preconditioning polynomial, and finally solve the preconditioned system using the same Krylov subspace method as in the first stage.

[57] states a polynomial preconditioner is of the form

$$M_m^{-1} = \sum_{j=0}^{m} y_{j,m} G^j \tag{5.56}$$

It then follows that the approximate inverse is a preconditioner with all coefficients set to 1.

From the definition of the polynomial preconditioner it follows that

$$M_m^{-1} A = \sum_{j=0}^{m} y_{j,m} G^j A \qquad = \delta_{0,m} A + \delta_{1,m} A^2 + \dots + \delta_{m,m} A^{m+1} = p_m(A) A \tag{5.57}$$

Where $p_m(A)$ is some polynomial in A.

So the preconditioned matrix $M_m^{-1} A$ is some polynomial in A and its spectrum is given by $\lambda_i p_m(\lambda_i)$ where λ_i is an eigenvalue of A.

It is then possible to define the polynomial $q_{m+1}(y) = yp_m(y)$ and this polynomial vanishes at $y = 0$ and has the property that $q_{m+1}(y)$ for $\lambda_{min} \leq y \leq \lambda_{max}$ is a continuous approximation of the spectrum of $M_m^{-1}A$.

The objective then becomes to find

$$\min_{q \in Q_{m+1}} \frac{\max_{\lambda_{min} \leq y \leq \lambda_{max}} q(y)}{min_{\lambda_{min} \leq y \leq \lambda_{max}} q(y)} \tag{5.58}$$

Where Q_{m+1} is the set of polynomials of degree $m+1$ or less which are positive on the interval $[\lambda_{min}, \lambda_{max}]$ and vanish at zero.

It can be proven [57] that the Chebychev iterative method minimizes this function and also that least-squares Legendre polynomial weights are a viable alternative to Chebychev polynomial weights.

5.8.6. Block Jacobi preconditioners

The Block-Jacobi preconditioner can be interpreted as a domain decomposition preconditioner.
This idea arises from the fact that sparse matrices can often be written in block form. A block matrix is a matrix whose elements themselves are matrices.
In this instance domain decomposition is achieved by defining block matrix that splits the domain of computation into (almost) independent blocks. The advantage here lies in the fact that the block splitting can make it easy to identify matrix parts that are independent and thus can be solved in parallel.

For example a 5-point difference scheme for a two dimensional system will result in an $N \times N$ matrix A that has five diagonals, a tridiagonal inner part plus two outer diagonals. This same system matrix can be written in tridiagonal block matrix form:

$$\begin{bmatrix} D_1 & E_2 & & & \\ F_2 & D_2 & E_3 & & \\ & \ddots & \ddots & \ddots & \\ & & F_{m-1} & D_{m-1} & E_{m-1} \\ & & & F_m & D_m \end{bmatrix} \tag{5.59}$$

Where the D matrices are tridiagonal $k \times k$ matrices and E and F diagonal $k \times k$ matrices, and N is mk

Here, every diagonal block represents a part of the domain and the off-diagonal entries represent dependencies between domains.

The simplest domain decomposition preconditioner is the block-Jacobi preconditioner, which is defined as

$$M = \begin{bmatrix} D_1 & & \\ & \ddots & \\ & & D_m \end{bmatrix} \tag{5.60}$$

In this preconditioned matrix the off-diagonal terms are discarded which makes the individual D matrices independent and thus invertible in parallel.

One of the problems with block-Jacobi is that it exhibits poor scaling behaviour, the number of iterations increase with the number of subdomains [59]. One strategy to solve this is to use a certain overlap between subdomains to improve propagation of information. Care should be taken for the variables in the overlapping domains as they will receive corrections from both domains they belong to.

A possible solution to the poor scaling of the block-Jacobi preconditioner is to construct the block matrix using completely independent blocks, which are perfectly parallelizable, and subsequently performing a coarse grid correction to compensate for the error of disconnecting the subdomains.

5.8.7. MULTICOLORING PRECONDITIONERS

As explained in section 5.5.1 for the Gauss-Seidel algorithm a multicoloring turns the sequential Gauss-Seidel method into a highly parallel scheme. The Red-Black ordering plus incomplete LU factorization of the problem matrix is also a form of preconditioning.

A more advanced coloring scheme is for example the RRB-method, that can serve as a preconditioner for the CG method. A nice implementation was done by De Jong [60] [61], and a performance analysis of the RRB accelerated CG method for the shallow water equations is presented in chapter 10.

RRB stands for repeated red-black ordering. A normal Red-Black ordered matrix has two levels that are independent and thus can be LU-factorized in parallel. The repeated red black ordering algorithm extends this by defining multiple levels depending on domain size and the preconditioner can then be constructed level-wise in parallel. On these levels an incomplete factorization is then performed.
An illustration of RRB numbering on an 8×8 grid can be observed in figure 5.7

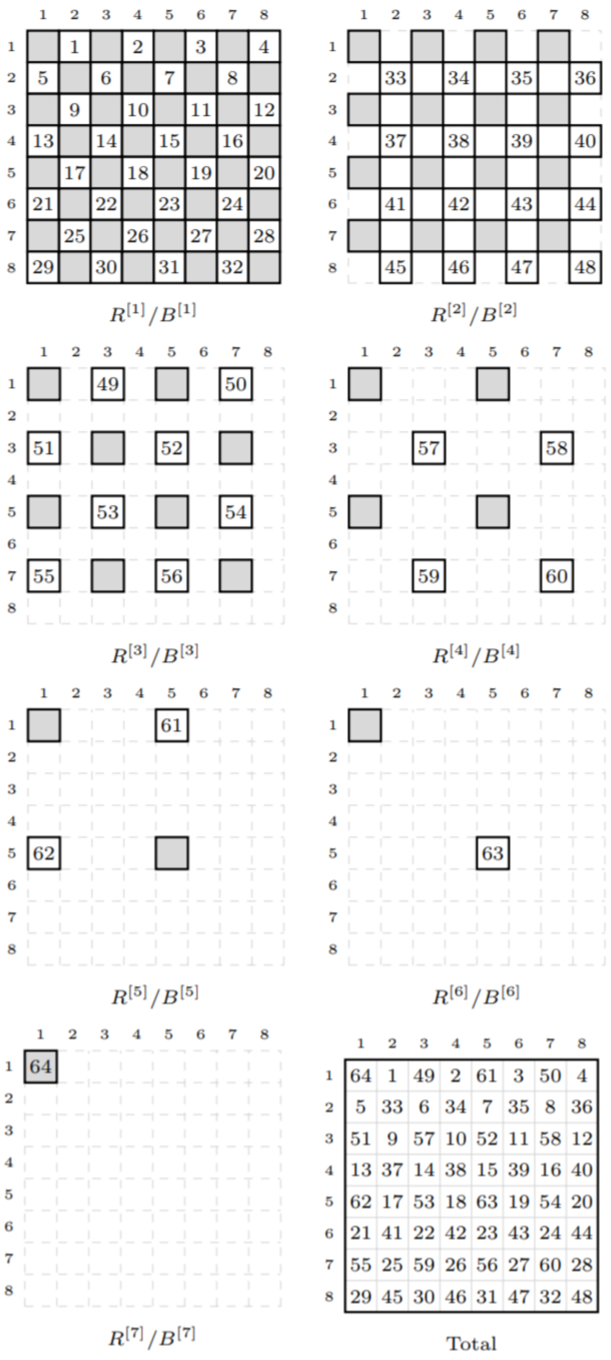


Figure 5.7: Illustration of the seven RRB numbering levels on an 8×8 grid.
Source: [60]

As can be observed, every successive RRB level becomes smaller, and thus the gain from adding this small extra level of parallelism also becomes smaller. At some point it might be beneficial to stop the RRB numbering at some chosen level, and proceed by computing a full Cholesky decomposition of the remaining nodes. The stopping level should be chosen such that using the Cholesky decomposition on the remainder is not much larger than incompletely factorizing it with the RRB method.

It should be mentioned that the level-wise LU decomposition algorithm described in section 5.4.1 is also a form of multicoloring where every parallel level corresponds to a color.

5.9. SOLVER SOFTWARE PACKAGES

Programming on a GPU is not as straightforward as classical CPU programming. As mentioned in chapter 4 it can be difficult to properly take advantage of the parallel architecture of a GPU without running into memory limitations. In many cases carefully optimizing a GPU method can yield as much of a performance improvement as the initial switch from CPU to GPU. Often a beginner trying to reinvent the wheel on a GPU will observe less than optimal performance.

Fortunately in order to make it easier for programmers with limited GPU programming experience to still be able to accelerate methods using a GPU many solver software packages exist that will provide acceptable results without the often frustrating optimization process.

5.9.1. PARALUTION

Paralution is a C++ library for sparse iterative methods that focues on multi-core CPU and GPU technology. It has a dual license model with either an open-source GPLv3 license and a commercial one. It has support for both OpenMP, OpenCL and CUDA, with plug-ins for FORTRAN, OpenFoam, MATLAB others [62]

One of the key selling points of the Paralution package is that it provides seamless portable integration which will run on the hardware it detects. Thus if a Paralution acceleration method intended for a GPU is written and then is used on a system without a GPU available the code will simply use the CPU instead.

The Paralution package contains the following solvers:

- Basic iterative methods: Jacobi, Gauss-Seidel and relaxed variants
- Chebyshev iteration
- Mixed precision defect correction
- Krylov subspace method: CG, CR, BiCGStab, Gmres
- Deflated preconditioned CG

- Both Geometric and Algebraic Multigrid
- Iterative eigenvalue solvers

Every solver method can also be used as a preconditioner, but additionally it supports a wide array of incomplete factorization preconditioners, approximate inverse preconditioners and colouring schemes. The preconditioner and solver design is such that you can mix and match preconditioners and methods as desired since everything is compatible and based on a single source code.

5.9.2. CUSOLVER & CUSPARSE

CuSOLVER and cuSPARSE are libraries developed and published by Nvidia and are included in the CUDA toolkit package in C and C++ language. CuSPARSE is a library that consists of mostly basic linear algebra subroutines that are optimized for sparse matrices, and is a useful library to consider when building a parallel program in CUDA. It is mentioned because it also contains a sparse triangular solver, tridiagonal solver and incomplete LU and Cholesky factorization preconditioners.

CuSOLVER is a high level package based on cuBLAS and cuSPARSE. The intent of cuSOLVER is to provide LAPACK-like features. This includes LU, QR and Cholesky decomposition. Unfortunately cuSOLVER is limited to direct solution methods [63].

5.9.3. AMGX

AmgX is an open source software package that offers solvers accessible through a simple C API that completely abstracts the GPU implementation. It contains a number of algebraic multigrid solvers, PCG, GMRES, BiCGStab and flexible variants. As preconditioners Block-Jacobi, Gauss-Seidel, incomplete LU, Polynomial and dense LU are available [64].

In addition it supports MPI and OpenMP which makes it very suitable for multi GPU systems, and as with Paralution the flexible implementation allows nested solvers, smoothers and preconditioners.

5.9.4. MAGMA

Another library that is freely available is MAGMA, which stands for Matrix Algebra on GPU and Multicore Architectures. It was developed by the team that also created LAPACK and it aims to provide a package that dynamically use the resources available in heterogeneous systems. It achieves this by using a hybridization methodology where algorithms are split into tasks of varying granularity and their execution is scheduled over the available components, where non-parallelizable tasks will often be assigned to the CPU while larger parallel tasks will be assigned to the GPU [65]

All MAGMA features are available for CUDA while most features are available in OpenCL or Intel Xeon Phi. The package includes

- LU, QR Cholesky factorization

- Krylov subspace methods: BiCG, BiCGStab, PCG, GMres
- Iterative refinement
- Transpose free quasi minimal residual algorithm
- ILU, IC, Jacobi ParILU, Block Jacobi and ISAI preconditioners

6

Implementation of the Stelling & Duinmeijer scheme

In the process of developing either an implicit or explicit model for the shallow water equations, the first step is to implement the Stelling & Duinmeijer scheme described in chapter 3. Implementing a method on a GPU is a highly complex matter, and thus it has been decided to first implement an explicit form of the scheme in MATLAB in order to familiarize with it, and subsequently implementing a CUDA C++-version.

The first test case is a square domain with an initial homogeneous water level of 1 meter and reflective boundary conditions. This system will be perturbed by a water droplet modeled by a Gaussian function.

6.1. Explicit implementation in MATLAB

As mentioned in chapter 3 an explicit scheme involves updating the solution vector by a matrix-vector product every time step. Storing this matrix however uses (costly) memory and it is often non-trivial to construct. It is also possible to implement an explicit method in a so called matrix-free form, which is more intuitive as it simply follows the discretised equations. A disadvantage of a matrix-free form is that it is harder to move from an explicit solution method to an implicit one, as constructing a matrix free implicit solution method is less intuitive.

As a first attempt at implementing the scheme explicitly the matrix-free approach has been chosen.

6.1.1. Staggered grid

Before the scheme can be implemented it is first necessary to define a grid with boundary and initial conditions. The Stelling & Duinmeijer scheme uses an Arakawa-C grid as described in chapter 3 and thus this will be the grid of choice. As before the water level H is defined on the centres of the grid squares while the velocities U and V are defined on the borders of the squares.

In order to construct the grid it is first important to consider the half-indices that occur due to the staggered nature of the grid, as matrix and vector indices in any programming language must be integers.

The solution is redefining the grid in the following way:

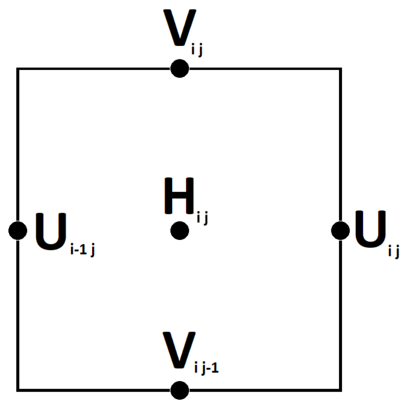


Figure 6.1: Shallow-water Arakawa C-grid with shifted velocity indices

This numbering amounts to all velocity-related indices being shifted by $-1/2$.

BOUNDARY CONDITIONS

This numbering becomes especially important when considering boundary conditions. If the grid consists of N squares, it is easy to deduce from figure 6.1 that the number of H variables is $N \times N$, the number of U variables is $(N+1) \times N$ and the number of V variables is $N \times (N+1)$.

In order to keep the three variable arrays the same size we introduce dummy rows and columns, which have been schematically represented in figure 6.2

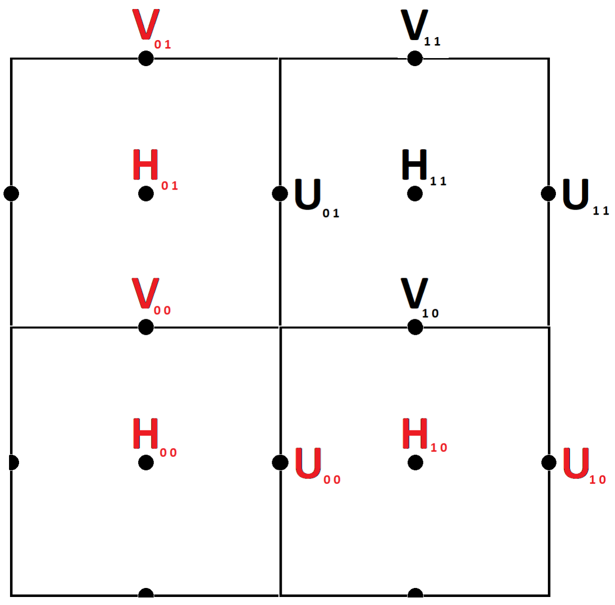


Figure 6.2: 1 × 1 Arakawa-C grid with dummy rows and columns introduced to allow U and V to be defined at the left and bottom boundaries. Note that the red coloured variables are the dummy variables which are not used in calculations.

6.1.2. STELLING & DUINMEIJER SCHEME

In order to aid the reader in understanding the implementation process, we first repeat the Shallow-Water equations and then the corresponding discretized Stelling & Duinmeijer scheme with $\theta = 0$, first described in chapter 3:

$$\begin{aligned}
\frac{\partial H}{\partial t} + \frac{\partial}{\partial x}(H\bar{u}_x) + \frac{\partial}{\partial y}\left(H\bar{u}_y\right) &= 0 \\
\frac{\partial \bar{u}_x}{\partial t} + \frac{\partial \bar{u}_x}{\partial x}\bar{u}_x + \frac{\partial \bar{u}_x}{\partial y}\bar{u}_y + g\frac{\partial \zeta}{\partial x} + \frac{g u_x ||\mathbf{u}||}{C^2 H^2} &= 0 \\
\frac{\partial \bar{u}_y}{\partial t} + \frac{\partial \bar{u}_y}{\partial x}\bar{u}_x + \frac{\partial \bar{u}_y}{\partial y}\bar{u}_y + g\frac{\partial \zeta}{\partial y} + \frac{g u_y ||\mathbf{u}||}{C^2 H^2} &= 0
\end{aligned} \tag{6.1}$$

$$\frac{h^{n+1}_{i,j}-h^n_{i,j}}{\Delta t}+\frac{h'^n_{i+1/2,j}u^n_{i+1/2,j}-h'^n_{i-1/2,j}u^n_{i-1/2,j}}{\Delta x}+\frac{h'^n_{i,j+1/2}v^n_{i,j+1/2}-h'^n_{i,j-1/2}v^n_{i,j-1/2}}{\Delta y}=0 \tag{6.2}$$

$$\begin{aligned}&\frac{u^{n+1}_{i+1/2,j}-u^n_{i+1/2,j}}{\Delta t}+u^n_{\rightarrow}\frac{u^n_{i+1/2,j}-u^n_{i-1/2,j}}{\Delta x}+v^n_{\uparrow}\frac{u^n_{i+1/2,j}-u^n_{i+1/2,j-1}}{\Delta y}\\&+g\frac{\zeta^n_{i+1,j}-\zeta^n_{i,j}}{\Delta x}+c_f\frac{u^{n+1}_{i+1/2,j}\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^n_{i+1/2,j}}=0\end{aligned} \tag{6.3}$$

$$\begin{aligned}&\frac{v^{n+1}_{i,j+1/2}-u^n_{i,j+1/2}}{\Delta t}+u^n_{\rightarrow}\frac{v^n_{i,j+1/2}-v^n_{i-1,j+1/2}}{\Delta x}+v^n_{\uparrow}\frac{v^n_{i,j+1/2}-v^n_{i,j-1/2}}{\Delta y}\\&+g\frac{\zeta^n_{i,j+1}-\zeta^n_{i,j}}{\Delta y}+c_f\frac{v^{n+1}_{i,j+1/2}\left|\left|\mathbf{u}^{=n}_{i,j+1/2}\right|\right|}{(h^{-y})^n_{i,j+1/2}}=0\end{aligned} \tag{6.4}$$

Where as before h' is the upwinded water level depending on flow direction, $\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|$ the magnitude of the averaged velocity vector at step n at $i+1/2, j$, and $u_{\rightarrow}$ and $v_{\uparrow}$ the velocity advection terms that can be chosen accordingly to the scheme's desired conservation properties, as explained in chapter 3.

The first problem here is that this scheme is not fully explicit. However, the implicit bottom friction terms can be easily calculated as they are completely independent and thus can be inverted at low cost:

$$\begin{aligned}\frac{u^{n+1}_{i+1/2,j}-u^n_{i,j}}{\Delta t}&=F\left(\mathbf{u}^n,h^n\right)-c_f\frac{u^{n+1}_{i+1/2,j}\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^n_{i+1/2,j}}\\u^{n+1}_{i+1/2,j}&=u^n_{i,j}+\Delta t\left(F\left(\mathbf{u}^n,h^n\right)-c_f\frac{u^{n+1}_{i+1/2,j}\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^n_{i+1/2,j}}\right)\\u^{n+1}_{i+1/2,j}\left(1+\Delta tc_f\frac{\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^n_{i+1/2,j}}\right)&=u^n_{i,j}+\Delta tF\left(\mathbf{u}^n,h^n\right)\\u^{n+1}_{i+1/2,j}&=\left(u^n_{i,j}+\Delta t\,F\left(\mathbf{u}^n,h^n\right)\right)/\left(1+\Delta tc_f\frac{\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^n_{i+1/2,j}}\right)\end{aligned} \tag{6.5}$$

Where $F\left(\mathbf{u}^n,H^n\right)$ represents the other explicit terms in equations 6.4.

6.1.3. EXPLICIT TERM FORMULATION

Now that the grid and relevant equations are properly defined it is also necessary to write all terms in equations 6.4 in terms of U, V or H, which are the water level and velocities in computer memory.
The first is

$$h'^{n}_{i+1/2,j} = Hx_{i,j} = \left(U_{i,j} > 0\right) H_{i,j} + \left(U_{i,j} < 0\right) H_{i+1,j} + (U_{i,j} == 0) \max\left(H(i,j), H_{i+1,j}\right) \tag{6.6}$$

With $h'^{n}_{i,j+1/2}$ is defined analogously with dynamic upwinding in the y-direction. Note that this definition can be vectorised by defining the comparative terms as boolean arrays. Also note that this term is undefined at the boundaries $i = n+1$ in the x-direction and $j = n+1$ in the y-direction. In these cases we take it equal to $H_{i,j}$.

The averaged water level in the x direction is defined as:

$$\left(h^{-x}\right)^{n}_{i+1/2,j} = \frac{H_{i,j} + H_{i+1,j}}{2} \tag{6.7}$$

Next is $\zeta^{n}_{i,j}$. Since $\zeta = h + d$ the sum of the bathymetry term and the water level, we simply set $d = 0$ for the initial implementation to obtain $\zeta^{n}_{i,j} = H_{i,j}$

For the friction term we define

$$\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right| = \sqrt{U^2_{i,j} + \left(\frac{V_{i,j} + V_{i,j-1} + V_{i+1,j} + V_{i+1,j-1}}{4}\right)^2} \tag{6.8}$$

With $\left|\left|\mathbf{u}^{=n}_{i,j+1/2}\right|\right|$ defined analogously

VELOCITY INTERPOLATION

The final term is $u^{n}_{\rightarrow}$. This term can be chosen to make the scheme either momentum conservative or energy-head conservative, the necessity of which depends on the velocity gradient.
The momentum conservative approach is quite complex while the energy-head conservative approach is simply the mean of the adjacent velocities, and thus has been chosen for a first implementation:

$$u^{n}_{\rightarrow} = \frac{U_{i,j} + U_{i-1,j}}{2} \tag{6.9}$$

With $v^{n}_{\uparrow}$ defined analogously.

For the momentum conservative formulation we have

$$u^{n}_{\rightarrow} = \frac{(q^{-x}_{u})_{i,j}}{(h^{-x})_{i+1/2,j}} \tag{6.10}$$

Where

$$(q^{-x}_{u})_{i,j} = \frac{Hx_{i,j}U_{i,j} + Hx_{i-1,j}U_{i-1,j}}{2} \tag{6.11}$$

VARIABLE FLOW DIRECTION

It is important to note that the set of equations 6.4 is only well defined for positive flow directions. The aim is to implement a scheme that is suitable for variable flow directions, and thus it needs to be modified. The correct approach then is to substitute each term that depends on flow direction by both its positive flow and negative flow counterpart, and multiply them with boolean values that represent whether the flow is actually positive or negative.

There exists another solution to the problem of variable flow directions in the velocity advection terms: central difference, which is symmetric:

$$\frac{\partial u^n_{i,j}}{\partial x} = \frac{U_{i+1,j} - U_{i-1,j}}{2\Delta x} \tag{6.12}$$

With the $y-$direction defined analogously.

In this case for the advection terms we take $u^n_{\rightarrow} = U_{i,j}$ and $v^n_{\uparrow} = \dfrac{V_{i,j} + V_{i,j-1} + V_{i+1,j} + V_{i+1,j-1}}{4}$ in the u equation of 6.4 and analogously in the v equation.

6.1.4. ADVECTIVE VELOCITY UPWINDING

As mentioned, the momentum advection terms in equations 6.4 are defined for positive flow directions only. In the case of the U term, or unidirectional advection, the expression for negative flow directions is straightforward. In the case of positive flow the expression is left looking, e.g. in the direction the information is coming from, so $\dfrac{\partial U_i}{\partial x} = \dfrac{U_i - U_{i-1}}{\Delta x}$.

For negative flow directions this simply changes to $\dfrac{\partial U_i}{\partial x} = \dfrac{U_{i+1} - U_i}{\Delta x}$.

The cross advection terms are a little more complicated, however. In this case the sign of the advective term decides the direction of the derivative, for positive directions the U derivative is taken in the direction of the V-flow, and then multiplied with the average of the adjacent V-velocities:

$$\frac{\Delta t}{\Delta y}\left(U_{i,j} - U_{j-1,i}\right)\frac{\left(V_{i,j-1} + V_{i+1,j-1}\right)}{2} \tag{6.13}$$

For negative V the U-derivative is upward looking and the upward V-terms are averaged:

$$\frac{\Delta t}{\Delta y}\left(U_{i,j+1} - U_{j,i}\right)\frac{\left(V_{i,j1} + V_{i+1,j}\right)}{2} \tag{6.14}$$

6.1.5. RESULTING EXPLICIT UPDATE SCHEME FOR POSITIVE FLOW DIRECTIONS

If these definitions are substituted into equation 6.4 the following explicit update expression is obtained for the energy head conservative scheme for positive flow directions:

$$
\begin{aligned}
U_{i,j} &= U_{i,j} - g\frac{\Delta t}{\Delta x}\left(H_{i+1,j} - H_{i,j}\right) \\
&- \frac{\Delta t}{\Delta x}\left(U_{i,j} - U_{i-1,j}\right)\frac{\left(U_{i,j} + U_{i-1,j}\right)}{2} \\
&- \frac{\Delta t}{\Delta y}\left(U_{i,j} - U_{j-1,i}\right)\frac{\left(V_{i,j-1} + V_{i+1,j-1}\right)}{2} \\
&/ \left(1 + \frac{2\Delta t c_f \sqrt{U_{i,j}^2 + \left(\frac{V_{i,j} + V_{i,j-1} + V_{i+1,j} + V_{i+1,j-1}}{4}\right)^2}}{H_{i+1,j} + H_{i,j}}\right)
\end{aligned} \tag{6.15}
$$

$$
\begin{aligned}
V_{i,j} &= V_{i,j} - g\frac{\Delta t}{\Delta y}\left(H_{i,j+1} - H_{i,j}\right) \\
&- \frac{\Delta t}{\Delta x}\left(V_{i,j} - V_{i-1,j}\right)\frac{\left(U_{i-1,j} + U_{i-1,j+1}\right)}{2} \\
&- \frac{\Delta t}{\Delta y}\left(V_{i,j} - V_{i,j-1}\right)\frac{\left(V_{i,j} + V_{i,j-1}\right)}{2} \\
&/ \left(1 + \frac{2\Delta t c_f \sqrt{+\left(\frac{U_{i,j} + U_{i,j+1} + U_{i-1,j} + U_{i-1,j+1}}{4}\right)^2 + V_{i,j}^2}}{H_{i,j} + H_{i,j+1}}\right)
\end{aligned} \tag{6.16}
$$

$$
\begin{aligned}
H_{i,j} &= H_{i,j} \\
&- \frac{\Delta t}{\Delta x}\left(Hx_{i,j}U_{i,j} - Hx_{i-1,j}U_{i-1,j}\right) \\
&- \frac{\Delta t}{\Delta y}\left(Hy_{i,j}V_{i,j} - Hy_{j-1,i}V_{j-1,i}\right)
\end{aligned} \tag{6.17}
$$

$$ \tag{6.18} $$

The complete scheme uses the sum of the negative and positive upwinding terms multiplied with boolean values that activate the corresponding term when the velocity in that grid node is positive or negative.

The scheme is integrated using the Sielecki method [66] [67], which is a form of the semi-implicit Euler method. First U_{t+1} is calculated using U_t,V_t, and H_t. Then V_{t+1} is calculated which uses U_{t+1}, and finally H_{t+1} which uses both U_{t+1} and V_{t+1}.

Matlab GPU implementation

Matlab is able to perform elementary linear algebra and arithmetic operations on an Nvidia GPU device. In order to instruct the Matlab compiler to perform calculations on a GPU, simply define the variables to be use in the calculations as a so called gpuArray.

Matlab takes care of the rest in the sense that the variables reside in device memory and are dynamically copied back and forth between device memory and host memory depending on where they are used.

The benchmark results of this implementation are presented in chapter 7.

6.2. GPU Cuda C++ implementation

The goal of the project is to build an acceptably optimized solution method in the Cuda C++ language, as Matlab is known to be non-optimal for GPU programming.

In order to translate the Matlab program to C++ a number of things should be taken into consideration, which are discussed in this section.

Grid operations

The first important thing about a Cuda program is that when doing operations on the grid this is not done through vectorisation or by using a for loop, but instead thread-wise where every thread handles a single grid point.
As described in chapter 4 a Cuda kernel will launch with a grid size and a block size, which can both have up to three spatial dimensions. Depending on the GPU an optimal block size must be chosen after which the grid dimensions will be set to the number of grid points in that dimension divided by the block size in that dimension.

Every grid point will have access to both its block index and thread index in both dimensions, and because of this the global array index can be reconstructed from these indices:

$$\begin{aligned}
&j = blockIdx.y * blockDim.y + threadIdx.y\\
&i = blockIdx.x * blockDim.x + threadIdx.x\\
&globalId = (j+1) * m + i + 1
\end{aligned} \tag{6.19}$$

Where n is the number of grid points in the x-direction.

As mentioned before, the grid has a dummy array at the boundary in order for the derivatives to be well defined at the nodes next to the boundary. Therefore only the internal nodes need to be calculated and not the boundary nodes, hence the offset on the global index. This also means that modifying and loading the boundary elements on the GPU needs to be done by threads on the computational domain as the boundary nodes do not get threads assigned to them.

Constant declaration

If certain variables are unchanging constants that are used during calculations on the GPU, such as the gravitational acceleration g, it is beneficial to store them in the GPU's constant memory. This is also discussed in chapter 4.

The reason for this is that constant memory is built in such a way that it is very efficient at broadcasting data across a warp in a block. Therefore if all threads in a warp read the same adress it is more efficient than global memory.

In order to have a constant available in GPU memory, first declare it as a constant using const float or const int and assign a value.
Then declare its GPU counterpart with __constant __float/int .
Finally copy the host constant to the GPU memory adress using the function "cudaMemcpyToSymbol".

INITIALIZING AND FILLING STORAGE ARRAYS

Much like the constants the arrays of H, U and V also need to be available on the GPU device. Before declaring the arrays, it is important to make sure that depending on the problem size they all fit on the GPU. The arrays that need to be stored are $H, U, V, U+$ and $V+$, where $U+$ and $V+$ are boolean arrays that indicate whether the velocity value on that grid block is positive or negative. The first three arrays are composed of single precision floating point numbers.
A single precision floating point number takes up 4 bytes of memory, and a boolean 1 byte.
This means that for an $n \times m$ grid we use $(12+2) \times n \times m$ bytes of device memory.

Every array must be initialized on device memory using "cudaMalloc". Next they need to be filled with the initial conditions. There are two methods to this: construct the arrays in host memory first and copy them to the GPU, or construct them directly on the GPU.
Since the transfer of data between host and device can take a long time as problem size increases, it has been chosen to fill the arrays directly on the device using a Cuda kernel.

In this kernel, every thread assigns the initial conditions to the variables at its respective gridpoints, while the boundaries are taken care of by their neighboring threads using an if- statement.

VECTOR UPDATE

The vector update procedure is the exact same as in its Matlab counterpart, except for a few details. As with the array fill function boundary elements must be loaded by their neighbors since they do not have their own dedicated thread.

Shared memory arrays are declared into which the global arrays are copied, since loading from and writing to shared memory is much faster.

Finally all operations that are not dependent on each other are calculated in parallel. For example the calculations of Hx can happen as soon as U finishes updating, and V

can start updating at the same time and does not depend on Hx which means these calculations can be performed in parallel. Also the newly calculated values of U can already be written to global memory while the calculations for V and H are still pending.

6.3. MULTITHREADED C++ IMPLEMENTATION

After initial benchmarking which is discussed in chapter 7, it was concluded that the Matlab implementation described in section 6.1 was not suitable for performance benchmarks and that it was necessary to write a multithreaded C++ program in order to compare GPU speedup more fairly.

When translating the CUDA program into regular C++, the first difficulty encountered was the fact that in the Cuda program every grid point is handled by a single thread. For large grid sizes this means a lot of threads which the GPU is optimized for, but unfortunately the CPU thread scheduler is not designed for such thread counts and this will incur a large performance penalty.

The solution was to divide the domain into a number of parts equal to the number of threads, which are launched using the "std:thread" built in C++ command.
Since every thread handles multiple grid points every grid operation was enclosed in a double for-loop iterating through the part of the domain assigned to the thread. Coalesced memory access was unsurprisingly important for CPU code as well, as changing the loop order after initially having the y-direction be the inner loop sped up the code by a factor 10. When dividing the domain into multiple threads this means that the borders of the subdomains need to be exchanged by the threads every timestep.

This was easy to implement however, since the the "Onlyborders" CUDA kernel does the exact same thing except the border is exchanged between blocks instead of between threads.

The CUDA kernels make heavy use of the "cudasyncthreads" command to ensure all threads have completed the previous vector update before reusing them. Unfortunately C++ has no built in thread synchronisation function. A thread synchronization function was developed using Mutex, a C++ object that can be modified by only one thread at a time. If a thread acquires the Mutex modification privileges, it checks whether all other threads have completed their work. If this is not the case it goes to sleep to wait for the last thread to wake it up.

6.4. IMPLICIT MATLAB IMPLEMENTATION

Implementing a complex scheme such as the shallow water equations implicitly may seem like a daunting task. Nonlinear equations can be solved explicitly but implicit solution methods are best suited for linear systems of equations and the Shallow-Water equations are highly nonlinear.

A simple solution to this problem is to follow the Stelling & Duinmeijer scheme with

θ set to 1, which is a linearised form:

$$\frac{h^{n+1}_{i,j}-h^{n}_{i,j}}{\Delta t}+\frac{h'^{n}_{i+1/2,j}u^{n+1}_{i+1/2,j}-h'^{n}_{i-1/2,j}u^{n+1}_{i-1/2,j}}{\Delta x}+\frac{h'^{n}_{i,j+1/2}v^{n+1}_{i,j+1/2}-h'^{n}_{i,j-1/2}v^{n+1}_{i,j-1/2}}{\Delta y}=0 \tag{6.20}$$

$$\begin{aligned}&\frac{u^{n+1}_{i+1/2,j}-u^{n}_{i+1/2,j}}{\Delta t}+u^{n}_{\rightarrow}\frac{u^{n}_{i+1/2,j}-u^{n}_{i-1/2,j}}{\Delta x}+v^{n}_{\uparrow}\frac{u^{n}_{i+1/2,j}-u^{n}_{i+1/2,j-1}}{\Delta y}\\&+g\frac{\zeta^{n+1}_{i+1,j}-\zeta^{n+1}_{i,j}}{\Delta x}+c_f\frac{u^{n+1}_{i+1/2,j}\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^{n}_{i+1/2,j}}=0\end{aligned} \tag{6.21}$$

$$\begin{aligned}&\frac{v^{n+1}_{i,j+1/2}-u^{n}_{i,j+1/2}}{\Delta t}+u^{n}_{\rightarrow}\frac{v^{n}_{i,j+1/2}-v^{n}_{i-1,j+1/2}}{\Delta x}+v^{n}_{\uparrow}\frac{v^{n}_{i,j+1/2}-v^{n}_{i,j-1/2}}{\Delta y}\\&+g\frac{\zeta^{n+1}_{i,j+1}-\zeta^{n+1}_{i,j}}{\Delta y}+c_f\frac{v^{n+1}_{i,j+1/2}\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-y})^{n}_{i,j+1/2}}=0\end{aligned} \tag{6.22}$$

In this case, only the ζ and bottom friction terms in the velocity equations and the velocity terms in the continuity equation are linearised and taken implicitly. The advective velocity terms remain explicit.

To obtain a matrix equation we substitute the u and v equations into the h equation after bringing the explicit terms to the right side similar to equation 6.5:

$$u^{n+1}_{i+1/2,j}=\left(-g\Delta t\frac{\zeta^{n+1}_{i+1,j}-\zeta^{n+1}_{i,j}}{\Delta x}+u^{n}_{i,j}+\Delta t\,F\left(\mathbf{u}^{n},h^{n}\right)\right)/\left(1+\Delta t c_f\frac{\left|\left|\mathbf{u}^{=n}_{i+1/2,j}\right|\right|}{(h^{-x})^{n}_{i+1/2,j}}\right) \tag{6.23}$$

Where again $F(\mathbf{u}^n,h^n)$ represents the velocity advection terms. The next step is substituting the velocity expressions into the continuity equation, so for readability we will call the bottom friction term $Bf_{i,j}$ and the explicit u terms $FU^{n}_{i,j}$ to obtain:

$$u^{n+1}_{i+1/2,j}=Bf_{i,j}\left[-g\Delta t\frac{\zeta^{n+1}_{i+1,j}-\zeta^{n+1}_{i,j}}{\Delta x}+u^{n}_{i,j}+FU^{n}_{i,j}\right] \tag{6.24}$$

Substituting this and its v counterpart into the h equation and using that $\zeta=h-d$ we obtain:

$$\begin{aligned}&h^{n+1}_{i,j}=h^{n}_{i,j}\\&+g\frac{\Delta t^2}{dx^2}\left[h'^{n}_{i+1/2,j}Bf_{i,j}\left(h^{n+1}_{i+1,j}-h^{n+1}_{i,j}+FU_{i,j}\right)-h'^{n}_{i-1/2,j}Bf_{i-1,j}\left(h^{n+1}_{i,j}-h^{n+1}_{i-1,j}+FU_{i-1,j}\right)\right]\\&+g\frac{\Delta t^2}{dy^2}\left[h'^{n}_{i,j+1/2}Bf_{i,j}\left(h^{n+1}_{i,j+1}-h^{n+1}_{i,j}+FV^{n}_{i,j}\right)-h'^{n}_{i,j-1/2}Bf_{i,j-1}\left(h^{n+1}_{i,j-1}-h^{n+1}_{i,j-1}+FV^{n}_{i,j-1}\right)\right]\end{aligned} \tag{6.25}$$

The bottom terms resulting from the substitution $\zeta = h - d$ are also absorbed into $FU_{i,j}$ resulting in the expression:

$$FU_{i,j} = u^n_{i+1/2,j} - g\frac{\Delta t}{\Delta x}\left(d_{i+1,j} - d_{i,j}\right) - u^n_{\rightarrow}\frac{\Delta t}{\Delta x}\left(u^n_{i+1/2,j} - u^n_{i-1/2,j}\right) - v^n_{\uparrow}\frac{\Delta t}{\Delta y}\left(u^n_{i+1/2,j} - u^n_{i+1/2,j-1}\right) \tag{6.26}$$

Where $u^n_{\rightarrow}$ and $v^n_{\uparrow}$ can be various expressions depending on which qualities are desired in the scheme, discussed in the previous sections.

Now that the substitution is complete it can be rearranged into $Ax = b$ form. When observing equation 6.25 the stencil representing A can be constructed quite easily. As before we rename $Bf_{i,j}h'^n_{i+1/2,j} = Hx_{i,j}$ and $Bf_{i,j}h'^n_{i,j+1/2} = Hy_{i,j}$

$$A = g\frac{\Delta t^2}{\Delta x^2}\begin{bmatrix} & Hy_{i,j-1} & \\ Hx_{i-1,j} & H_{sum} & Hx_{i,j} \\ & Hy_{i,j} & \end{bmatrix} \tag{6.27}$$

With $H_{sum} = Hy_{i,j-1} + Hx_{i-1,j} + Hx_{i,j} + Hy_{i,j}$

This is a stencil very similar to the 2d-Poisson equation which produces a pentadiagonal system, except in this case the terms are weighted with H_x and H_y.

The linear system that now needs to be solved at every timestep becomes

$$(I - A)\,h^{n+1} = b \tag{6.28}$$

Where

$$b = h^n - \frac{\Delta t}{\Delta x}\left[Hx_{i,j}FU_{i,j} - Hx_{i-1,j}FU_{i-1,j}\right] - \frac{\Delta t}{\Delta y}\left[Hy_{i,j}FV_{i,j} - Hy_{i,j-1}FV_{i,j-1}\right] \tag{6.29}$$

In this case Hx, Hy, FU and FV are all vectors, however indices are still necessary because the expression contains a discretized differential equation and thus the vectors need to be left-shifted by one index in their respective directions.

After h^{n+1} is calculated u^{n+1} and v^{n+1} can be obtained by adding the missing implicit term to FU and FV

$$u^{n+1} = FU - g\frac{\Delta t}{\Delta x}Bf_{i,j}\left(h^{n+1}_{j,i+1} - h^{n+1}_{j,i}\right) \tag{6.30}$$

6.4.1. Crank-Nicolson method

As described in chapter 3, the implicit system formulation in this section amounts to an implementation of the Backwards Euler method, which is known to approximate the solution with an error of order $O(\Delta t)$.

In practice this implementation was seen to greatly dampen the total amount of energy in the system for timesteps larger than prescribed by the CFL condition. This means that while a larger timestep could be taken, doing so is a tradeoff between numerical accuracy and speed.

An easy improvement here is to modify the system to a Crank-Nicolson scheme, which is also suggested by Stelling & Duinmeijer. As described in chapter 3 equation 3.2 a discretized differential equation is of the form

$$\frac{\phi_{n+1}-\phi_n}{\Delta t} = aF(\phi_{n+1}) + (1-a)F(\phi_n) \tag{6.31}$$

The Euler Backwards method uses $a = 1$ while Euler forward uses $a = 0$. Crank-Nicolson uses $a = 1/2$ which produces second order accuracy in time with an error of order $O(\Delta t^2)$.

This means the extra explicit terms must be added to 6.26:

$$FU_{CN} = FU - g\frac{\Delta t}{2\Delta x} Bf_{i,j}\left(h^n_{j,i+1} - h^n_{j,i}\right) \tag{6.32}$$

This results in $A_{CN} = \frac{A}{2}$ and that

$$u^{n+1}_{CN} = FU_{CN} - g\frac{\Delta t}{2\Delta x} Bf_{i,j}\left(h^{n+1}_{j,i+1} - h^{n+1}_{j,i}\right) \tag{6.33}$$

It should be noted that using this Crank-Nicolson implementation to solve the Shallow-Water equations will not produce results second-order accurate in time, as only the parts relating to the continuity equation are solved with this scheme. The nonlinear advective velocity terms are still solved explicitly and thus have an $O(\Delta t)$ error, and the bottom friction is taken fully implicitly also with an $O(\Delta t)$ error.

7 BENCHMARK RESULTS OF THE PRELIMINARY EXPLICIT IMPLEMENTATION

As mentioned in chapter 6, the explicit Stelling & Duinmeijer scheme has been implemented in Matlab on a square grid with reflective boundary conditions and a Gaussian perturbation simulating a water drop as initial condition.

After this was done and it was confirmed that the scheme was stable and performed adequately a Matlab GPU implementation was built as well as a Cuda C++ implementation, as described in chapter 6.

It is however important to note that this implementation is not perfect. At the boundaries the differential equations will be unbalanced due to zero terms from the ghost points, and there is a not-insignificant degree of numerical dissipation. The bottom profile is not implemented yet and the model is unable to deal with drying and wetting. Currently the model calculates with only a fixed time step of .01 seconds instead of a CFL condition limited timestep.

Implementing these features was the next research step and is the reason the method discussed in this section is called a preliminary implementation.

Since the scheme itself is already numerically dissipative, it has been chosen to set the bottom friction coefficient to 0, as the natural dissipation has an effect similar to a friction term

It should be noted that the explicit CUDA C++ implementation benchmarked in this section was later found to have some serious design flaws, which will be discussed in chapter 8.

In figures 7.1 and 7.2 both the computation time and the speedup of 100 timesteps are presented for the Matlab CPU implementation benchmarked on an Intel i7 8086k, Matlab GPU implementation and three different versions of the Cuda C++ implementation, benchmarked on an Nvidia RTX 2080 Ti and an Nvidia Quadro K1000M.

The Intel i7 8086k is operating at 5000 Mhz.
The RTX 2080 Ti has 68 streaming multiprocessors with a total of 4235 Cuda cores operating at 2160 Mhz. It can handle up to 1024 threads per block, 1024 threads per SM and 64 registers per thread. It has 11 GB of DDR6 Vram with a 352 bit memory bus with a maximum bandwidth of 616 GB/s. Furthermore it has a maximum of 64 Kbyte of shared memory per SM.

The Quadro K1000M has 1 streaming multiprocessor with a total of 192 Cuda cores operating at 850 Mhz. It can handle up to 1024 threads per block, 2048 threads per SM and 32 registers per thread. It has 2 GB of DDR3 Vram with a 128 bit memory bus with a maximum bandwidth of 28.8 GB/s. Furthermore it has a maximum of 48 KByte of shared memory per SM.

The three Cuda kernels differ in the following way:

UPDATE
This kernel is a direct translation of the Matlab implementation described in chapter 6 into Cuda C++. Every iteration the variables $H, U, V, Upos$ and $Vpos$ are loaded into global memory and every time a variable is finished updating it is written back to global memory while the rest of the thread executes.

UPDATENOBOOL
Initial benchmarking showed high memory utilization compared to computational utilization of the GPU device, implying a possible memory bottleneck. Since the booleans $Upos$ and $Vpos$ are comparatively cheap to calculate, in this kernel they are now only calculated within a thread.
If a neighboring value is needed, the neighboring velocity U or V is loaded from shared memory to calculate the boolean value. This local calculation does result in a small increase in the necessary number of registers per thread.

UPDATEBORDERS
The writing of updated variables to Global memory is necessary when the data needs to be in global memory after every iteration, for example to update boundary conditions or to visualize it in a surface plot. However when multiple timesteps are performed without this requirement, it makes no sense to perform costly global memory reads and writes.
Global memory writes cannot be completely eliminated however, since shared memory is only accessible to threads within the same block. Every iteration threads on the edge of the block need data from their neighbors, which belong to a different block.
Therefore in this kernel version only the block borders are written and read from global memory every iteration and only after all iterations are done is the entire grid written to global memory.

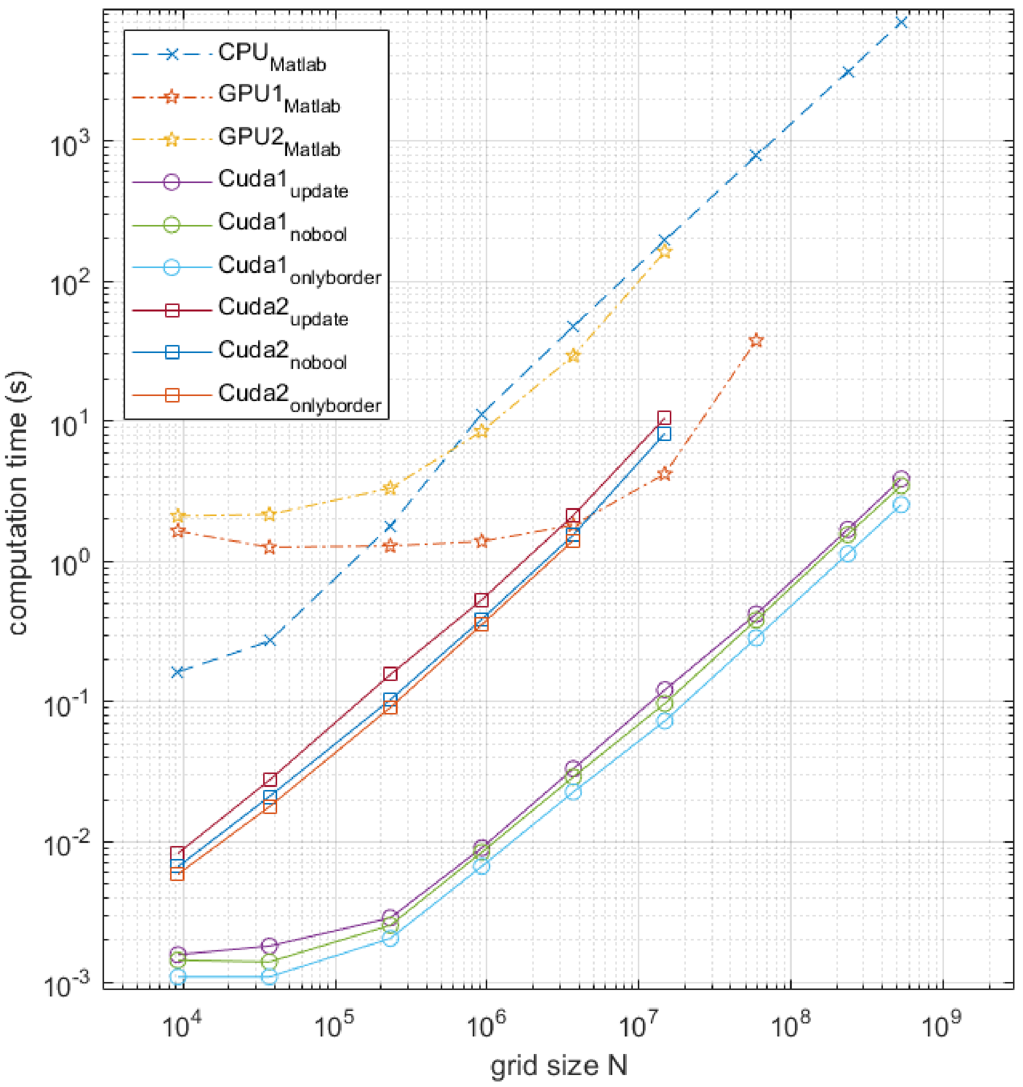


Figure 7.1: Computation time of 100 iterations vs grid size for Matlab CPU implementation, Matlab GPU implementation and 3 different Cuda kernels running on an Nvidia RTX 2080 Ti (GPU1/Cuda1) and an Nvidia Quadro K1000M (GPU2/Cuda2)

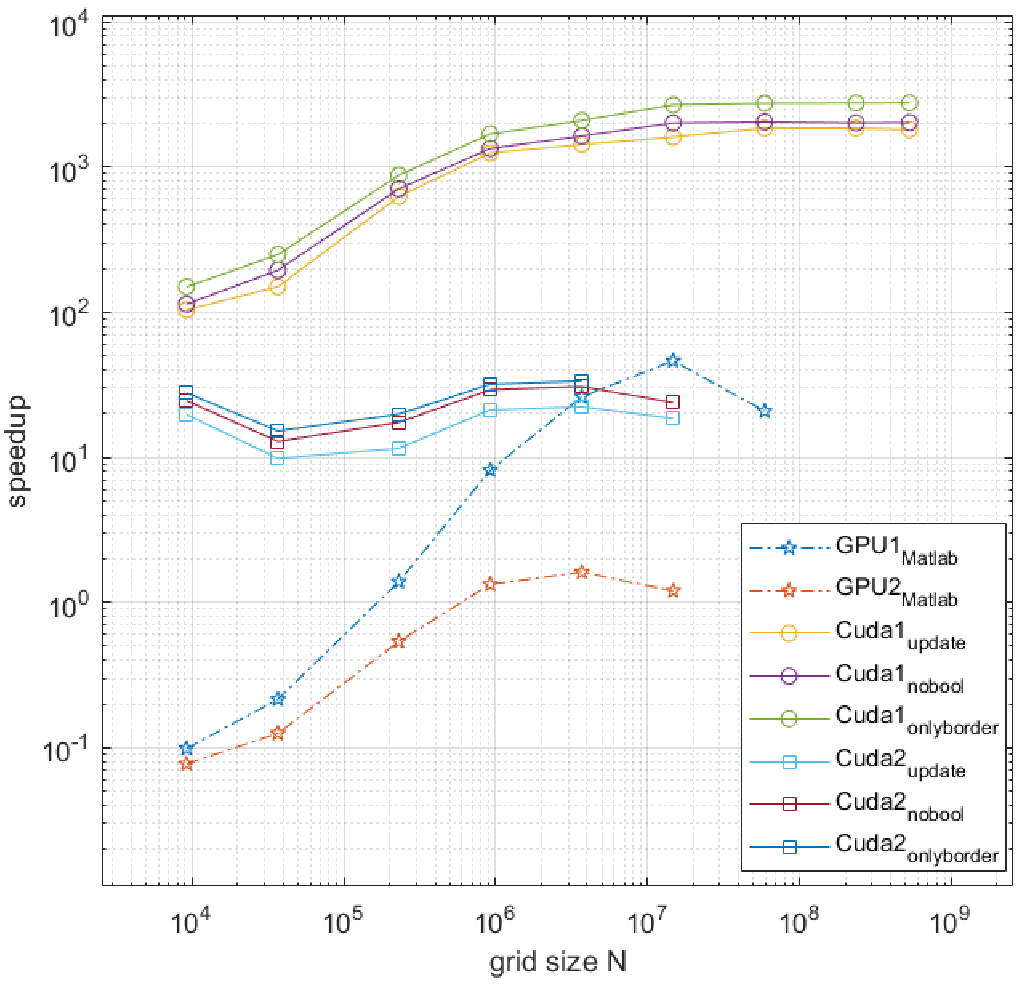


Figure 7.2: Speedup of 100 iterations vs grid size for the Matlab GPU implementation and 3 different Cuda kernels compared to the Matlab CPU implementation running on an Nvidia RTX 2080 Ti (GPU1/Cuda1) and an Nvidia Quadro K1000M (GPU2/Cuda2)

DISCUSSION

As can be observed in figure 7.1, the Matlab CPU implementation is the slowest for grid sizes above 1 million. The Matlab GPU implementation exhibits high startup times in the order of a few seconds, but provides a moderate speedup for larger grid sizes even for the Quadro K1000M device which is relatively old and slow. It is important to note that for smaller grid sizes the CPU implementation will be faster as the GPU device overhead will outweigh the added GPU parallelism and computation power.

Another notable observation is that the Matlab GPU implementation exhibits poor scaling, with computation times increasing superlinearly on a log-log scale, while it would be expected to scale linearly with problem size once the full device is being utilized, as can be observed for both the CPU implementation and the Cuda C++ implementations. The reason for this probably has to do with the way Matlab internally translates CPU to GPU code.

Another thing to note is that the matlab GPU implementation consumed significantly more memory than the Cuda C++ implementation, leading to a maximum problem size of $(40 \times 96)^2$ on the Quadro and $(80 \times 96)^2$, while the RTX 2080 Ti. The Cuda C++ implementation could handle up to $(240 \times 96)^2$ grid points before running out of memory.

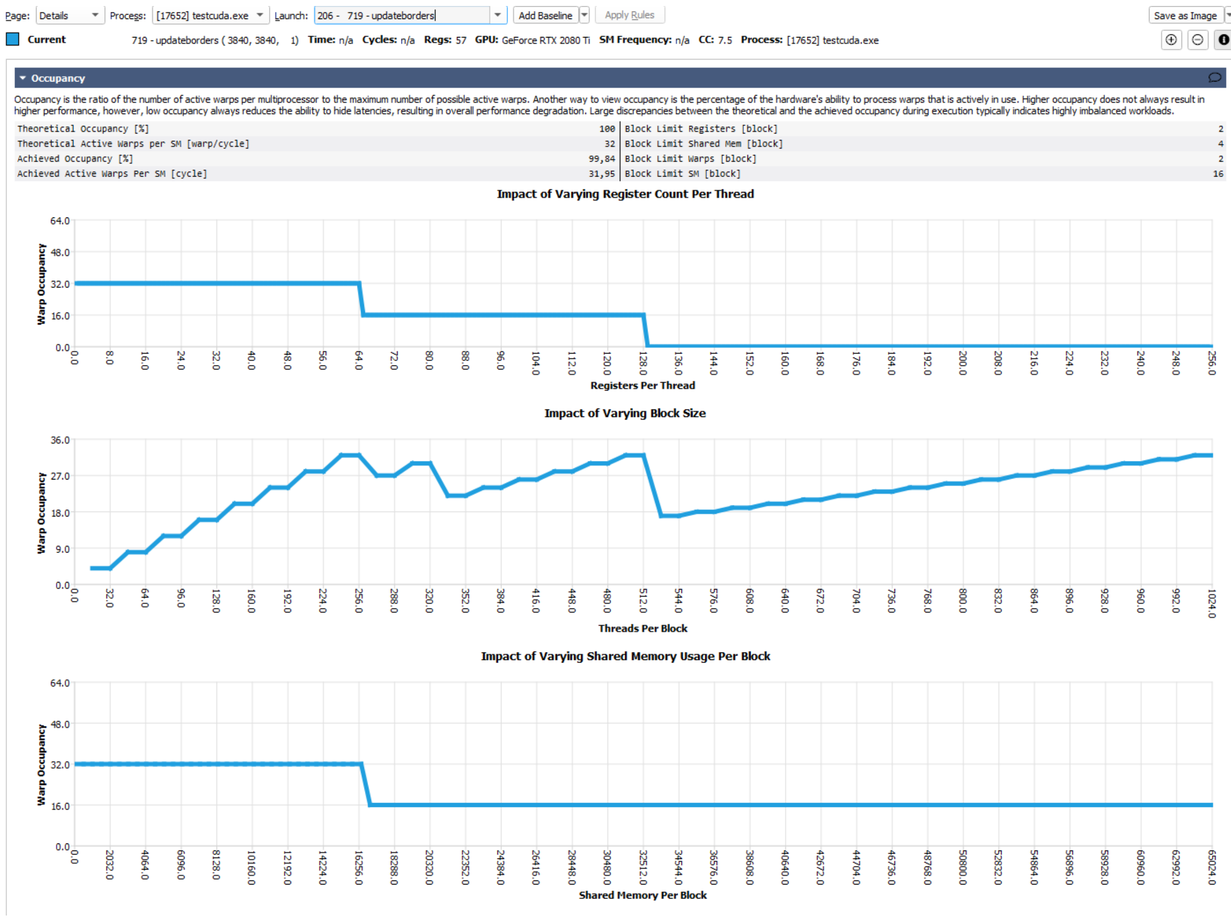
The Cuda C++ implementation benchmarked RTX 2080 Ti provides an extraordinary speedup of up to 3000 times, which is so much that it had to be double checked whether this speedup even lies within the theoretical capabilities of the device.
For verification the values of H have been compared after 100 iterations for a grid of 32×32 for all Cuda C++ kernels on both GPU devices and were found to be equal to the Matlab CPU result up to a satisfactory precision.
For larger grid sizes it has only been checked whether the kernels had water heights that lay between 0 and 2 unequal to the initial water level and no 'NaN' values.

However it is important to note that it is unfair to compare a Matlab implementation to a Cuda C++ program, as the Matlab language is inherently slower than C++. Future benchmarks will be performed with a C++ CPU implementation for fair comparison.

When comparing the three different Cuda kernels, it is clear that the "onlyborder" kernel is the fastest, followed by the "nobool" kernel and finally the naive "update" kernel. This is to be expected as the "onlyborder" kernel has a significant reduction in the amount of global memory traffic, while the "nobool" kernel has a small reduction in global memory traffic. This also means that global memory traffic is a factor significantly limiting the kernel execution speed.

7.0.1. Block size

Using the Nsight compute visual profiler it is possible to profile the warp occupancy as a function of the number of threads per block. This has been done for the only-border-exchanging update kernel on the RTX 2080 Ti GPU. The results can be observed in figure 7.3.


Page: Details
Process: [17652] testcuda.exe
Launch: 206 - 719 - updateborders
Add Baseline
Apply Rules
Save as Image
Occupancy
Impact of Varying Register Count Per Thread
Warp Occupancy
Registers Per Thread
Impact of Varying Block Size
Threads Per Block
Impact of Varying Shared Memory Usage Per Block
Shared Memory Per Block

🟦 Current 719 - updateborders (3840, 3840, 1) Time: n/a Cycles: n/a Regs: 57 GPU: GeForce RTX 2080 Ti SM Frequency: n/a CC: 7.5 Process: [17652] testcuda.exe

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	2
Theoretical Active Warps per SM [warp/cycle]	32	Block Limit Shared Mem [block]	4
Achieved Occupancy [%]	99,84	Block Limit Warps [block]	2
Achieved Active Warps Per SM [cycle]	31,95	Block Limit SM [block]	16

Figure 7.3: Nsight profiler results for the border exchanging update kernel

As can be observed, maximum occupancy is achieved for kernels of size 256, 512 and 1024, which result in block sizes of 16 × 16, 32 × 16 and 32 × 32 respectively.

The kernel requires 16 Kilobyte of shared memory per block and a maximum of 64 registers per thread to achieve a warp occupancy of 32. We see that the kernel uses 57 registers with an achieved occupancy of 99.84% and 31.95 out of 32 achieved active warps per SM.

We see that both the amount of available registers and the maximum warps per block are limiting the maximum of parallel blocks active per SM to 2. Since a high occupancy is achieved and there are multiple factors limiting the active blocks it can be concluded that 2 active blocks per SM is optimal for this Kernel.

In figure 7.4 the computation time result for all three kernels for varying blocksizes are presented:

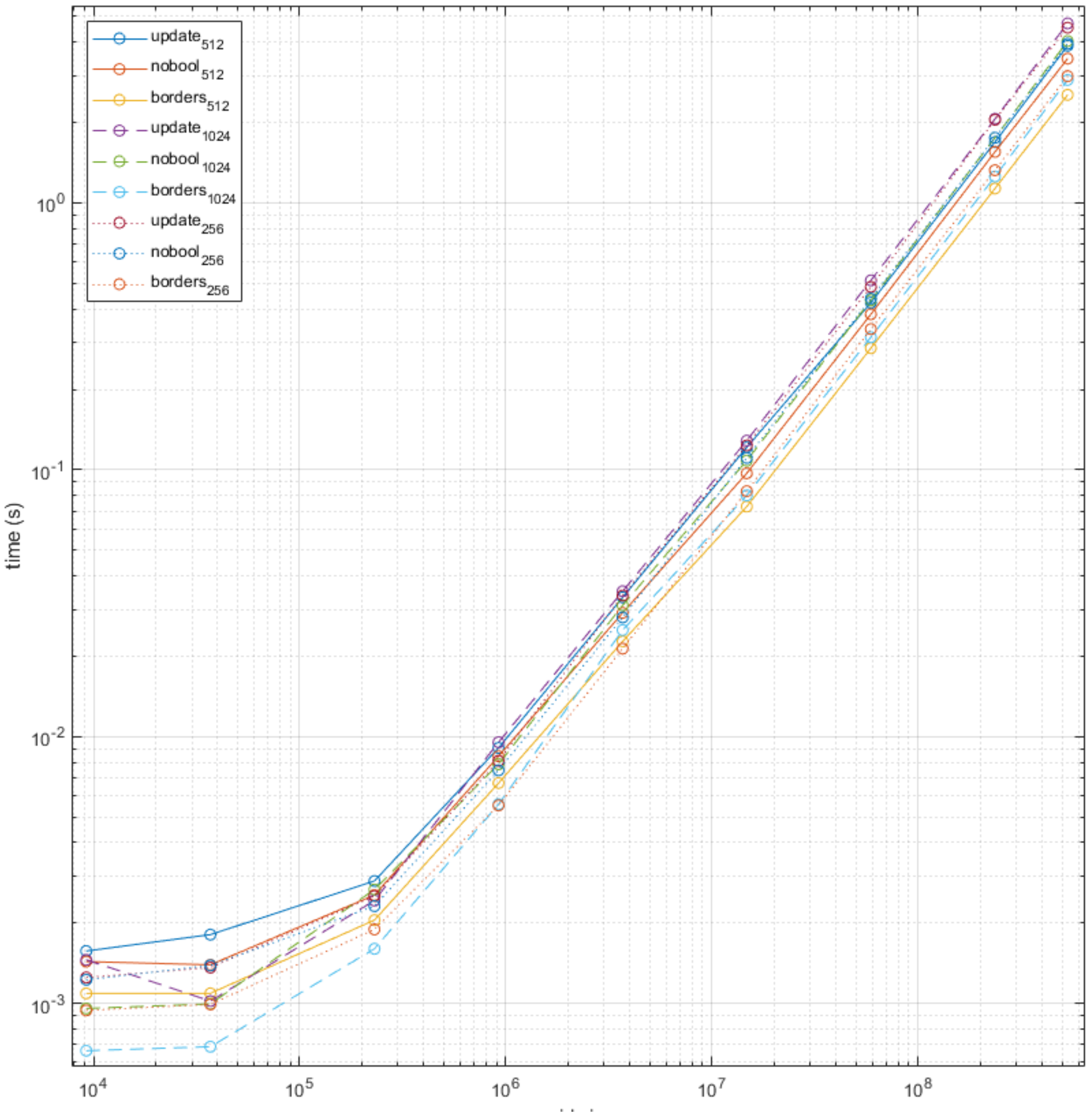


Figure 7.4: Computation time of 100 iterations vs grid size for 3 different Cuda kernels running on an Nvidia RTX 2080 Ti with a block size of 256, 512 and 1024 threads.

The data is very clustered making it hard to read the figure. This is to be expected as the occupancy analysis predicted a warp size of 32 threads for all three configurations. It is however interesting to look at the outliers.

For smaller grids the naive update kernel with a block size of 512 performs the worst, while the "borders" kernel with a block size of 1024 is the fastest by a significant margin.

For larger grid sizes the borders1024 kernel is first overtaken by borders256 and finally by borders512 for the fastest spot. Also the update1024 kernel performs the worst for in this scenario followed closely by update256 making update512 the fastest of the update kernels.

The conclusion here is that for small problem sizes for all three kernels a block size of 1024 threads is optimal, while for large problem sizes a block size of 512 threads is optimal. The reason for this is currently unknown. It is important to note that these optimal block sizes are generally GPU architecture specific.

7.0.2. Initialization time

When looking at the speedup of a GPU versus a CPU program it is also important to investigate startup costs in addition to raw computation time, as a large startup cost can easily outweigh the benefits of a smaller computation time. In figure 7.5 the startup times for various grid sizes of the Matlab CPU and GPU implementations as well as the Cuda C++ implementation for both GPU devices is presented.

For smaller grid sizes the CPU is faster but for larger grids it is the slowest, presumably since the initialization phase can be executed in parallel too. For grid sizes above 1 million points we observe a linear relationship on a log-log scale between startup time and grid size. The CPU and Cuda C++ implementations have the same gradient, while the Matlab GPU implementation scales worse.

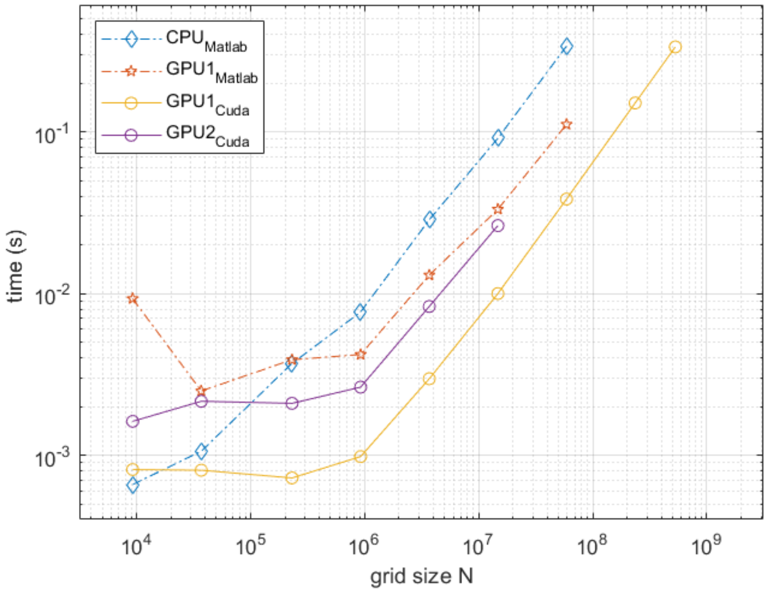


Figure 7.5: Startup time vs grid size for the Matlab CPU, Matlab GPU and Cuda C++ implementations. GPU1 is the RTX 2080 Ti and GPU2 is the Quadro K1000M.

8
Implementation of the modified Stelling & Duinmeijer scheme

As mentioned in chapter 7, the initial implementation had a number of room for improvement, such as a lack of a drying and wetting mechanic for grid points, a lack of bathymetry and relatively high numerical dissipation.

A second version has been implemented aiming to improve in these areas and ultimately mimic Deltares' Delft3D-Flow as closely as possible.

It was also found that the three CUDA kernels described in chapter 7 had various issues with thread synchronization which were previously overlooked. These flaws and their solutions are discussed at the end of this chapter.

8.1. Improvements of the scheme

8.1.1. Velocity advection

Evaluating the behaviour of the Stelling & Duinmeijer scheme and its observed numerical dissipation it was suggested that this dissipation could be reduced by modifying the velocity advection terms described in subsection 6.1.4. The velocity advection terms use first order upwinding which is great for stability but also introduces numerical dissipation.

It was chosen to calculate the length advection term with central difference and the cross advection term by second order upwind, also called the Reduced Phase Error Method by G.S. Stelling [68]. This leads to:

$$U\frac{\partial U}{\partial x} = U_{i,j}\frac{U_{i+1,j} - U_{i-1,j}}{2\Delta x} \tag{8.1}$$

and

$$V\frac{\partial U}{\partial y} = \frac{3U_{i,j} - 4U_{i-1,j} + U_{i-2,j}}{4\Delta y}\left(V_{i,j-1} + V_{i+1,j-1}\right), V_{j,i} > 0 \tag{8.2}$$

$$V\frac{\partial U}{\partial y} = \frac{-3U_{i,j} + 4U_{i+1,j} - U_{i+2,j}}{4\Delta y}\left(V_{i,j} + V_{i+1,j}\right), V_{j,i} < 0 \tag{8.3}$$

8.1.2. Control booleans

One of the problems with the initial implementation is that the derivatives are not well defined at the boundaries of the domain. Ghost points have been implemented in order to make sure the difference equations all use readable memory adresses, but without setting the correct values on the ghost points the difference equations will be incorrect.

There are a number of possible solutions to this. The first is to simply use different equations at the boundary. For example a point on the left boundary of the domain could use right-looking forward differencing instead of central.

A problem with the aforementioned method is that the boundary points need to be treated separately from the internal points. Delft3D-Flow is written for execution on vector computers which means it is beneficial for every grid point to evaluate the exact same expression. Coincidentally the same holds for GPU since they calculate 32 grid points at the same time and a single divergent thread will slow execution speed by 50%.

Thread expression homogeneity is achieved by introducing so called control booleans, kenmerkarrays in Dutch, which are multiplied with parts of the difference equation to dynamically modify it depending on whether the neighboring velocity points have defined values. The control boolean for the x-velocity is called kfu. For example, the central difference equation for the velocity advection then becomes:

$$U\frac{\partial U}{\partial x} = \frac{kfu_{i+1,j}U_{i+1,j} - kfu_{i-1,j}U_{i-1,j} + \left(kfu_{i-1,j} - kfu_{i+1,j}\right)U_{i,j}}{\left(1 + kfu_{i-1,j}kfu_{i+1,j}\right)\Delta x} kfu_{i,j}U_{i,j} \tag{8.4}$$

This may seem like a complicated expression at first, but if either $kfu_{i+1,j} = 0$ or $kfu_{i-1,j} = 0$ is substituted it becomes either a forward or backward difference equation.

8.1.3. Wetting and drying

The first implementation of the Stelling & Duinmeijer scheme had no way of dealing with very low water levels, as they could easily become negative due to numerical inaccuracies which is both non-physical and highly unstable.

Delft3D-FLOW solves this using the control booleans from the previous subsection. If the water level interpolated at a velocity point falls below a certain threshold, the corresponding kfu or kfv is set to zero. If it rises above another threshold it is set back to one.

Additionally, the expressions for calculating U and V at the next timestep are multiplied with kfu and kfv respectively, which means that if the interpolated water level falls below the threshold the velocity becomes zero in that point.

This also means that if the interpolated water levels on all four velocity points surrounding a water level point fall below the threshold, all four velocities also become zero and thus the water level will remain constant.

8.1.4. BATHYMETRY

Usually the problem of wetting and drying only arises when some form of bathymetry is introduced. For example when simulating a tide on a beach there is an area where the water level approaches zero and the beach begins.

The introduction of a bathymetry into the scheme follows from the expressions

$$H = \zeta + D \tag{8.5}$$

Where H is the water surface level from reference point, ζ the total water height and D the bathymetry from reference point. The bathymetry term D can have two possible orientations, where a positive term can either mean the bathymetry level lies above or below reference point. Since H is defined in the positive direction in seemed most intuitive to also define D in the positive direction.

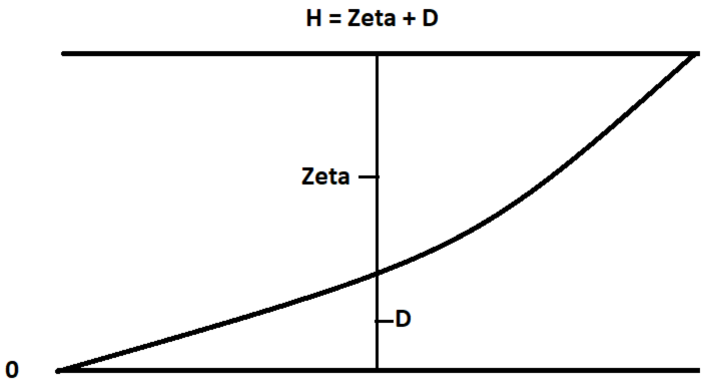


Figure 8.1: Illustration of the relationship between H, ζ and D.

Additionally we have the relation

$$\frac{\partial H}{\partial t} = \frac{\partial \zeta}{\partial t} + \frac{\partial D}{\partial t} = \frac{\partial \zeta}{\partial t} \tag{8.6}$$

and

$$\frac{\partial H}{\partial x} = \frac{\partial \zeta}{\partial x} + \frac{\partial D}{\partial x} \tag{8.7}$$

Which means that when switching from H to ζ in the scheme it is only necessary to add the spatial D derivatives in the calculation of the velocity terms as the bathymetry is time invariant.

8.2. TEST CASES

CASE 1

As mentioned in chapter 7, the initial test consisted of a square grid with reflective boundary conditions and a Gaussian perturbation of the water level to simulate a water drop.

For the updated scheme it was necessary to also test the wetting and drying and thus a Gaussian bathymetry was added that rises 3 cm above the water level in order to simulate a small island.

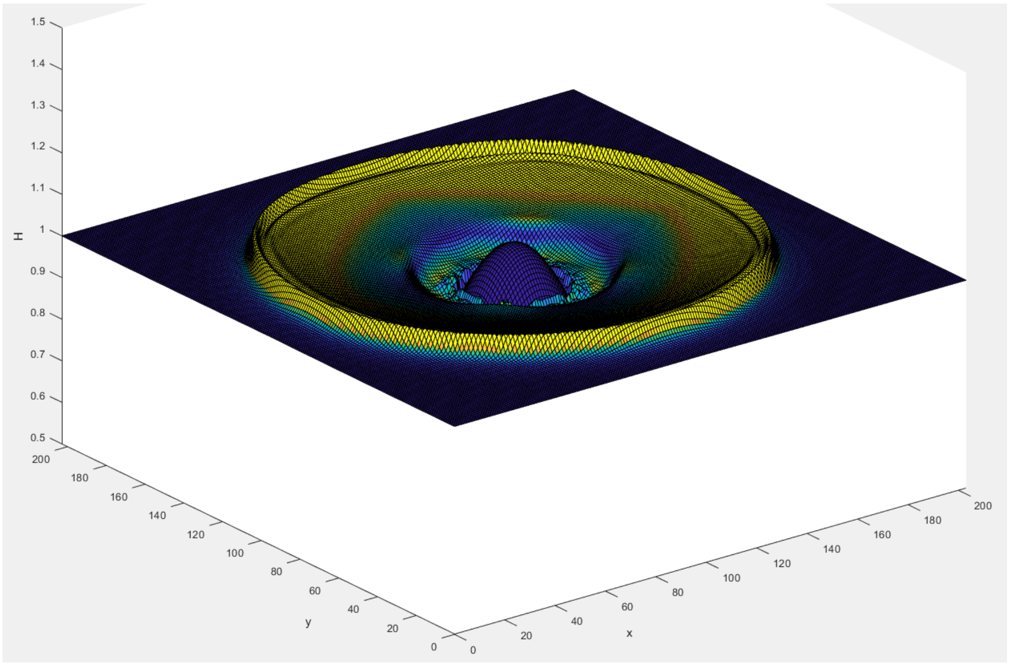


Figure 8.2: Surface plot of test case 1 after 330 iterations of .1 second on a 200m by 200m grid and an initial H of 1m.

CASE 2

The second test case is a reproduction of the case tested by L. Peeters in her thesis "Salt marsh modelling implemented on a GPU" which was also conducted in collaboration with Deltares [69].

Unfortunately in her case the maximum stable timestep was significantly lower than the timestep imposed by the CFL condition. The purpose of reusing this test case is to see if the scheme developed in this project does not suffer from this limitation and to compare it with the earlier Delft3D-Flow result that were also used by Peeters for comparison.

This test case consists of a square domain with a width and length of 600 meters. The bathymetry imposed starts at 0 on the leftmost boundary and increases linearly to 4.5 meters for 250 meters in the x-direction and is uniform in the y-direction.

A tidal water level is imposed on the leftmost boundary, modeled as a sinusoidal function that varies between 1 and 5 meters with a period of 12 hours, and with an initial water level of 3 meters.

Since all variables are initialized uniformly in the y-direction the y-velocity will remain 0 for the entire period of the test case which means it is a pseudo-1D case.

An illustration of test case 2 can be observed in figure 8.3.

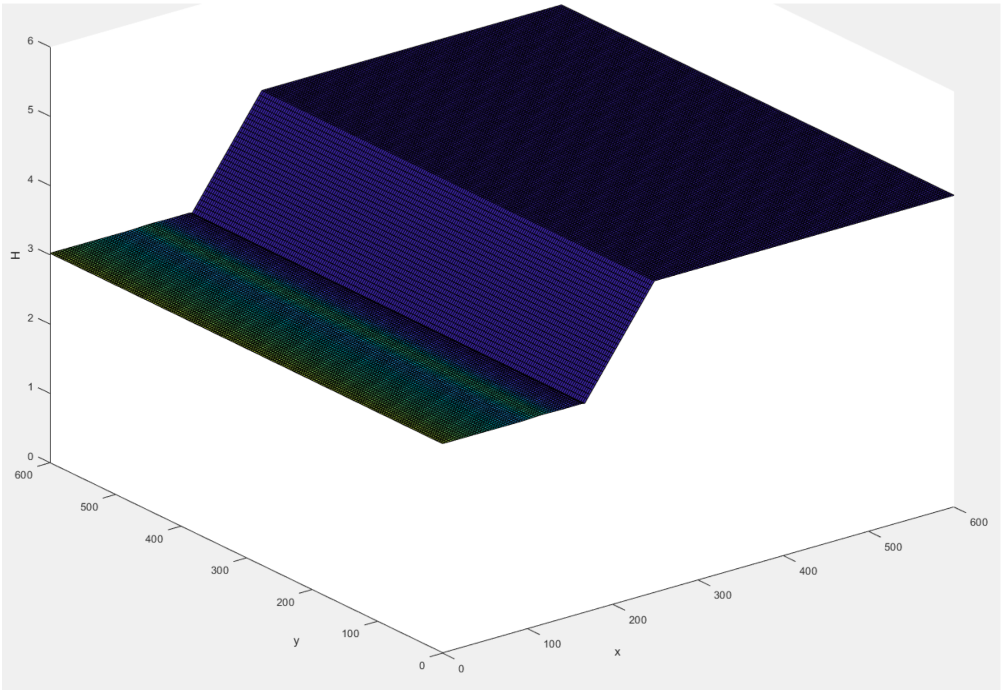


Figure 8.3: Surface plot of test case 2 on a 200 × 200 grid after 100 seconds.

In figure 8.4 a 2D flow velocity vector plot is shown of an adjusted test case 2, adding two islands to make the case two dimensional. Observe the flow velocity curves around the two islands placed in the domain with the largest velocities located in the channel between the islands as expected and velocities tapering off when they hit the dry bed on the right side of the domain.

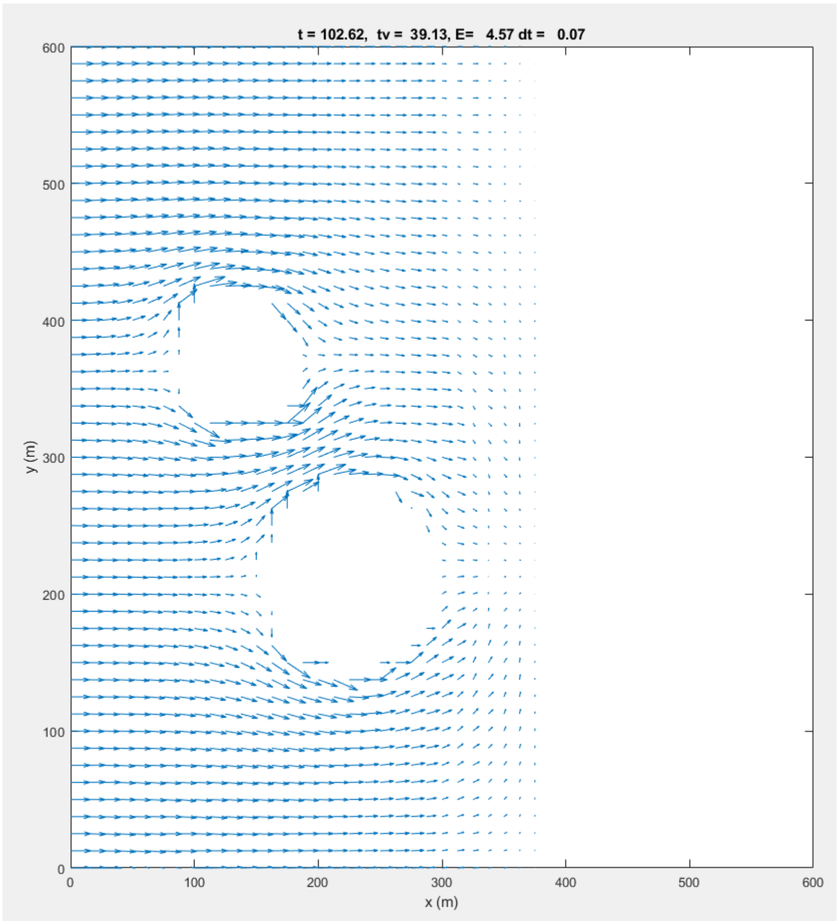


Figure 8.4: 2D flow velocity vector plot of test case 2 after 102.6 seconds on a 50 × 50 grid. Arrow size represents relative velocity magnitude.

8.2.1. Timestep size

Using the CFL condition it is possible to calculate the maximum stable timestep for the scheme at the beginning of each iteration, which was done in the MATLAB implementation of the scheme. However the maximum stable timestep depends on the maximum of the water level ζ across the entire grid, and finding the maximum value of ζ on the GPU array requires writing a separate maximum function kernel which is not trivial. After some testing it was found that for test case 1 the maximum timestep under the CFL condition remains fairly constant, and thus it has been chosen to not use a dynamically calculated time step for test case 1.

For test case 2, it is expected that the maximum timestep is very closely related to the tidal boundary condition. Thus the timestep for case 2 is calculated dynamically by assuming the maximum value of ζ does not exceed the tidal level by more than 10% and substituting this maximum into the CFL condition.

8.3. CUDA C++ implementation

8.3.1. Branch avoidance

In the CUDA implementation the computational grid is partitioned into blocks 32×32 threads as this is the maximum allowable block size on most GPU devices. It has been chosen to have the maximum allowable size as boundary values need to be communicated between the block, and the ratio of internal points to boundary points is better when the block size is large.

Every block thus has a 32×32 block of computational points and another 4 rows and 4 columns of boundary points that overlap with neighboring blocks, as the second order upwind scheme uses information up to two grid points away.

Since only the internal points have threads assigned to them, this means that the loading of the boundary points from memory must be handled by internal points as well. It would be intuitive to let the boundary points be handled by their closest neighbor, i.e the leftmost computational column with threadId.x = 0. This however clashes with the way a GPU operates. In chapter 4 it was stated that the GPU executes a thread block 1 warp consisting of 32 threads at a time. The important thing here is that the warps are taken row-wise, so in a 32×32 block the first warp will execute the first row of 32 threads.

Now if during execution of a warp threads within a warp diverge, the entire warp is executed multiple times, once for every instance of thread divergence. If the left and rightmost boundary are handled by the first and last column of computational points, this means that every warp on the grid will have to execute twice. Once for the boundary points and once for the internal points, which is very inefficient.

On the top and bottom boundary the system is better, only the top and bottom warps execute twice as a single warp handles the entire boundary. Thus to improve the efficiency of the boundary handling the second thread row was assigned to both the left and right boundary through a coordinate mapping. This results in the left and right boundary columns being initialized by a single warp of threads each without any performance loss.

To reduce complexity this restricts the Cuda kernel to having symmetric thread block sizes, but since testing in chapter 7 concluded that the performance difference between a 32×32 block size and a 32×16 block size was negligible this was deemed an acceptable compromise.

8.3.2. Grid corners

During the initial implementation it was concluded that since the difference equations only use information from neighboring points in the x-direction or the y-direction but never diagonally it was not necessary to load the 16 corner ghost-point values from global memory into shared memory as the values would never be used.
However after implementing the second order velocity upwinding it was quickly found that the second order upwind term does require these corner values, and therefore it is necessary to load them from global into shared memory.

Unfortunately these 16 corner points are not loaded by a single warp without branching, making it an inefficient operation.

8.3.3. Block-level synchronization

A serious problem with the initial implementation described in chapter 6 is the lack of block-level synchronization. As mentioned in chapter 7 the validation of the results consisted of inspecting array values from the command line and checking for anomalies. This proved to be insufficient as further testing by surface plotting the data revealed anomalous behaviour.

The solution was found to be non-continuous along thread block boundaries. After investigation it was concluded that this is the result of a lack of block-level synchronization in the kernel. To avoid excessive loading from global memory the only border communicating kernel was developed as described in chapter 6. The problem with this approach is that due to a lack of a block-level synchronization there is nothing that ensures that different blocks are calculating the same time step. When the computational grid becomes large not all threads will reside on the device at the same time, which leads to blocks becoming desynchronized.

CUDA Cooperative Groups introduced in CUDA 9 would allow for in-kernel synchronisation, but restricts CUDA grid size with respect to device capabilities leading to higher code complexity. Cooperative groups are well suited for problems with high initialization costs such as this one.

Cooperative groups restrict the CUDA grid size to the maximum number of active threads of the device. The reason for this once a block is assigned to an SM, it will keep it occupied until all block threads finish execution as there is unfortunately no way for an active block to become inactive again.
This means that a block-level synchronization where the number of blocks exceeds the

number of multiprocessors will result in a deadlock: The active blocks are waiting for the inactive blocks to reach the synchronization point and keeping the SMs occupied, and the inactive blocks are waiting for available SMs to start execution on.

There is however one other way to ensure all blocks execute in sync. Host launched Cuda kernels execute sequentially on the device unless asynchronicity is specified. This means that if every timestep consists of a separate kernel launch it is ensured that all blocks are calculating the same point in time. As explained in chapter 6, at every kernel launch the entire grid must be loaded from global memory into shared memory, so moving back to one kernel per iteration introduces significant memory overhead which was tested to incur a performance penalty of about 15-20% as described in chapter 7 for the initial implementation.

SUB-ITERATION SYNCHRONIZATION

After reverting the kernel architecture to one kernel per iteration, small anomalies still occurred when waves crossed block boundaries, especially in the 1-D case where the anomalies consisted of partially reflected waves that would eventually grow catastrophically large. After a long time investigating possible causes it was eventually found to be fairly obvious. Since the shallow water equations are a coupled system of differential equations, trying to solve them using the Euler forward method described in chapter 3 results in an unstable scheme. To improve stability every variable update uses the most recent available information during calculation.

The update scheme consists of updating first U, then V and finally H. This means U is calculated fully explicitly and after U_{t+1} is calculated the newly updated value is used in the calculation of V_{t+1}. Finally both updated values U_{t+1} and V_{t+1} are used to calculate H_{t+1}.
This also means that every time a variable is updated, a block-level synchronization is necessary after which boundary values must be exchanged again. The initial implementation however only exchanged boundary values once at the beginning of the iteration, while it turns out three exchanges are necessary per time step.

This means that to ensure all blocks use updated values from their neighbors, every timestep must be split into three kernels, one for every variable that needs to be updated. As every kernel requires a complete load and subsequently write of the computational grid out of and into global memory, doing this three times per timestep is very inefficient and the performance impact of this will be further investigated in chapter 9.

One possible solution is surmised to be padding every block with additional ghost points which can be used to calculate the otherwise out of sync velocities within the thread block and thus eliminating the need for communication or synchronization. This will be part of possible future research.

One possible alleviation of this problem is the fact that the shallow-water scheme is also stable under the CFL condition if both U and V are calculated fully explicitly and

then after synchronisation calculate H with the updated U and V. This method would require one fewer synchronisation operations per timestep.
This is also called the Hansen [70] scheme.

It should be noted that the CPU C++ implementation is less hampered by this phenomenon, as CPU threads are able to go to sleep and then later resume execution. This means the barrier immediately synchronizes at block level as opposed to CUDA's thread level synchronization.

8.3.4. COMPARISON OF CODE COMPLEXITY

Code complexity and readability are important factors when considering moving production code from a CPU to a GPU platform. If GPU code is extremely complex and has poor readability then translation, integration and maintenance is difficult.

Below are two code snippets that calculate Hx, Hy, kfU, kfV and finally U. The first is MATLAB code, the second GPU Cuda C++ code. Unfortunately due to formatting the longer code expressions are poorly legible, but the takeaway is that the two are very similar.

A major difference, however, is that in GPU code updating the U values requires a separate kernel. This means the kernel requires pointers to the vectors in device memory, then shared memory must be allocated and the relevant variables must be copied into the shared memory. Finally the main variables are stored in device memory in one dimensional arrays and thus requires a coordinate mapping for 2D access.

The update expressions themselves do not differ substantially, and the conclusion is that when translating a Matlab program into Cuda C++ the main difficulty lies in designing the overarching program structure and the device specific considerations.

```
%%% MATLAB

%define computational domain
 i=3:n+1;
 j=3:n+1;

% Define Hx and Hy
Hx(j,i) = (U(j,i)>0).*H(j,i) + (U(j,i)<0).*H(j,i+1) +
 (U(j,i)==0).*max(H(j,i),H(j,i+1));
Hy(j,i) = (V(j,i)>0).*H(j,i) + (V(j,i)<0).*H(j+1,i) +
 (V(j,i)==0).*max(H(j,i),H(j+1,i));

% wetting and drying
 kfu(j,i-1) = Ones - (Hx(j,i-1)< droogval);
 kfv(j-1,i) = Ones - (Hy(j-1,i)< droogval);
 kfh(j,i)    = (1-(1-kfv(j,i)).*(1-kfv(j-1,i)).*(1-kfu(j,i)).*(1-kfu(j
     ,i-1)));

%% Update Vectors

% Update U

U(j,i) = (kfu(j,i).*(U(j,i)  -g*dt/dx* (H(j,i+1)-H(j,i)+(D(j,i+1)-D(
     j,i)))  ...  %Gravity term

-dt./(dx*(1+kfu(j,i-1).*kfu(j,i+1))).*( kfu(j,i+1).*(U(j,i+1)-U(j,i)
     )
+kfu(j,i-1).*(U(j,i)-U(j,i-1)) ).*U(j,i)... % Central diff length
     advection

+(V(j,i)>0).*(-    dt./((1+kfu(j-2,i))*dy).* kfu(j-1,i).*
 ( (1+2*kfu(j-2,i)).*U(j,i)-(1+3*kfu(j-2,i)).*U(j-1,i)
+kfu(j-2,i).*U(j-2,i)).*(V(j-1,i)+V(j-1,i+1))/2) ...  %advection V+

+(V(j,i)<0).*(-    dt./((1+kfu(j+2,i))*dy).* kfu(j+1,i).*
 (-(1+2*kfu(j+2,i)).*U(j,i)+(1+3*kfu(j+2,i)).*U(j+1,i)
-kfu(j+2,i).*U(j+2,i)).*(V(j,i)+V(j,i+1))/2))...     %advection V-

 ./(1+kfu(j,i).*dt*cf*sqrt((U(j,i).^2+((V(j,i)+V(j-1,i))/2).^2))
 ./( max(droogval/1000,Hx(j,i)))); %Bottom friction (implicit)
```

```
%%%%% CUDA C++

__global__ void
__launch_bounds__(MAX_THREADS_PER_BLOCK, MIN_BLOCKS_PER_MP)
updateUorV(float *h, float *U, float *V, float* Utemp, float* Vtemp,
float *D, float dt)
{

__shared__        float    s_h[BLOCK_SIZE_y + 4][BLOCK_SIZE_x + 4];
__shared__        float    s_U[BLOCK_SIZE_y + 4][BLOCK_SIZE_x + 4];
__shared__        float    s_V[BLOCK_SIZE_y + 4][BLOCK_SIZE_x + 4];
__shared__           __int8 s_kfU[BLOCK_SIZE_y + 4][BLOCK_SIZE_x + 4];
__shared__           __int8 s_kfV[BLOCK_SIZE_y + 4][BLOCK_SIZE_x + 4];

unsigned int j = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
unsigned int si = threadIdx.x + 2; // local i for shared memory access
unsigned int sj = threadIdx.y + 2; // local j for shared memory access
unsigned int globalIdx = (j + 2) * n_d + i + 2;

float s_hymin; float s_hxmin; float s_hx; float s_hy;

//load from global memory
s_U[sj][si] = U[globalIdx];
s_V[sj][si] = V[globalIdx];
s_h[sj][si] = h[globalIdx];

//Borders
** load border values **

//Calculate hx and hy
s_hx = (s_U[sj][si] > 0) *s_h[sj][si] +
(s_U[sj][si] < 0) *s_h[sj][si + 1] +
(s_U[sj][si] == 0) *fmaxf(s_h[sj][si], s_h[sj][si + 1]);

s_hy = (s_V[sj][si] > 0)* s_h[sj][si]
+ (s_V[sj][si] < 0) * s_h[sj + 1][si]
+ (s_V[sj][si] == 0) * fmaxf(s_h[sj + 1][si], s_h[sj][si]);

__syncthreads();

//wetting/drying
s_kfU[sj][si] = 1 - (s_hx < droogval);
s_kfV[sj][si] = 1 - (s_hy < droogval);

__syncthreads();

//update U
Utemp[globalIdx] = s_kfU[sj][si] * (s_U[sj][si] +
-g * dt/dx * (s_h[sj][si+1]-s_h[sj][si] + D[globalIdx+1]-D[globalIdx])
    +
-dt / (dx*(1+s_kfU[sj][si-1]*s_kfU[sj][si+1]))*(s_kfU[sj][si+1] *
(s_U[sj][si+1] - s_U[sj][si]) + s_kfU[sj][si-1]
* (s_U[sj][si] - s_U[sj][si-1]))* s_U[sj][si] +
(s_V[sj][si] > 0) *
-dt / ((1 + s_kfU[sj-2][si])*dy) * s_kfU[sj-1][si] *
((1 + 2 * s_kfU[sj-2][si])*s_U[sj][si] - (1 + 3 * s_kfU[sj - 2][si])*
s_U[sj-1][si] + s_kfU[sj-2][si] * s_U[sj-2][si])  *
(s_V[sj-1][si] + s_V[sj-1][si+1]) / 2+

(s_V[sj][si] < 0) *
-dt / ((1 + s_kfU[sj + 2][si])*dy) * s_kfU[sj + 1][si] *
(-(1 + 2 * s_kfU[sj + 2][si])*s_U[sj][si] + (1 + 3 * s_kfU[sj + 2][si])
    *
s_U[sj + 1][si] - s_kfU[sj + 2][si] * s_U[sj + 2][si]) *
(s_V[sj][si] + s_V[sj][si + 1]) / 2);

/(1+s_kfU[sj][si]*dt*cf*sqrtf((s_U[sj][si]*s_U[sj][si]+
( (s_V[sj][si]+s_V[sj-1][si])/2)*(s_V[sj][si]+s_V[sj-1][si])/2)))
/( fmaxf(droogval*.01,s_hx)))
```

9 BENCHMARK RESULTS OF THE MODIFIED STELLING & DUINMEIJER SCHEME

As described in chapter 8 the initial Stelling & Duinmeijer scheme described in chapter 6 was modified in order to mimic Deltares' Delft3D-FLOW as closely as possible. In addition a C++ CPU implementation has been done in order to be able to compare CPU and GPU performance more fairly.

9.1. COMPUTATION TIME

In this section the computation time and speedup for two test cases are compared in order to provide a measure of the speedup that can be obtained by doing shallow-water simulations on a GPU. The two test cases are described in section 8.2.

GPU KERNELS

In subsection 8.3.3 it is described that after additional research it was concluded that the initial implementation of the scheme described in chapter 6 needed to communicate subgrid borders more than once per timestep, which the initial implementation lacked.

Subgrid communication is a very expensive operation on a GPU as it involves writing the contents of the shared memory to global memory for all thread blocks, followed by writing the updated values back into shared memory. In order to measure the performance impact of this additional synchronisation, two different GPU kernels were developed as described in 8.3.3:
One that uses the original Sielecki scheme which requires three synchronisation operations per timestep, and the Hansen that calculates both U and V fully explicit requiring only two synchronisation operations per timestep. Comparing execution times of these

two kernels will then provide a measure of the performance impact of a single synchronisation operation.

Multithreading

A common criticism of studies that measure GPU acceleration is that a highly parallel GPU program is compared with a single threaded CPU program which is not very fair. Nowadays most computer systems have a minimum of four cores but higher core counts are becoming increasingly common. Additionally server computers often sacrifice single core performance for a vastly increased core count.

Because a comparison with a multi-threaded CPU program is both relevant and more fair such a program has been implemented as described in chapter 6, together with all modifications to the scheme described in chapter 8. Similar to a GPU implementation, multithreading requires additional synchronisation steps that add overhead to the program, and as such multi core performance is not simply a matter of dividing the single core performance by the number of cores. Therefore in order to compare multi-core scaling benchmarks for various thread counts have been performed.

Since both test systems described in the next subsection can use hyperthreading allowing for more concurrent active threads than there are physical cores on the CPU, benchmarks include the case where the thread count is twice the physical core count of the processor. To see if there is any benefit to be gained by increasing thread count above the maximum amount of concurrent threads benchmarks also include this scenario.

Test systems

The benchmarks have been performed on two different computer systems, the same as described in chapter 7. System one is a high-end consumer desktop that contains top of the line components at the time of writing. The second system is a 7 year old laptop that would be considered low-end by todays standards.

The first system uses a 6 core, 12 threads Intel i7 8086k together with an Nvidia RTX 2080 Ti. The Intel i7 8086k is operating at 5000 Mhz with 32 GB of DDR4 DRAM operating at 3200 Mhz with Cas Latency 14. The RTX 2080 Ti has 68 streaming multiprocessors with a total of 4235 Cuda cores operating at 2160 Mhz. It can handle up to 1024 threads per block, 1024 threads per SM and 64 registers per thread. It has 11 GB of DDR6 Vram with a 352 bit memory bus with a maximum bandwidth of 616 GB/s. Furthermore it has a maximum of 64 Kbyte of shared memory per SM.

The second system has a 4 core, 8 thread Intel i7 3610QM an Nvidia Quadro K1000M. The Intel i7 3610QM is operating at 3100 Mhz with 8GB of DDR3 Dram operating at 1600Mhz Cas Latency 11. The Quadro K1000M has 1 streaming multiprocessor with a total of 192 Cuda cores operating at 850 Mhz. It can handle up to 1024 threads per block, 2048 threads per SM and 32 registers per thread. It has 2 GB of DDR3 Vram with a 128 bit memory bus with a maximum bandwidth of 28.8 GB/s. Furthermore it has a maximum of 48 KByte of shared memory per SM.

9.1.1. VALIDATION

As no analytical solution exists for the Shallow-Water equations except for specific cases, it is not possible to validate the model results by comparing the numerical solution to an analytical one. We do know however, that the spatial discretizations described in chapter 6 and the improvements in chapter 8 give the numerical scheme first order accuracy in time because of the Euler forward scheme, first order accuracy in space for the initial implementation and second order accuracy in space for the improved implementation.

The results from the MATLAB implementation and both the GPU CUDA C++ and CPU C++ have been compared by exporting the computed values to a .txt file and then importing them into Matlab. Results were the same up to acceptable accuracy. Single precision floating point arithmetic is only guaranteed to be accurate to up to 6 digits, and observed numerical differences were smaller.

All three implementations were also validated by visual inspection through plotting the resulting velocities and water levels using MATLAB and checking for discontinuities and general subjective "water like" behaviour.

9.1.2. TEST CASE 1

In figure 9.1 and table 9.1 the test system 1 CPU vs GPU comparison graph is presented for test case 1, described in section 8.2. Surprisingly the GPU already outperforms the CPU on a 100×100 grid, which is so small that it only occupies 9 out of the 68 multiprocessors on the device. From the curvature in both GPU graphs can be concluded that computation time does not linearly scale with problem size for grids smaller than 10^5. For the CPU kernel the highly multithreaded $CPU12$ and $CPU24$ runs also do not scale linearly for the smallest two grids. This is expected as dividing the problem into many threads introduces a significant synchronisation overhead, which is only compensated for when computational load is large, similar to a GPU. If we inspect table 9.1 we can observe that both CPU and GPU scale almost linearly with problem size for the larger grids which is to be expected.

In figure 9.2 we can observe that for grids larger than around 4 million points the fastest GPU kernel provides a speedup of around 45 times compared to the fastest CPU implementation.

When comparing the performance of the CPU C++ implementation for various thread-counts, it can be observed that for the smallest chosen grid the additional overhead induced by increasing the threadcount to 12 or 24 hampers computation time. For larger grids we see that 12 threads has the best performance, closely followed by 24 and 6 threads. This means that Intel Hyperthreading does offer a small performance gain over 1 thread per core, but increasing thread count beyond the CPU's maximum concurrent thread count has no positive effect.

If the single and multi threaded computation times are compared it can be observed that increasing the number of threads from 1 to 2 provides an almost 2x speedup for most grid sizes. Moving from 2 to 3 also provides a near 1.5 times speedup. However when moving from 3 to 6 threads the scaling falls short of being linear, seemingly independent of problem size. This indicates that for high thread counts a significant overhead is introduced.
The thread barrier function that is used to synchronize the different threads is self-written and probably nowhere near maximum efficiency, which could be a possible explanation for the suboptimal multicore scaling. Another possible factor is that the L3 cache on this CPU is shared between all cores, which means there is less L3 cache per core available as the number of active cores increases.

SYSTEM 1

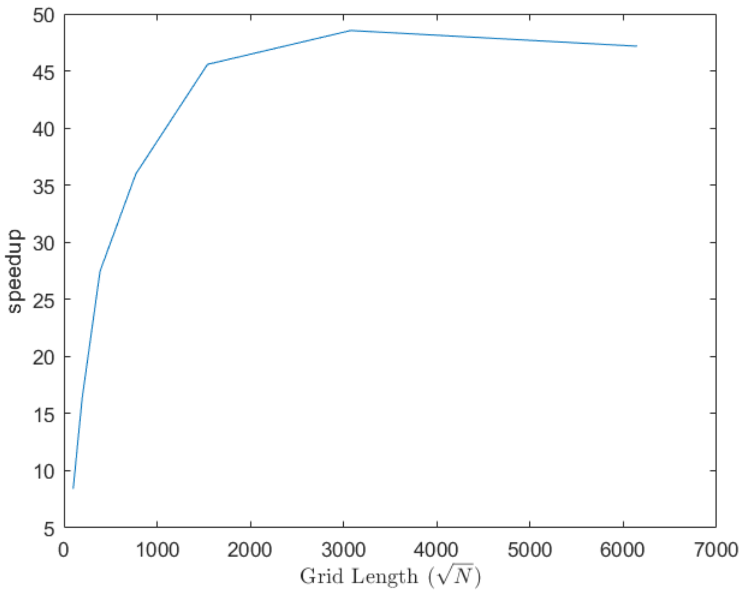


Figure 9.2: Plot of speedup versus grid length in both directions, comparing the Hansen Cuda kernel with the 12 thread CPU C++ implementation for test system 1

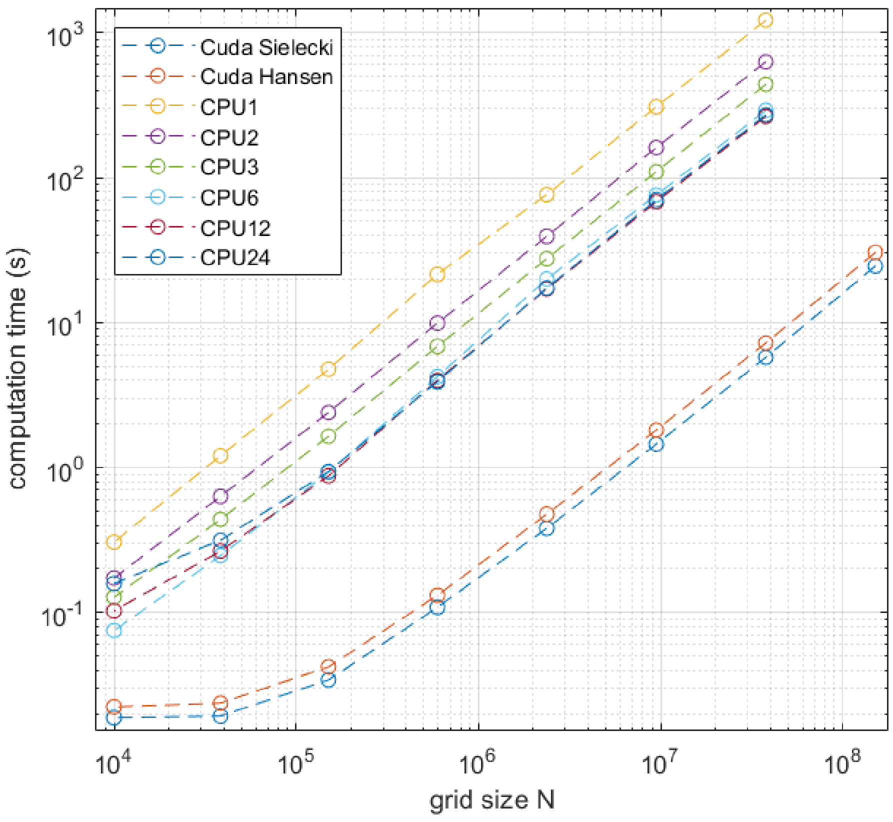


Figure 9.1: Plot of computation time of 1000 time steps in seconds for test case 1 on system 1 for various grid sizes. Cuda Hansen is the modified kernel that exchanges borders twice per timestep, Cuda Sielecki exchanges three times. The CPU rows are the computation times for the C++ CPU implementation for various thread counts.

$\sqrt{N}$	100	196	388	772	1540	3076	6148	12292
Hansen	0.018800	0.019270	0.034192	0.10853	0.38058	1.44965	5.7570	24.589
Sielecki	0.022322	0.023685	0.042107	0.13113	0.47666	1.81252	7.2415	30.543
CPU1	0.305687	1.20825	4.76128	21.5301	76.5302	308.639	1222.5	
CPU2	0.172804	0.632221	2.39847	9.93223	39.3897	161.265	630.70	
CPU3	0.127358	0.438293	1.64414	6.86199	27.5986	109.936	441.34	
CPU6	0.074952	0.246507	0.929399	4.23524	19.9891	76.0721	291.55	
CPU12	0.102501	0.26558	0.875854	3.99464	17.1241	68.1345	264.36	
CPU24	0.158379	0.314748	0.939437	3.90622	17.3451	70.3667	271.53	

Table 9.1: Table of computation time of 1000 time steps in seconds for test case 1 on system 1 for various grid sizes. Hansen is the modified kernel that exchanges borders twice per timestep, Sielecki exchanges three times. The CPU rows are the computation times for the C++ CPU implementation for various thread counts.

System 2

In figure 9.3 and table 9.2 the computation times of test case 1 for system 2 can be observed. Two additional GPU results are present: The first two kernels use a 32×32 block size for a total of 1024 threads per block, which means that it is possible for two active blocks per SM. The 16×16 kernels have 256 threads per block which allows for 8 active blocks on the SM.

If we compare the 4 GPU kernels, it is no surprise that again the Hansen kernel is superior to the Sielecki kernel.
When observing table 9.2 it can be noted that the 16×16 kernel is superior to the 32×32 one. However, for test system 1 there was no discernible difference between the two configurations. One possible explanation is that since the GPU from test system 2 is of an older generation it might lack some scheduling optimizations.
Another possible explanation could be that since the Quadro K1000M only has a single SM, block scheduling becomes more important. If the two resident blocks for the 32×32 kernel finish at the same instant, the device idles while two new blocks are initialized. However with 8 resident blocks the initialization could be hidden through block switching. The 2080 Ti contains 68 SMs instead of a single one, which would make it much less sensitive to this kind of scheduling problems.
It must be noted that block-size performance dependency can be seen as guesswork as the inner workings of a GPU device are very complex. Official Cuda programming guides generally advise to optimize for block size through trial-and-error. Therefore there is a fair chance that the performance differences between the 16×16 kernel and the 32×32 kernel on both devices have different origins than theorized above.

When we observe figure 9.3 core scaling and grid scaling is similar to that of system 1. The GPU results lie much closer to the CPU results, but that is to be expected as the system has a relatively weak GPU and strong CPU. Observing table 9.2 we see both CPU and GPU performance approximately linear with problem size, apart from the 100×100 grid. This is to be expected as the Quadro GPU is only able to have 2048 concurrently active threads, which means we expect linear scaling starting at around a 70×70 grid.

When observing the speedup in figure 9.4 the best-case GPU implementation provides a roughly 1.7 times speedup compared to the best case scenario 8-thread CPU implementation. This leads to the conclusion that a moderate GPU speedup can be obtained even without dedicated high-end graphics cards.

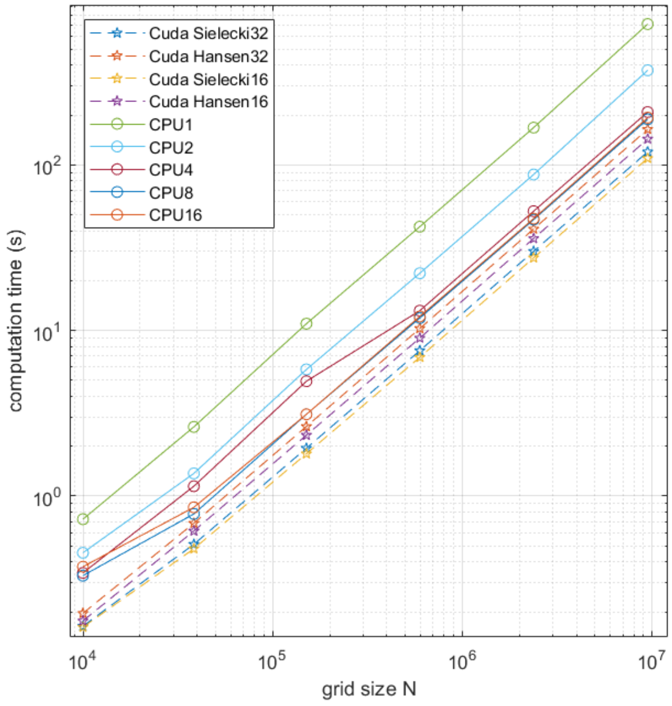


Figure 9.3: Plot of computation time of 1000 time steps in seconds for test case 1 on system 2 for various grid sizes. Cuda Hansen32 is the modified kernel with a block size of 32 × 32 that exchanges borders twice per timestep, Cuda Sielecki16 uses a 16 × 16 block size and exchanges three times. The CPU rows are the computation times for the C++ CPU implementation for various thread counts.

$\sqrt{N}$	100	196	388	772	1540	3076
Hansen32	0.163908	0.511419	1.94641	7.55688	30.0658	119.98
Sielecki32	0.195926	0.685671	2.63318	10.2788	41.0195	163.975
Hansen16	0.161542	0.480747	1.80186	6.89172	27.4649	109.595
Sielecki16	0.176315	0.615512	2.3326	9.00504	35.936	143.347
CPU1	0.724834	2.61743	10.9859	42.3639	167.968	708.631
CPU2	0.454378	1.37038	5.81773	22.1663	87.4908	372.718
CPU4	0.343363	1.14705	4.93875	13.1716	52.532	208.498
CPU8	0.330632	0.780039	3.11883	11.896	46.7138	188.548
CPU16	0.374418	0.855233	3.11677	12.1545	47.3572	192.824

Table 9.2: Table of computation time of 1000 time steps in seconds for test case 1 on system 2 for various grid sizes. Hansen32 is the modified kernel with a block size of 32 × 32 that exchanges borders twice per timestep, Sielecki16 uses a 16 × 16 block size and exchanges three times. The CPU graphs are the computation times for the C++ CPU implementation for various thread counts.

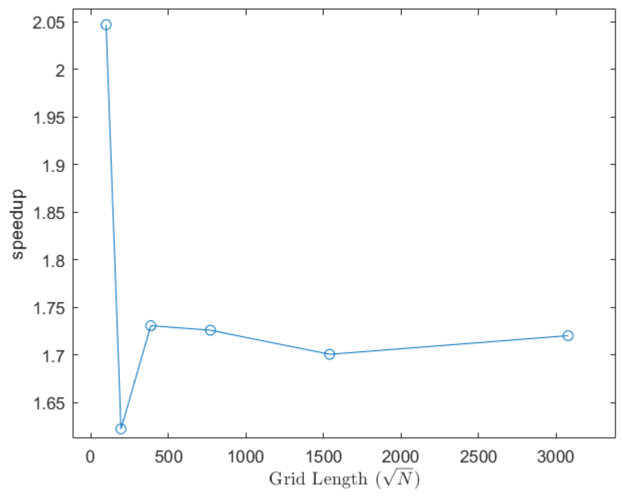


Figure 9.4: Plot of speedup versus square root of grid size, comparing the GPU16$_{2ex}$ kernel with the 8 thread CPU C++ implementation for test system 2

9.1.3. TEST CASE 2

Test case 2 as described in section 8.2. Instead of calculating a fixed number of timesteps as in test case 1, it uses a dynamic timestep depending on the tidal level and terminates after 24 simulated hours, or 86400 seconds. This means the amount of work to be done no longer scales linearly with problem size. The CFL condition and thus the timestep scales with the root the grid size, while the computation time for a single iteration scales linearly. This a combined factor of $t \sim N^{1.5}$. For example this means that when increasing grid size from $N = 100 \times 100$ to 1000×1000 the grid is 100 times larger and the timestep is 10 times smaller for a combined workload increase of a factor 1000.

However, due to the extreme computational size of this test case computations will be limited to the RTX 2080Ti GPU and grid sizes of 1 million points or less, which is similar to the grid sizes tested by Peeters [69]. In test case 1 it was observed that computation time does not start to scale linearly with problem size until around 2×10^5 points. It is then no surprise that the results in table 9.3 do not yet exhibit this scaling behaviour. For the GPU$_{2ex}$ kernel the computation time increase from 100×100 to 1060×1060 was roughly a factor 115, 10 times less than the predicted factor 1200. It follows that the overhead for smaller problem sizes is large.

Another observation is that the number of iterations performed scale linearly with the root of the problem size, which is expected behaviour.

If we compare the results from table 9.3 with the results from Peeters in table 9.4, we see that both kernels on the RTX 2080 Ti outperform both the FirePro D700 and the HD Graphics 4000. This is a meaningless comparison however, as it compares two different implementations on two different machines, making it impossible to ascribe performance differences to either with any certainty.

It is however interesting to compare with the Delft3D-FLOW benchmark in table 9.5 with the results from table 9.3. Comparing the smallest grid we observe a speedup of roughly a factor 7 or 8 accounting for the grid size difference. The most striking result is a comparison of the largest grid sizes, which results in a speedup of a factor 126 when accounting for the grid size difference.

$\sqrt{N}$	96	192	384	1056
time steps	237670	465642	922757	2510297
Hansen	4.34653	8.89584	30.2175	503.034
Sielecki	5.07782	10.678	36.6837	659.177

Table 9.3: Table of computation time in seconds for 24 simulated hours for various grid sizes on test system 1. The Hansen kernel exchanges borders twice per iteration and the Sielecki kernel three times.
The time steps row denotes the number of iterations required to simulate 24 hours.

$\sqrt{N}$	96	128	256	1024
time steps	864000	864000	1728000	8640000
FirePro D700	32	32	152	6120
HD Graphics 4000	3960	4320		

Table 9.4: Table of computation time in seconds for 24 simulated hours for various grid sizes benchmarked by Peeters [69] on an AMD Radeon HD - FirePro D700 and an Intel HD Graphics 4000.

$\sqrt{N}$	100	1000
time steps	22880	28800
Delft3D-FLOW	40	57600

Table 9.5: Table of computation time in seconds for 24 simulated hours for various grid sizes benchmarked by Delft3D-FLOW by Lotte Peeters [69].

10 Benchmark results of the implicit implementation and preconditioners

In section 6.4 the MATLAB implementation of the developed implicit shallow-water solver is described. The Shallow-Water equations are split into an implicit gravity-wave system and an explicit momentum-advection system. The CFL condition states the time step is most stringently restricted by the gravity-wave system, and thus solving this system implicitly allows for larger timesteps. Formulating and solving the implicit system takes longer than solving the explicit system, and thus the question is whether the larger timestep compensates for the additional work per iteration.

The implicit system was implemented in CUDA C++ using a Conjugate Gradient solver, described in chapter 4. The Conjugate Gradient solver implemented uses NVIDIA's CUSparse and CUBlas libraries for vector operations.
The system matrix is constructed by a GPU kernel in CSR format. Theoretically the diagonal format is the most efficient for a pentadiagonal system, however the diagonal format is not supported by the CUSParse library and thus the CSR format was chosen.

The performance of the CG method is highly dependent on the spectrum of the system matrix. In the case of CG the convergence rate is tied to the ratio of the smallest and largest eigenvalues of the system matrix. Therefore an efficient preconditioner that brings the eigenvalues closer together without changing the solution is often used to accelerate the method's convergence. However, the construction of the preconditioner plus the additional work of solving a preconditioned system also takes time.
Because of the preconditioner the CG algorithm will converge in fewer iterations, and for a positive acceleration it is necessary that more time is saved by the reduction in CG iterations than the cost of preconditioning.

CUSP LIBRARY

As part of preconditioner testing, the CUSP solver library has been integrated into the system. The library provides great compatibility with an externally constructed system matrix and right-hand side. It provides an implementation of the CG algorithm and offers the following preconditioners:

- Diagonal
- Smoothed Aggregation Algebraic Multigrid
- Scaled Bridson Approximate Inverse [71]
- Lin et al.[72] Approximate Inverse

CUSP documentation on the SAAMG preconditioner [73]:

"Smoothed aggregation is expensive to use but is a very effective preconditioning technique to solve challenging linear systems. The default configuration uses a symmetric strength measure, MIS-based aggregation in device memory, sequential aggregation in host memory, Jacobi smoothing is applied to the tentative prolongator, Jacobi relaxation on each level of hierarchy and LU to solve the coarse matrix in host memory."

The scaled Bridson Approximate Inverse preconditioner uses Bridson's [71] dropping strategies, either static tolerance or a fixed number of non-zeroes per row. The Lin et al. [72] uses the dropping strategy proposed by Lin to restrict fill in.

The CUSP library was chosen for benchmarking because it is a C++ template library, which means source files need to be included but no CMake building is necessary, which leads to very easy integration into existing programs. However, the latest release of the library is targeted at CUDA 5.5. Attempting to use the library in a CUDA 10 project led to build errors which thankfully were resolved by a GitHub forum user who provided a solution.

MARIN REPEATED RED BLACK SOLVER

The Maritime Research Institute Netherlands, or MARIN has provided their Repeated Red Black CG pentadiagonal solver developed by De Jong [60] [61] in order to compare performance. The solver unfortunately only accepts host memory input which it then copies to the GPU internally. As the SWE system matrix is constructed on the GPU, it needs to be copied to the host in order to present it to the RRB solver, which then copies it back internally. This is highly inefficient but it should be possible to modify the solver to accept device input instead.

10.1. RESULTS

10.1.1. SOLVE TIME

In figure 10.1 and table 10.1 the average computation time per timestep is presented, comparing the explicit implementation described in chapter 8, the CUBlas/CUsparse implicit CG implementation, the MARIN RRB solver and the CUSP solver with various preconditioners for test case 1. The implicit timestep was chosen to be a factor 100 times larger than the explicit timestep, and the explicit results are thus the average computation time of 100 iterations for a fair comparison. For the CG method the starting solution was taken to be the solution of the previous time step, and an absolute stopping criterion of $1e^{-5}$ was used. Testing absolute and relative stopping criteria provided similar results.

From the results it is evident that the explicit solver is approximately 8 times faster than the fastest implicit implementation. Among the implicit solvers we see the CUBlas/-CUSparse implementation is the fastest method, but it is beaten by the RRB solver on small grids. The Multigrid solver is competitive with CUSPs bare CG and diagonally preconditioned CG, and all three AINV solvers have very poor performance. It should be noted that the Bridson 1 preconditioner which uses a static 10 % dropping strategy did not converge.

Finally it should be noted that the shown computation times for the Marin RRB solver include unnecessary data transfer between CPU and GPU. The fact that the RRB solver is faster than the CUBlas/CUSparse implementation on smaller grids is likely due to the fact that the extra data transfer between host and device becomes more significant as grid size increases.

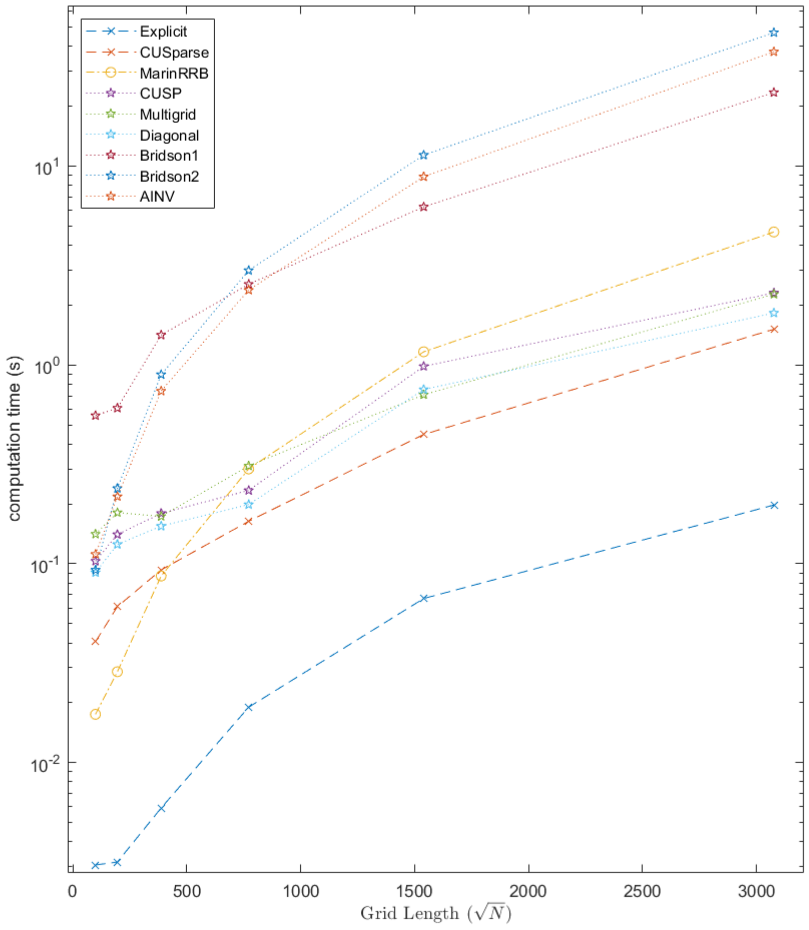


Figure 10.1: Average timestep calculation time in seconds versus grid length for various solvers. Implicit timestep was 100 times the explicit timestep size.

$\sqrt{N}$	96	192	384	768	1536	3072
explicit	0.00305674	0.00316147	0.00591104	0.0189923	0.0668797	0.197481
CUSparse	0.0405446	0.060885	0.0931468	0.16353	0.447088	1.51235
CUSP	0.103157	0.140323	0.179233	0.23392	0.983736	2.30398
Diag	0.0903021	0.125185	0.154624	0.198562	0.752405	1.82552
SAAMG	0.141092	0.180848	0.173174	0.310631	0.708295	2.2692
Bridson 2	0.0930085	0.23944	0.894763	2.98523	11.3594	46.9521
AINV	0.111692	0.217752	0.73924	2.37956	8.84294	37.6046
MarinRRB	0.0174882	0.028635	0.0867973	0.30004	1.16445	4.66386

Table 10.1: Average timestep calculation time versus grid length for various solvers. Implicit timestep was 100 times the explicit timestep size.

10.1.2. PRECONDITIONER ITERATIONS

In table 10.2 the average number of Conjugate Gradient iterations before the residual was of the same order as the single machine precision are presented for various implicit time step size factors on a grid of length 1536. If we observe the implicit and CUSP rows we can see they require a similar number of iterations, which is to be expected as both methods implement the bare CG algorithm. The number of iterations required scales almost linearly with time step factor, which is expected behaviour as the system matrix becomes less diagonally dominant as time step size increases which slows CG convergence speed.

The Diagonal preconditioner had no effect for smaller time step sizes, but led to an approximately 10% reduction for the larger step sizes. Since a diagonal preconditioner can be implemented at very low cost, this means that for larger time steps it is preferred over the bare CG implementation.

The best performer is the Marin RRB preconditioner, which reduces the required iterations by approximately 90 % for the largest time step size. It is closely followed by the SAAMG preconditioner, which is seen to have similar scaling but requires 3-4 times as many CG iterations before convergence.

The Bridson 1 preconditioner with a static drop tolerance did not produce a result for larger time steps and thus can be concluded as ineffective. The Bridson 2 preconditioner which allows 10 non-zero entries per row performed slightly better than the Lin et al. AINV preconditioner.

dt_{impl}/dt_{expl}	1	10	50	100
CUSparse	4	41	255	542
CUSP	3	41	254	541
Diag	3	42	231	496
SAAMG	2	8	16	20
Bridson 1	3	NaN	NaN	NaN
Bridson 2	1	12	69	148
AINV	2	16	88	187
MarinRRB	1	2	4	7

Table 10.2: Table of required number of Conjugate Gradient iterations required for residual $||r||_2 < 1e^{-5}$ for various time step sizes on a grid of length 1536. The row dt_{impl}/dt_{expl} indicates the fraction between the used implicit time step size and the explicit time step size.

10.1.3. PRECONDITIONER CONSTRUCTION TIME

In table 10.3 the average time in milliseconds necessary to construct the preconditioner for various grid sizes is presented with a timestep of 100 times the explicit time step. It should be noted that the preconditioner construction time was found be independent of the time step size, which is to be expected.
When comparing the results the diagonal preconditioner is the fastest which is also expected behaviour.

While in the preceding subsection it appears that the SAAMG preconditioner and the Marin RRB preconditioner were both highly effective, the RRB construction time beats the SAAMG preconditioner by a factor 60. This is not surprising as the CUSP documentation states the SAAMG preconditioner is partially constructed on the CPU, while the RRB preconditioner is constructed fully on the GPU.

Comparing the three AINV preconditioners the static drop tolerance Bridson 1 preconditioner is the fastest. Unfortunately it was found in the preceding sections to not converge for larger time steps. The Lin et al preconditioner is 33% faster than the Bridson 2 preconditioner, and even though it requires more iterations before convergence as seen in figure 10.1 it is still slower than the Bridson 2 preconditioner.

$\sqrt{N}$	96	192	384	768	1536	3072
Diag	0.087776	0.28731	0.35554	0.44882	1.005	6.11632
MarinRRB	0.48714	0.55299	0.75062	3.5846	8.663	31.5106
SAAMG	57.287	74.278	113.38	212.15	565.81	1855.03
Bridson1	6.8364	23.101	86.778	331.13	1375.2	12669.5
Bridson 2	46.938	169.53	666.81	2692.1	10799	45607
AINV	30.447	121.41	474.48	1903	7819.8	33611.1

Table 10.3: Table of average preconditioner construction time in milliseconds for various grid sizes with a timestep of 100 times explicit

The previous sections showed the Marin RRB preconditioner to be the fastest when considering the combination of preconditioner setup time and CG iterations.
The question remains whether it is actually faster than the explicit method or the bare CUBlas/CUSparse implementation. For this reason the benchmark was modified by measuring the total time taken by all the copy operations between CPU and GPU and subtracting it from the total run time.
This benchmark was performed for an implicit timestep size both 10 and 100 times the explicit timestep. The results are presented in tables 10.4 and 10.5.

$\sqrt{N}$	100	196	388	772	1540	3076	6148
explicit	0.0008568	0.0008954	0.002055	0.0075267	0.027688	0.103217	0.4100
CUSparse	0.036858	0.0403108	0.052227	0.0927517	0.255135	0.908019	3.5782
MarinRRB	0.01192	0.0159365	0.029711	0.0738711	0.167786	0.454324	1.5484

Table 10.4: Computation time of the equivalent of 100 time explicit time steps in seconds for various grid sizes with an implicit time step 10 times the size of the explicit time step.

$\sqrt{N}$	100	196	388	772	1540	3076	6148
explicit	0.004998	0.00448	0.010604	0.035240	0.12883	0.52132	1.8883
CUSparse	0.1571	0.232	0.309392	0.560343	1.71688	5.81244	22.4183
MarinRRB	0.0089	0.01059	0.019798	0.038388	0.090922	0.25789	0.88327

Table 10.5: Computation time of the equivalent of 500 time explicit time steps in seconds for various grid sizes with an implicit time step 100 times the size of the explicit time step. 500 steps are taken as otherwise the implicit method would only calculate a single timestep which is prone to error.

If we compare the computation times in table 10.4 it is evident that the explicit solver outperforms the Marin RRB solver by about a factor 4 for the larger grid sizes. On the other hand, the Marin RRB solver proved to be more than twice as fast as the CUSparse/CUBlas CG implementation on the larger grids. Comparing the smaller grid sizes it is evident that the smaller grid sizes heavily favor the explicit method.

On the other hand, if we compare the computation times in table 10.5 we see that for a very large time step of 100 times explicit the Marin RRB solver is about twice as fast as the explicit method on the largest two grids, while on the smaller grids they are comparable. The performance difference between the CUSparse/CUBlas implementation and the RRB solver is also increased significantly compared to table 10.4. This is no surprise as the CUSparse/CUBlas implementation goes from 40 to 550 required CG iterations per timestep, while the RRB solver moves from 2 to 7 iterations.

Another interesting question is at which time step size the Marin RRB-solver and the explicit solver have similar performance. In figure 10.2 and table 10.6 the average computation time of the equivalent of 5 implicit timesteps is plotted for varying time step sizes is presented. This means that if the timestep factor was 50, 5 implicit time steps were measured and 250 explicit time steps. Evident from the plot is that the explicit and implicit both suffer in performance as a larger timestep is taken. The explicit method because it has to calculate more time steps to match the implicit solver, and the CUSparse/CUBlas method suffers because the number of CG iterations per timestep increases with time step, as seen in table 10.2. The RRB Solver, on the other hand, has an almost constant computation time independent of timestep and thus overtakes the explicit method in performance somewhere between 50 and 60.

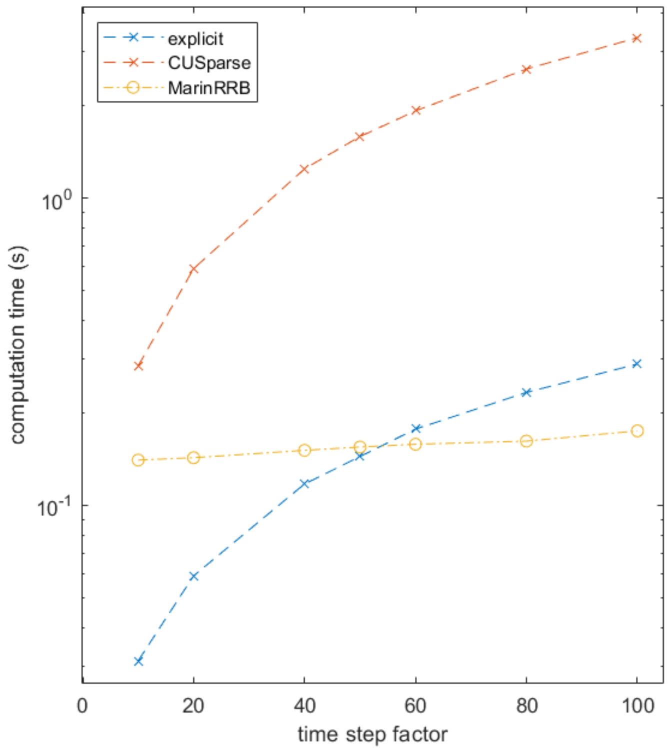


Figure 10.2: Plot of average computation time in seconds of the equivalent of 5 implicit timesteps for varying time step sizes

dt_{impl}/dt_{expl}	10	20	40	50	60	80	100
explicit	0.031132	0.058927	0.11742	0.14411	0.177161	0.2331	0.28858
CUSparse	0.284537	0.590082	1.24751	1.58977	1.92637	2.6366	3.3355
Marin RRB	0.140407	0.142916	0.15102	0.15478	0.158114	0.1618	0.1747

Table 10.6: Table of average computation time in seconds of the equivalent of 5 implicit timesteps for varying time step sizes

The conclusion here is that the Marin RRB solver, when implemented without the memory copy overhead, outperforms the explicit implementation on a large enough grid large enough implicit time step.

However, the implicit method's wetting and drying mechanism has been observed to become unstable if the implicit timestep is taken too large. Additionally, the models numerical accuracy is also dependent on timestep size, with large implicit time steps having been observed to introduce significant damping.

Therefore, the choice between the RRB method and the explicit method becomes a choice of model accuracy versus speed.

It should be noted that the benchmark cases simulate a very shallow depth. Large implicit timesteps are expected to be more stable in deeper water, and this should be tested in future work.

In real world scenarios Delft3D-FLOW uses implicit timesteps of up to 10 times the CFL prescribed time step, which makes the explicit method the preferred choice in these cases.

11 SUMMARY AND CONCLUSIONS

11.0.1. SUMMARY

The main research question of the project is “Which numerical method is best suited for solving the shallow water equations on a GPU in terms of model accuracy, robustness and speed?”

In order to answer this research question a number of numerical methods have been implemented and their performance compared.

First a literature review was conducted in order to define the shallow-water equations and explore possible discretization methods and subsequently numerical methods to solve them, which are chapter 1 through 3. In addition, possible GPU platforms and architecture considerations were discussed in chapter 4. Finally some implicit parallel solvers and their GPU suitability were discussed in chapter 5.

As a starting point for an implementation, the Stelling & Duinmeijer scheme [21] was taken. After modifying it to an explicit Sielecki scheme [66] it was suitable for an initial MATLAB implementation for testing. Next a GPU CUDA C++ translation of the MATLAB code was implemented, which both are described in chapter 6. The results were benchmarked, compared and discussed in chapter 7.

In order to mimic Deltares' Delft3D-FLOW shallow water solver a number of modifications to the initial implementation were developed, described in chapter 8, and in addition a CPU C++ implementation of the same code was developed in order to provide a more meaningful CPU and GPU comparison.

Finally, an implicit scheme was developed and implemented in both Matlab and Cuda C++ using the Conjugate Gradient method with various preconditioners from the CUSP [73] library to solve the linear system of equations every time step. In addition, in collaboration with MARIN their pentadiagonal solver developed by De Jong [60] [61] was integrated and performance was compared.

11.1. Conclusions

11.1.1. Research questions

What are the tradeoffs involved in solving the shallow water equations on a GPU using explicit methods compared to implicit?

From the results in chapter 10 it can be concluded that the fastest implicit solution method was faster than the explicit method with a time step more than 50 times larger. At such large time steps, the implemented wetting and drying mechanic has been observed to cause instability. Also, numerical error resulting from such a large time step was found to be significant for model behaviour.

The conclusion is that using explicit compared to implicit methods involves a tradeoff of accuracy and robustness versus speed, with the implicit method being faster but less accurate and not able to incorporate wetting and drying properly.

How does the performance of existing software packages compare to a self-built solver?

The implementation of the solver software packages described in chapter 5, most notably MAGMA, Paralution and AmgX, did unfortunately not succeed. However, the CUSP GPU library has been successfully integrated and tested, described in chapter 10. The CUSP bare CG implementation proved to converge to the desired tolerance in exactly the same number of iterations as the self built solver. However, it was found to be slower in general. Using CUSPS available preconditioners closed the gap, with the diagonal and SAAMG preconditioners having the best performance. However neither of them came remotely close to the RRB preconditioner developed by Marin. It might be possible for other packages to reach the performance of a self-built solver, as the CUSP package is a very small sample size. Thus more packages will need to be benchmarked for a definitive verdict.

The conclusion is that using the CUSP library is not optimal but it does have acceptable performance while being very easy to implement.

What are the tradeoffs involved in solving the shallow water equations on a GPU in 32-bit floating point precision compared to 64 bit and 16 bit?

During the course of the literature review it was concluded that a consumer GPU, such as the RTX 2080 Ti used in the course of this project, has 32 times the 64-bit performance when calculating in 32-bit, as explained in chapter 4. After some consideration it was chosen not to test this in practice as 32-bit precision is sufficient for a shallow-water model. 16 bit precision was proposed as Nvidia's Tensor can do calculations faster in 16-bit precision when compared to 32-bit. However when it became apparent it was not possible to utilize the Tensor cores in a shallow-water solver it was decided not to test 16 bit performance either.

WHICH METHOD OR SOLVER LIBRARY IS BEST SUITED FOR INTEGRATION INTO DELTARES' EXISTING FORTRAN BASED SOLVERS?

Because it was chosen to develop an explicit and implicit method from the ground up, Deltares' Delft3D-FLOW implementation was not examined. This research question can thus not be answered on the basis of the work done in this project. It is however highly likely that it is possible to do pre- and post processing in FORTRAN and then call CUDA C++ code to do the calculations. Whether this is indeed the case is recommended future research.

WHICH NUMERICAL METHOD IS BEST SUITED FOR SOLVING THE SHALLOW WATER EQUATIONS ON A GPU IN TERMS OF MODEL ACCURACY, ROBUSTNESS AND SPEED?

On the basis of the performance benchmarks of the implicit and explicit methods tested in the course of this project it can be concluded that an explicit method is best suited for GPU implementation on a high resolution grid with low water depth. For these cases the explicit implementation was the most accurate, most stable and fastest method. However specific cases can exist with high water depth and no wetting and drying where a large time step can be taken and then an implicit implementation will be preferable.

11.1.2. ADDITIONAL CONCLUSIONS

EXPLICIT IMPLEMENTATION

For an explicit time integration method it is possible to do the entire initialization phase on the GPU device itself, as opposed to CPU construction followed by costly data transfer between CPU and GPU.
In addition, the inherent parallelism of an explicit method makes it able to utilize a multi-core GPU architecture very well.

The explicit time integration method is also compatible with the addition of drying and wetting mechanics due to the small time step size it is restricted to by the CFL condition. An implicit method that uses larger time steps needs additional measures to ensure stability with the wetting and drying, but the small explicit time step means wetting and drying can be implemented at low cost.

However, after an initial CUDA C++ explicit implementation described in chapter 6 it was found that this parallelism is limited. Thread synchronization between vector updates turned out to be necessary and this requires separate kernel launches inducing significant overhead. In chapter 9 it was found that moving from 2 to 3 thread synchronizations per time step induces a 15-20 % performance penalty.

CUDA Cooperative Groups introduced in CUDA 9 would allow for in-kernel synchronisation, but restricts CUDA grid size with respect to device capabilities leading to higher code complexity. Cooperative groups are well suited for problems with high initialization costs such as this one. Thus it is expected that an implementation utilizing it will be faster, making it a good candidate for future research.

A GPU implementation was designed to have every thread executing the same expressions, achieved using control booleans described in chapter 8. Deltares' Delft3D-

FLOW was designed for vector computers with the exact same restrictions, making it well suited for GPU translation.

On the high-end test system 1 a speedup of a factor 45 was obtained when comparing the explicit GPU implementation with the same CPU C++ code. Results will vary by system but this illustrates significant speedups can be attained. On test system 2 a 1.7 times speedup was observed, illustrating that even with a low-end GPU a positive speedup is still possible.

A comparison with sequential semi-implicit CPU Delft3D-FLOW yielded a 120 times speedup for a 24 hour simulation of a tidal test case.

IMPLICIT IMPLEMENTATION

An implicit implementation that lifts most stringent water-level dependent restriction on the time step size was successfully developed. In this implementation described in chapter 6 and 10 it was found possible to construct the system matrix directly on the GPU in parallel, which is orders of magnitude faster than CPU construction plus transfer. The implicit implementation was found to be compatible with the wetting and drying mechanic introduced in chapter 8, given that the implicit time step was not taken too large.

The implicit implementation was found to have similar performance when using a CUSParse/CUBlas conjugate gradient method to solve the linear system when compared to CUSPs CG solver.

For unpreconditioned CG, the number of iterations required to reach single precision tolerance was found to scale approximately linear with the time step size. This had the consequence of a larger implicit time step only moderately improving performance.

A number of preconditioners were tested to see if CG performance could be improved to a point where the implicit implementation outperformed the explicit.
CUSPs smoothed aggregation algebraic multigrid preconditoner and Marins repeated red black preconditioner were both extremely effective. However, the SAAMG preconditioner had a high construction time while the RRB preconditioner was very fast to construct. All approximate inverse preconditioners tested had poor performance, and a diagonal preconditioner had a small speedup at larger time steps.

The RRB preconditioned implicit implementation was found to outperform the explicit implementation for implicit time step that were more than a factor 55 larger than the explicit time step.

Such a large implicit time step is rarely used in practice, making the explicit implementation the best performer on the GPU in most scenarios. However it could be possible that for scenarios with deeper water without the wetting/drying such a large time step can indeed be taken which would make an implicit method the best option.

11.2. Recommendations for future work

Research questions

In the course of the project not every research question was properly answered. Some of them turned out to be of a lower priority than initially thought. The comparison of 16- 32- and 64 bit precision was not made, only a single solver package was tested and no research was conducted into integrating the implementations built in the course of the project with Deltares' existing software. These three research questions do remain relevant, and answering them could be part of possible future work on the subject.

Benchmark cases

During the evaluation of the benchmark results it became apparent that for the tested cases the explicit method was superior. However the fact that for very large time steps the implicit method was faster for very large time steps means a test case should have been tested where such a large time step was a viable option. An example would be a test case with deeper water without wetting or drying. A good recommendation for future work is thus to benchmark new such test cases to verify the expectation that the implicit implementation will be the optimal choice in these scenarios.

Improvement of the RRB solver

As mentioned in chapter 10, the Marin RRB solver assumes the system matrix is constructed on the CPU and then passed to the solver method. This leads to many superfluous copy operations between CPU and GPU memory when the system matrix already exists on the GPU. For the benchmarking of the solver the copy time was simply measured and subtracted from the total run time. However a better solution would obviously be modifying the RRB solver so that it can also accept pointers to arrays that are already in GPU memory.

Cooperative groups

Both the implicit method and the explicit method suffer performance penalties that result from synchronization requirements. On a GPU thread synchronization is performed by launching a new kernel with high initialization costs. Especially the conjugate gradient method suffers from this as every matrix or vector operation requires a separate kernel, many of which occur every iteration of the method. As mentioned before a possible solution to this could be CUDA cooperative groups. This feature introduced in CUDA 9 allows for easier synchronization of thread groups across blocks and even across devices if so desired within a single kernel. A drawback of the cooperative groups feature is that it will not work with a consumer graphics card in a Windows operating system environment. Still the potential for a large reduction in synchronisation time is there and certainly warrants further investigation. If cooperative group is not an option, the introduction of ghost points similar to practices used in the MPI protocol could be investigated to reduce synchronization for the explicit method.

Curvilinear structured and unstructured grids

As explained in chapter 2 it is possible to transform a rectangular grid as the one used in the implementation from this project into a different shape, such as a river or a lake, through a curvilinear transformation. This would enable comparison with a wider range of shallow water models and benchmark cases.
Similarly for domains where certain parts of the domain require a very high grid density a finite element implementation would be ideal.

Alternative wetting and drying models

As mentioned before the current wetting and drying approach is not suitable for large time steps, as the approach is inherently explicit. Alternative solutions to the problem exist, such as the one proposed by van't Hof and Vollebregt [74]. Successfully implementing an approach that is suitable for larger time steps would improve the versatility of the implicit implementation.

A

Bibliography

References

[1] Delft Hydraulics and Rijkswaterstaat, “Storm surge barrier eastern scheldt; evaluation of water movement studies for design and construction of the barrier,” Tech. Rep., 1989.

[2] E. D. De Goede, “Historical overview of 2D and 3D hydrodynamic modelling of shallow water flows in the Netherlands,” *Ocean Dynamics*, Jan 2020. [Online]. Available: https://doi.org/10.1007/s10236-019-01336-5

[3] A. B. d. Saint-Venant, “Théorie du mouvement non permanent des eaux, avec application aux crues des rivières et a l’introduction de marées dans leurs lits,” *Comptes Rendus de l’Académie des Sciences*, no. 73, pp. 147–154 and 237–240, 1871.

[4] D. J. Acheson, *Elementary fluid dynamics.* Oxford; New York: Clarendon Press ; Oxford University Press, 1990, oCLC: 20296032.

[5] C. B. Vreugdenhil, *Numerical Methods for Shallow-Water Flow*, ser. Water Science and Technology Library. Springer Netherlands, 1994. [Online]. Available: https://www.springer.com/gp/book/9780792331643

[6] L. D. Landau and E. M. Lifshitz, *Fluid Mechanics.* Elsevier, Aug. 1987, google-Books-ID: eVKbCgAAQBAJ.

[7] R. Manning, “On the flow of water in open channels and pipes.” *Transactions of the Institution of Civil Engineers of Ireland , Vol. XX*, pp. 161–207, 1891.

[8] A. Sommerfeld, *Partial Differential Equations in Physics.* New York: Academic Press, New York, 1949.

[9] R. Courant and D. Hilbert, *Methods of Mathematical Physics, Vol. 1*, 1st ed. Weinheim: Wiley-VCH, Jan. 1989.

[10] J. Oliger and A. Sundström, “Theoretical and Practical Aspects of Some Initial Boundary Value Problems in Fluid Dynamics,” *SIAM Journal on Applied Mathematics*, vol. 35, no. 3, pp. 419–446, Nov. 1978. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/0135035

[11] J. v. Kan, F. Vermolen, and A. Segal, *Numerical Methods in Scientific Computing*. Delft: VSSD, Mar. 2006.

[12] W. E. Hammond and N. M. F. Schreiber, “Mapping Unstructured Grid Problems to the Connection Machine,” 1992.

[13] A. Arakawa and V. R. Lamb, “Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model,” in *Methods in Computational Physics: Advances in Research and Applications*, ser. General Circulation Models of the Atmosphere, J. Chang, Ed. Elsevier, Jan. 1977, vol. 17, pp. 173–265. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780124608177500094

[14] H. Weller, “Numerics: The analysis and implementation of numerical methods for solving differential equations,” University of Reading, Tech. Rep. [Online]. Available: http://www.met.reading.ac.uk/~sws02hs/teaching/PDEsNumerics/PDEsNumerics_2_student.pdf

[15] H. P. Gunawan, “Numerical simulation of shallow water equations and related models,” Ph.D. dissertation, Universit Paris-Est, 2015.

[16] L. Euler, *Institutionum calculi integralis.* imp. Acad. imp. Saènt., 1769, google-Books-ID: cA8OAAAAQAAJ.

[17] J. C. Butcher, “Runge–Kutta Methods,” in *Numerical Methods for Ordinary Differential Equations.* John Wiley & Sons, Ltd, 2008, pp. 137–316. [Online]. Available: http://onlinelibrary.wiley.com/doi/abs/10.1002/9780470753767.ch3

[18] J. Crank and P. Nicolson, “A practical method for numerical evaluation of solutions of partial differential equations of the heat conduction type,” *Mathematical Proceedings of the Cambridge Philosophical Society, 43*, pp. 50–67, 1947.

[19] P. E. Aackermann, P. J. D. Pedersen, A. P. Engsig-Karup, T. Clausen, and J. Grooss, “Development of a GPU-Accelerated Mike 21 Solver for Water Wave Dynamics,” in *Facing the Multicore-Challenge III.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7686, pp. 129–130. [Online]. Available: http://link.springer.com/10.1007/978-3-642-35893-7_15

[20] J.-L. Lions, Y. Maday, and G. Turinici, “A “parareal” in time discretization of PDE’s,” *Comptes Rendus de l’Académie des Sciences. Série I. Mathématique*, vol. 332, Jan. 2001.

[21] G. S. Stelling and S. P. A. Duinmeijer, “A staggered conservative scheme for every Froude number in rapidly varied shallow water flows,” *International Journal for*

Numerical Methods in Fluids, vol. 43, no. 12, pp. 1329–1354, Dec. 2003. [Online]. Available: http://doi.wiley.com/10.1002/fld.537

[22] "General-purpose computing on graphics processing units," May 2019, page Version ID: 896005463. [Online]. Available: https://en.wikipedia.org/w/index.php?title=General-purpose_computing_on_graphics_processing_units&oldid=896005463

[23] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.

[24] "NVIDIA GeForce RTX 2080 Ti Specs." [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305

[25] K. Lemmens, "Introduction to Parallel Programming on the GPU," Delft Institute for Applied Mathematics, Tech. Rep., 2019.

[26] "Texture Memory in CUDA: What is Texture Memory in CUDA programming." [Online]. Available: http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

[27] S. W. Williams, "Auto-tuning Performance on Multicore Computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2008. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html

[28] "Roofline model," May 2019, page Version ID: 896109879. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Roofline_model&oldid=896109879

[29] "r/nvidia - The new Titan V has both the highest FP64-performance as the best FP64/price ratio on a Nvidia GPU ever." [Online]. Available: https://www.reddit.com/r/nvidia/comments/7iduuh/the_new_titan_v_has_both_the_highest/

[30] "NVIDIA Tesla V100 PCIe 16 GB Specs." [Online]. Available: https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-16-gb.c2957

[31] "NVIDIA GeForce GTX TITAN Specs." [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gtx-titan.c1996

[32] "NVIDIA GeForce GTX TITAN BLACK Specs." [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gtx-titan-black.c2549

[33] "Radeon Pro," May 2019, page Version ID: 897140975. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Radeon_Pro&oldid=897140975

[34] K. Freund, "Is NVIDIA Unstoppable In AI?" [Online]. Available: https://www.forbes.com/sites/moorinsights/2018/05/14/is-nvidia-unstoppable-in-ai/

[35] "Programming Tensor Cores in CUDA 9," Oct. 2017. [Online]. Available: https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

[36] N. Oh, “The NVIDIA Titan V Deep Learning Deep Dive: It’s All About The Tensor Cores.” [Online]. Available: https://www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive

[37] A.-K. Cheik Ahamed and F. Magoulès, “Conjugate gradient method with graphics processing unit acceleration: CUDA vs OpenCL,” *Advances in Engineering Software*, vol. 111, pp. 32–42, Sep. 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S096599781630477X

[38] “OpenACC Programming and Best Practices Guide,” p. 64. [Online]. Available: https://www.openacc.org/resources

[39] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, “CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application,” in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 136–143.

[40] NVIDIA/PGI, “Free Fortran, C, C++ Compilers & Tools for CPUs and GPUs.” [Online]. Available: https://www.pgroup.com/

[41] “CUDA,” May 2019, page Version ID: 897356499. [Online]. Available: https://en.wikipedia.org/w/index.php?title=CUDA&oldid=897356499

[42] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[43] K. He, S. X. Tan, H. Wang, and G. Shi, “GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1140–1150, Mar. 2016.

[44] A. Duffy, “Creating and Using a Red-Black Matrix,” 2010. [Online]. Available: http://computationalmathematics.org/topics/files/red_black.pdf

[45] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*. National Bureau of Standards, 1952. [Online]. Available: http://archive.org/details/jresv49n6p409

[46] J. R. Shewchuk, “An Introduction to the Conjugate Gradient Method Without the Agonizing Pain,” Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep., 1994.

[47] “Gradient descent,” May 2019, page Version ID: 897146259. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Gradient_descent&oldid=897146259

[48] A. van der Sluis and H. A. van der Vorst, “The rate of convergence of Conjugate Gradients,” *Numerische Mathematik*, vol. 48, no. 5, pp. 543–560, Sep. 1986. [Online]. Available: http://link.springer.com/10.1007/BF01389450

[49] H. van der Vorst, “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, Mar. 1992. [Online]. Available: https://epubs.siam.org/doi/10.1137/0913035

[50] Y. Saad and M. H. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, Jul. 1986. [Online]. Available: http://epubs.siam.org/doi/10.1137/0907058

[51] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial, 2nd Edition*, Jan. 2000.

[52] "Multigrid method," Apr. 2019, page Version ID: 893614381. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multigrid_method&oldid=893614381

[53] T. Washio and C. Oosterlee, "On the Use of Multigrid as a Preconditioner," *Ninth International Conference on Domain Decomposition Methods*, 1998. [Online]. Available: http://www.numerical.rl.ac.uk/reports/gsRAL95026.pdf

[54] P. Sanan, "linear algebra - How is Krylov-accelerated Multigrid (using MG as a preconditioner) motivated?" 2015. [Online]. Available: https://scicomp.stackexchange.com/questions/19786/how-is-krylov-accelerated-multigrid-using-mg-as-a-preconditioner-motivated

[55] R. D. Falgout, "An Algebraic Multigrid Tutorial," Institute for Mathematics and its Applications, Tech. Rep., 2010.

[56] N. I. M. Gould and J. A. Scott, "On approximate-inverse preconditioners," Computing and Information Systems Dpeartment, Rutherford Appleton Laboratory, Tech. Rep., 1995. [Online]. Available: http://www.numerical.rl.ac.uk/reports/gsRAL95026.pdf

[57] O. G. Johnson, C. A. Micchelli, and G. Paul, "Polynomial Preconditioners for Conjugate Gradient Calculations," *SIAM Journal on Numerical Analysis*, vol. 20, no. 2, pp. 362–376, 1983. [Online]. Available: http://www.jstor.org/stable/2157224

[58] M. van Gijzen, "A polynomial preconditioner for the GMRES algorithm," *Journal of Computational and Applied Mathematics*, vol. 59, no. 1, pp. 91–107, Apr. 1995. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/037704279400015S

[59] ——, "Iterative methods for linear systems of equations," Delft University of Technology, Tech. Rep., 2008.

[60] M. de Jong and C. Vuik, "GPU Implementation of the RRB-solver," *Reports of the Delft Institute of Applied Mathematics, issn 1389-6520, volume 16-06*, p. 53, 2016.

[61] M. de Jong, A. van der Ploeg, A. Ditzel, and C. Vuik, "Fine-grain parallel rrb-solver for 5-/9-point stencil problems suitable for gpu-type processors," *Electronic Transactions on Numerical Analysis*, vol. 46, pp. 375–393, 2017.

[62] *PARALUTION – Documentation.* [Online]. Available: https://www.paralution.com/documentation/

[63] "cuSPARSE." [Online]. Available: http://docs.nvidia.com/cuda/cusparse/index.html

[64] "AmgX," Nov. 2013. [Online]. Available: https://developer.nvidia.com/amgx

[65] "MAGMA: MAGMA Users' Guide." [Online]. Available: https://icl.cs.utk.edu/projectsfiles/magma/doxygen/

[66] A. Sielecki, *Mathematical Weather Rev.* U.S. Department of Agriculture, 1968, vol. 96.

[67] K. Lindenberg, K. Vuik, and P. W. J. van Hengel, "Stability analysis for numerical methods applied to an inner ear model," *Involve*, vol. 10, no. 2, pp. 181–196, 2017. [Online]. Available: https://doi.org/10.2140/involve.2017.10.181

[68] G. S. Stelling, "On the construction of computational methods for shallow water flow problems," Ph.D. dissertation, Delft University of Technology, 1983.

[69] L. Peeters, "Salt marsh modelling: implemented on a gpu," Master's thesis, Delft University of Technology, 2018. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Aab1242b2-72e9-4052-a7e8-bbe77a7a4d5a

[70] W. Hansen, *Theorie zur Errechnung des wasserstandes und der Strömingen in Randmeeeren nebst Anwendungen.* Tellus, 1956, vol. 96.

[71] R. Bridson and W.-P. Tang, "Refining an approximate inverse," *Journal of Computational and Applied Mathematics*, vol. 123, pp. 293–306, 2000.

[72] C. Lin and More, "Incomplete cholesky factorizations with limited memory," *SIAM Journal on Scientific and Statistical Computing*, pp. 24–45, 1999.

[73] "Cusp documentation." [Online]. Available: https://cusplibrary.github.io/classcusp_1_1precond_1_1scaled__bridson__ainv.html

[74] B. van't Hof and E. A. H. Vollebregt, "Modelling of wetting and drying of shallow water using artificial porosity," *International Journal for Numerical Methods in Fluids*, vol. 48, pp. 1199–1217, 2005.