DELFT UNIVERSITY OF TECHNOLOGY

Suitability of Shallow Water solving methods for GPU Acceleration

Literature Review

Author:
Floris Buwalda

Supervisor TU Delft:
Prof. dr. ir. C. Vuik
Supervisors Deltares:
E. de Goede
M. Pronk

May 19, 2019

CONTENTS

INTRODUCTION

Deltares is a research institute in the Netherlands that specializes applied research in the field of water and subsurface. This research includes building applications that can simulate shallow-water flow in a variety of environments. One such application is Delft3D-FLOW [1].

As the demands on scale and resolution of these simulation models increases, it becomes more attractive to consider doing the computations on a GPU, a Graphics Processing Unit as opposed to the traditional CPU, Central Processing Unit. The idea is that moving computations to the GPU will is beneficial to the large-scale viability of simulation models as GPUs are both more cost-efficient and energy-efficient compared to a CPU.

In 2017 a GPU computing initiative was started at Deltares by Maarten Pronk and Erik de Goede, and as a result a proposal was developed for exploring the possibility of GPU implementation of Shallow-Water models together with the department of computer science at the TU Delft as part of a master thesis project.

The aim of the project is to first develop a satisfactory GPU-accelerated explicit solving method and then aim to do the same for an implicit method. An explanation of explicit and implicit methods can be found in chapter 3. The main resulting research question is then:

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

In this literature review the theoretical background necessary for the development of these methods is presented. A number of literature research questions have been proposed:

1. What are the Shallow Water Equations and which form will be solved?
2. What discretization method exist and which is most suitable?
3. Which time integration methods exist and are suitable?
4. What GPU architecture aspects will need to be taken into consideration?
5. What linear solvers exist and are suitable for GPU implementation?
6. Is there a possible use of the new Nvidia Tensor cores for accelerating solver computations?

Chapters 1 through 5 will aim to provide the background necessary to answer the literature research questions, the conclusion of which will be presented in chapter 6.

1

THE SHALLOW WATER EQUATIONS

1.1. INTRODUCTION

The shallow water equations 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.
They 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 [2]. The Navier-Stokes equations are the full set of equations describing viscous fluid flow. These equations are hard to solve due to their inherent nonlinearity 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.

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 [3]:

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

Where $\mathbf{u}$ is a 3-dimensional flow velocity vector, ρ_0 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 [3] that $\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{u})$. Substituting this into 1.1 we obtain:

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

We know 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 also the continuity equation then become:

$$\frac{\partial(\mathbf{u})}{\partial t} + \nabla\cdot\left(\mathbf{u}\mathbf{u}^T\right) = \frac{1}{\rho_0}\left(-\nabla p + \nabla\cdot\tau\right) + g \tag{1.3}$$

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

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.

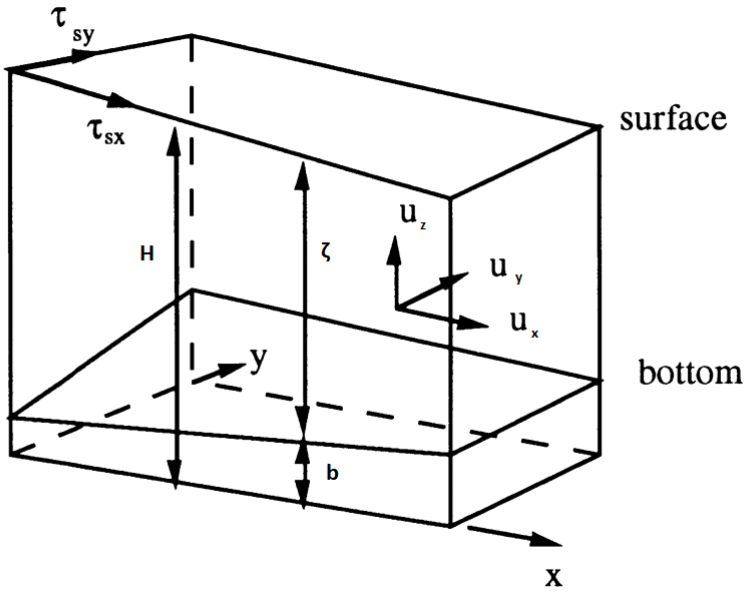


Figure 1.1: A schematic of the shallow water domain. $u_{x,y,z}$ are the flow velocities in their respective directions, source: [4]

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 $z = -b$ 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 free surface $z = \zeta$ 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 bottom 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 only look 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 regarding 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:

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

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

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

1.5 together with 1.6 are what we call 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 [5] 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 [5] as $\frac{\tau_{bx}}{\rho_0} = \frac{g u_x ||\mathbf{u}||}{C^2 H}$, where C is the Chézy coefficient [6]. 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 often quite 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 [4] 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 technically linear but second order derivatives also complicate things so it is neglected for now.

The bottom friction is approximated by a constant C that is proportial 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 has very little meaning 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.

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 [7] 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 [8], 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 [4].

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 [9] 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. [4] 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, and a derivative is a continuous limit. 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 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 integrated using Newton-Cotes integration [10]. 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 guarantee conservation 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.

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 subsection of the domain of computation, the grid can be refined only on that subsection and remain coarse on the rest of the domain which reduces computational complexity.

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.

2.5.1. 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 that there is no need for a connectivity matrix: simply all neighboring matrix entries are connected, 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 [10]. This solves the boundary problem but reformulation of the original problem into curvilinear coördinates is often nontrivial.

An illustration of curvilinear reformulation is given in figure 2.1.

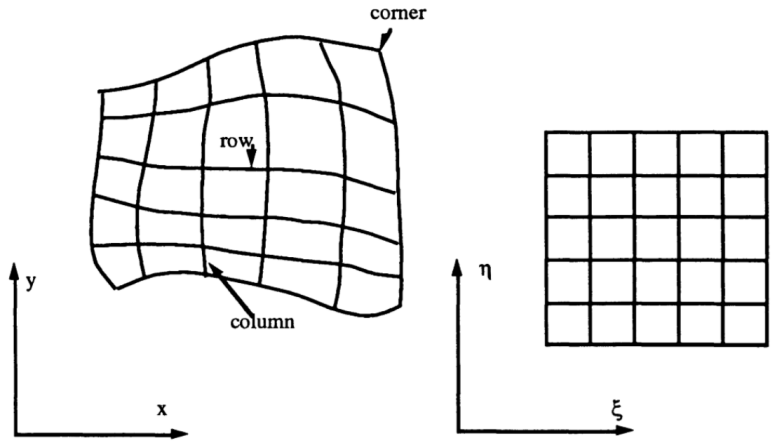


Figure 2.1: An illustration of boundary fitting a structured grid using curvilinear coordinates. Source: [4]

2.5.2. 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.

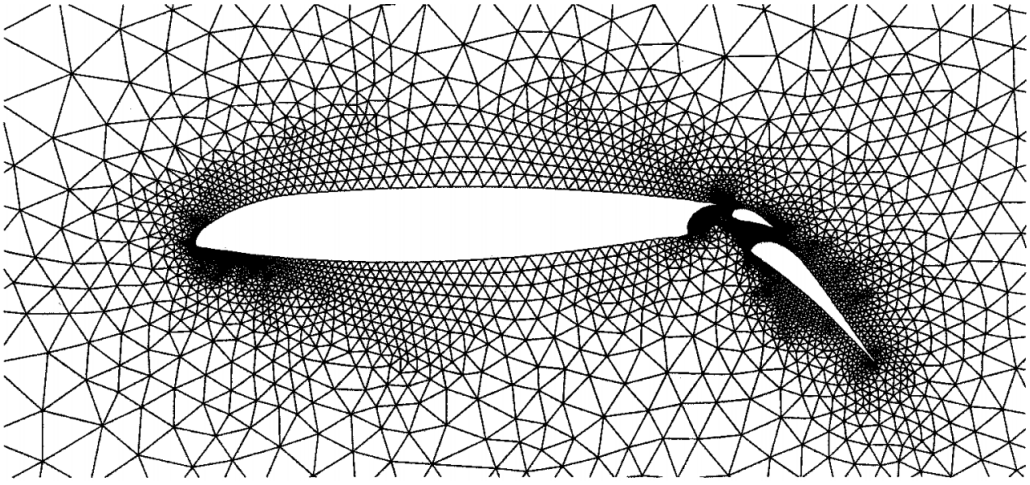


Figure 2.2: An example of an unstructured grid using triangular elements. Source: [11]

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 [10].

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 [12] proposed four different staggered grids. According to [13], 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 checkerboarded solutions, and allows for a larger grid size as variable density is reduced.

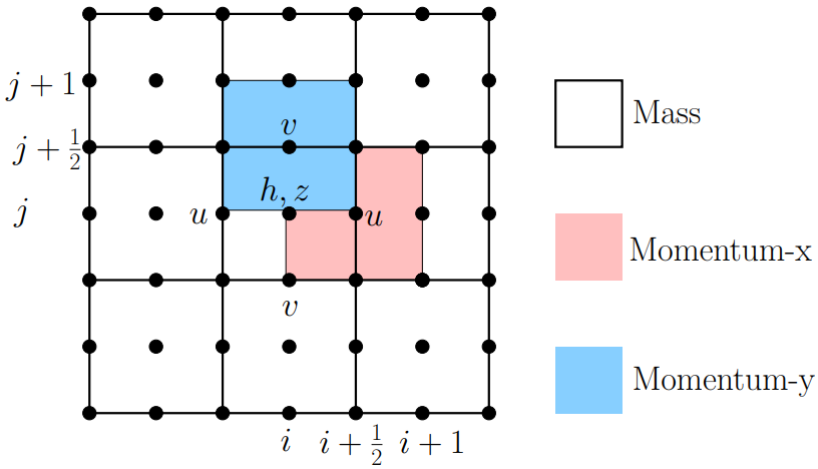


Figure 2.3: An illustration of the Arakawa C-grid. u, v are the flow velocity in the x, y directions respectively, h is the water depth and i, j are the node indices. The control volumes for the conserved variables are coloured white, pink and blue. Source: [14]

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:

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

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 dat defines the (partial) differential equation, and θ 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 combi-

nation 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}:

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

Euler [15] 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 explicit methods ideal for implementation on a GPU, which will be explained further in chapters 4 and 5.

3.2.1. STABILITY

When numerically integrating a hyperbolic partial differential equation, it is important to know when the method is stable or not.

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 grid distance. 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 [8].

$$C = \frac{\mathbf{u}_x dt}{dx} + \frac{\mathbf{u}_y dt}{dy} \leq C_{max} \tag{3.5}$$

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.

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 [16].

The CFL condition is a necessary but not sufficient condition for stability, however. 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. 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 purely on the function value $F(\phi_{n+1})$. This means that solving the equation is now not as straightforward as with equation 3.3. This method is also called the Euler backward method. The complexity is highly dependent on the form of the function F. The big advantage of implicit time integration is that it is unconditionally stable with respect to the size of the time step. Do note that this does not mean that the solution will be accurate for all time steps, however.

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 to which the solution can be obtained by inverting the matrix $\mathbf{A} + I$. This means that every time step a linear system of equations needs to

be solved which is computationally expensive. An advantage is that since the method is unconditionally stable often a larger time step can be used which can offset this.

Solving a system of linear equations is computationally expensive and not trivial to parallelize. Chapter 5 describes various solving methods and their discusses 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 [17]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 systems which are comparatively computationally cheap to solve.
Aackermann & Pedersen [18] 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 [16]. As mentioned before the Runge-Kutta 4 method has remainded 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, which John C. Butcher developed 60 years after the RK methods were developed. [16]:

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}
	b_1	b_2	$\cdots$	b_s

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 [16] 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 important concept in time integration is a method called Parareal [19]. When parallelizing the solving process of a partial differential equation, usually the system of equations that needs to be solved at every time step is parallelized. Parareal however, attempts to put the parallelization one level higher 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. NUMERICAL SCHEMES FOR THE SHALLOW WATER EQUATIONS

In chapter 1 the shallow water equations have been derived, in chapter 2 various discretization methods have been described and in this chapter various time integration methods have been described.
Now what remains is to use one of the discretization and time integration methods to construct a numerical scheme that solves the SWE.

3.6.1. STELLING & DUINMEIJER SECOND ORDER SCHEME

Stelling & Duinmeyer [20] developed a first order finite difference scheme for the shallow water equations that can be modified for second order accuracy.
They start with the non-conservative one 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

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

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 tems, 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 conserving approach, which is defined as $eh = \frac{u^2}{2g} + \zeta$ in one dimension.

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:

$$\frac{du_{i+1/2,j}}{dt} + \frac{\left(q_u^{-x}\right)_{i,j}}{h^{-x}_{i+1/2,j}} \frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} + \frac{\left(q_v^{-x}\right)_{i,j-1/2}}{h^{-x}_{i+1/2,j}} \frac{u_{i+1/2,j} - u_{i+1/2,j-1}}{\Delta y}$$
$$+ g\frac{\zeta_{i+1,j} - \zeta_{i,j}}{\Delta x} + c_f \frac{u_{i+1/2,j}||\mathbf{u}_{i+1/2,j}||}{h^{-x}_{i+1/2,j}} = 0 \tag{3.15}$$

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 $h^{-x}_{i+1/2,j} = (h_{i,j} + h_{i+1,j})/2$, with the y equation defined analogously.

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

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

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

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

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

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

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.

3.6.2. BAGHERI ET AL. FOURTH ORDER SCHEME

Bagheri et al. [21] have constructed an implicit finite difference scheme that is fourth order accurate on a rectangular grid.

The method starts with the non-conservative SWE including bottom friction, equal to equation 1.8. The equation can be written in the form:

$$\frac{\partial \mathbf{u}}{\partial t} + A\frac{\partial \mathbf{u}}{\partial x} + B\frac{\partial \mathbf{u}}{\partial y} = S(\mathbf{u}) \tag{3.20}$$

Where

$$\mathbf{u} = \begin{bmatrix} h \\ u_x \\ u_y \end{bmatrix} \quad A = \begin{bmatrix} 0 & 1 & 0 \\ -u_x^2 + gh & 2u_x & 0 \\ -u_x u_y & u_y & u_x \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 1 \\ -u_x u_y & u_y & u_x \\ -u_y^2 + gh & 0 & 2u_y \end{bmatrix} \quad S = \begin{bmatrix} 0 \\ -g\left(\frac{\partial z}{\partial x} + \frac{u_x||\mathbf{u}||}{C^2 h}\right) \\ -g\left(\frac{\partial z}{\partial y} + \frac{u_y||\mathbf{u}||}{C^2 h}\right) \end{bmatrix} \tag{3.21}$$

The idea now is to improve the accuracy of the scheme by using a second order finite difference approximation. Recalling the Taylor expansion from chapter 1 2.1 and substracting the forward and backwards approximations:

$$\begin{aligned} f(x+dx) &= f(x) + dx f'(x) + dx^2 f''(x)/2 + dx^3 f'''(x)/6 + O(dx^4) \\ f(x-dx) &= f(x) - dx f'(x) + dx^2 f''(x)/2 - dx^3 f'''(x)/6 + O(dx^4) \\ \frac{f(x+dx) - f(x-dx)}{dx} - f'(x) &= \frac{dx^2}{6} f'''(x) + O(dx^4) \end{aligned} \tag{3.22}$$

Bagheri proposes to substract an approximation of the third order derivatives $\frac{dx^2}{6}\left(A\frac{\partial^3 \mathbf{u}}{\partial x^3}\right)$ and $\frac{dy^2}{6}\left(B\frac{\partial^3 \mathbf{u}}{\partial y^3}\right)$ from 3.20 which themselves are approximated by differentiating 3.20.

Since the x and y differentiation procedure is analogous, we will restrict to the x direction for readability. Differentating with respect to x twice produces:

$$-\left(A\frac{\partial^3 \mathbf{u}}{\partial x^3}\right) = \frac{\partial^2 A}{\partial x^2}\frac{\partial \mathbf{u}}{\partial x} + 2\frac{\partial A}{\partial x}\frac{\partial^2 \mathbf{u}}{\partial x^2} + \frac{\partial^2 B}{\partial x^2}\frac{\partial \mathbf{u}}{\partial y} + 2\frac{\partial B}{\partial x}\frac{\partial^2 \mathbf{u}}{\partial x \partial y} + B\frac{\partial^3 \mathbf{u}}{\partial y \partial x^2} - \frac{\partial^2 \mathbf{S}(\mathbf{u})}{\partial x^2} \tag{3.23}$$

If we denote the discrete central difference operator as δ_n with $n = x, y$ and substitute into 3.20:

$$\begin{aligned}&\left[A\delta_x + B\delta_y\right]\mathbf{u} + \\ &\frac{dx^2}{6}\left[\left(\delta_x^2 A\delta_x + 2\delta_x A\delta_x^2 + \delta_x^2 B\delta_y + 2\delta_x B\delta_x\delta_y + B\delta_y\delta_x^2\right)\mathbf{u} + \delta_x^2\mathbf{S} + O(dx^2, dy^2)\right] + \\ &\frac{dy^2}{6}\left[\left(\delta_y^2 B\delta_y + 2\delta_y B\delta_y^2 + \delta_y^2 A\delta_x + 2\delta_y A\delta_y\delta_x + A\delta_x\delta_y^2\right)\mathbf{u} + \delta_y^2\mathbf{S} + O(dx^2, dy^2)\right] \\ &= \mathbf{S} + O(dx^4, dy^4)\end{aligned} \tag{3.24}$$

Bagheri proposes introducing several terms for readability:

$$\begin{aligned}C &= A + \frac{dx^2}{6}\delta_x^2 A + \frac{dy^2}{6}\delta_y^2 A \\ D &= B + \frac{dx^2}{6}\delta_x^2 B + \frac{dy^2}{6}\delta_y^2 B \\ E &= 2dx^2\delta_x B + 2dy^2\delta_y A \\ F &= \frac{dx^2}{3}\delta_x A \\ G &= \frac{dy^2}{3}\delta_y B \\ J &= 1 + \frac{dx^2\delta_x^2 + dy^2\delta_y^2}{6}\end{aligned} \tag{3.25}$$

Finally the time derivative is added to the source term:
$S(\mathbf{u}) = S(\mathbf{u}) - \frac{\partial \mathbf{u}}{\partial t}$ and $\frac{\partial \mathbf{u}}{\partial t}$ is discretised as $\delta_t^+ \mathbf{u}^n = \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{dt}$ where $\mathbf{u}^n$ is the value of the vector u at time $t = n$.
Adding all this into equation 3.24 and rearranging leads to:

$$\left[JI\delta_t^+ C\delta_x + D\delta_y + F\delta_x^2 + G\delta_y^2 + \frac{1}{6}\left(dy^2 A\delta_x\delta_y^2 + dx^2 B\delta_y\delta_x^2 + E\delta_x\delta_y\right)\right]\mathbf{u}^n = JS + O(dx^4, dy^4) \tag{3.26}$$

Now if we call $C\delta_x + D\delta_y + F\delta_x^2 + G\delta_y^2 + \frac{1}{6}\left(dy^2 A\delta_x\delta_y^2 + dx^2 B\delta_y\delta_x^2 + E\delta_x\delta_y\right) = K$ and introduce the variable μ that decides the implicit/explicit factor of the scheme we obtain:

$$JI\delta_t^+ \mathbf{u}^n + (1-\mu)K\mathbf{u}^n + \mu K\mathbf{u}^{n+1} = (1-\mu)S^n + \mu S^{n+1} + O(dt^p, dx^4, dy^4) \tag{3.27}$$

This equation can be written into matrix-vector form with the following procedure:

$$\sum_{k_1=-1}^{1}\sum_{k_2=-1}^{1} w_{i+k_1,j+k2}\mathbf{u}_{i+k_1,j+k_2}^{n+1} = \sum_{k_1=-1}^{1}\sum_{k_2=-1}^{1} w'_{i+k_1,j+k2}\mathbf{u}_{i+k_1,j+k_2}^{n} + 24dtJ\left[\mu S_{ij}^{n+1} + (1-\mu)S_{ij}^{n}\right] \tag{3.28}$$

with

$$w_{i+k_1,j+k_2} = p_{i+k_1,j+k_2} + q_{i+k1,j+k_2} \tag{3.29}$$

$$w'_{i+k_1,j+k_2} = (\mu-1)\frac{dt}{dx^2dy^2}p_{i+k_1,j+k_2} + q_{i+k1,j+k_2} \tag{3.30}$$

They can be represented by a 9-point stencil in the following way:

$$\mathbf{w} = \mu\frac{dt}{dx^2dy^2}\begin{bmatrix} p_{i-1,j-1} & p_{i-1,j} & p_{i-1,j+1} \\ p_{i,j-1} & p_{i,j} & p_{i,j+1} \\ p_{i+1,j-1} & p_{i+1,j} & p_{i+1,j+1} \end{bmatrix} + \begin{bmatrix} 0 & 4I & 0 \\ 4I & 8I & 4I \\ 0 & 4I & 0 \end{bmatrix} \tag{3.31}$$

and the elements of the p-matrix:

$$\begin{aligned}
p_{i-1,j-1} &= -2dxdy^2A_{ij} - 2dx^2dyB_{ij} + 6dxdyE_{ij} \\
p_{i,j-1} &= -12dx^2dyD_{ij} + 4dx^2dyB_{ij} + 24dx^2G_{ij} \\
p_{i+1,j-1} &= -2dxdy^2A_{ij} + 2dx^2dyB_{ij} - 6dxdyE_{ij} \\
\\
p_{i-1,j} &= -12dxdy^2C_{ij} + 4dxdy^2A_{ij} + 24dy^2F_{ij} \\
p_{i,j} &= -12dy^2F_{ij} - 12dx^2G_{ij} \\
p_{i+1,j} &= 12dxdy^2C_{ij} - 4dxdy^2A_{ij} + 24dy^2F_{ij} \\
\\
p_{i-1,j+1} &= -2dxdy^2A_{ij} + 2dx^2dyB_{ij} - 6dxdyE_{ij} \\
p_{i,j+1} &= 12dx^2dyD_{ij} - 4dx^2dyB_{ij} + 24dx^2G_{ij} \\
p_{i+1,j+1} &= 2dxdy^2A_{ij} + 2dx^2dyB_{ij} + 6dxdyE_{ij}
\end{aligned} \tag{3.32}$$

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 used exclusively for video game rendering.

Later people realised that the computing capabilities of a GPU could be harnessed for other uses, which they proceeded to do by rewriting their 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 problems 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.

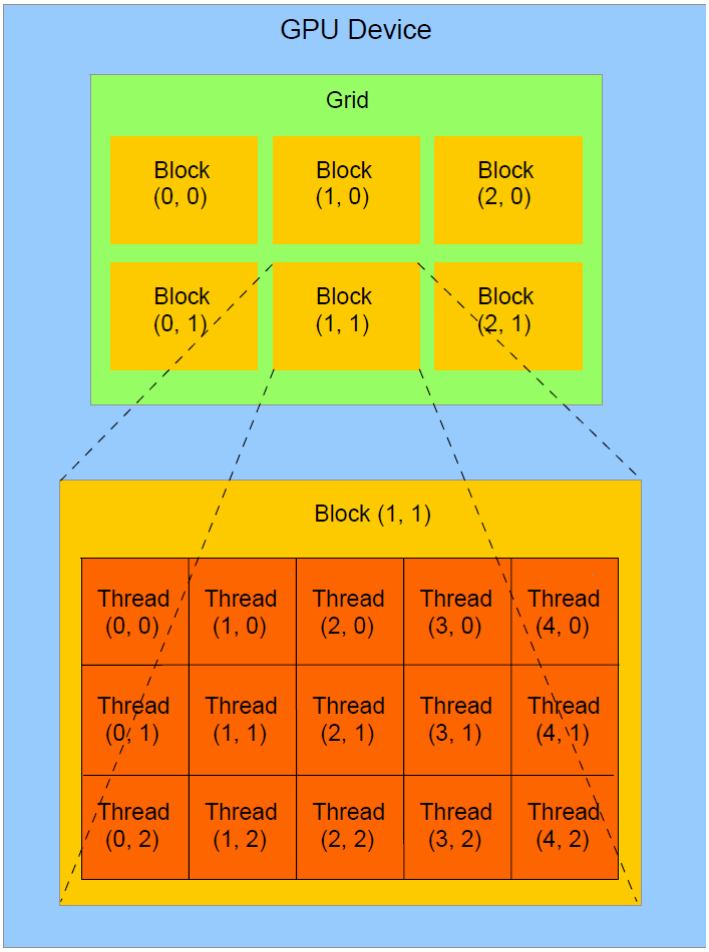


Figure 4.1: A schematic representation of GPU program structure Source: [25]

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 shall be 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.

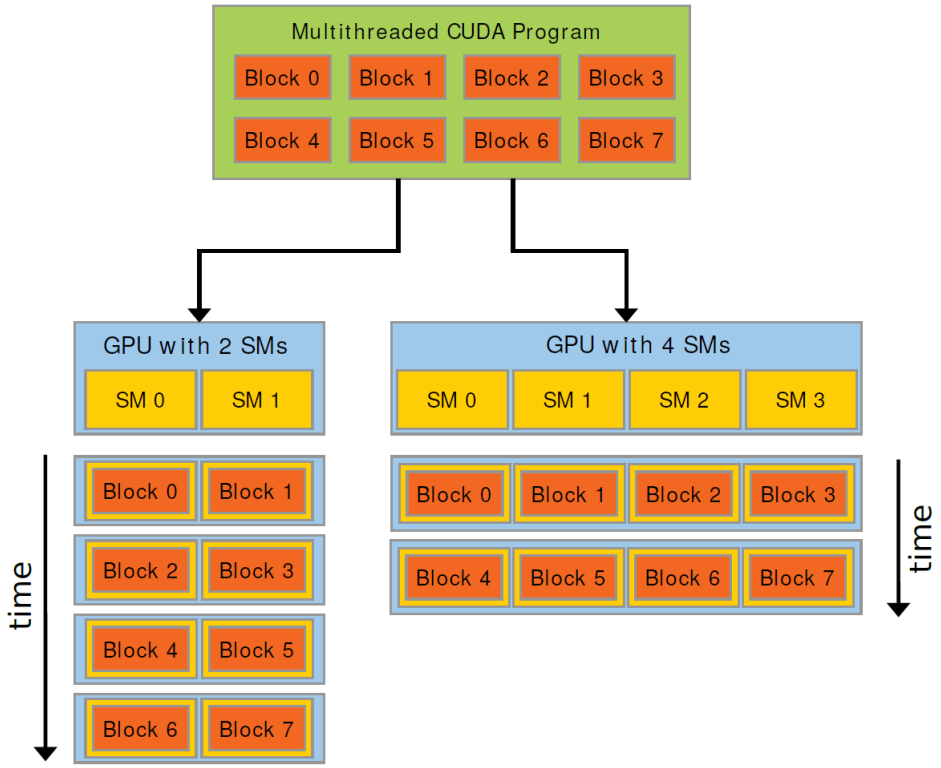


Figure 4.2: A schematic representation of GPU program execution structure Source: [25]

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

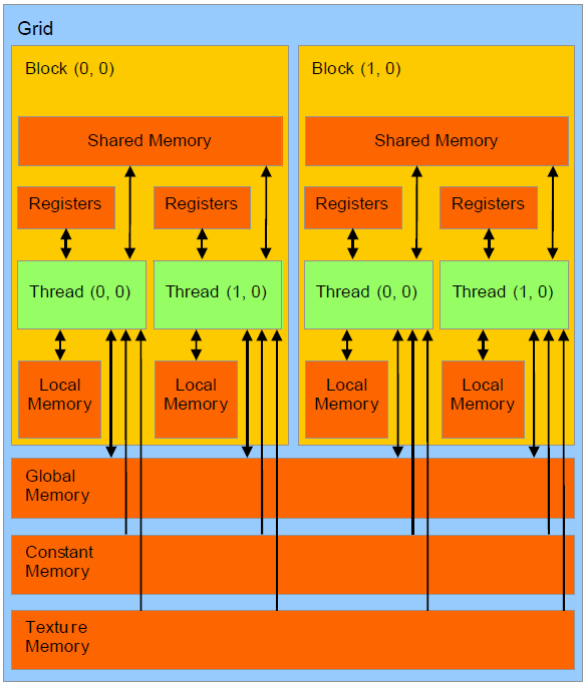


Figure 4.3: A schematic representation of GPU memory structure Source: [25]

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 local memory available to a thread larger than the register memory, but 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 benificial for -limited applications. Texture memory is optimized for spatial locality [26], which means that threads in the same warp acess 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 of 616 Gigabyte 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 .

4.3.2. LATENCY

Bandwidth limits the maximum data transfer rate through a memory bus. This figure however is only important when it is exceeded. When transferring data when not exceeding the bandwidth capacity there is still a delay between sending and receiving 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 per transfer operation.

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.

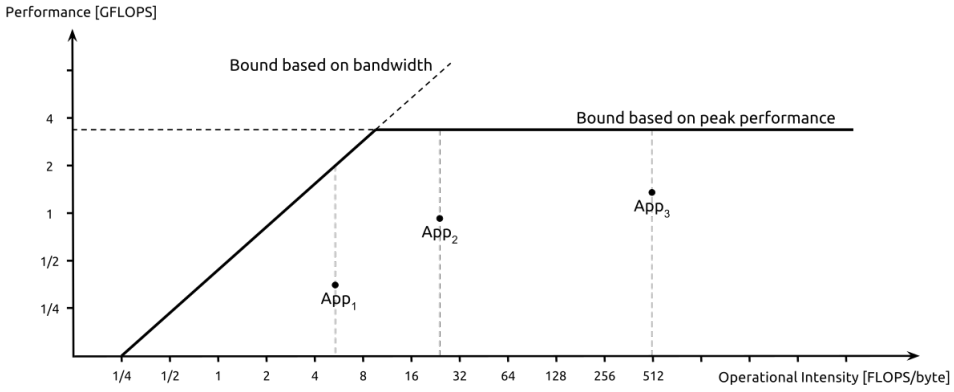


Figure 4.4: A schematic representation of the Roofline model,
maximum program Gflops vs program Operational intensity, where Operational Intensity is $\frac{W}{Q}$ Source: [28]

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 some-

thing 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.

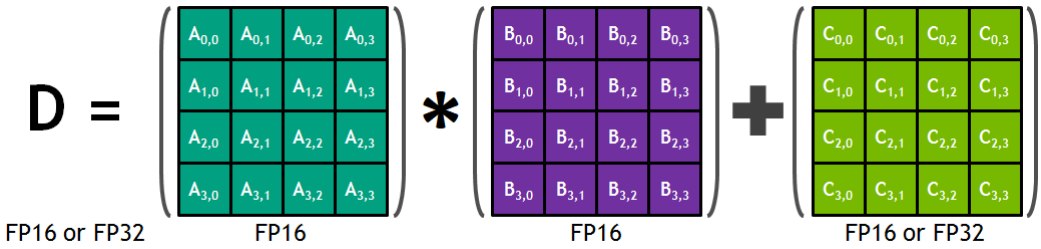
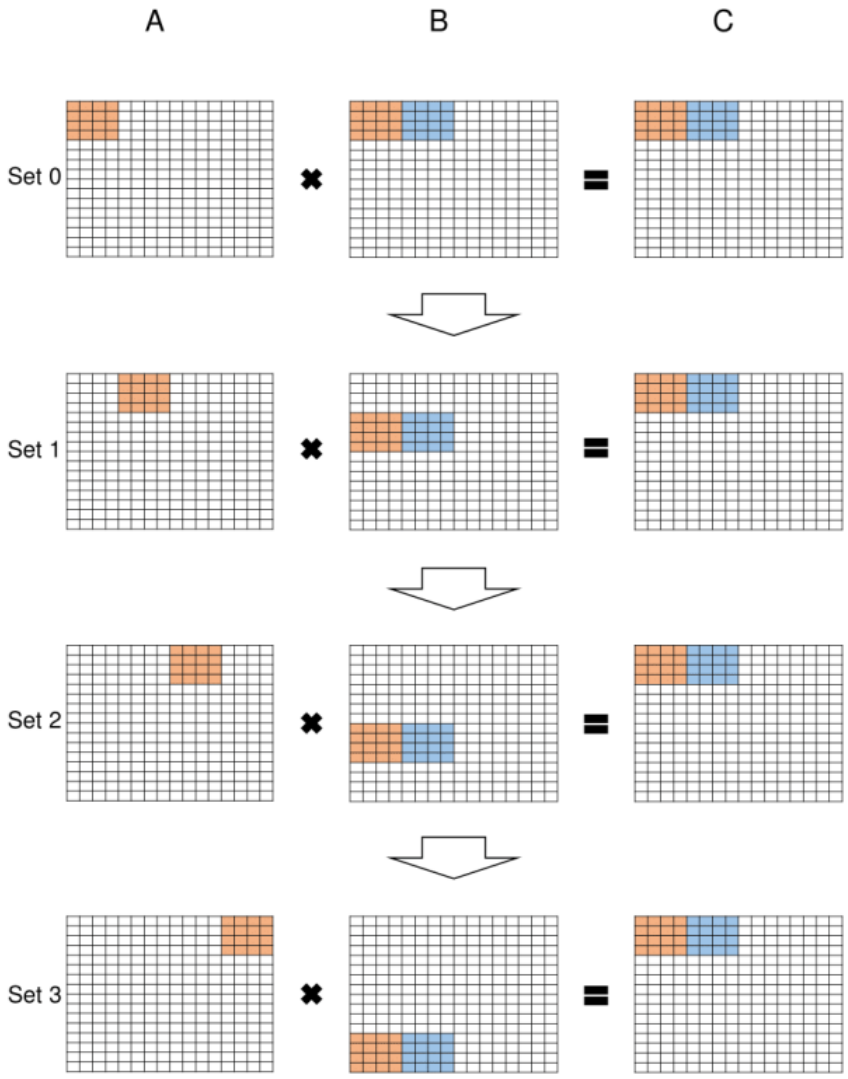


Figure 4.5: A schematic representation of a Tensor core operation Source: [35]

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:

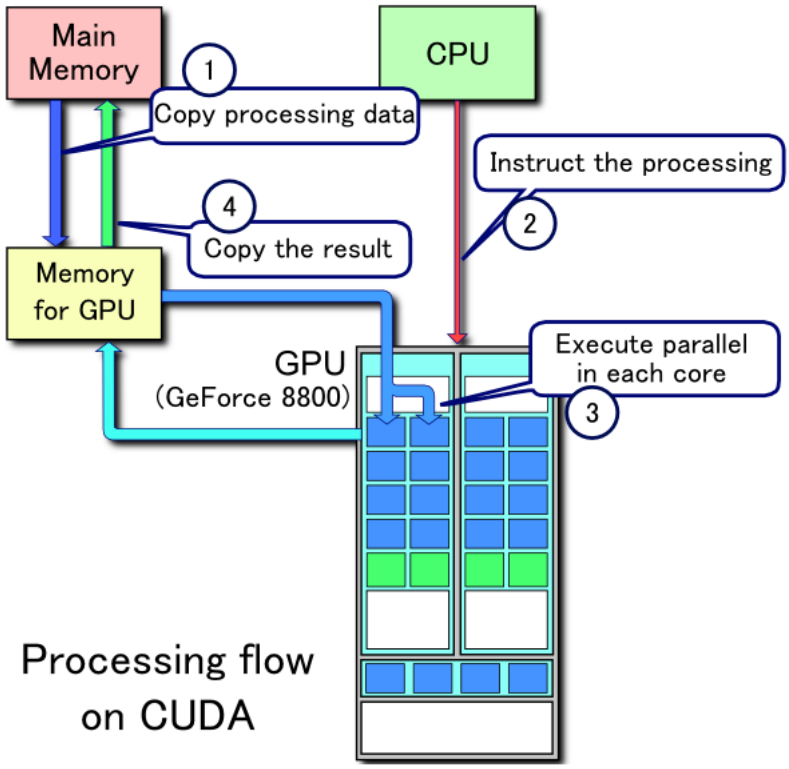


Figure 4.7: A schematic representation of CUDA processing flow Source: [41]

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. This 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

Often after constructing the matrix in a construction format it is 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.

5.3.1. TENSOR CORES FOR SPARSE MATRIX VECTOR MULTIPLICATION

As described in section 4.5 the TU102 can utilize Tensor cores for fast matrix-matrix multiplication and accumulation. On first glance they seem ill suited for matrix vector multiplication as the matrix sizes need to be an integer multiple of 4 in both dimensions.

However, the shallow water equations consist of three coupled equations which have to be solved in parallel. Since for an explicit method the equations are independent, the system could be formulated in such a way that an $N \times N \times 3 * N \times 1 \times 3$ operation must be done every time step, with N the total number of grid points.

If we then add a dummy row filled with zeroes to both systems, we obtain an $N \times N \times 4 * N \times 1 \times 4$ system, which can be computed in parallel using Tensor cores as it involves N parallel $N \times 4 * N \times 4$ multiplications, as long as N is a multiple of 4.

Adding the dummy row reduces computational efficiency by 33%, but this is compensated for by the fact that the TU102 GPU has an 8 fold increase in compute capability when performing operations using the Tensor cores.

The question remains whether the Tensor cores will be able to exploit the sparsity of the matrices.

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 5.1 and 5.2.
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 5.2.

Since the column levels are factorized sequentially it is important to distribute the work among levels as equally as possible, to ensure that a single level is the bottleneck while on other levels the majority the GPU is idling. This technique is also called load balancing. This is why in figure 5.2 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.

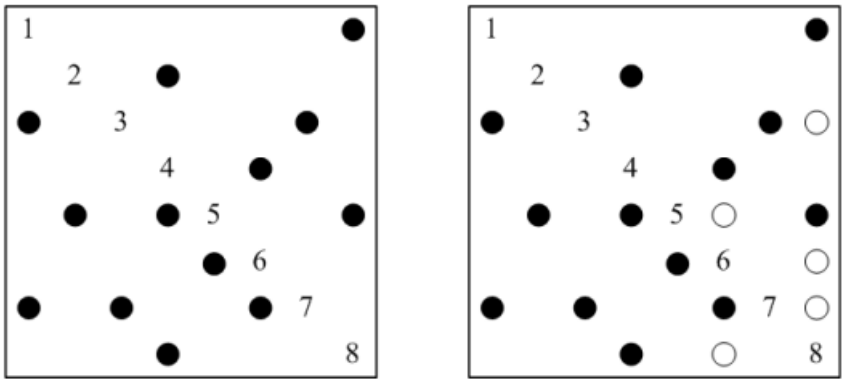


Figure 5.1: Representation of expected fill-in of a simple matrix with 8 rows and columns, where the white entries on the right are the predicted fill-in elements.
Source: [43]

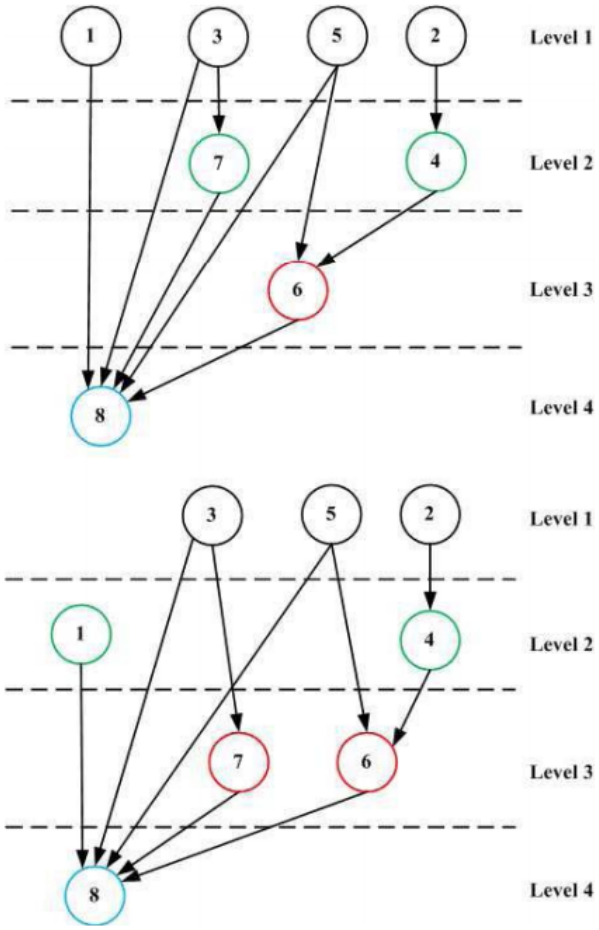


Figure 5.2: Levelization tree of the columns of the right matrix shown in figure 5.1.
The top figure shows naive column leveling, the bottom figure shows equalized column leveling.
Source: [43]

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

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

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 known. 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 then the error will increase. This is a vague statement, 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:

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

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

5.6.1. THE KRYLOV SUBSPACE

A second very important class of methods for solving linear systems of equations are 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:

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

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:

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

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:

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

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.

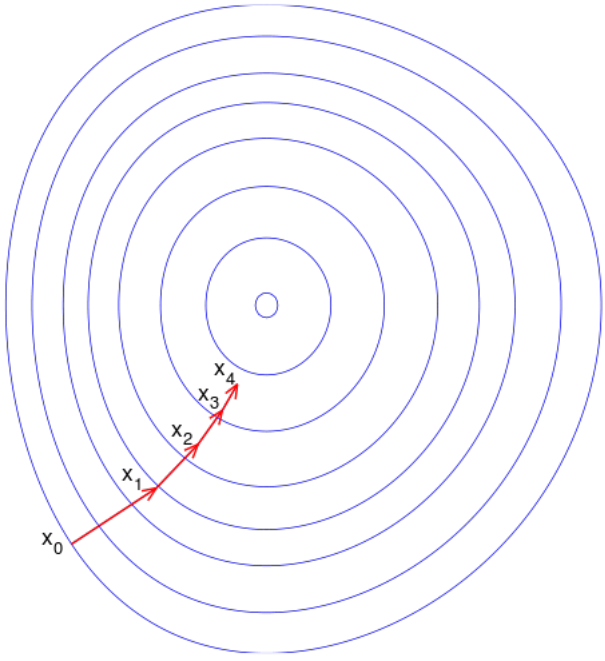


Figure 5.4: Illustration of successive steps of the Gradient Descent method
Source: [47]

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 func-

tion 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:

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

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

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

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:

$$\tilde{\mathbf{r}}_i = Q_i(A)\,\mathbf{r}_i = (I-\omega_1 A)\ldots(I-\omega_i A)(A)\,\mathbf{r}_i \tag{5.45}$$

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 Gram-Schmidt orthogonalization process that 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 problem.

Thus when preconditioning GMRES often agressive 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 has since then become quite popular due to their computational efficiency. Properly implemented Multigrid exhipbits 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.

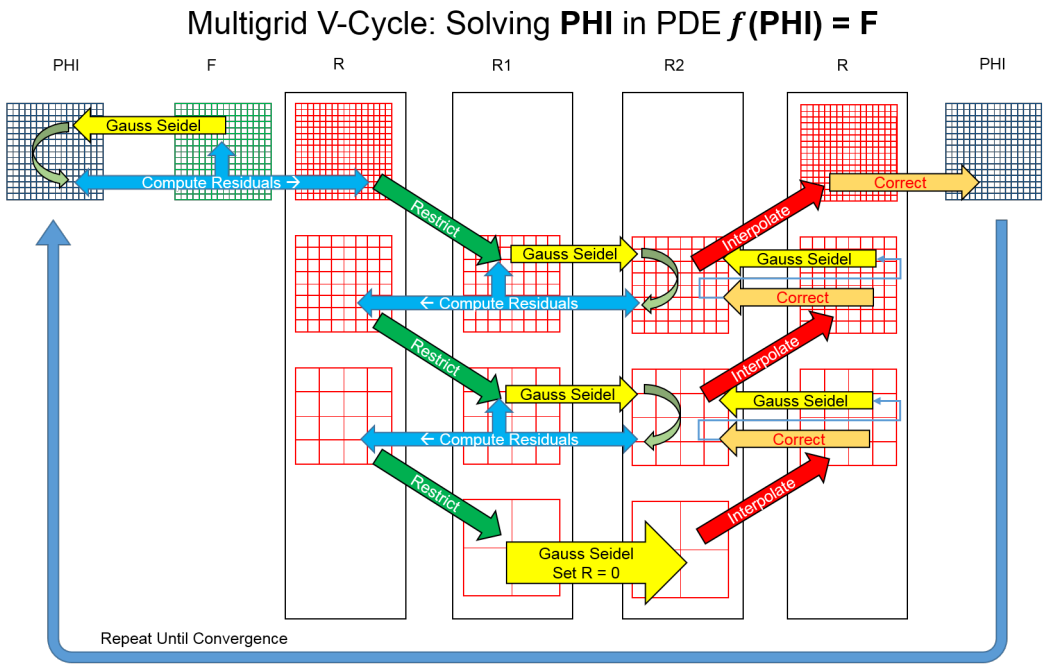


Figure 5.5: Illustration of an iteration of a V-cycle multigrid method
Source: [52]

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.

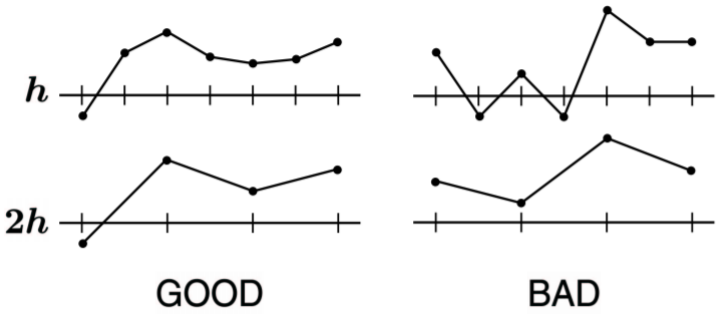


Figure 5.6: Illustration of coarsening error for a smooth function (left) and an oscillatory function (right) when moving from a fine grid with grid distance h to a coarse grid with grid distance $2h$.
Source: [51]

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 here 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 in 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 $M = PP^T$.

5

5.8.3. MULTIGRID AS A PRECONDITIONER

The multigrid algorithm is theoretically optimal 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 $\frac{N}{2} \times \frac{N}{2}$ 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

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

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, and then using the Chebychev or Richardson's iterative method [58] to construct the preconditioner. In practice this is done by starting with the normal 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

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

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 $q_{m+1}(y) = y p_m(y)$ 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.

The concept of domain decomposition usually involves defining block matrix that splits the domain of computation into (almost) independent blocks. 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.

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 independently.

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.

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 colouring scheme is for example the RRB-method, that can serve as a preconditioner for the CG method [60]. 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

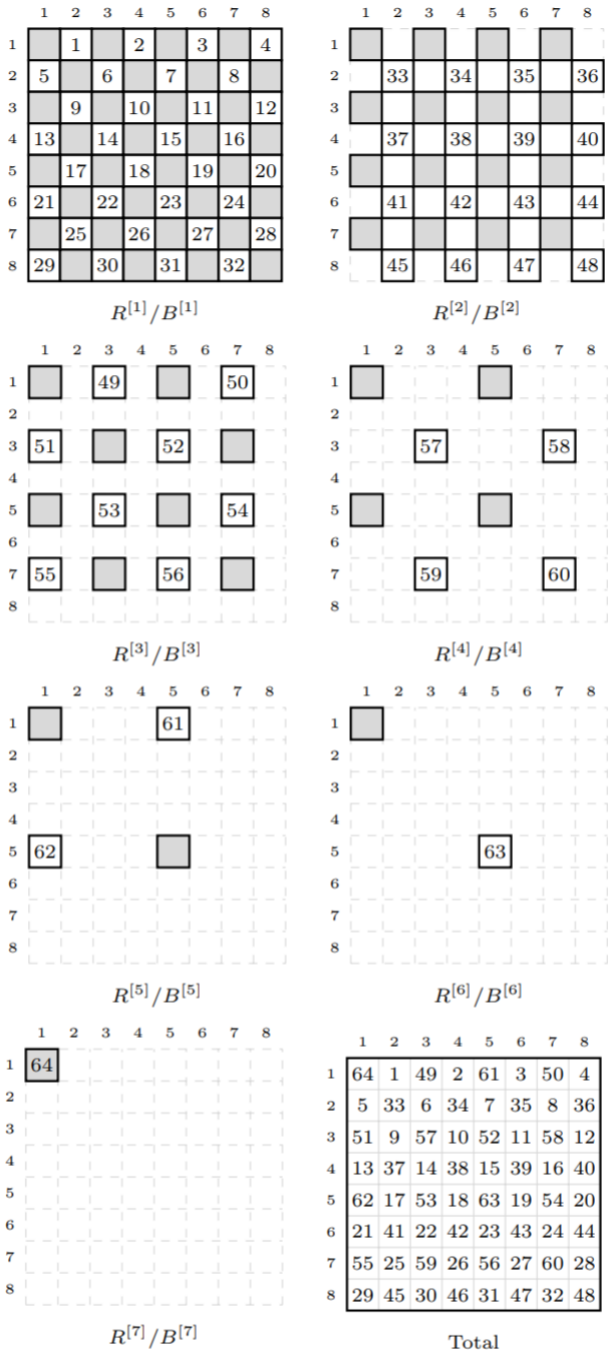


Figure 5.7: Illustration of the seven RRB numbering levels on an 8 × 8 grid.
Source: [60]

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. For this reason beginners to GPU programming are often warned to be prepared to chase after the state of the art performance and fail miserably.

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 commersial one. It has support for both OpenMP, OpenCL and CUDA, with plug-ins for FORTRAN, OpenFoam, MATLAB others [61]

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 high level packages 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 [62].

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 [63].

In addition it supports MPI and OpenMP which makes it very suitable for multi GPU systems, and as with Paralution the flexible implementation allows nexted 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 [64]

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

CONCLUSIONS

6.1. LITERATURE REVIEW CONCLUSIONS

WHAT ARE THE SHALLOW-WATER EQUATIONS AND WHICH FORM WILL BE SOLVED?

In the course of the project the linearised form chosen by Stelling & Duinmeijer [20] will be used as a governing system of equations, see also 1.9 and 3.19.

WHAT DISCRETIZATION METHOD EXIST AND WHICH IS MOST SUITABLE?

The main three methods to discretize a partial differential equation are Finite Volumes, Finite Differences and Finite Elements, all of which are described in chapter 3. As the first step in the project is to implement the Stelling & Duinmeijer scheme which is discretized by finite differences, this is the method of choice.

As a starting point the finite difference scheme will be implemented on a regular structured rectangular staggered Arakawa-C grid described in chapter 2.

WHICH TIME INTEGRATION METHODS EXIST AND ARE SUITABLE?

The first step in the project is to implement the Stelling & Duinmeijer scheme which uses the θ method with $\theta = 0.5$. As this results in an implicit system this can not be used as a starting point for an explicit solving method.
The explicit method will integrate of the same system of equations with θ set to 0 with the semi-implicit Euler method described in chapter 3.

WHAT GPU ARCHITECTURE ASPECTS WILL NEED TO BE TAKEN INTO CONSIDERATION?

In chapter 4 GPU architecture is described.
After review of the options with regards to device vendor and programming language the conclusion is that GPU calculations will be performed on an Nvidia TU-102 GPU, specifically a Nvidia 2080 Ti GPU. The program will be written in CUDA.

With regards to then design of GPU solvers most low-level methods will result in a memory bottleneck. Therefore the method should be designed to maximize coalesced memory access and use of shared memory. It was also found that single precision calculations will be up to 32 times faster on the chosen device.

WHAT LINEAR SOLVERS EXIST AND ARE SUITABLE FOR GPU IMPLEMENTATION?

An implicit time integration method will result in a linear system of equations that needs to be solved at every time step, as described in chapter 3. In chapter 5 an overview of possible linear system solvers has been given. It is proposed that after successful implementation of an explicit method on the GPU an implicit method will be implemented as well where the linear system is solved at every time step using the Preconditioned Conjugate Gradient method, which is very suitable for GPU implementation.

If this proves successful the aim is to then compare a number of preconditioners to find one suitable for acceleration of the Conjugate Gradient method.

To compare the viability of using solver software package solutions several will be tested as a linear solver for an implicit method.

IS THERE A POSSIBLE USE OF THE NEW NVIDIA TENSOR CORES FOR ACCELERATING SOLVER COMPUTATIONS?

As described in chapter 4, Tensor cores are extremely efficient at performing 4×4 matrix-matrix multiplications. Unfortunately most explicit and implicit methods avoid matrix-matrix multiplications as they are computationally very expensive. One possible candidate is the LU-factorization method described in 5 as it does involve them.

6.2. FURTHER RESEARCH SUBQUESTIONS

After the conclusion of the literature study it will be attempted to design and implement various shallow-water solvers as mentioned in the preceding section.
A number of additional research questions have been formulated that will be answered after the conclusion of the project.

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?

A

BIBLIOGRAPHY

REFERENCES

[1] "Delft3d-FLOW User Manual." [Online]. Available: https://oss.deltares.nl/web/delft3d/download

[2] 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.

[3] D. J. Acheson, *Elementary fluid dynamics*. Oxford; New York: Clarendon Press ; Oxford University Press, 1990, oCLC: 20296032.

[4] 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

[5] L. D. Landau and E. M. Lifshitz, *Fluid Mechanics*. Elsevier, Aug. 1987, google-Books-ID: eVKbCgAAQBAJ.

[6] 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.

[7] . Sommerfeld, *Partial Differential Equations in Physics*. New York: Academic Press, New York, 1949.

[8] R. Courant and D. Hilbert, *Methods of Mathematical Physics, Vol. 1*, 1st ed. Weinheim: Wiley-VCH, Jan. 1989.

[9] 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

[10] J. v. Kan, F. Vermolen, and A. Segal, *Numerical Methods in Scientific Computing*. Delft: VSSD, Mar. 2006.

[11] W. E. Hammond and N. M. F. Schreiber, "Mapping Unstructured Grid Problems to the Connection Machine," 1992.

[12] 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

[13] D. H. Weller, "Numerics: The analysis and implementation of numerical methods for solving differential equations," p. 49.

[14] H. P. Gunawan, "Numerical simulation of shallow water equations and related models," p. 167.

[15] L. Euler, *Institutionum calculi integralis*. imp. Acad. imp. Saènt., 1769, google-Books-ID: cA8OAAAAQAAJ.

[16] 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

[17] J. Crank and P. Nicolson, "A practical method for numerical evaluation of solutions of partial differential equations of the heat conduction type," pp. 50–67, 1947.

[18] 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*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. Keller, D. Kramer, and J.-P. Weiss, Eds. 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

[19] 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.

[20] 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

[21] J. Bagheri and S. Das, "Modeling of Shallow-Water Equations by Using Implicit Higher-Order Compact Scheme with Application to Dam-Break Problem," *Applied and computational mathematics*, vol. 2, Jan. 2013.

[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," 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," 2001.

[54] "linear algebra - How is Krylov-accelerated Multigrid (using MG as a preconditioner) motivated?" [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," p. 98.

[56] N. I. M. Gould and J. A. Scott, "On approximate-inverse preconditioners," p. 24.

[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] M. van Gijzen, "Iterative Methods for Linear Systems of Equations," Oct. 2008. [Online]. Available: http://ta.twi.tudelft.nl/nw/users/gijzen/CURSUS_DTU/LES1/TRANSPARANTEN/les1.pdf

[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.

[61] *PARALUTION – Documentation*. [Online]. Available: https://www.paralution.com/documentation/

[62] "cuSPARSE." [Online]. Available: http://docs.nvidia.com/cuda/cusparse/index.html

[63] "AmgX," Nov. 2013. [Online]. Available: https://developer.nvidia.com/amgx

[64] "MAGMA: MAGMA Users' Guide." [Online]. Available: https://icl.cs.utk.edu/projectsfiles/magma/doxygen/