Acceleration of a non-linear water wave model using a GPU

Morten Gorm Madsen

Kongens Lyngby
November 2010
IMM-MASTER

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MASTER

Preface

This thesis is submitted as a partial fulfillment of the requirements of the Danish M.Sc. degree at the Technical University of Denmark (DTU). The work has been carried out at the Department of Informatics and Mathematical Modelling with supervision from Assistant Professor Allan Peter Engsig-Karup, Associate Professor Bernd Dammann and Assistant Professor Jeppe Revall Frisvad.

Summary

Acceleration of a non-linear water wave model using a GPU

The primary objective of this work is to use a GPU (massively parallel hardware) to accelerate an existing optimized sequential algorithm, solving a potential flow problem. The potential flow problem poses an initial value problem at a 2D surface, coupled with a 3D Laplace problem. A low storage Defect Correction method with a multigrid preconditioner is used to solve a flexible order approximation of the Laplace problem. The widely used explicit RK4 method is applied for time integration.
The primary reason for porting this particular solver, is that both Defect Correction and the preconditioner are expected to be well suited for GPUs, given that the right discretization is used. The work focuses on both analysis and implementation of the multigrid method, and understanding how it should be configured in order to be an efficient preconditioner for the Defect Correction algorithm. Only little attention is given to the standard 4 stage Runge Kutta method.
The most significant results of the work is that rethinking the memory layout both provides a significant increment in problem size and gives a boost to the solution time, even for a naive CUDA implementation. In particular the program developed can hold a Laplace problem of up to 100,000,000 degrees of freedom in 4GB RAM. For problems of this size, the iterative solution to the Laplace problem is improved by a decimal within a matter of seconds. This is up to 10 times faster than the existing CPU implementation. Although the target platform is the Compute Capability 1.3 Tesla architecture, it is also shown that moving the program to a Fermi architecture GPU, accelerates the code even further with a resulting speedup of up to 42 times faster than the existing CPU

code. Remarkably the speedup on the Fermi-architecture is achieved with the naive implementation of the program.

Resumé (danish)

Acceleration af en ikke-lineær vandbølgemodel ved brug af en GPU

Hovedmålet med dette arbejde er at bruge en GPU (massivt parallel hardware) til at accelerere en eksisterende optimeret sekventiel algoritme til løsning af et potentialflowproblem. Problemet er et begyndelsesværdiproblem for en 2D overflade og koblet med et 3D Laplace-problem. Til at løse Laplace-problemet til variabel spatial orden, bruges Defect Correction-metoden med en multigrid preconditioner, som begge har lavt hukommelsesforbrug. Til integration i tid bruges standard RK4.
Den primære grund til at portere netop denne løser er, at både Defect Correction og preconditioneren forventes at være passe godt til GPU'er, givet den rette diskretiseringsmetode bruges. Arbejdet fokuserer på både analyse og implementation af multigridmetoden samt forståelse for hvordan den skal konfigureres for at være en effektiv preconditioner til Defect Correction-algoritmen. Der arbejdes kun i begrænset omfang med Runge Kutta-metoden.
De vigtigste resultater af dette arbejde er at en revurdering af hukommelseslayoutet tillader en signifikant forøgelse af problemstørrelsen og et boost af løsningstiden, selv for en naiv CUDA-implementation. Mere specifikt, kan det udviklede program arbejde med et Laplaceproblem med op til 100.000.000 frihedsgrader i 4GB RAM. For problemer af denne størrelse, forbedres den iterative løsning til Laplaceproblemet med et decimal på blot sekunder. Dette er op til 10 gange hurtigere end for en lignende eksisterende CPU-implementation. Selvom udviklen har været målrettet Tesla-arkitekturen (Compute Capability 1.3), har det vist sig, at afvikling af programmet på et Fermi-kort er op til 42 gange hurtigere end den eksisterende CPU-kode. Forbedringen på Fermi-kortet er opnået alene med den naive implementation af programmet.

Declarations

The work described in this thesis has been published at the CUDA seminar, held by Glass and Time, Roskilde University, November 24^{th} 2010.

Contents

Chapter 1

Introduction

During the two last decades, computer graphics hardware has developed from a fixed pipeline processor with no level of programmability to a flexible high performance hardware platform which can be used for purposes other than computer graphics. Specially the CUDA (Compute Unified Device Architecture) programming model by NVIDIA has become popular in the high performance computing community due to its C/C++-like language and that it is easy to deploy even to existing programs. The difference between GPUs and CPUs is that the GPU is a massively parallel piece of hardware which is highly specialized for high throughput applications. Although using GPUs for high performance purposes is a relatively new field, many examples of applications already exist [7].

The goal of this work is to implement a GPU version of a solver for a potential flow problem previously described by e.g. [1], formally presented in section 1.1. The focus is primary on analysis of a multigrid solver as an efficient low memory preconditioner for a Defect Correction algorithm. The Defect Correction algorithm presented in [12] with a multigrid preconditioner is shown by A. Engsig-Karup in [5] to be an efficient method for solving a potential flow problem. In particular, the algorithm is shown to have $O(n)$ scalability properties for both memory usage and solution time which makes it suitable also for large scale problems. In chapter 6 it is shown that problem sizes of up to 100 million degrees of freedom can be processed by a device with 4GB RAM.

For a discussion of the field of application of the fluid potential flow problem we will refer to [6, 1].

1.1 Potential flow formulation

A Cartesian coordinate system is used with the xy-plane located at the surface still water level and the z-axis pointing upwards. The still water depth is given as $h(x, y)$ where x and y are the 2D spatial coordinates. The position of the free surface is given as a single-valued function $\eta(x, y, t)$ with t being the current time. Since η is single valued, the model does not allow for overturning waves. An inviscid fluid with irrational flow can be described by its velocity potential/potential flow which is here denoted $\tilde{\phi}$ at surface level and ϕ in the fluid. For $\boldsymbol{u}$ being flow velocity, the velocity potential is given by $\boldsymbol{u} = \nabla\phi$. The governing equations are then given by

$$\partial_t \eta = -\nabla\eta \cdot \nabla\tilde{\phi} + \tilde{w}(1 + \nabla\eta \cdot \nabla\eta) \tag{1.1}$$

$$\partial_t \tilde{\phi} = -g\eta - \frac{1}{2}(\nabla\tilde{\phi} \cdot \nabla\tilde{\phi} - \tilde{w}^2(1 + \nabla\eta \cdot \nabla\eta)) \tag{1.2}$$

$$\tilde{w} = \partial_z \tilde{\phi}, \quad \tilde{\phi} = \phi|_{z=\eta} \tag{1.3}$$

$$\phi = \tilde{\phi}, \quad z = \eta \tag{1.4}$$

$$\nabla^2\phi + \partial_{zz}\phi = 0, \quad -h \leq z < \eta \tag{1.5}$$

$$\partial_z \phi + \nabla h \cdot \nabla\phi = 0, \quad z = -h \tag{1.6}$$

In order to not confuse ∇ of the surface equations (1.1) and (1.2) with a 3D gradient, it is consistently used as the 2D gradient, also in the Laplace equation.

$$\nabla \equiv [\partial_x \;\; \partial_y]^T \Rightarrow \nabla^2 \equiv \partial_x^2 + \partial_y^2$$

In order to evaluate η and surface velocity potential $\tilde{\phi}$ over time, the vertical gradient of the fluid velocity potential $\tilde{w}$ is needed. In order to calculate the vertical gradient $\tilde{w}$ of the velocity potential, the velocity potential needs to be available for the entire domain. The velocity potential must conform to the Laplace equation (1.5) and $\tilde{w}$ can thus be found when the solution to the Laplace problem is known. The solution to the Laplace equation will depend on the surface velocity potential $\tilde{\phi}$ and the water depth h. The computational domain of the Laplace equation is 'where the water is' as illustrated in fig. 1.1 and therefore varies over time which is rather inconvenient for a numerical approach to solve the Laplace problem. By transformation of the vertical variable, a problem mathematical equivalent to the Laplace equation can be set up such

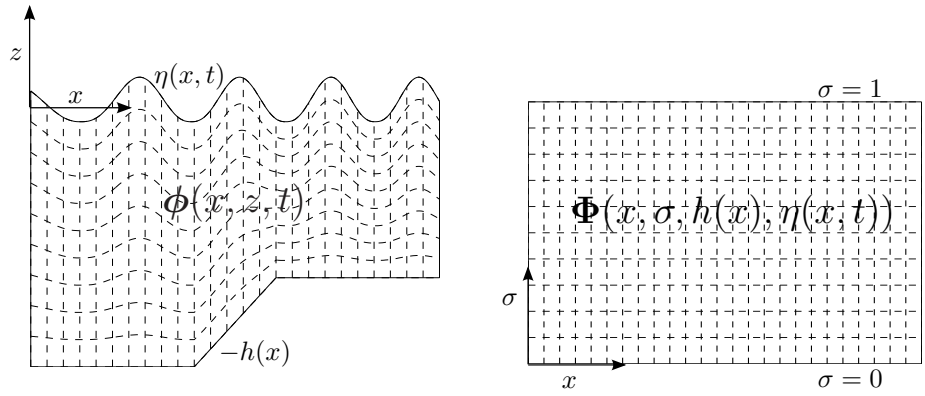

Figure 1.1: The Laplace problem is transformed from the physical domain (left) to a regular time invariant domain (right). Transformation give a better numeric approach although transformed problem is rather unintuitive.

that the domain is regular and independent of time. The linear mapping of the vertical variable is given by

$$\sigma \equiv \frac{z + \eta(x,y,t)}{d}, \quad d = \eta(x,y,t) + h(x,y) \tag{1.7}$$

The cost is that the Laplace equation expands into a large and somewhat unintuitive form. The derivations of the transformed Laplace equation primarily involve use of the chain rule. Even though only one variable is changed, the calculations are rather lengthy and full length derivations are therefore located in appendix A. The Laplace problem in the transformed domain becomes[1]

$$\Phi = \tilde{\phi}, \qquad \sigma = 1 \tag{1.8}$$

$$\nabla^2\Phi + \nabla^2\sigma\left(\partial_z\Phi\right) + 2\nabla\sigma\cdot\nabla\left(\partial_\sigma\Phi\right) + \left(\nabla\sigma\cdot\nabla\sigma + \left(\partial_z\sigma\right)^2\right)\partial_{\sigma\sigma}\Phi = 0, 0 \leq \sigma < 1 \tag{1.9}$$

$$\left(\partial_z\sigma + \nabla h\cdot\nabla\sigma\right)\left(\partial_z\Phi\right) + \nabla h\cdot\nabla\Phi = 0, \qquad \sigma = 0 \tag{1.10}$$

[1]Later use in figure: By transforming the vertical variable the grid is stretched and squeezed into a cuboid time invariant domain.

Here $\Phi(\boldsymbol{x}, \sigma, t) = \phi(\boldsymbol{x}, z, t)$. The derivatives of σ are found as well.

$$\partial_z \sigma = \frac{1}{d} \tag{1.11}$$

$$\nabla \sigma = (1-\sigma)\frac{\nabla h}{d} - \sigma \frac{\nabla \eta}{d} \tag{1.12}$$

$$\nabla^2 \sigma = \frac{1-\sigma}{d}\left(\nabla^2 h - \frac{\nabla h \cdot \nabla h}{d}\right) - \frac{\sigma}{d}\left(\nabla^2 \eta - \frac{\nabla \eta \cdot \nabla \eta}{d}\right) \tag{1.13}$$

$$-\frac{1-2\sigma}{d^2}\nabla h \cdot \nabla \eta - \frac{\nabla \sigma}{d} \cdot (\nabla h - \nabla \eta) \tag{1.14}$$

The relation from (1.1) and (1.2) to the Laplace equation is through $\tilde{w}$. For the transformed Laplace equation, the relation is found using the chain rule.

$$\tilde{w} = \partial_z \phi|_{z=\eta} = \partial_z \Phi|_{\sigma=1} = \partial_\sigma \Phi \partial_z \sigma|_{\sigma=1} \tag{1.15}$$

The problem (differential equation and transformed Laplace equation) posed above is considered as 'the full' problem.

For small amplitude waves ($\eta \ll h$) and flat bottom, all derivatives in η and h vanishes from the surface equations (1.1) and (1.2). Since $\eta \ll h$, there is also almost no difference from water depth to still water depth. In particular $d = h$ can be assumed. Given that also $\tilde{w}^2$ vanishes, the surface equation take linear form:

$$\partial_t \eta = \tilde{w} \tag{1.16}$$

$$\partial_t \tilde{\phi} = -g\eta \tag{1.17}$$

And for the transformed Laplace equation

$$\Phi = \tilde{\phi}, \quad \sigma = 1 \tag{1.18}$$

$$\nabla^2 \Phi + (\partial_z \sigma)^2 \partial_{\sigma\sigma} \Phi = 0, \quad 0 \leq \sigma < 1 \tag{1.19}$$

$$(\partial_\sigma \Phi)\, \partial_z \sigma = 0, \quad \sigma = 0 \tag{1.20}$$

$$\partial_z \sigma = d^{-1}, \quad d = h \tag{1.21}$$

There exist no general analytic solution for the full problem although solutions to certain cases are available.

1.1.1 Initial value problem

Evolving surface elevation and potential is an initial value problem and evaluation of (1.1) and (1.2) require a solution to the Laplace problem. Given a way

to evaluate (1.1) and (1.2) at a certain time step, the remaining initial value problem (IVP) needs to be solved. For the solver to the IVP, two objectives will be in question: Namely stability requirements and desired order of accuracy. For now, the stability requirements of the solver will be analyzed.

First an analysis of linearly coupled differential equations. In general linear differential equations are given by

$$\partial_t \boldsymbol{x} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} \tag{1.22}$$

By the introduction of $\boldsymbol{y} = \boldsymbol{S}\boldsymbol{x}$, where $\boldsymbol{S} = [\boldsymbol{s}_1\ \boldsymbol{s}_2\ ...\ \boldsymbol{s}_n]$, the system can be rewritten into a system of linearly independent ODEs:

$$\boldsymbol{S}\partial_t \boldsymbol{y} = \boldsymbol{A}\boldsymbol{S}\boldsymbol{y} + \boldsymbol{b} \tag{1.23}$$

$$\partial_t \boldsymbol{y} = \boldsymbol{S}^{-1}\boldsymbol{A}\boldsymbol{S}\boldsymbol{y} + \boldsymbol{S}^{-1}\boldsymbol{b} \tag{1.24}$$

$$\partial_t \boldsymbol{y} = \boldsymbol{D}\boldsymbol{y} + \tilde{\boldsymbol{b}}, \quad \boldsymbol{D} = diag(\lambda_1, \lambda_2, ...\lambda_n) \tag{1.25}$$

Here $\lambda_1, \lambda_2, ..., \lambda_n$ are the eigenvalues associated with the eigenvectors $\boldsymbol{s}_1, \boldsymbol{s}_2, ..., \boldsymbol{s}_n$. All of the differential equations in the ODE presented by (1.22) are independent since $\boldsymbol{D}$ is a diagonal matrix. The general solution to the system is therefore given by

$$y_k = \left(c_k + \int e^{-\lambda_k t}\tilde{b}_k \, \mathrm{d}t\right) e^{\lambda_k t}$$

where c_k is the k^{th} integration constant. Introducing the matrix exponential[2], the solution can be written in matrix form

$$\boldsymbol{y} = \left(\boldsymbol{c} + \int e^{-\boldsymbol{D}t}\tilde{\boldsymbol{b}} \, \mathrm{d}t\right) e^{\boldsymbol{D}t}$$

Substituting $\boldsymbol{y} = \boldsymbol{S}\boldsymbol{x}$ back into the equation reveal

$$\boldsymbol{x} = \boldsymbol{S}^{-1}\left(\boldsymbol{c} + \int e^{-\boldsymbol{D}t}\tilde{\boldsymbol{b}} \, \mathrm{d}t\right) e^{\boldsymbol{D}t} \tag{1.26}$$

For homogeneous systems ($\boldsymbol{b} = \boldsymbol{0}$), the integral simply vanishes, leaving

$$\boldsymbol{x} = \boldsymbol{S}^{-1}\left(\boldsymbol{c} + e^{\boldsymbol{D}t}\right) \tag{1.27}$$

What is important to notice here, is that the eigenvalues, represented by $\boldsymbol{D}$, determine the solution. Judging from the solution (1.27) to the homogeneous linear ODE, eigenvalues with positive real parts imply that the magnitude of the solution will grow over time. For negative real parts, the magnitude will be dampened over time, and finally for an absent real part, the result will keep

[2] $e^{\mathbf{X}} = \sum_{k=0}^{\infty} 1/k! \mathbf{X}^k$. For $\boldsymbol{A} = diag(a_{1,1}, a_{2,2}, ..., a_{n,n}) \Rightarrow e^{\boldsymbol{A}} = diag(e^{a_{1,1}}, e^{a_{2,2}}, ..., e^{a_{n,n}})$

the magnitude of its initial condition. The imaginary part of the eigenvalues determine whether the system is oscillatory or not; for no imaginary part, the system will not oscillate. For imaginary parts different from 0, the system will oscillate with an angular velocity corresponding to the size of the imaginary part of the eigenvalue.

In [6], A. Engsig-Karup shows that the eigenvalues for the linear case has absent real parts and that the magnitude of the eigenvalue is determined by the wave number kd and gravity g. We will not cover the physics of the potential flow formulation but the energy should generally be conserved due to the lack of viscous forces. It is therefore expected that the solution to the non linear problem is not damped but oscillatory. The eigenvalues will thus have absent real parts and the magnitude of the imaginary part will further be time dependent due to the non linearity of the problem.
A reasonable requirement is therefore that the stability region of the IVP solver must cover parts of the imaginary axis, which is the case with e.g. the standard 4^{th} order Runge-Kutta scheme. A. Engsig-Karup has further shown that the full non linear problem is convergent when a 4^{th} order Runge-Kutta scheme is used. Therefore the standard 4^{th} order Runge-Kutta method is considered as a sufficient choice of iteration scheme.

$$\boldsymbol{k}_1 = \Delta t \boldsymbol{f}(\boldsymbol{x}_t, t) \tag{1.28}$$

$$\boldsymbol{k}_2 = \Delta t \boldsymbol{f}(\boldsymbol{x}_t + \frac{\boldsymbol{k}_1}{2}, t + \frac{\Delta}{2} t) \tag{1.29}$$

$$\boldsymbol{k}_3 = \Delta t \boldsymbol{f}(\boldsymbol{x}_t + \frac{\boldsymbol{k}_2}{2}, t + \frac{\Delta}{2} t) \tag{1.30}$$

$$\boldsymbol{k}_4 = \Delta t \boldsymbol{f}(\boldsymbol{x}_t + \boldsymbol{k}_3, t + \Delta t) \tag{1.31}$$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \frac{1}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4), \quad \boldsymbol{x} = [\ \eta \quad \tilde{\phi}\ ]^T$$

Provided a way to calculate $\boldsymbol{k}_1$, $\boldsymbol{k}_2$, $\boldsymbol{k}_3$ and $\boldsymbol{k}_4$, advancing in time is trivial. The primary objective is therefore to evaluate (1.1) and (1.2) efficiently, which again primarily concerns solving the transformed Laplace equation.

1.1.2 Stability region of RK4

Runge-Kutta methods are a set of Taylor series methods; in order to eliminate the unwanted terms of a Taylor expansion, the Runge-Kutta methods approximate the solution through multiple stages. According to [10] The standard 4^{th}

order Runge-Kutta simplifies to

$$x_{t+1} = x_t + \Delta t x'_t + \frac{1}{2}(\Delta t)^2 x''_t + \frac{1}{6}(\Delta t)^3 x_t^{(3)} + \frac{1}{24}(\Delta t)^4 x_t^{(4)} + O(h^5) \quad (1.32)$$

where the higher order temporal derivatives are approximated through multiple stages per times step. In general, the stability region for an IVP solver is defined to be the region that ensure a linear homogeneous differential equation converges. For linear homogeneous differential equations, the solution given in (1.26) can be used to determine the stability region of the Runge Kutta method. The stability region is found by simple insertion of the higher order derivatives of (1.22) into (1.32). To obtain the high order derivatives, (1.22) is applied to itself;

$$\partial_t \boldsymbol{x} = \boldsymbol{A}\boldsymbol{x} \Rightarrow \partial_{t^2} \boldsymbol{x} = \boldsymbol{A}^2\boldsymbol{x} \Rightarrow \partial_{t^3} \boldsymbol{x} = \boldsymbol{A}^3\boldsymbol{x} \Rightarrow \partial_{t^4} \boldsymbol{x} = \boldsymbol{A}^4\boldsymbol{x} \quad (1.33)$$

By insertion of (1.33) into (1.32), the stability region of the method can be found.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \Delta t \boldsymbol{A}\boldsymbol{x}_t + \frac{1}{2}(\Delta t)^2 \boldsymbol{A}^2\boldsymbol{x}_t + \frac{1}{6}(\Delta t)^3 \boldsymbol{A}^3\boldsymbol{x}_t + \frac{1}{24}(\Delta t)^4 \boldsymbol{A}^4\boldsymbol{x}_t + O((\Delta t)^5)$$

To analyze the system, we use (1.25); when the method converges for the equivalent problem, the method will converge for the original problem as well.

$$\boldsymbol{y}_{t+1} = \boldsymbol{y}_t + z\boldsymbol{y}_t + \frac{1}{2}z^2\boldsymbol{y}_t + \frac{1}{6}z^3\boldsymbol{y}_t + \frac{1}{24}z^4\boldsymbol{y}_t + O((\Delta t)^5), \quad z = \Delta t \boldsymbol{D} \quad (1.34)$$

If the standard 4^{th} order Runge Kutta method should converge, it is therefore crucial that $\max|z| < 1$. Recall that $\boldsymbol{D}$ is a diagonal matrix holding the eigenvalues of $\boldsymbol{A}$. Hence the Runge Kutta method presented is stable for

$$|\lambda_{max}\Delta t| < 1 \quad (1.35)$$

Since the eigenvalues in general are complex, the stability region is best illustrated by a complex plot of (1.34). As seen in fig. 1.2, the stability region of RK4 cover parts of the imaginary axis which as mentioned is a requirement for this particular problem. For general ODE's, the temporal step size can be chosen arbitrary as long as $|\lambda_{max}\Delta t| < 1$ is not violated. In [6], A. Engsig-Karup suggests that it will be reasonable to choose time step from the CFL[3] conditions which is more strict than the actual stability region of the solver. The CFL conditions are given by

$$\Delta t \leq C\frac{u}{\Delta x}, \quad C < 1 \quad (1.36)$$

[3]Courant-Friedrichs-Lewy

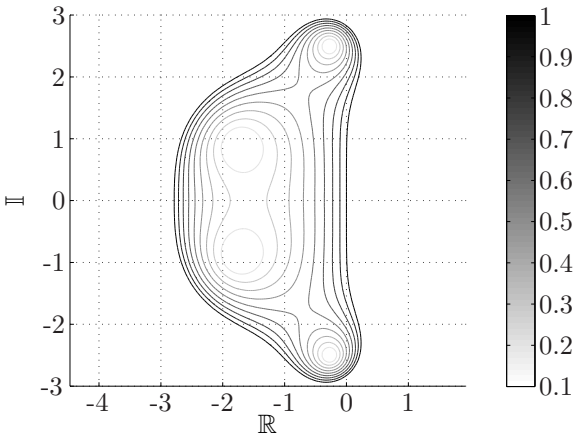

Figure 1.2: Illustration of $|\lambda_{max}\Delta t| < 1$ for (1.34). The edge of the region represent the boundary locus.

where u is the fluid velocity $\frac{L(\text{wave length})}{T(\text{wave period})}$. The CFL conditions relate to the Courant number, which is given by $\nu = \frac{u \cdot \Delta t}{\Delta x}$. In order to ensure convergence $C < 1$ should be chosen. The CFL condition arise when explicit time-marching schemes (as e.g. RK4) are used for the numerical solution of certain PDEs, including the surface potential flow formulation. The advantage of having an explicit time scheme is that the time integration is embarrassing parallel which is a potential advantage to any parallel implementation of a program. The disadvantage of the CFL-conditions is that they are more strict than (1.35).

1.2 Approximating the Laplace equation 1

To solve the differential equation, a Method of Lines approach (MOL) is used. MOL separates the evaluation of the differential equation into a spatial and temporal discretization. For the spatial discretization, the idea is to simply exchange every term in the transformed Laplace equation with a discrete approximation. In general some initial surface elevation and surface potential will be known and only at the discrete positions x_i, $i = 1, ..., N$. Everything else need to be approximated which is done with finite difference.

Finite difference in general uses a weighted sum based on Taylor expansion to approximate any order derivatives. Increasing the number of grid points used in the weighted sum increases accuracy as well as the order of derivative which can be approximated. Since finite difference is a weighted sum, the approximation

of some derivative throughout an entire grid can be calculated using a matrix vector product. For a vector representation $\boldsymbol{U}$of some grid, the approximated second order derivative $\boldsymbol{U}_{xx}$ in each grid point can be found by

$$\boldsymbol{U}_{xx} = \boldsymbol{A}_{xx}\boldsymbol{U} \tag{1.37}$$

where $\boldsymbol{A}_{xx}$ contains the appropriate weighting for finite difference approximations in this particular grid.
It is important to emphasize that the intension is to avoid the usage of fully generated matrices in the implementation of the program although matrices are used for a formalized presentation of the methods.
For a 1D structured grid with uniform distributed grid nodes, the 2^{nd} order discretization is given by

$$\boldsymbol{U} = \begin{bmatrix} U_1 & U_2 & \cdots & U_{n-1} & U_n \end{bmatrix}^T \tag{1.38}$$

$$\boldsymbol{A}_{xx} = \frac{1}{h^2}\begin{bmatrix} 2 & -5 & 4 & -1 & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & 1 & -4 & 5 & -2 \end{bmatrix} \tag{1.39}$$

where h is the grid spacing. Similarly the Laplacian can be approximated using similar matrices containing weights for derivatives in the remaining directions.

$$\nabla^2\boldsymbol{U} = \boldsymbol{A}\boldsymbol{U}, \quad \boldsymbol{A} = \boldsymbol{A}_{xx} + \boldsymbol{A}_{yy} + \boldsymbol{A}_{zz} \tag{1.40}$$

For the Laplace problem, the value of the Laplacian $\nabla^2\boldsymbol{U}$ is given rather than the grid values $\boldsymbol{U}$. In general, there exist no particular solution to a PDE if not at least one Dirichlet boundary condition is specified. In order to solve the discrete approximation to the Laplace equation given by (1.40) boundary conditions thus have to be specified. For a discrete approximation with absent Dirichlet conditions, missing boundary conditions is manifested by an underdetermined system. To approximate some solution U to a Laplace problem, the following system should be solved

$$\boldsymbol{A}_{xx}\boldsymbol{U} + \boldsymbol{A}_{yy}\boldsymbol{U} + \boldsymbol{A}_{zz}\boldsymbol{U} + \boldsymbol{A}_{B.C}\boldsymbol{U} = \boldsymbol{0} + \boldsymbol{b}_{B.C} \tag{1.41}$$

where $\boldsymbol{A}_{B.C}$ represents the boundary conditions of the system and $\boldsymbol{b}_{B.C}$ the values of the boundary conditions. For a 1D system with uniform grid spacing

h, the matrices and boundary values are given by

$$\boldsymbol{b}_{B.C} = \begin{bmatrix} U_1 & 0 & \cdots & 0 & \phi_n \end{bmatrix}^T \tag{1.42}$$

$$\boldsymbol{A}_{xx} = \frac{1}{h^2}\begin{bmatrix} 0 & & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & 1 & -4 & 5 & -2 \end{bmatrix}, \quad \boldsymbol{A}_{B.C} = \begin{bmatrix} 1 & & & & \\ & 0 & & & \\ & & \ddots & & \\ & & & 0 & \\ & & \frac{1}{2h} & -\frac{2}{h} & \frac{3}{2h} \end{bmatrix} \tag{1.43}$$

Here a Dirichlet boundary condition is applied at U_0 and a 2^{nd} order accurate flow condition (Neumann condition) at U_n.

Due to the MOL approach, the surface variables (η, h and derivatives hereof) will be independent from the transformed Laplace problem although some are time dependent. For this reason, they are static values hence explicit in the transformed Laplace problem. This is also the case for σ and its derivatives. Although independent of the Laplace problem, they do vary over the spatial domain.

$$\partial_z\sigma(\partial_\sigma\Phi) \rightarrow \begin{bmatrix} \sigma_{z,1}\Phi_{\sigma,1} \\ \sigma_{z,2}\Phi_{\sigma,2} \\ \vdots \\ \sigma_{z,N}\Phi_{\sigma,N} \end{bmatrix} \tag{1.44}$$

$$\begin{bmatrix} \sigma_{z,1}\Phi_{\sigma,1} \\ \sigma_{z,2}\Phi_{\sigma,2} \\ \vdots \\ \sigma_{z,N}\Phi_{\sigma,N} \end{bmatrix} = \begin{bmatrix} \sigma_{z,1} & & & \\ & \sigma_{z,2} & & \\ & & \ddots & \\ & & & \sigma_{z,N} \end{bmatrix} \begin{bmatrix} \Phi_{\sigma,1} \\ \Phi_{\sigma,2} \\ \vdots \\ \Phi_{\sigma,N} \end{bmatrix} = \mathcal{D}(\boldsymbol{\sigma}_z)\boldsymbol{\Phi}_\sigma \tag{1.45}$$

$$(\partial_z\sigma)^2(\partial_{\sigma\sigma}\Phi) \rightarrow \mathcal{D}(\boldsymbol{\sigma}_z^2)\boldsymbol{\Phi}_\sigma, \quad \boldsymbol{\sigma}_z^2 \equiv \mathcal{D}(\boldsymbol{\sigma})\boldsymbol{\sigma} \tag{1.46}$$

where $\boldsymbol{\Phi}_\sigma$ and $\boldsymbol{\Phi}_{\sigma\sigma}$ are first and second derivatives of the discrete valued function $\boldsymbol{\Phi}$. We can now describe the transformed Laplace problem. The linear approximation is given by

$$\Phi = \tilde{\phi}, \quad \sigma = 1 \tag{1.47}$$

$$\nabla^2\Phi + (\partial_z\sigma)^2\partial_{\sigma\sigma}\Phi = 0, \quad 0 \leq \sigma < 1 \tag{1.48}$$

$$(\partial_\sigma\Phi)\partial_z\sigma = 0, \quad \sigma = 0 \tag{1.49}$$

$$\downarrow$$

$$\boldsymbol{A}_{D.C}\boldsymbol{\Phi} = \tilde{\boldsymbol{\phi}}, \quad \sigma = 1 \tag{1.50}$$

$$\left(\boldsymbol{A}_{xx} + \boldsymbol{A}_{yy} + \mathcal{D}(\boldsymbol{\sigma}_z)^2\boldsymbol{A}_{\sigma\sigma}\right)\boldsymbol{\Phi} = 0, \quad 0 \leq \sigma < 1 \tag{1.51}$$

$$(\boldsymbol{A}_\sigma\boldsymbol{\Phi})\mathcal{D}(\boldsymbol{\sigma}_z) = 0, \quad \sigma = 0 \tag{1.52}$$

here $\boldsymbol{A}_{D.C}$ describe the Dirichlet condition at surface level.

For the non linear case, (1.51) and (1.52) should be substituted by the non linear approximations presented by (1.57) and (1.60) respectively.

$$\nabla^2\Phi \rightarrow \boldsymbol{A}_{xx}\boldsymbol{\Phi} + \boldsymbol{A}_{yy}\boldsymbol{\Phi} \tag{1.53}$$

$$\nabla^2\sigma\left(\partial_\sigma\Phi\right) \rightarrow \mathcal{D}(\boldsymbol{\sigma}_{xx} + \boldsymbol{\sigma}_{yy})\boldsymbol{A}_\sigma\boldsymbol{\Phi} \tag{1.54}$$

$$2\nabla\sigma\cdot\nabla\left(\partial_\sigma\Phi\right) \rightarrow 2\mathcal{D}(\boldsymbol{\sigma}_x)\boldsymbol{A}_{x\boldsymbol{\sigma}}\boldsymbol{\Phi} + 2\mathcal{D}(\boldsymbol{\sigma}_y)\boldsymbol{A}_{y\sigma}\boldsymbol{\Phi} \tag{1.55}$$

$$\left(\nabla\sigma^T\nabla\sigma + \left(\partial_z\Phi\right)^2\right)\partial_{\sigma\sigma}\Phi \rightarrow \mathcal{D}\left(\boldsymbol{\sigma}_x^2 + \boldsymbol{\sigma}_y^2 + \boldsymbol{\sigma}_z^2\right)\boldsymbol{A}_{\sigma\sigma}\boldsymbol{\Phi} \tag{1.56}$$

$$\begin{aligned}\boldsymbol{A}_{xx}\boldsymbol{\Phi} + \boldsymbol{A}_{yy}\boldsymbol{\Phi} + \mathcal{D}(\boldsymbol{\sigma}_{xx} + \boldsymbol{\sigma}_{yy})\boldsymbol{A}_\sigma\boldsymbol{\Phi} + 2\mathcal{D}(\boldsymbol{\sigma}_x)\boldsymbol{A}_{x\sigma}\boldsymbol{\Phi} + 2\mathcal{D}(\boldsymbol{\sigma}_y)\boldsymbol{A}_{y\sigma}\boldsymbol{\Phi}\\ + \mathcal{D}\left(\boldsymbol{\sigma}_x^2 + \boldsymbol{\sigma}_y^2 + \boldsymbol{\sigma}_z^2\right)\boldsymbol{A}_{\sigma\sigma}\boldsymbol{\Phi} = \boldsymbol{0}, \quad 0 \le \sigma < 1\end{aligned} \tag{1.57}$$

Even though many more terms have been added, the system is still linear in $\boldsymbol{\Phi}$ so the problem is still to solve a linear system. As for the boundary conditions to the full problem, they are approximated by

$$(\partial_z\sigma + \nabla h\cdot\nabla\sigma)(\partial_\sigma\Phi) \rightarrow \mathcal{D}(\boldsymbol{\sigma_z} + \mathcal{D}(\boldsymbol{h}_x)\boldsymbol{\sigma}_x + \mathcal{D}(\boldsymbol{h}_y)\boldsymbol{\sigma}_y)\boldsymbol{A}_\sigma\Phi \tag{1.58}$$

$$\nabla h\cdot\nabla\Phi \rightarrow \mathcal{D}(\boldsymbol{h}_x)\boldsymbol{A}_x\Phi + \mathcal{D}(\boldsymbol{h}_y)\boldsymbol{A}_y\Phi \tag{1.59}$$

$$\begin{aligned}\mathcal{D}(\boldsymbol{\sigma_z} + \mathcal{D}(\boldsymbol{h}_x)\boldsymbol{\sigma}_x + \mathcal{D}(\boldsymbol{h}_y)\boldsymbol{\sigma}_y)\boldsymbol{A}_\sigma\boldsymbol{\Phi}\\ + \mathcal{D}(\boldsymbol{h}_x)\boldsymbol{A}_x\boldsymbol{\Phi} + \mathcal{D}(\boldsymbol{h}_y)\boldsymbol{A}_y\boldsymbol{\Phi} = 0, \quad \sigma = 0\end{aligned} \tag{1.60}$$

In order to evaluate the actual differential equation, the velocity potential must be known throughout the domain. The Laplace equation is solved with the current surface potential as boundary condition on the surface prior to evaluation of the differential equation. Using the previous notation, the approximation to (1.1) and (1.2) is here presented in matrix form

$$\partial_t\boldsymbol{\eta} = -(\mathcal{D}(\boldsymbol{\eta}_x)\tilde{\boldsymbol{\phi}}_x + \mathcal{D}(\boldsymbol{\eta}_y)\tilde{\boldsymbol{\phi}}_y) + \mathcal{D}(\tilde{\boldsymbol{w}})(1 + \boldsymbol{\eta}_x^2 + \boldsymbol{\eta}_y^2) \tag{1.61}$$

$$\partial_t\tilde{\boldsymbol{\phi}} = -g\boldsymbol{\eta} - \frac{1}{2}\left(\tilde{\boldsymbol{\phi}}_x^2 + \tilde{\boldsymbol{\phi}}_y^2 - \mathcal{D}(\tilde{\boldsymbol{w}}^2)(1 + \boldsymbol{\eta}_x^2 + \boldsymbol{\eta}_y^2)\right) \tag{1.62}$$

As mentioned earlier, the only coupling to the Laplace equation is through $\tilde{w}$ which is approximated from the calculated solution to the transformed Laplace problem.

$$\tilde{w} = \partial_\sigma\Phi\partial_z\sigma|_{\sigma=1} \tag{1.63}$$

$$\downarrow$$

$$\tilde{\boldsymbol{w}} = \mathcal{D}(\boldsymbol{\sigma}_z)\boldsymbol{A}_\sigma\boldsymbol{\Phi}|_{\sigma=1} \tag{1.64}$$

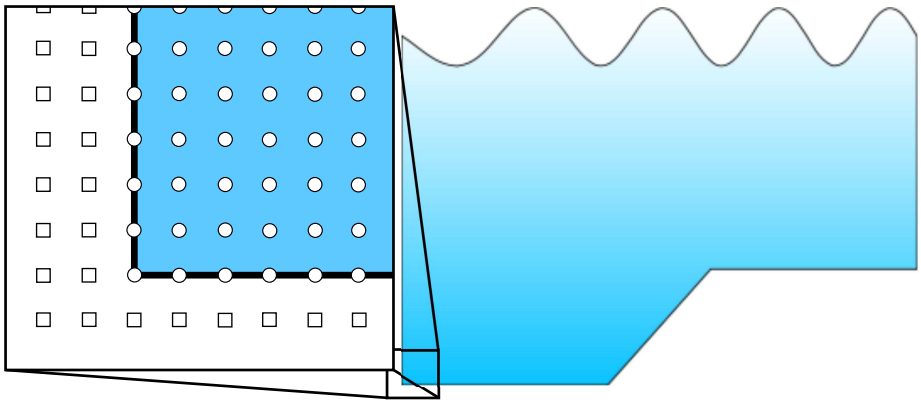

Figure 1.3: A number of layers of ghost points (squares) are used both in the horizontal directions and along the bottom in order to imprint boundary values. The internal domain is represented by the round dots.

1.2.1 Boundary conditions

A typical way to implement boundary conditions is shown in the 1D sample discretization (1.43) in previous section. In the example, off-centered approximations are needed near the boundary but this is neither practical from a GPU parallelization point of view nor good for certain classes of solvers[4]. [1] suggest to use one or more layers of ghost points outside the computational domain to impose boundary conditions (illustrated by fig. 1.3). The ghost points can be either eliminated from the system or simply added as additional equations in the system.

We will not eliminate the ghost points since the GPU architecture used for implementation is slowed down when the program has divergent branches.

Since the ghost points are added as additional equations in the system, they should be up to date prior to any operation on the grid, be that smoothing, restriction, prolongation[5] or anything else. For a bounded domain, the ghost update procedure should implement zero-flow at bottom and sides of the domain.

[4]Particularly the Jacobi method and Gauss-Seidel methods which we will use requires the system matrix to be positive definite

[5]Smoothing, restriction and prolongation are components of Coarse Grid Correction. See section 2.1.

We will use the ghost points to fulfill following equations

$$\partial_x \phi = 0, \quad x = 0 \vee x = L_x \tag{1.65}$$
$$\partial_y \phi = 0, \quad y = 0 \vee y = L_y \tag{1.66}$$
$$\partial_z \phi = 0, \quad z = -h \tag{1.67}$$

Neither (1.65) nor (1.66) alters in the transformed Laplace equation. On the other hand (1.67) will be given by (1.10):

$$\partial_x \Phi = 0, \quad x = 0 \vee x = L_x \tag{1.68}$$
$$\partial_y \Phi = 0, \quad y = 0 \vee y = L_y \tag{1.69}$$
$$\left(\partial_z \sigma + \nabla h \cdot \nabla \sigma\right)\left(\partial_z \Phi\right) + \nabla h \cdot \nabla \Phi = 0, \qquad \sigma = 0 \tag{1.70}$$

For the x and y directions, the number of ghost points added should always be large enough to allow (1.57) to be evaluated using central approximations only. The amount of ghost points in the horizontal directions is therefore 1 or more. The order of accuracy on the border should be the same level as for the internal points. For a 2^{nd} order accurate approximation to (1.68), a horizontal ghost point can therefore be isolated from

$$\partial_x \Phi = 0 \Rightarrow \tag{1.71}$$
$$\frac{1}{2\Delta x}\left(U_{-\Delta x,y,\sigma} + U_{\Delta x,y,\sigma}\right) = 0 \Leftrightarrow \tag{1.72}$$
$$U_{-\Delta x,y,\sigma} = U_{\Delta x,y,\sigma} \tag{1.73}$$

where U is the discrete approximation to Φ. For higher order approximations, there will be more approximations available to $\partial_x \Phi = 0$. Given a 4^{th} order approximation scheme, 2 layers[6] of ghost points should be used. For the 4^{th} order approximation, there are two available approximations to $\partial_x \Phi = 0$ (a forward and a central) which touches a ghost point.

$$\frac{1}{12\Delta x}\begin{bmatrix} -3 & \underline{-10} & 18 & -6 & 1 \end{bmatrix} \tag{1.74}$$
$$\downarrow$$
$$\frac{1}{12\Delta x}(\underbrace{-3U_{-\Delta x,y,\sigma}}_{\text{Ghost point}} - 10U_{0,y,\sigma} + 18U_{\Delta x,y,\sigma} - 6U_{2\Delta x,y,\sigma} + U_{3\Delta x,y,\sigma}) = 0 \tag{1.75}$$

$$\frac{1}{12\Delta x}\begin{bmatrix} 1 & -8 & \underline{0} & 8 & -1 \end{bmatrix} \tag{1.76}$$
$$\downarrow$$
$$\frac{1}{12\Delta x}(\underbrace{U_{-2\Delta x,y,\sigma} - 8U_{-\Delta x,y,\sigma}}_{\text{Ghost points}} + 0U_{0,y,\sigma} + 8U_{\Delta x,y,\sigma} - U_{2\Delta x,y,\sigma}) = 0 \tag{1.77}$$

[6] 3 for a 6^{th} order approximation, 4 for a 8^{th} order approximation etc.

At least (1.77) should be fulfilled since it contains all ghost points. The question is then whether it is necessary to also fulfill (1.75). Experience shows that it is enough to fulfill (1.77) by simply letting the ghost points be a mirror of the domain values:

$$U_{-2\Delta x,y,\sigma} = U_{2\Delta x,y,\sigma} \tag{1.78}$$
$$U_{-\Delta x,y,\sigma} = U_{\Delta x,y,\sigma} \tag{1.79}$$

This should be done in both horizontal directions. For the ghost points at $x = 0 \wedge y = 0$, the mirroring is expressed by

$$U_{-2\Delta x,-2\Delta y,\sigma} = U_{2\Delta x,2\Delta y,\sigma} \tag{1.80}$$
$$U_{-2\Delta x,-\Delta y,\sigma} = U_{2\Delta x,\Delta y,\sigma} \tag{1.81}$$
$$U_{-\Delta x,-2\Delta y,\sigma} = U_{\Delta x,2\Delta y,\sigma} \tag{1.82}$$
$$U_{-\Delta x,-\Delta y,\sigma} = U_{\Delta x,\Delta y,\sigma} \tag{1.83}$$

The bottom boundary condition should be updated according to (1.60). For the low order linear transformed Laplace problem, the update reduces to mirroring in the boundary. For higher order approximations the ghost point value should be isolated from the finite difference sum U_σ. $\hat{U}_\sigma$ is defined as the part of the finite difference sum which does not use the ghost point. In particular

$$U_\sigma = \hat{U}_\sigma + S_\sigma^{(ghost)} U^{(ghost)}$$

where $S_\sigma^{(ghost)}$ is the stencil value associated with the ghost point $U^{(ghost)}$. The resulting ghost update is therefore generally given by

$$(\partial_z \sigma + \nabla h \cdot \nabla \sigma)\, U_\sigma + \nabla h \cdot \nabla U = 0 \tag{1.84}$$
$$(\partial_z \sigma + \nabla h \cdot \nabla \sigma) \left(\hat{U}_\sigma + S_\sigma^{(ghost)} U^{(ghost)} \right) + \nabla h \cdot \nabla U = 0 \tag{1.85}$$
$$U_{x,y,-\Delta\sigma} = -\frac{1}{S_\sigma^{(ghost)}} \left[\hat{U}_\sigma + a \nabla h \cdot \nabla U \right] \tag{1.86}$$
$$a = \frac{1}{(\partial_z \sigma + \nabla h \cdot \nabla \sigma)} \tag{1.87}$$

To illustrate, a 4^{th} order example in 2D is presented. The finite difference sums of (1.60) are represented by the stencils

$$S_\sigma = \frac{1}{12\Delta\sigma} \begin{bmatrix} -1 \\ 6 \\ -18 \\ \underline{10} \\ 3 \end{bmatrix} \tag{1.88}$$

$$S_x = \frac{1}{12\Delta x} \begin{bmatrix} 1 & -8 & \underline{0} & 8 & -1 \end{bmatrix} \tag{1.89}$$

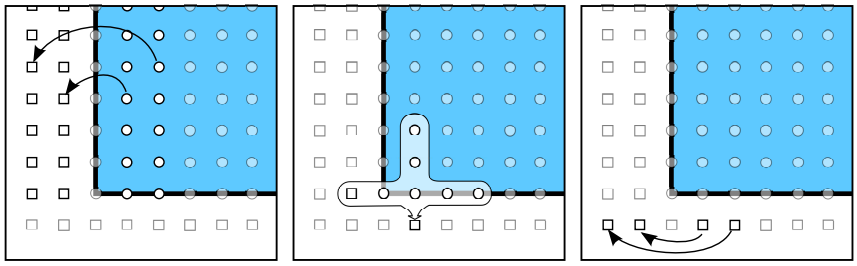

Figure 1.4: Updating the ghostpoints using a 4^{th} order accurate non linear approximation. First the horizontal layer should be updated by mirroring (left). Then the bottom layer using (1.91) (center, only one point illustrated), and finally the corners (right).

where the underlined element of the stencils is a bottom boundary element. The stencils are applied to the grid

$$\frac{\partial_z\sigma + \partial_x h\,\partial_x\sigma}{12\Delta\sigma}\left(3U_{x,\sigma-\Delta\sigma} + 10U_{x,\sigma} - 18U_{x,\sigma+\Delta\sigma} + 6U_{x,\sigma+2\Delta\sigma} - U_{x,\sigma+3\Delta\sigma}\right)$$
$$+ \frac{\partial_x h}{12\Delta x}\left(U_{x-2\Delta x,\sigma} - 8U_{x-\Delta x,\sigma} + 0U_{x,\sigma} + 8U_{x+\Delta x,\sigma} - U_{x+2\Delta x,\sigma}\right) = 0 \quad (1.90)$$

The value of the ghost point is then obtained by isolation of $U_{x,\sigma-\Delta\sigma}$

$$U_{x,\sigma-\Delta\sigma} = -\frac{1}{3}\left[\left(10U_{x,\sigma} - 18U_{x,\sigma+\Delta\sigma} + 6U_{x,\sigma+2\Delta\sigma} - U_{x,z+3\Delta\sigma}\right)\right.$$
$$\left. + a\left(U_{x-2\Delta x,\sigma} - 8U_{x-\Delta x,\sigma} + 0U_{x,\sigma} + 8U_{x+\Delta x,\sigma} - U_{x+2\Delta x,\sigma}\right)\right] \quad (1.91)$$

$$a = \frac{\partial_x h(x)}{\partial_z\sigma(x) + \partial_x h(x)\partial_x\sigma(x)} \quad (1.92)$$

Near the sides of the domain, the horizontal derivatives will cover other ghost points as well. The ghost points on the side of the domain should therefore be up to date before updating the bottom boundary layer.

Since a central stencil is applied in the horizontal direction, the ghost points in the bottom corners of the domain will need to be updated subsequently. Updating the corner ghost points is performed by mirroring in the horizontal axes. The entire update procedure is illustrated for a 2D case in fig. 1.4

1.2.2 Periodic boundary conditions

One very fortunate consequence of using ghost points explicitly rather than eliminating them from the system is that periodic boundary conditions can be implemented fairly easy. Instead of letting the ghost points be a mirror, they should hold a copy of the values from the other end of the domain. In particular for the x-direction, in a domain with dimensions $0 \leq x \leq L_x$ the update is given by

$$U_{-\Delta x,y,\sigma} = U_{L_x,y,\sigma} \tag{1.93}$$

$$U_{-2\Delta x,y,\sigma} = U_{L_x-\Delta x,y,\sigma} \tag{1.94}$$

$$U_{L_x+\Delta x,y,\sigma} = U_{0,y,\sigma} \tag{1.95}$$

$$U_{L_x+2\Delta x,y,\sigma} = U_{\Delta x,y,\sigma} \tag{1.96}$$

Again the easiest way to update the corners is to update ghost points along first one direction, synchronize and then update along the second direction.

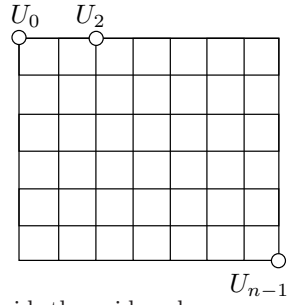

Figure 1.5: In a structured grid, the grid nodes are numbered by position. Coordinates and connectivity can therefore be calculated rather than looked up in tables.

1.3 Approximating the Laplace equation 2

Knowing what to solve, it is time to decide on how to solve it. The intention of this project is to increase the problem size while also allowing the order of approximation to be adjustable. Increasing the problem size requires more memory. Increasing order of the spatial approximation of the Laplace equation gives higher stability demands to the numerical solver. Due to the Method of Lines approach, the Laplace problem and evaluation of temporal derivatives will be separated. Evaluation of the Laplace problem will be the most computation and storage consuming part; the Laplace equation is a global problem in a 3D domain whereas the evaluation of the actual differential equation is an explicit operation in a 2D domain.

The primary choice about discretization will be whether to use a structured or unstructured grid. The advantage of the unstructured grid is the possibility to insert grid elements wherever higher accuracy is needed. On the other hand, it will also make computations complex and expensive and require for position and lookup tables to identify grid connectivity. For a structured grid, the storage requirements are lower since vertex position and connectivity is known beforehand. The disadvantage of using a structured grid is that it is not as easy to get more details in local areas of the grid. Because of the low storage requirements of the structured grid and that GPUs has an advantage for structured memory, an ordered grid will be used. The structured grid will be with a regular layout (fig. 1.5). Regardless of grid structure, there is also an option to whether or not the system matrix should be generated. The system matrix compared to the grid is large and in other words very memory consuming. Memory is a scarce resource so creation of the system matrix should be avoided. The cost of not saving the matrix is that the matrix will need to be generated more than once. Computational power on the other hands is not as scarce so on the fly matrix generation is considered the better choice. Also, it is expected that the memory latency hiding feature of CUDA devices (also described in chapter 3) will render

the matrix generation a more or less 'free' operation: While a thread waits for a variable to be transfered from memory it is paused, freeing up resources to perform calculations elsewhere.
Because of the regular layout of the grid, the finite difference stencils are very much alike. The only difference between them will actually be a scalar determined by the grid distance in the direction of the approximation. Approximating the second derivatives on a rectangular grid, the same stencil can be used in all directions:

$$S_{xx} = \frac{1}{(\Delta x)^2}[S_{nn}]_x, \quad S_{yy} = \frac{1}{(\Delta y)^2}[S_{nn}]_y, S_{\sigma\sigma} = \frac{\sigma_z}{(\Delta \sigma)^2}[S_{nn}]_\sigma \tag{1.97}$$

$$S_{nn} = [1 \ \underline{-2} \ 1] \tag{1.98}$$

The benefit of using the same stencil all over the domain is that it can be held in the cached constant memory of the CUDA device (discussed in chapter 3). To calculate the finite difference stencils of various order, the stable procedure developed by Fornberg [8], here implemented as `fdcoeffF`, will be used.

The equations to be approximated, are partial derivatives of 1^{st} and 2^{nd} order as well as a few mixed derivatives. The partial derivatives are calculated directly using `fdcoeffF` and the mixed derivatives as a tensor product of the partial derivatives. An example of generating mixed derivatives using the 1D stencils is provided in appendix B.

1.4 Expectations to speedup

Prior to further analysis, the expected of speedup from a CUDA implementation will be estimated. Two common notions of scalability is strong and weak scaling:

Strong scaling : How does solution time vary with number of available processors for fixed problem size?

Weak scaling : How does solution time vary with problem size for a fixed number of processors?

It has already been shown that the surface potential problem using a transformed Laplace problem for the viscous term has weak scaling properties for the Defect Correction with a Multigrid preconditioner [5]. What is interesting is therefore how large a speedup can be obtained for a parallel version of the algorithm. If the memory bandwidth is not limited, and there is sufficiently

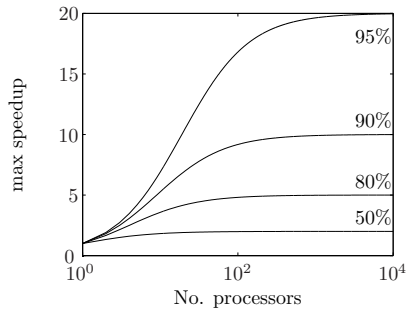

Figure 1.6: Ahmdahls law for fixed parallel fractions of a code.

work to do, the parallel speedup of a program is determined by Ahmdahl's law.

$$\text{max speedup} = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.99}$$

where P is the parallel fraction of a program and N the number of processors available. If on the other hand, the domain size is small such not all processors have work to do, the speedup is bound by Gustafsons' law

$$\text{max speedup} = N - (1-P)(N-1) \tag{1.100}$$

where again P is the parallel fraction of the program. When the number of processors is in surplus, a parallel problem will thus have strong linear scaling. Regardless of Gustafson's or Ahmdahl's law apply to a particular size of the problem, one must be aware that a sequential overhead is associated with the launch of a kernel[7]. For small problems, the GPU implementation is therefore most likely slower than a CPU implementation since the initialization of the device simply eats up too much time.

Computing finite difference stencils is not particularly complex, so the algorithm is expected to be memory bound rather than compute bound.

Besides the computational power, also the bandwidth should be considered; A Tesla C1060/Quadro FX 5800 GPU has 244 CUDA cores[8] which each process instructions for 32 threads (a warp) at the time. On these GPUs the upper limit of active threads is thus 7808 at the time, and the bandwidth per thread is therefore relatively limit. Given full occupancy, the bandwidth per thread is therefore $\frac{102.4}{7808} GB/s$ =0.013GB/s which is very little compared to CPU bandwidth. Currently, host RAM bandwidth is up to approximately 17GB/s[9] which

[7]GPU programs are called kernels.
[8]System specifications available in appendix C.
[9]DDR3, PC17000. 6-12GB/s is more common though

in comparison per processor is much more. If the number of computations per transfered element is low, the speedup will thus depend on the ratio between host and device bandwidth rather than the increased number of threads. It is non trivial to calculate the ratio between computations and transfered memory for a kernel to be memory bound since the hardware is fairly advanced.

Chapter 2

Analysis of the iterative Laplace solver

The key to perform well in solving the surface potential flow equations is to solve the transformed Laplace equation efficiently; the Laplace equation pose a global/implicit problem while evolving surface quantities is a local/explicit problem due to the choice of an explicit time integration scheme. The target size of the Laplace equation will be in the scale of several million degrees of freedom and $O(n)$ scalability is therefore crucial for the Laplace solver. Further it is requested that the solver should use double precision accuracy.

The ordinary multigrid method (Coarse Grid Correction - CGC) is famous for its $O(n)$ scaling properties and it is further interesting because it basically is an embarrassing parallel algorithm. Since the number of concurrent active threads on GPUs presently is in the scale of thousands, particularly embarrassing parallel algorithms are expected to gain from a parallel implementation. Although interesting, it has during this project come clear that the CGC components are not sufficiently stable[1] to solve high order approximations to neither ordinary nor transformed Laplace problem.

The Defect Correction scheme (algorithm 1) is recently[2] proven to 'fix' the

[1] Investigated further in section 2.2
[2] 2010, [5], parallel with this project

stability problems with CGC; when DC is preconditioned with a low order linear CGC method, the scheme is stable, efficient and still embarrassing parallel.

Algorithm 1 Defect Correction: Solve $\boldsymbol{Au} = \boldsymbol{b}$

1: $k \to 0$
2: **repeat**
3: $\quad \boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{Au}^{[k]}$
4: $\quad \boldsymbol{d}^{[k]} \leftarrow \boldsymbol{\mathcal{M}}^{-1}\boldsymbol{r}$ ▷ $\boldsymbol{\mathcal{M}}$ is a low order approximation to $\boldsymbol{A}$
5: $\quad \boldsymbol{u}^{[k+1]} \leftarrow \boldsymbol{u}^{[k]} + \boldsymbol{d}^{[k]}$ ▷ Compensate for defect $\boldsymbol{d}$
6: $\quad k \to k + 1$
7: **until** $||\boldsymbol{d}^{[k]}|| < ||\boldsymbol{d}^{[0]}||\cdot$rel.tol + abs.tol ▷ Convergence criteria

Also the Generalized Minimal Residual (GMRES) method using a low order, either a direct solution or iterative multigrid preconditioner is known to be an efficient solution strategy [1]. [5] shows that GMRES outperform DC for the Whalin shoal problem[3] with up to approximately 20% shorter solution time for a gridsize of $257 \times 21 \times 6$. GMRES is a Krylow subspace method and therefore associated with finding a number of orthogonal vectors. The method is therefore not embarrassing parallel and also uses more memory than the Defect Correction algorithm. Defect Correction with a low order multigrid preconditioner is therefore the choice of algorithm.

2.1 Coarse Grid Correction

Defect Correction is in itself a trivial (and embarrassing parallel) algorithm. The preconditioner - Coarse Grid Correct - on the other hand is non trivial. CGC (algorithm 2) belong to a subset of methods called Multigrid Methods. The idea of multigrid approaches is to remove some modes of the error, re-sample and handle the remaining error components subsequently. Which modes and which kind of re-sampling used depends on the chosen multigrid scheme.

CGC deals with high frequency error components first and low frequency components in a recursive fashion (illustrated by fig. 2.1). To deal with the high frequency components, the scheme utilizes the smoothing properties of other iterative solvers. Both the Jacobi and the Gauss-Seidel method are examples of solvers with such properties. Although neither of the methods are excessively good for solving a system alone, they have smoothing properties; information propagates fast locally for both methods, hence remove high frequency error components relatively fast (illustrated in fig. 2.2 for the Jacobi method). CGC

[3]The Whalin shoal problem is also for validation of the mode. Eventually see section 6.2.

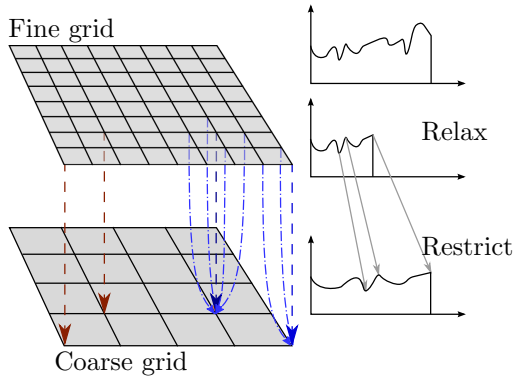

Figure 2.1: Multigrid removes high frequencies of the Fourier form residual by smoother relaxations. The remaining residual is transfered to a coarse grid using direct transfer (red arrows/left) or full weighting (blue arrows/right) which stretches the frequency form residual.

deals with high frequency error components first and low frequency components in a recursive fashion (illustrated by fig. 2.1). To deal with the high frequency components, the scheme utilizes the smoothing properties of other iterative solvers. Both the Jacobi and the Gauss-Seidel method are examples of solvers with such properties. Although neither of the methods are excessively good for solving a system alone, they have smoothing properties; information propagates fast locally for both methods, hence remove high frequency error components relatively fast (illustrated in fig. 2.2 for the Jacobi method).

Algorithm 2 Coarse Grid Correction: Solve $\boldsymbol{Au} = \boldsymbol{f}$

1: **repeat**
2: Relax ν_1 times on $\boldsymbol{Au} = \boldsymbol{f}$ with initial guess $\boldsymbol{v}$
3: Compute coarse grid residual: $\check{\boldsymbol{r}} \leftarrow \boldsymbol{R}(\boldsymbol{f} - \boldsymbol{Av})$
4: Solve coarse grid approximation $\check{\boldsymbol{A}}\check{\boldsymbol{e}} = \check{\boldsymbol{r}}$ ▷ Possibly recursive
5: Correct fine grid approximation: $\boldsymbol{v} \leftarrow \boldsymbol{v} + \boldsymbol{I}\check{\boldsymbol{e}}$
6: Relax ν_2 times on $\boldsymbol{Au} = \boldsymbol{f}$ with initial guess $\boldsymbol{v}$
7: **until** converged ▷ As preconditioner, do not repeat

The algorithm basically contain 4 components and a bunch of coarse approximations to the original problem. A coarse approximation in this context is simply a discretization of same order with double distance between grid nodes. The 4 components of Coarse Grid Correction are smoother, restriction operator, coarse grid solver and interpolation operator. The tools and ideas used in CGC resemble operations found in Digital Signal Processing (DSP) and can be analyzed thoroughly using Local Fourier Analyses (LFA). The components of this multigrid method is analyzed in section 2.2 and section 2.3. The convergence criteria can be the same as used in DC but is irrelevant when used as a

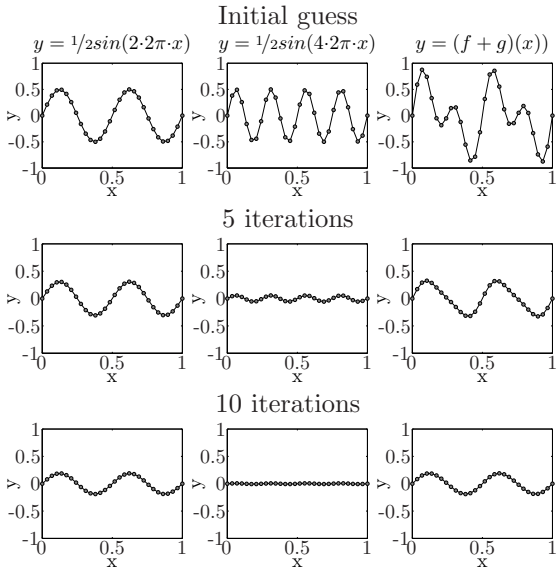

Figure 2.2: The Jacobi method applied to a 1D Laplace problem $\partial_{xx}\mathbf{x} = \mathbf{0}$ with different initial guesses on a 30 element grid. The magnitude of a high frequency error component (center) is reduced much faster than a low frequency error component (left).

preconditioner; it should perform only one iteration rather than iterating until convergence is reached.

In order to fully understand how the multigrid solver works, the core components are presented in detail in the upcoming sections 2.1.1-2.1.5. Especially the smoother (2.1.5) is an important component of the algorithm.

2.1.1 Restriction operator

Restriction cover the fine-to-coarse grid transfer and is done such that the integral sum in the transfered area will not change. The coarse grid transfer should thus approximate u_c:

$$\int_{\Omega_f} u_f \, \mathrm{d}\Omega_f = \int_{\Omega_f} u_c \, \mathrm{d}\Omega_c, \quad \forall i \tag{2.1}$$

Indices f and c denote fine and coarse grid variables respectively. The simplest grid transfer is direct transfer which is first order accurate. For a given grid

node, the integral can be approximated locally by the rectangle method

$$\int_{-h_{f,a}}^{h_{f,b}} \boldsymbol{u}_f \, \mathrm{d}\Omega_f = u_{f,i} h_{f,a} + u_{f,i}(h_{f,b}) + O(h) = u_{f,i}(h_{f,a} + h_{f,b}) + O(h_f^2) \quad (2.2)$$

where h is a measurement for step length (not the still water depth). Similarly, the integral for the coarse grid can be approximated locally using the midpoint rule

$$\int_{-h_{f,a}}^{h_{f,b}} \boldsymbol{u}_c \, \mathrm{d}\Omega_c = u_{c,i}(h_{f,a} + h_{f,b}) + O(h_c^3) \quad (2.3)$$

Combining (2.2) and (2.3) reveals the Direct Transfer method.

$$u_{c,j}(h_{f,a} + h_{f,b}) = u_{f,i}(h_{f,a} + h_{f,b}) \Rightarrow \quad (2.4)$$

$$u_{c,j} = u_{f,i} \quad (2.5)$$

Here u_i and u_j are considered 'on top' of each other. The order of the direct transfer correspond to the lowest order of the approximation to the integral. In other words, the direct transfer is a local 2^{nd} order, global 1^{st} order approximation scheme. The order of accuracy is increased fairly easy by exchanging the fine grid integral approximation with the trapezoidal rule.

$$\int_{-h_{f,a}}^{h_{f,b}} \boldsymbol{u}_c \, \mathrm{d}\Omega_c = \frac{u_{f,i-1} + u_{f,i}}{2} h_{f,a} + \frac{u_{f,i} + u_{f,i+1}}{2} h_{f,b} + O(h_f^3) \quad (2.6)$$

$$= u_{f,i-1} \frac{h_{f,a}}{2} + u_{f,i} \frac{h_{f,a} + h_{f,b}}{2} + u_{f,i+1} \frac{h_{f,b}}{2} + O(h_f^3) \quad (2.7)$$

Combining (2.3) and (2.7) reveal the 2^{nd} order accurate Full Weighting operator

$$u_{c,j}(h_{f,a} + h_{f,b}) = u_{f,i-1} \frac{h_{f,a}}{2} + u_{f,i} \frac{h_{f,a} + h_{f,b}}{2} + u_{f,i+1} \frac{h_{f,b}}{2} \quad (2.8)$$

$$u_{c,j} = u_{f,i-1} \frac{1}{2} \frac{h_{f,a}}{h_{f,a} + h_{f,b}} + u_{f,i} \frac{1}{2} + u_{f,i+1} \frac{1}{2} \frac{h_{f,b}}{h_{f,a} + h_{f,b}} \quad (2.9)$$

Again u_i and u_j are considered 'on top'. The grid values u_{i-1} and u_{i+1} on the other hand do not have 'on top' counterparts on the coarse grid (fig. 2.3). For cuboid grids $h_{f,a} = h_{f,b}$ result in the stencil R:

$$u_{c,j} = \frac{1}{4} [u_{f,i-1} + 2u_{f,i} + u_{f,i+1}] \quad (2.10)$$

$$R = \frac{1}{4} [1 \;\; \underline{2} \;\; 1] \quad (2.11)$$

As for the mixed derivatives described in section 1.3, the 2D and 3D restriction and prolongation operators can be calculated by the 1D tensor product. For higher order restriction operators, the coarse grid integral in (2.1) can no longer be approximated with one coarse grid point only. It is therefore not possible to make an explicit restriction operator with order higher than 2. An implicit operator is expensive and should therefore be avoided if not crucial to maintain effectiveness and precision of the algorithm.

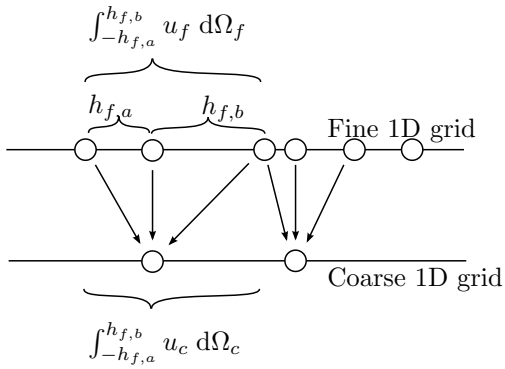

Figure 2.3: The fine grid integral should match the coarse grid integral. Grid values u_f and u_c are given only at the grid nodes.

2.1.1.1 Updating along the edges

With an exception of the surface values, the coarsening can be calculated using the stencil given by (2.9) anywhere; the ghost points introduced in section 1.2.1 should merely be updated prior to the restriction. For the surface values, the stencil will be 'poking out' of the domain though. There are two solutions to this: Either make a local first order approximation to (2.1) (ie. copy the values at the surface) or just ignore that the stencil is too large and insert zeros outside the domain. In practice ignoring that the stencil is too wide works just fine.

2.1.2 Prolongation operator

Prolongation cover the coarse-to-fine grid transfer. The prolongation operator is simply interpolation between coarse and fine grid; if the grid nodes are 'on top', the value is transfered directly. If the grid point is in between, the value is interpolated to some order of accuracy. The sum of the order of restriction and order of interpolation should exceed the order of differentiation in the approximated system [12]. For the transformed Laplace equation, the highest order derivatives are 2. order partial derivatives, thus

$$m_r + m^i > 2 \tag{2.12}$$

where m_r and m^i is the order of the restriction operator and prolongation operator respectively. Remaining is the question to whether direct injection should be used rather than full weighting for restriction.

The 1^{st} order accurate prolongation operator is linear interpolation between the coarse grid points. Increasing the interpolation accuracy by a single order,

the schemes will be either explicit and use a larger neighborhood (polynomial interpolation) or implicit (e.g. spline interpolation). Neither method is less computational expensive than the 2^{nd} order restriction operator. Therefore the best choice of restriction and prolongation fulfilling (2.12) will be Full Weighting ($m_r = 2$) and linear interpolation ($m^i = 1$).

Linear interpolation is generally given by

$$u_i = \frac{h_a u_{j_a} + h_b u_{j_b}}{h_a + h_b} \tag{2.13}$$

where h_a and h_b is the distance to the nearest neighboring grid points, u_{j_a} and u_{j_b}. For coarse to fine grid transfers, the interpolation generalize to two cases due to the rectilinear grid:

$$u_i = \begin{cases} u_j & u_i \text{ over } u_j \\ \frac{u_{j_a} + u_{j_b}}{2} & u_i \text{ between } u_{j_a} \text{ and } u_{j_b} \end{cases} \tag{2.14}$$

For 2D and 3D, linear interpolation generalizes to bi- and tri-linear interpolation.

2.1.3 Restriction strategy

The typical grid size for the type of problems we are interested in will have few points in the vertical direction and lots in the horizontal directions. Restriction in all directions is therefore not feasible; for a grid of size $65 \times 17 \times 3$, restriction in all directions cannot be done since the vertical grid size cannot be decreased. Restriction in the horizontal directions on the other hand is possible. Not restricting in all directions at once is called semi-coarsening. Semi-coarsening in both horizontal directions reveals a coarse grid of size $33 \times 9 \times 3$. Instead of applying semi-coarsening along both x- and y-direction, semi-coarsening along just one direction is possible as well. For semi-coarsening, the restriction operator should work in the restricted directions only. That is (2.1) should approximate the integral of the restricted directions only. Semi-coarsening can in general be applied instead of full coarsening if chosen to. A general coarsening strategy based on analysis of the smoothers is provided in section 2.4.

2.1.4 Coarse grid solver

The job of the coarse grid operator is to find a solution to the coarse grid approximation. If the grid is suitable for another division of size, CGC can be used to solve this problem as well. At some point it will no longer be possible to do a

restriction operation on the grid. At this point, the coarse grid operator should solve the system completely such that remaining low frequency components are removed. This can also be done at an earlier state if CGC proves inefficient for small grids.

When CGC is used as coarse grid operator, the algorithm is obviously recursive. Depending on the order in which CGC is invoked, different recursive patterns called cycles arise. Three particularly wide used strategies are called F-, V- and W-cycles due to their F-, V- and W-like recursion patterns. The V- and W-cycle generalize to the μ-cycle (algorithm 3). Using $\mu = 1$ reveal the V-cycle and $\mu = 2$ the W-cycle.

Algorithm 3 μ-cycle

repeat
  Relax ν_1 times on $\boldsymbol{Au} = \boldsymbol{f}$ with initial guess $\boldsymbol{v}_0$
  Compute coarse grid residual: $\check{\boldsymbol{r}} = \boldsymbol{R}(\boldsymbol{f} - \boldsymbol{Av_1})$
  Set initial guess $\check{\boldsymbol{e}} = \boldsymbol{0}$
  if possible **then**
    Update $\check{\boldsymbol{e}}$ using CGC on $\check{\boldsymbol{A}}\check{\boldsymbol{e}} = \check{\boldsymbol{r}}$ μ times
  else
    Solve $\check{\boldsymbol{A}}\check{\boldsymbol{e}} = \check{\boldsymbol{r}}$
  end if
  Correct fine grid approximation: $\boldsymbol{v}_2 = \boldsymbol{v}_1 + \boldsymbol{I}\check{\boldsymbol{e}}$
  Relax ν_2 times on $\boldsymbol{Au} = \boldsymbol{f}$ with initial guess $\boldsymbol{v}_1$
until converged

2.1.5 Smoothers

In Coarse Grid correction, the objective of the smoother is to eliminate or at least reduce the high frequency error components of the system. Although all components are needed in order to make the algorithm work, the effectiveness of the smoother proves to be the key to make the algorithm efficient. In the following sections we will introduce a number of smoothers and look into their pros and cons.

Since the preconditioner is low order and ghost points take care of the boundary, the approximation to the transformed Laplace equation is everywhere presented by central approximations. On a uniform ordered grid, the central approximation to the second derivative of some discrete valued function is given by the stencil

$$S = \frac{1}{h^2}\begin{bmatrix} 1 & \underline{-2} & 1 \end{bmatrix} \tag{2.15}$$

where h is the step length. Applying the above stencil to (1.51), the full 3D stencil is obtained:

$$S_{3D} = \frac{1}{h_x^2}[\ 1 \quad \underline{-2} \quad 1\ ]_x + \frac{1}{h_y^2}[\ 1 \quad \underline{-2} \quad 1\ ]_y + (\partial_z\sigma)^2\frac{1}{h_\sigma^2}[\ 1 \quad \underline{-2} \quad 1\ ]_\sigma \tag{2.16}$$

All the smoothers presented are derived from some splitting of the original using either present ($u_i^{[k]}$) or future iterates ($u_i^{[k+1]}$) of the grid points. A neighboring future iterate can be obtained by e.g. updating on top of the old value. It is therefore convenient to present the methods using forward and backward stencils denoted S^+ and S^- where $S = S^+ + S^-$. A forward stencil is a stencil applied to future iterates of the grid and its backward counterpart applied to present values of the grid. Performing a relaxation on the system can, regardless of method presented, be written in matrix form

$$\boldsymbol{G}^+x^{[k+1]} + \boldsymbol{G}^-x^{[k]} = \boldsymbol{b} \tag{2.17}$$

$$\boldsymbol{G}^+x^{[k+1]} = \boldsymbol{b} - \boldsymbol{G}^-x^{[k]} = \tilde{\boldsymbol{b}} \tag{2.18}$$

where $\boldsymbol{G}^+$ and $\boldsymbol{G}^-$ represent the matrices generated by applying all of S^+ and S^- to the problem.

In general for methods which can be presented in matrix form, it is required that the spectral radius of the iteration matrix $\boldsymbol{G}$ to be at max 1; the magnitude of a matrix vector product is limited by the spectral radius, formally described by (2.20).

$$\rho(\boldsymbol{G}) = \max_i |\lambda_i| \tag{2.19}$$

$$||\boldsymbol{G}^k\boldsymbol{v}|| \leq \rho(\boldsymbol{G})^k||\boldsymbol{v}|| \tag{2.20}$$

$$\rho(\boldsymbol{G}) \leq 1 \Rightarrow ||\boldsymbol{G}^k\boldsymbol{v}|| \leq ||\boldsymbol{v}|| \tag{2.21}$$

where λ_i are the eigenvalues of the iteration matrix $\boldsymbol{G}$. For an iterative method using forward and backward stencil, $\boldsymbol{G}$ is given by

$$\boldsymbol{G} = (\boldsymbol{G}^+)^{-1}\boldsymbol{G}^- \tag{2.22}$$

2.1.5.1 The Jacobi Method

The Jacobi method is derived from a splitting of the linear system:

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b} \tag{2.23}$$

$$(\boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U})\boldsymbol{u} = \boldsymbol{b} \tag{2.24}$$

$$\boldsymbol{D}\boldsymbol{u} = -(\boldsymbol{L} + \boldsymbol{U})\boldsymbol{u} + \boldsymbol{b} \tag{2.25}$$

$$\boldsymbol{u}^{[k+1]} = \boldsymbol{D}^{-1}(-(\boldsymbol{L} + \boldsymbol{U})\boldsymbol{u}^{[k]} + \boldsymbol{b}) \tag{2.26}$$

where $\boldsymbol{L}, \boldsymbol{D}$ and $\boldsymbol{U}$ are lower, upper and diagonal elements of $\boldsymbol{A}$ respectively. Another presentation uses the residual rather than $\boldsymbol{L}$ and $\boldsymbol{U}$: Continuing from (2.25) it is found that

$$\boldsymbol{D}\boldsymbol{u} = -(\boldsymbol{L} - \boldsymbol{D} + \boldsymbol{D} + \boldsymbol{U})\boldsymbol{u} + \boldsymbol{b} \tag{2.27}$$

$$\boldsymbol{D}\boldsymbol{u} = \boldsymbol{D}\boldsymbol{u} - (\boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U})\boldsymbol{u} + \boldsymbol{b} \tag{2.28}$$

$$\boldsymbol{D}\boldsymbol{u}^{[k+1]} = \boldsymbol{D}\boldsymbol{u}^{[k]} + (-\boldsymbol{A}\boldsymbol{u}^{[k]} + \boldsymbol{b}) \tag{2.29}$$

$$\boldsymbol{u}^{[k+1]} = \boldsymbol{u}^{[k]} + \boldsymbol{D}^{-1}\boldsymbol{r}^{[k]} \tag{2.30}$$

The forward stencil notation is obtained immediately from (2.29) related to (2.18) and (2.16):

$$\boldsymbol{G}^{+} = \boldsymbol{D}, \quad \boldsymbol{G}^{-} = \boldsymbol{D} - \boldsymbol{A} \Rightarrow \tag{2.31}$$

$$S^{+} = \frac{1}{(\Delta x)^2}[\ 0 \quad \underline{-2} \quad 0\ ]_x + \frac{1}{(\Delta y)^2}[\ 0 \quad \underline{-2} \quad 0\ ]_y + (\partial_z\sigma)^2\frac{1}{(\Delta\sigma)^2}[\ 0 \quad \underline{-2} \quad 0\ ]_\sigma \tag{2.32}$$

$$S^{-} = -\left(\frac{1}{(\Delta x)^2}[\ 1 \quad \underline{0} \quad 1\ ]_x + \frac{1}{(\Delta y)^2}[\ 1 \quad \underline{0} \quad 1\ ]_y + (\partial_z\sigma)^2\frac{1}{(\Delta\sigma)^2}[\ 1 \quad \underline{0} \quad 1\ ]_\sigma\right) \tag{2.33}$$

The Jacobi method exists in a slightly modified version, called damped Jacobi. Damped Jacobi is given by

$$\boldsymbol{u}^{[k+1]} = \boldsymbol{u}^{[k]} + \lambda\boldsymbol{D}^{-1}\boldsymbol{r} \tag{2.34}$$

where λ is the dampening factor. The forward stencil notation is thus

$$\boldsymbol{G}^{+} = \boldsymbol{D}, \quad \boldsymbol{G}^{-} = \boldsymbol{D} - \lambda\boldsymbol{A} \Rightarrow \tag{2.35}$$

$$S^{+} = \frac{1}{(\Delta x)^2}[\ 0 \quad \underline{-2} \quad 0\ ]_x + \frac{1}{(\Delta y)^2}[\ 0 \quad \underline{-2} \quad 0\ ]_y + \frac{(\partial_z\sigma)^2}{(\Delta\sigma)^2}[\ 0 \quad \underline{-2} \quad 0\ ]_\sigma \tag{2.36}$$

$$S^{-} = -\lambda\left(\frac{1}{(\Delta x)^2}\begin{bmatrix} 1 \\ \frac{2}{\lambda} - 2 \\ 1 \end{bmatrix}^T_x + \frac{1}{(\Delta y)^2}\begin{bmatrix} 1 \\ \frac{2}{\lambda} - 2 \\ 1 \end{bmatrix}^T_y + \frac{(\partial_z\sigma)^2}{(\Delta\sigma)^2}\begin{bmatrix} 1 \\ \frac{2}{\lambda} - 2 \\ 1 \end{bmatrix}^T_\sigma\right) \tag{2.37}$$

$\boldsymbol{G}^{+}$ contains only the diagonal elements of $\boldsymbol{A}$. The Jacobi and damped Jacobi method is therefore embarrassing parallel; all elements of $\boldsymbol{u}^{[k+1]}$ can be calculated concurrently with no synchronization needed since $\boldsymbol{u}^{[k]}$ and $\boldsymbol{u}^{[k+1]}$ are distinct vectors.

2.1.5.2 Gauss-Seidel methods

Gauss-Seidel methods overwrite values of $\boldsymbol{x}^{[k]}$ rather than saving $\boldsymbol{x}^{[k+1]}$ in a distinct vector. The grid values are thus a mix of future and present values. The order in which the values are updated determines which Gauss-Seidel method is being used. The two most widely used Gauss-Seidel methods are Lexicographic Gauss-Seidel (GSLEX) and Red Black Gauss-Seidel (RBGS). 3D GSLEX is given by the two stencils

$$S^- = \frac{1}{(\Delta x)^2}[\ 0 \quad \underline{0} \quad 1\ ]_x + \frac{1}{(\Delta y)^2}[\ 0 \quad \underline{0} \quad 1\ ]_y + (\partial_z \sigma)^2 \frac{1}{(\Delta \sigma)^2}[\ 0 \quad \underline{0} \quad 1\ ]_\sigma \tag{2.38}$$

$$S^+ = \frac{1}{(\Delta x)^2}[\ 1 \quad \underline{-2} \quad 0\ ]_x + \frac{1}{(\Delta y)^2}[\ 1 \quad \underline{-2} \quad 0\ ]_y + (\partial_z \sigma)^2 \frac{1}{(\Delta \sigma)^2}[\ 1 \quad \underline{-2} \quad 0\ ]_\sigma \tag{2.39}$$

Notice that the only difference from the Jacobi method is that some values have moved from S^- to S^+. This particular Gauss-Seidel method starts in a corner of the domain and works from there. Because GSLEX work diagonally across the domain, it is not embarrassing parallel thus not particularly well suited for parallel use.

The Red Black Gauss-Seidel method update every second value concurrently: A Jacobi relaxation is performed, first on the red marked grid points and then black marked grid points afterwards. Red and Black points are interleaved. RBGS applied to the 1D Laplace equation is given by

$$U_i^{[k+1]} = \begin{cases} \frac{1}{2}\left(U_{i-1}^{[k]} + U_{i+1}^{[k]}\right) & \text{red grid points} \\ \frac{1}{2}\left(U_{i-1}^{[k+1]} + U_{i+1}^{[k+1]}\right) & \text{black grid points} \end{cases} \tag{2.40}$$

The method is therefore better suited for parallel use even when there are many processing units available; half the domain can be processed at the time. RBGS is a so called 2 grid operator which does not allow for the method to be written directly in forward and backward notation.

2.1.5.3 Line smoothing

Jacobi and Gauss-Seidel are point smoothers which generally have the drawback that they are inefficient for systems with anisotropic stencils[4]. A line solver on the other hand suffer less from anisotropic stencils[5]. Line smoothers update

[4] Investigated further in section 2.3

[5] 2D or 3D stencils: Anisotropy occurs if the elements in one direction generally are larger than in the remaining directions.

an entire line in the physical domain at the time rather than a point at the time. The remaining directions are updated using a point smoothing strategy. In forward stencil notation, the 2^{nd} order accurate σ-line smoother applied to the transformed Laplace equation is given by

$$S^- = S_x^- + S_y^- + (\partial_z\sigma)^2 \frac{1}{(\Delta\sigma)^2}[\ 0 \quad \underline{0} \quad 0\ ]_\sigma \tag{2.41}$$

$$S^+ = S_x^+ + S_y^+ + (\partial_z\sigma)^2 \frac{1}{(\Delta\sigma)^2}[\ 1 \quad \underline{-2} \quad 1\ ]_\sigma \tag{2.42}$$

Since the line smoother work in one direction only, S_x^-, S_x^+, S_y^- and S_y^+ are determined by the strategy (Jacobi, GSLEX, RBGS, ...) used in the other directions. For a domain of size[6] $N_x \times N_y \times N_\sigma$, a σ-line Jacobi smoother consists of $N_x \cdot N_y$ decoupled linear systems which can be solved independently. $\boldsymbol{G}^+$ is in that case given by the S^+ stencils alone; when $\boldsymbol{x}_\sigma$ define the elements of a line in the σ-direction in the domain, (2.18) can be written as

$$\widehat{\boldsymbol{G}}^+ \boldsymbol{u}_\sigma = \widehat{\boldsymbol{b}} \tag{2.43}$$

$$\widehat{\boldsymbol{G}}^+ = -\frac{2}{(\Delta x)^2}\begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \end{bmatrix} - \frac{2}{(\Delta y)^2}\begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{0} \end{bmatrix} + \frac{(\partial_z\sigma)^2}{(\Delta\sigma)^2}\begin{bmatrix} 1 & & & & & & \\ 1 & -2 & 1 & & & & \\ & & \ddots & & & & \\ & & 1 & -2 & 1 & & \\ & & & 1 & -2 & 1 & \\ & & & -1 & & 1 & \end{bmatrix} \tag{2.44}$$

where $\boldsymbol{0}$ is a row of 0's and $\boldsymbol{I}$ an $N_\sigma \times N_\sigma$ identity matrix. Further $\widehat{\boldsymbol{G}}$ is only part of $\boldsymbol{G}$ and $\widehat{\boldsymbol{b}}$, the corresponding elements of $\tilde{\boldsymbol{b}}$. The first row specifies the Dirichlet boundary conditions at the top. The last row apply the 2^{nd} order Neumann condition at the bottom through a ghost point as specified in section 1.2.1. Because of the ghost point, $\widehat{\boldsymbol{G}}^+$ is a $(N_\sigma+1)\times(N_\sigma+1)$ matrix. For Gauss-Seidel type smoothers in the x, y-directions, the line relaxation will be determined by the same system as well; when the smoother has reached the line in question, it locally performs a Jacobi relaxation.

$\widehat{\boldsymbol{G}}^+$ is tri-diagonal matrix with a single out-lier for the boundary element. When the diagonal elements are determined by $\boldsymbol{a}_{-1}, \boldsymbol{a}_0$ and $\boldsymbol{a}_1$ and the out-liner e, $\widehat{\boldsymbol{G}}^+$

[6] Ghost point not included

is given by

$$\widehat{\boldsymbol{G}}^{+} = \begin{bmatrix} a_{0,1} & a_{1,1} & & & & \\ a_{-1,2} & a_{0,2} & a_{1,2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & a_{-1,N_\sigma} & a_{0,N_\sigma} & a_{1,N_\sigma} & \\ & & e & a_{-1,N_\sigma+1} & a_{0,N_\sigma+1} & a_{1,N_\sigma+1} \end{bmatrix} \tag{2.45}$$

$$e = -1 \tag{2.46}$$

$$\boldsymbol{a}_{-1} = \frac{1}{(\Delta\sigma)^2}\begin{bmatrix} 0 \\ 1 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{a}_0 = \begin{bmatrix} 1 \\ -2(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{(\partial_z\sigma)^2}{(\Delta\sigma)^2}) \\ \vdots \\ -2(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{(\partial_z\sigma)^2}{(\Delta\sigma)^2}) \\ 1 \end{bmatrix}, \quad \boldsymbol{a}_1 = \frac{1}{(\Delta\sigma)^2}\begin{bmatrix} 0 \\ 1 \\ \vdots \\ 1 \\ 0 \end{bmatrix} \tag{2.47}$$

The system can be solved efficiently using Gaussian elimination. For tridiagonal matrices, Gaussian elimination is also known as the Thomas Algorithm. Algorithm 4 present the Thomas Algorithm modified to include the out-lier e.

Replacing the point smoother with a line smoother does not change the scalability of the algorithm; updating with a point smoother is an $O(1)$ operation performed N_σ times resulting in $O(N_\sigma)$ operations to update a row. For the line smoother, the cost is $O(c \cdot (N_\sigma + 1)) = O(N_\sigma)$ operations (c is a constant). Although the cost of a relaxation is higher, the algorithm therefore still has the same scaling properties as a point smoother.

2.2 Digital Signal Processing tools

Tools from Digital Signal Processing (DSP) will be used for an in depth analysis of Coarse Grid Correction; DSP is concerned with the representation of signals by a sequence of numbers and processing these signals. If the grid values are thought of as a signal, smoothing, restriction and prolongation can be thought of as processing the signal. The processes correspond to convolve with a filter and/or change sample rate. Exactly due to the similarity to convolution, DSP gets interesting as a tool for analysis of the smoother: Convolution in the spatial domain corresponds to multiplication in the frequency domain and the Fourier form of the coarse grid correction operation can therefore be used to analyze its convergence properties. When the magnitude of the Fourier form is known, it can be used to provide an upper limit to how fast some particular error

Algorithm 4 Thomas Algorithm

$\boldsymbol{a} \equiv \boldsymbol{a}_{-1}, \quad \boldsymbol{b} \equiv \boldsymbol{a}_0, \quad \boldsymbol{c} \equiv \boldsymbol{a}_1$ ▷ in place variables for improved readability
$\boldsymbol{d} \equiv \tilde{\boldsymbol{b}}$

for $i \leftarrow 2, N_\sigma$ **do** ▷ Forward elimination
    $scale \leftarrow \frac{a_i}{b_{i-1}}$
    $a_i \leftarrow a_i - b_{i-1} \cdot scale$
    $b_i \leftarrow b_i - c_{i-1} \cdot scale$
    $d_i \leftarrow d_i - d_{i-1} \cdot scale$
end for

$i \leftarrow N_\sigma + 1$ ▷ Apply boundary condition
$scale \leftarrow \frac{e}{b_{i-2}}$
$a_i \leftarrow a_i - c_{i-2} \cdot scale$
$d_i \leftarrow d_i - d_{i-2} \cdot scale$

$scale \leftarrow \frac{a_i}{b_{i-1}}$ ▷ Forward elimination - last row
$a_i \leftarrow a_i - b_{i-1} \cdot scale$
$b_i \leftarrow b_i - c_{i-1} \cdot scale$
$d_i \leftarrow d_i - d_{i-1} \cdot scale$

$d_i \leftarrow \frac{d_i}{b_i}$ ▷ Backward elimination
for $i \leftarrow N_\sigma, 1$ **do**
    $d_i \leftarrow d_i - d_{i+1} \frac{c_i}{b_i}$
    $d_i \leftarrow \frac{d_i}{b_i}$
end for

$\tilde{\boldsymbol{b}} \leftarrow \boldsymbol{d}$

component decay. Further it will also be easily seen if the method is divergent for some particular approximation: If the Fourier form for any frequency has a magnitude greater than 1, the smoother will be divergent! First a short introduction to DSP (for details, see [9]).

Convolution is an operation defined for both continuous and discrete domains.

$$(f * g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau = \int_{-\infty}^{\infty} g(\tau)\, f(t-\tau)\, d\tau \tag{2.48}$$

$$(f * g)(n) \equiv \sum_{m=-\infty}^{\infty} f(m)\, g(n-m) = \sum_{m=-\infty}^{\infty} g(m)\, f(n-m) \tag{2.49}$$

The discrete version relates to CGC since it is a formalized way of calculating a weighted neighborhood sum using the same weight allover the domain. Finite difference uses a weighted neighborhood sum with a weight which might change over the domain. Notice that the operation is symmetric; $(f * g)(n) \equiv (g * f)(n)$. In other words it does not matter which of f or g is signal or filter. Mathematically this means that the convolution satisfies the commutative law. Convolution is also an associative and distributive operation.

Commutativity

$$f(n) * g(n) = g(n) * f(n) \tag{2.50}$$

Associativity

$$[f(n) * g_1(n)] * g_2(n) = f(n) * [g_1(n) * g_2(n)] \tag{2.51}$$

Distributivity

$$f(n) * [g_1(n) + g_2(n)] = f(n) * g_1(n) + f(n) * g_2(n) \tag{2.52}$$

For analysis of the smoother, the Fourier transform[7] of an infinite finite-energy discrete valued function will be needed

$$1D : F(\omega) = \sum_{k=-\infty}^{\infty} f(n) e^{-i\pi k\omega}, \omega \in]-1;1] \tag{2.53}$$

For a finite discrete valued function $x_0, ..., x_{n-1}$, the Fourier form is given by a set of Fourier coefficients $X_0, ..., X_{n-1}$

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{k}{N} n} \tag{2.54}$$

[7] Alternatively Z-transform can be used: $F = \sum_{k=-\infty}^{\infty} f(n) z^k$

Be aware that both $F(\omega)$ and X_k are complex valued. For real valued functions on large grids, $X_k \to F(\pm\frac{k}{N})$ and it will for analytic purposes therefore suffice to use (2.53). For the analysis tools to be valid, the system must further be Linear Time-Invariant (LTI). That is signal and filter does not change during the convolution; if a value of the signal or filter has been processed, it must not change before the entire signal has been processed. This is not always the case for the smoother of CGC since the discretization/the filter might change near the borders. This is the case for a high order approximation to the non-linear transformed Laplace problem since the discretization changes for every grid node due to the non-linear terms. Further, the finite difference stencils might change near the edge of the domain. Although convolution and smoothing is not totally equivalent, it is still possible to analyze what happens to the signal locally. One is called Fourier Analysis and the other Local Fourier Analysis (LFA). The strategy in either case is to find the frequency response of the filter.

2.2.1 1D Local Fourier Analysis

Since the smoother depends on both problem and discretization, both should be chosen prior to the analysis. For the ordinary Laplace equation in 1D, a symmetric 2^{nd} order stencil is given by

$$h^{-2}\left(U_{i-1} - 2U_i + U_{i+1}\right) = U_i'' \tag{2.55}$$

A single Jacobi update on a Poisson equation corresponds to update the element U_i by a weighted sum of a neighborhood around U_i. In case of the 2^{nd} order stencil presented here, the update is reduced to

$$U_i^{[k+1]} = \frac{1}{2}\left(U_{i-1}^{[k]} + U_{i+1}^{[k]} + b_i\right) \tag{2.56}$$

In order to analyze the method using LFA, $\boldsymbol{b}$ is set to $\mathbf{0}$. In underline notation[8], the filter to analyze is given by

$$f(n) = \left[\begin{array}{ccc} \frac{1}{2} & \underline{0} & \frac{1}{2} \end{array}\right] \tag{2.57}$$

The frequency response of the filter is found using the Fourier transform for

[8] If unknown to the reader, this is elaborated in Appendix B

discrete time signals.

$$F(\omega) = \sum_{n=-\infty}^{\infty} f(n)e^{in\omega\pi}, \omega \in]-1;1] \tag{2.58}$$

$$F(\omega) = f(-1)e^{-i\omega\pi} + f(0)e^{0} + f(1)e^{i\omega\pi} \tag{2.59}$$

$$F(\omega) = \frac{1}{2}(\cos(-\omega\pi) + i\sin(-\omega\pi)) + \frac{1}{2}(\cos(\omega\pi) + i\sin(\omega\pi)) \tag{2.60}$$

$$F(\omega) = \cos(\omega\pi) \tag{2.61}$$

Damped Jacobi is very similar to the Jacobi method. Given the Jacobi iterate $\tilde{U}_i^{[k+1]}$, damped Jacobi is given by

$$U_i^{[k+1]} = (1-\lambda)U_i^{[k]} + \lambda\tilde{U}_i^{[k+1]} \tag{2.62}$$

where $\lambda > 0$. Insertion of (2.56) gives the damped Jacobi iteration for a 2^{nd} order ordinary Laplace equation

$$U_i^{[k+1]} = (1-\lambda)U_i^{[k]} + \lambda\frac{1}{2}\left(U_{i-1}^{[k]} + U_{i+1}^{[k]}\right) \tag{2.63}$$

Since convolution is distributive, the frequency response F_λ for damped Jacobi is easily computed using the frequency response F for the corresponding Jacobi method.

$$F_\lambda(\omega) = (1-\lambda)e^{i0\omega\pi} + \lambda F(\omega) \tag{2.64}$$

$$= 1 - \lambda(1 - \cos(\omega\pi)) \tag{2.65}$$

Varying λ thus change the magnitude response of the frequencies. Figure 2.4 shows frequency responses for the analyzed situation for various choices of λ. What is interesting to notice is that for any $0 < \lambda < 1$, the magnitude of the response of the high frequencies (the dyed area of fig. 2.4) are less than 1 thus damped.

Normally the goal for a solver is to find a solution to some system using that solver over and over. In CG, the main purpose for the smoother is to reduce the error of the high frequency components rather than solving the system. Altering the weight in damped Jacobi changes the frequency response, so a straight forward idea will be to find an optimal value of λ. Most often the grid will be re-sampled by doubling the step length which is nearly the same as using half the number of grid points. Halving the sample rate cause the frequency spectrum of the grid values to be stretched to double length which is elaborated in section 2.2.2. Frequencies in the range 0.5 to 1 will thus need to be cut off [9];

$$1/2 \leq \omega_{high} \leq 1$$

[9]The cut frequency can be narrowed a bit further down due to the grid is of finite size; the grid can only contain resolution up to $\omega = 1 - \frac{L}{h}$. $1/2 \leq \omega_{high} \leq 1$ is asymptotic behavior

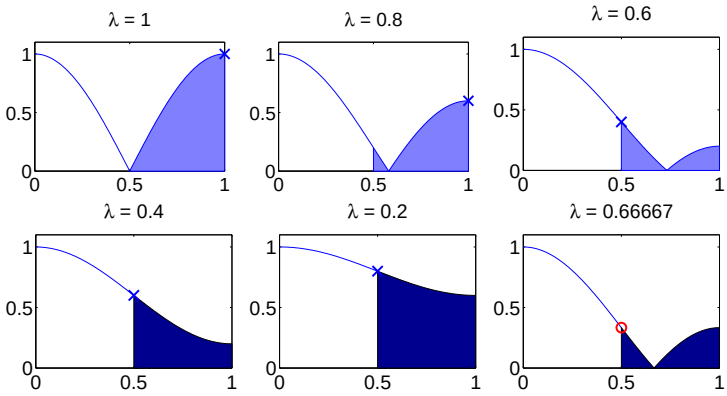

Figure 2.4: Damped Jacobi frequency response for 2^{nd} order approximation to the Laplace equation. Largest magnitude frequency response in ($\frac{1}{2} \leq \omega \leq 1$) for given λ marked with $\times$ or o (also see fig. 2.5)

The objective will therefore be to minimize the maximal value of the frequency response in this area.

$$\min \sup\{|F_\lambda(\omega)| \mid \omega \in \omega_{high}, 0 \leq \lambda\} \tag{2.66}$$

For this particular frequency response, the maximum absolute value in the given interval is found in the end of the interval (ie. $\omega = \frac{1}{2}$ or $\omega = 1$). Hence the optimization problem reduces to

$$\min \max(|F_\lambda(\frac{1}{2})|, |F_\lambda(1)|) \tag{2.67}$$

$$F_\lambda(\frac{1}{2}) = 1 - \lambda(1 - \cos(\frac{\pi}{2})) = 1 - \lambda \tag{2.68}$$

$$F_\lambda(1) = 1 - \lambda(1 - \cos(\pi)) = 1 - 2\lambda \tag{2.69}$$

Above equations are visualized in fig. 2.5. The minimal maximum frequency response in the domain is $1/3$ and is obtained with $\lambda = 2/3$. $\lambda = 2/3$ is also the suggested weight of A Multigrid Tutorial [2].

In order to determine if the algorithm is convergent also for high order approximations, the local Fourier analysis generalized for approximations to the Laplace equation. The weighted neighborhood sum and corresponding filter used in the Jacobi update will in general be given by

$$U_i^{[k+1]} = \frac{1}{c_0}\left[\sum_{k=-\infty}^{-1} U_{i+k}^{[k]} c_k + \sum_{k=1}^{\infty} U_{i+k}^{[k]} c_k\right] \tag{2.70}$$

$$f(n) = \frac{1}{c_0}\left[\begin{array}{ccccccc} \cdots & c_{-2} & c_{-1} & \underline{0} & c_1 & c_2 & \cdots \end{array}\right] \tag{2.71}$$

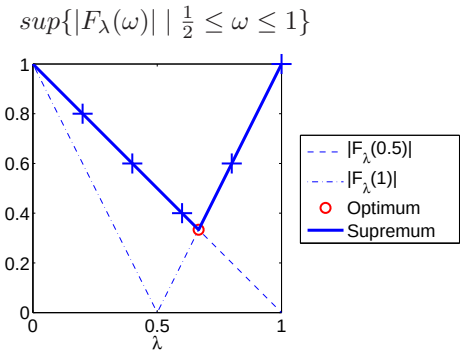

Figure 2.5: Maximum value of frequency response in high frequency interval ($\frac{1}{2} \leq \omega \leq 1$) depends on λ. Laplace equation 2^{nd} order approximation. Samples shown in fig. 2.4 marked with $+$.

with corresponding frequency response

$$F(\omega) = \frac{... + c_{-2}e^{-2i} + c_{-1}e^{-i} + 0 + c_1 e^{i} + c_2 e^{2i} + ...}{c_0}$$

The particular values of c_k can be calculated using the previously mentioned procedure `fdcoeffF` by Fornberg. The frequency response for damped Jacobi is found by insertion of (2.70) into (2.62). Figure 2.6 shows frequency responses for various *central* approximations with various values of λ. For the 3D problem, off-centered approximation are needed near the surface. The central element of the stencil will always be element no. 2 regardless of order. For the very same reason the 2^{nd} order approximation will never be 'too wide'. Figure 2.7 shows frequency responses for off-center stencils. The frequency response for some of the high order off-center approximations have values larger than 1 regardless of λ. The interpretation is that damped Jacobi will be divergent for those stencils. When the smoother is divergent, CGC will also be divergent. The conclusion is that damped Jacobi is an insufficient smoother for the linear Laplace problem if the spatial approximation order is greater than 4.

2.2.2 Restriction and prolongation

In the previous Section it was illustrated how the Jacobi method acts as a smoothening filter. As pointed out in section 2.1.1, CGC seek to eliminate the high frequency error components and deal with low frequency components. In order to form high frequency components from low frequency components, the grid values are transfered to a coarse grid. A grid transfer corresponds for

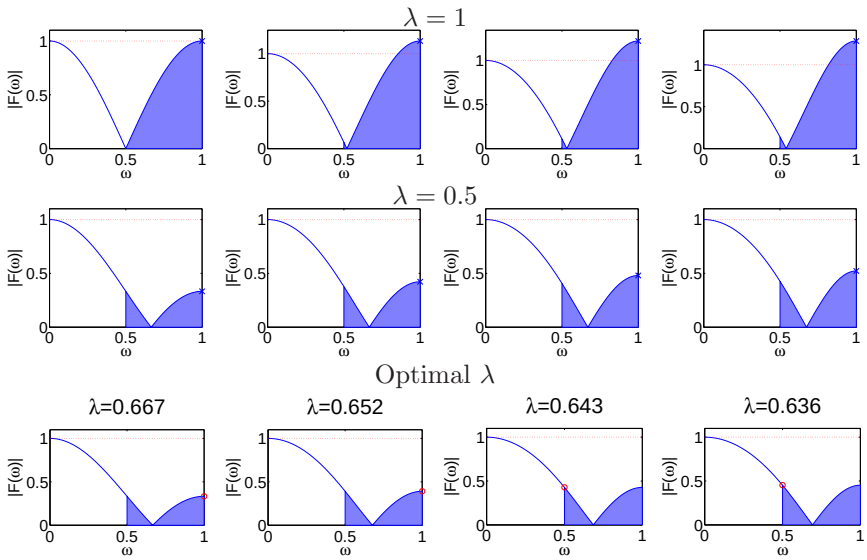

Figure 2.6: Frequency response magnitude of damped Jacobi using different weights and with various *central* approximations to the ordinary Laplace equation. From left approximation orders are 2, 4, 6 and 8.

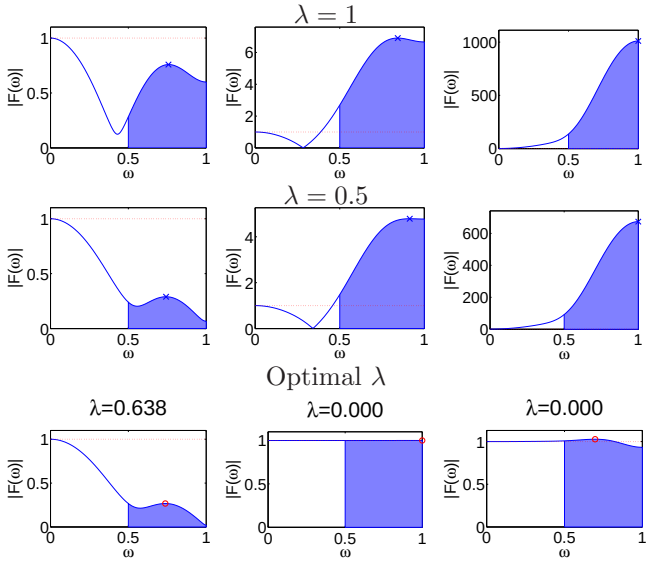

Figure 2.7: Frequency response magnitude of damped Jacobi using different weights and with various *asymmetric* approximations to the ordinary Laplace equation. From left approximation orders are 4, 6 and 8.

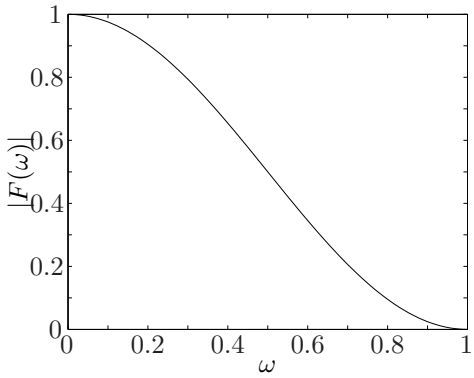

Figure 2.8: Frequency response magnitude for full weight restriction - high frequencies are removed although not very effectively.

DSP to re-sample the signal. Throughout this project, only matching grids are considered. The re-sampling rate for restriction is therefore given by

$$\frac{n_{coarse}}{n_{fine}} = \frac{\frac{n-1}{2}}{n} \to \frac{1}{2} \quad \text{for } n \to \infty$$

When a signal is re-sampled, the Fourier form of the signal is either stretched or compacted. For down-sampling (restriction), the signal is stretched by the down-sampling factor. The down-sampling factor is here always 2 since the number of grid points is always halved. If the signal carriers high frequencies, they will result in aliasing components which is a potential problem: The aliasing components might result in a frequency response which is larger than 1 which imply that the scheme is divergent. Fortunately, full weighting help limiting the aliasing components since it serve as a pre re-sampling filtering. In the FW used, the weighting of the grid points is the tensor product of

$$\frac{1}{4}[1 \ \underline{2} \ 1] \tag{2.72}$$

In 1D, the response is given by

$$\frac{1}{4}z^{-1} + \frac{1}{2} + \frac{1}{4}z^{1} = \frac{1}{2} + \frac{1}{2}\cos\ \omega\pi \tag{2.73}$$

The frequency response is illustrated in fig. 2.8 and seem to approximate a high cut filter thus reduce aliasing components.

When the signal is up-sampled (prolongated), the frequency response is compacted by the up-sampling factor. For up-sampling a number of zeros are injected into the signal; if the up-sampling factor is the integer I, $I-1$ zeros are injected between every signal value. Resulting, a lot of aliasing components

are introduced to the signal and the signal power is reduced by a factor of I. FW corresponds to add a post filtering process to the re-sampling. For the FW procedure used in this project, the filter and corresponding response is the same as the pre down-sampling filtering ((2.72) and (2.73)).

2.2.3 Combining smoother and grid transfer procedures

We have now analyzed the basic components of Coarse Grid Correction individually. The remaining question is how they perform when used together. For that purpose we will use the convolution theorem: Convolution in the spatial domain corresponds to multiplication in the frequency domain. In particular for a signal $f(n)$, convolved with two filters h_1 and h_2 with frequency response H_1 and H_2 respectively,

$$\hat{f}(n) = (f * h_1 * h_2)(n) = (f * h_2 * h_1)(n) \Leftrightarrow \tag{2.74}$$

$$\hat{F}(\omega) = (F \cdot H_1 \cdot H_2)(\omega) = (F \cdot H_2 \cdot H_1)(\omega) \tag{2.75}$$

The frequency response of the entire system is thus calculated by simple multiplication of the Fourier form of the individual operators. A 1 grid correction scheme will have total system frequency response $\mathcal{R}$

$$\mathcal{R} = S^{\nu_1} \cdot R_{FW} \cdot \mathcal{S} \cdot I_{FW} \cdot S^{\nu_2} \tag{2.76}$$

$$\tag{2.77}$$

Here $\mathcal{S}$ represent the response of the coarse grid solver. S, R, and I are the frequency responses of the smoother, restriction and prolongation procedures. More important, the magnitude of $\mathcal{R}$ is bounded by the magnitude of the components:

$$|\mathcal{R}| \leq |S|^{\nu_1} \cdot |R_{FW}| \cdot |\mathcal{S}| \cdot |I_{FW}| \cdot |S|^{\nu_2} \tag{2.78}$$

The coarse grid solver ideally drive the error to zero for the coarse grid. For practical purposes, it is sufficient to perform just a number of additional smoothings.

2.3 3D Local Fourier Analysis

The goal of the project is to create an efficient solver for a 3D non linear problem. Up until now it is shown how Local Fourier Analysis can be used to analyze how the solver works and how it performs in a 1D case. This will now be generalized to 3D. In the upcoming sections LFA will be used to analyze both the simple Jacobi smoother and some of the better smoothers presented in section 2.1.5.

The important tool is still the Fourier transform. For short notation, the Z-transform will be used rather than the complex exponential

$$2D: F(z) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f(k,l) z_x^k z_y^l \tag{2.79}$$

$$3D: F(z) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} f(k,l,m) z_x^k z_y^l z_\sigma^m \tag{2.80}$$

The Z-transform is a generalization of the Fourier transform; insertion of $z = e^{i\omega\pi}$ reveal the Fourier transform. The Z-transform applied to some 2D filter is shown below

$$\begin{bmatrix} f_{-1,-1} & f_{-1,0} & f_{-1,1} \\ f_{0,-1} & \underline{f_{0,0}} & f_{0,1} \\ f_{1,-1} & \overline{f_{1,0}} & f_{1,-1} \end{bmatrix} \tag{2.81}$$

$$\downarrow$$

$$F = sum\left(\begin{bmatrix} z^{-l}z^{-k}f_{-1,-1} & z^{-l}f_{-1,0} & z^{-l}z^{k}f_{-1,1} \\ z^{-k}f_{0,-1} & f_{0,0} & z^{k}f_{0,1} \\ z^{l}z^{-k}f_{1,-1} & z^{l}f_{1,0} & z^{l}z^{k}f_{1,1} \end{bmatrix} \right) \tag{2.82}$$

$$\begin{aligned} F = {} & z_x^{-k} z_y^{-l} f_{-1,-1} + z_y^{-l} f_{0,-1} + z_x^k z_x^{-l} f_{1,-1} + z_x^{-k} f_{-1,0} \\ & + f_{0,0} + z_x^k f_{1,0} + z_x^{-k} z_y^l f_{-1,1} + z_y^l f_{0,1} + z_x^k z_y^l f_{1,1} \end{aligned} \tag{2.83}$$

For the analysis we will use the forward and backward stencils introduced in section 2.1.5.

Given the frequency responses F^+ and F^- for S^+ and S^-, the frequency response associated with the method is then found by

$$F = -\frac{F^-}{F^+} \tag{2.84}$$

The frequency responses in the different directions are important; if the high cut effect is little in a direction which will be restricted, aliasing components will ruin the signal informations and result in overall poor convergence of the algorithm. In order to investigate the effectiveness of a smoother, the partial frequency response magnitude is introduced. The partial frequency response

magnitudes are given by

$$|F_x(\omega_x)| = \sup_{T_x} |F(\boldsymbol{\omega})| = \sup |F(\boldsymbol{\omega})| \quad , 0 \leq \omega_y < 1, 0 \leq \omega_\sigma < 1 \tag{2.85}$$

$$|F_y(\omega_y)| = \sup_{T_y} |F(\boldsymbol{\omega})| = \sup |F(\boldsymbol{\omega})| \quad , 0 \leq \omega_x < 1, 0 \leq \omega_\sigma < 1 \tag{2.86}$$

$$|F_\sigma(\omega_\sigma)| = \sup_{T_\sigma} |F(\boldsymbol{\omega})| = \sup |F(\boldsymbol{\omega})| \quad , 0 \leq \omega_x \leq 1, 0 \leq \omega_y \leq 1 \tag{2.87}$$

2.3.1 Damped Jacobi method

We will now look at the response for the damped Jacobi method applied to the transformed Laplace equation. Using the stencil from the ordinary Jacobi method ((2.16)) and the distributive properties of the Fourier transform demonstrated for (2.62), the response of the damped Jacobi is found to

$$F_J = 1 - \lambda + \lambda \frac{\frac{1}{(\Delta x)^2}(z^{-k} + z^k) + \frac{1}{(\Delta y)^2}(z^{-l} + z^l) + \frac{(\partial_z \sigma)^2}{(\Delta \sigma)^2}(z^{-m} + z^m)}{-\left(-2\left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{(\partial_z \sigma)^2}{(\Delta \sigma)^2}\right)\right)} \tag{2.88}$$

$$F_J = 1 - \lambda + \lambda \frac{\frac{1}{(\Delta x)^2}\cos\omega_x\pi + \frac{1}{(\Delta y)^2}\cos\omega_y\pi + \frac{(\partial_z \sigma)^2}{(\Delta \sigma)^2}\cos\omega_z\pi}{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{(\partial_z \sigma)^2}{(\Delta \sigma)^2}} \tag{2.89}$$

In the 1D case we saw that the optimal smoothing factor was given at $\lambda = 2/3$. This is not the case for 3D since the oscillatory modes for the other direction mixes has an influence. The problem can be reduced by insertion of $\frac{\Delta\sigma}{\partial_z \sigma} = \epsilon \Delta y = \epsilon \Delta x$ where ϵ controls the anisotropy of the stencil. Notice that $\frac{\Delta\sigma}{\partial_z \sigma}$ correspond to the physical step-length in the vertical direction; $\frac{\Delta\sigma}{\partial_z \sigma} = \frac{\Delta\sigma}{d} = \Delta z$.

$$F_J = 1 - \lambda + \lambda \frac{\frac{1}{(\Delta x)^2}\cos\omega_x\pi + \frac{1}{(\Delta x)^2}\cos\omega_y\pi + \frac{1}{\epsilon^2(\Delta x)^2}\cos\omega_\sigma\pi}{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta x)^2} + \frac{1}{\epsilon^2(\Delta x)^2}} \tag{2.90}$$

$$= 1 - \lambda + \lambda \frac{(\cos\omega_x\pi + \cos\omega_y\pi)\epsilon^2 + \cos\omega_\sigma\pi}{2\epsilon^2 + 1} \tag{2.91}$$

$$\xrightarrow{\epsilon \to 0} 1 - \lambda + \lambda \cos\omega_\sigma \tag{2.92}$$

$$\xrightarrow{\epsilon \to \infty} 1 - \lambda + \lambda \frac{\cos\omega_x\pi + \cos\omega_y\pi}{2} \tag{2.93}$$

$$\tag{2.94}$$

The frequency response presented here shows the response of the full filter. The partial frequency response magnitude in the σ-direction is given by

$$|F_{J,\sigma}| = \sup_{T_\sigma} |1 - \lambda + \lambda \frac{(\cos\omega_x\pi + \cos\omega_y\pi)\epsilon^2 + \cos\omega_\sigma\pi}{2\epsilon^2 + 1}| \tag{2.95}$$

$$= \max |1 - \lambda + \lambda \frac{\pm 2\epsilon^2 + \cos\omega_\sigma\pi}{2\epsilon^2 + 1}| \tag{2.96}$$

$$\xrightarrow{\epsilon\to 0} |1 - \lambda + \lambda \cos\omega_\sigma| \tag{2.97}$$

$$\xrightarrow{\epsilon\to\infty} = 1 \tag{2.98}$$

The same can be done for the other directions as well. The important conclusion to draw here is that if the step length in one direction is much larger than the others ($\epsilon \to \infty$), damped Jacobi will be highly ineffective in smoothing in that direction regardless of λ. From (2.96) we get that the dampening factor for the high frequency range of the partial response is found to

$$|F_{J,\sigma}| = \max |1 - \lambda + \lambda \frac{2\epsilon^2 + 0}{2\epsilon^2 + 1}|, |1 - \lambda + \lambda \frac{-2\epsilon^2 - 1}{2\epsilon^2 + 1}| \tag{2.99}$$

$$= \max |1 - \lambda \left(1 - \frac{2\epsilon^2}{2\epsilon^2 + 1}\right)|, |1 - 2\lambda| \tag{2.100}$$

This points out that $\lambda < 1$ should be chosen in order to ensure convergence since $\lambda < 1 \Rightarrow |F_{J,\sigma}| < 1$. As in the 1D case, we can find a value of λ which maximizes the damping of the high frequency range. This time it will obviously depend on ϵ. Since $0 < \frac{2\epsilon^2}{2\epsilon^2+1} < 1 \ \forall\epsilon$, the optimal value of λ is found in the interval $\frac{1}{2} \leq \lambda \leq 1$. Hence the maximal smoothing is found at

$$1 - \lambda \left(1 - \frac{2\epsilon^2}{2\epsilon^2 + 1}\right) = 2\lambda - 1 \tag{2.101}$$

$$\lambda \left(3 - \frac{2\epsilon^2}{2\epsilon^2 + 1}\right) = 2 \tag{2.102}$$

$$\lambda = \frac{2}{3 - \frac{2\epsilon^2}{2\epsilon^2+1}} \tag{2.103}$$

Although this is the optimal λ for damping the partial frequency response, the problem is coupled to the remaining partial frequency responses as well: Altering ϵ affects how the remaining directions are smoothed. Since λ depends on ϵ, it is non-trivial to choose the optimal damping. For isotropic stencils ($\epsilon = 1$), the optimal value of λ is given by (2.103) though:

$$\lambda_{\epsilon=1} = \frac{2}{3 - \frac{2}{2+1}} = \frac{6}{7} \tag{2.104}$$

By insertion into (2.100), the partial frequency response magnitude is found to

$$F_{J,\epsilon=1} = 2\lambda_{\epsilon=1} - 1 = 2\frac{6}{7} - 1 = \frac{5}{7} \tag{2.105}$$

2.3.2 Lexicographic Gauss-Seidel

Using a forward and backward stencil notation, the 1D GSLEX applied to the Laplace problem takes form

$$S^+U^{t+1} + S^-U^t = 0 \tag{2.106}$$

$$S^+ = [\ 1 \quad \underline{-2} \quad 0\ ], \quad S = [\ 0 \quad \underline{0} \quad 1\ ] \tag{2.107}$$

The corresponding frequency response of the 1D GSLEX is then given by

$$F^- = 0z^{-1} + 0 + 1z^1 \tag{2.108}$$

$$F^+ = 1z^{-1} - 2 + 0z^1 \tag{2.109}$$

$$F_{GSLEX} = -\frac{z}{z^{-1} - 2} \tag{2.110}$$

The 3D stencil is given by (2.39). The corresponding frequency response is found to

$$F^- = \frac{1}{(\Delta x)^2}z_x + \frac{1}{(\Delta y)^2}z_y + (\partial_z\sigma)^2\frac{1}{(\Delta\sigma)^2}z_\sigma \tag{2.111}$$

$$F^+ = \frac{1}{(\Delta x)^2}(z_x^{-1} - 2) + \frac{1}{(\Delta y)^2}(z_y^{-1} - 2) + (\partial_z\sigma)^2\frac{1}{(\Delta\sigma)^2}(z_\sigma^{-1} - 2) \tag{2.112}$$

$$F_{GSLEX} = -\frac{\frac{1}{(\Delta x)^2}z_x + \frac{1}{(\Delta y)^2}z_y + (\partial_z\sigma)^2\frac{1}{(\Delta\sigma)^2}z_\sigma}{\frac{1}{(\Delta x)^2}(z_x^{-1} - 2) + \frac{1}{(\Delta y)^2}(z_y^{-1} - 2) + (\partial_z\sigma)^2\frac{1}{(\Delta\sigma)^2}(z_\sigma^{-1} - 2)} \tag{2.113}$$

Inserting $\frac{\Delta\sigma}{\partial_z\sigma} = \epsilon\Delta y = \epsilon\Delta x$ we can analyze the smoothing properties for

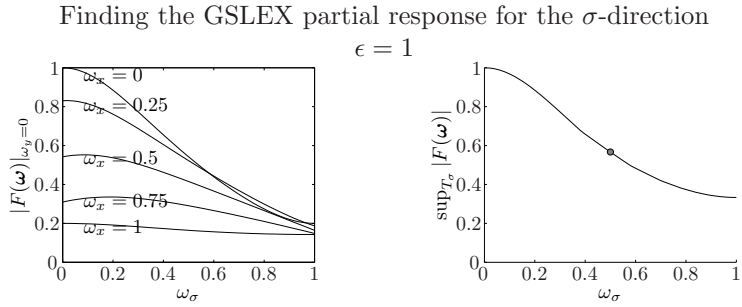

Figure 2.9: The partial response is found numerically by evaluation of $|F_\sigma|$ at various $\boldsymbol{\omega}$ (left). The maximum partial high frequency response is given by $\max \sup_{T_y} |F(\boldsymbol{\omega})|, 0.5 \leq \omega_\sigma < 1$, (right, marked with a circle).

anisotropic stencils.

$$F = -\frac{\frac{1}{(\Delta x)^2} z_x + \frac{1}{(\Delta x)^2} z_x + \frac{1}{\epsilon^2 (\Delta x)^2} z_\sigma}{\frac{1}{(\Delta x)^2}(z_x^{-1} - 2) + \frac{1}{(\Delta x)^2}(z_y^{-1} - 2) + \frac{1}{\epsilon^2 (\Delta x)^2}(z_\sigma^{-1} - 2)} \tag{2.114}$$

$$= -\frac{(z_x + z_y)\epsilon^2 + z_\sigma}{(z_x^{-1} + z_y^{-1} - 4)\epsilon^2 + z_\sigma^{-1} - 2} \tag{2.115}$$

$$\xrightarrow{\epsilon \to 1} \frac{z_x + z_y + z_\sigma}{6 - z_x^{-1} - z_y^{-1} - z_\sigma^{-1}} \tag{2.116}$$

$$\xrightarrow{\epsilon \to 0} \frac{z_\sigma}{2 - z_\sigma^{-1}} \tag{2.117}$$

$$\xrightarrow{\epsilon \to \infty} \frac{z_x + z_y}{4 - z_x^{-1} - z_y^{-1}} \tag{2.118}$$

GSLEX also prove inefficient for highly anisotropic stencils; the dampening in the σ-direction here vanishes for $\epsilon \to \infty$. For equally sized step lengths, the smoother is better than damped Jacobi though; using a simple numerical method (illustrated by fig. 2.9, the damping factor at $\epsilon = 1$ is found at $z_x = z_y \approx e^{i\frac{1}{10}\pi}, z_\sigma = e^{i\frac{1}{2}\pi}$.

$$D_{GSLEX,\epsilon=1}(e^{i\frac{1}{10}\pi}, e^{i\frac{1}{10}\pi}, e^{i\frac{1}{2}\pi}) \approx 0.57 \tag{2.119}$$

This is $(\frac{1-0.57}{1-0.71})$ 62% better than the damped Jacobi method.

2.3.3 Line smoother

A line smoother is applied in one direction with a point smoothing strategy in the others. Using damped Jacobi in the horizontal directions, the method can be presented in forward stencil notation:

$$S^- = \frac{1}{h_x^2}[\ 1 \quad \underline{0} \quad 1\ ]_x + \frac{1}{h_y^2}[\ 1 \quad \underline{0} \quad 1\ ]_y + (\partial_z\sigma)^2\frac{1}{h_\sigma^2}[\ 0 \quad \underline{0} \quad 0\ ]_\sigma \tag{2.120}$$

$$S^+ = \frac{1}{h_x^2}[\ 0 \quad \underline{-2} \quad 0\ ]_x + \frac{1}{h_y^2}[\ 0 \quad \underline{-2} \quad 0\ ]_y + (\partial_z\sigma)^2\frac{1}{h_\sigma^2}[\ 1 \quad \underline{-2} \quad 1\ ]_\sigma \tag{2.121}$$

The frequency response of the σ-line Jacobi smoother is then given by

$$F^- = \frac{1}{h_x^2}(z_x + z_x^{-1}) + \frac{1}{h_y^2}(z_y + z_y^{-1}) \tag{2.122}$$

$$F^+ = \frac{1}{h_x^2}(-2) + \frac{1}{h_y^2}(-2) + (\partial_z\sigma)^2\frac{1}{h_\sigma^2}(z_\sigma - 2 + z_\sigma^{-1}) \tag{2.123}$$

$$F = -\frac{\frac{1}{h_x^2}(z_x + z_x^{-1}) + \frac{1}{h_y^2}(z_y + z_y^{-1})}{-\frac{1}{h_x^2}2 - \frac{1}{h_y^2}2 + (\partial_z\sigma)^2\frac{1}{h_\sigma^2}(z_\sigma - 2 + z_\sigma^{-1})} \tag{2.124}$$

We again use $\frac{h_\sigma}{\partial_z\sigma} = \epsilon h_y = \epsilon h_x$ to analyze the smoothing properties of the algorithm

$$F = \frac{\frac{1}{h_x^2}2\cos\omega_x\pi + \frac{1}{h_x^2}2\cos\omega_y\pi}{\frac{1}{h_x^2}2 + \frac{1}{h_x^2}2 + \frac{1}{\epsilon^2 h_x^2}(2 - 2\cos\omega_\sigma\pi)} \tag{2.125}$$

$$= \frac{(\cos\omega_x\pi + \cos\omega_y\pi)\epsilon^2}{2\epsilon^2 + 1 - \cos\omega_\sigma\pi} \tag{2.126}$$

$$\xrightarrow{\epsilon\to 0} \begin{cases} 0 & \omega_\sigma \neq 0 \\ \frac{\cos\omega_x\pi + \cos\omega_y\pi}{2} & \omega_\sigma = 0 \end{cases} \tag{2.127}$$

$$\xrightarrow{\epsilon\to\infty} \frac{\cos\omega_x\pi + \cos\omega_y\pi}{2} \tag{2.128}$$

The partial response in the direction of the line smoothing differs from the other directions.

$$|F_\sigma| = \sup_{T_\sigma}\left|\frac{(\cos\omega_x\pi + \cos\omega_y\pi)\epsilon^2}{2\epsilon^2 + 1 - \cos\omega_\sigma\pi}\right| \tag{2.129}$$

$$= \frac{2\epsilon^2}{2\epsilon^2 + 1 - \cos\omega_\sigma\pi} \tag{2.130}$$

$$\xrightarrow{\epsilon\to 0} \begin{cases} 0 & \omega_\sigma \neq 0 \\ 1 & \omega_\sigma = 0 \end{cases} \tag{2.131}$$

$$\xrightarrow{\epsilon\to\infty} 1 \tag{2.132}$$

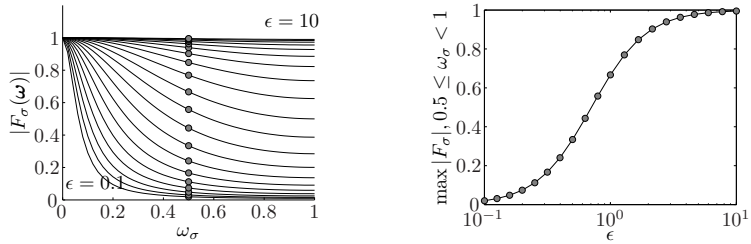

Figure 2.10: The partial frequency response (left) depend on ϵ. The maximum partial high frequency response therefore also depend on ϵ (right).

Decreasing the step size in the direction of the line smoother increases the smoothing effect of the line smoother. If the step size on the other hand is larger than in the other directions, the smoothing factor is very limited which is illustrated for large ϵ in fig. 2.10.

The smoothing in the other directions is not very different from the point smoother used.

$$|F_{xy}| = \sup \left| \frac{\frac{1}{h_x^2} 2\cos\omega_x\pi + \frac{1}{h_x^2} 2\cos\omega_y\pi}{\frac{1}{h_x^2} 2 + \frac{1}{h_x^2} 2 + \frac{1}{\epsilon^2 h_x^2}(2 - 2\cos\omega_\sigma\pi)} \right|, \quad 0 \leq \omega_\sigma < 1 \tag{2.133}$$

$$= \frac{|\cos\omega_x\pi + \cos\omega_y\pi|\epsilon^2}{2\epsilon^2} \tag{2.134}$$

$$= \frac{|\cos\omega_x\pi + \cos\omega_y\pi|}{2} \tag{2.135}$$

It is immediately noticed that this is also the size of the frequency repose magnitude of the 2D Jacobi point smoother. It is therefore reasonable that applying a line smoother in some direction generally improves the partial smoothing in all directions. For the Jacobi method in particular we already know that damping the iterate will improve the smoothing properties. Although a damping factor will improve the smoothing effect of $\max |F_{xy}|$, it will reduce the smoothing effect of $|F_\sigma|$.

2.4 Results of the analysis

Regardless of smoother, the grid should never be restricted in a given direction if the error is not sufficiently smooth; the signal will be disturbed by aliasing effects otherwise. It is therefore crucial that the error is sufficiently smooth in a given direction prior restriction. The smoothness of the error is unknown but

knowing the smoothing properties of the smoother, it is possible to estimate when it will be save to restrict in a certain direction.

We have seen that stencil anisotropy has big influence on the high cut effect of both line and point smoothers. Generally for point smoothers, the smoothing is best in the directions with the largest value of $1/(\Delta x)^2$, $1/(\Delta y)^2$ and $\sigma_z^2/(\Delta\sigma)^2$, corresponding to the direction with the smallest physical step length. For a vertical line smoother, the vertical stencil asymmetry has no influence on smoothing in the horizontal directions although the horizontal anisotropy has influence on the smoothing in the vertical direction.

The smoothness of the error in a particular direction is not known directly but knowing how the smoother performs, a reasonable guess can be made to which direction is save to restrict.

Since the horizontal effectiveness of the smoother rely on the step lengths, it is a logical choice to let the coarsening strategy depend on the step lengths as well. Algorithm 5 was developed to provide an all purpose coarsening strategy for xy-rectangular grids when a vertical line smoother is used. The algorithm take the effectiveness of a point smoother into regard when choosing if restriction along a particular direction should be allowed. A single parameter α determines how large a difference in step-length is allowed for the algorithm to perform coarsening in a particular direction. Setting $\alpha = 0$, the algorithm provides an always semi-coarsening strategy. Setting $\alpha = \infty$, the algorithm will restrict all directions whenever possible. During this project, $\alpha = 1$ ($\max(\Delta x, \Delta y) \leq 2\min(\Delta x, \Delta y)$) has proven to be a general good choice; the method is not seen to be divergent and the memory usage is lower than for $\alpha = 0$. The overall algorithm efficiency with respect to α as not investigated further.

Algorithm 5 All purpose coarsening strategy

1: assert(N_x, N_y, N_z all odd)
2: $\beta \leftarrow \alpha + 1$
3: $limit \leftarrow \beta \min(\Delta x, \Delta y)$;
4: **if** $\Delta x \leq limit \wedge \left(\frac{N_x - 1}{2} \mod 2\right) == 0$ **then**
5: $\quad \widehat{N}_x \leftarrow \frac{N_x - 1}{2} + 1$
6: **end if**
7: **if** $\Delta y \leq limit \wedge \left(\frac{N_y - 1}{2} \mod 2\right) == 0$ **then**
8: $\quad \widehat{N}_x \leftarrow \frac{N_y - 1}{2} + 1$
9: **end if**
10: **if** $\frac{\Delta\sigma}{\sigma_z} \leq limit \wedge \left(\frac{N_z - 1}{2} \mod 2\right) == 0$ **then**
11: $\quad \widehat{N}_x \leftarrow \frac{N_z - 1}{2} + 1$
12: **end if**

Regardless of coarsening strategy, the physical step size $\Delta z = {}^{\Delta\sigma}/_{\sigma_z}$ will most often be the smallest at the coarse grid levels; the solver is expected to be applied on physical domains which are large in the x and y directions and small in the z direction. It is therefore vital to use a line smoother at the coarse grid levels in order to obtain an efficient smoothing in the horizontal directions. It is on the other hand an open question whether it will be better to use a line smoother when the step size in x and y directions are smaller than in z.

One thing, the analysis has not given direct answers to is the loss of information in grid transfers: Whenever the grid values are transfered from a fine to a coarse grid, some information might be lost due to aliasing effects and truncation errors. If a particular residual has a very low frequency and therefore not propagated into the high frequencies until after a series of grid-transfers, much of the information might get lost in the process. The resulting error correction will then try and correct something which might not be the original residual and introduce even more error to the problem. This is discussed further in section 6.3.

Chapter 3

C for CUDA

There is no big difference from a device program or kernel to an ordinary C program. One is written in C and the other in C for CUDA which resembles C. When an ordinary C-program is compiled, it is translated into low-level and somewhat more complex (and architecture dependent) Assembler code which is a low-level programming language. The Assembler code will be translated to machine code/object code subsequently and finally object code is linked with the rest of the program to a binary file (runtime library or executable). When a device program is compiled, it is translated into PTX (Parallel Thread eXecution) code which is much alike Assembler code. The PTX code is compiled into a binary file (often postfixed cubin for CUDA binary) which is then loaded and executed by the device on demand. It is possible to write programs directly in PTX which is 'as easy' as writing programs in Assembler or manual perform post parsing optimizations to the code. In order to load the CUDA binary onto the device, nVidia supplies the CUDA Driver API. The CUDA driver API is a lower-level C API which provides functions to load kernels as modules of CUDA binary or assembly code, to inspect their parameters, and to launch them [4].

The CUDA Driver API might for some be a bit cumbersome since it requires a great deal of code in order to execute even simple kernels. To ease things up, nVidia developed the CUDA Runtime API which works on top of the CUDA Driver API. For the Runtime API, initialization, context, and module management are all implicit and resulting code is more concise [4]. As for the

compilation procedure, the PTX and CUDA binary is by default embedded in the compiled object files and later on also in the executable. If needed, the compiler can be told to instead generate separate PTX and/or cubin files which will then be loaded into the program at runtime [3]. For this particular compilation configuration, it is possible to take advantage of the conciseness of the runtime API and in the same time perform manual tuning of the PTX files.

Regardless of either CUDA Driver API or CUDA Runtime API is used, the kernel execution is asynchronous to the CPU. After invoking a kernel call, the CPU process immediately returns to the remainder of the program. CPU and GPU synchronization done explicitly[1] and presently the units are synchronized using busy-wait (spinning if one wants). It is possible through events to reduce the busy-wait to polling.

There is no doubt that C for CUDA and in general HPC on GPUs is still being developed as it has not yet come to a state of maturity; although the runtime API is built on top of the driver API, it was not possible to have interoperability between driver and runtime API until the recent release of CUDA 3.0. C for CUDA is also extended to include bindings to other languages than C/C++ which at present count .Net Framework (CUDA.NET) and Java (Jacuzzi Project). Fortran is also able to use CUDA although through C; linking CUDA code with Fortran is basically the same as calling C/C++ from Fortran.

3.1 Tesla and Fermi hardware architecture

In order to develop efficient parallel programs on a GPU it is important to understand the hardware; solid knowledge about the hardware often point out the parts of a program which will gain performance from a GPU implementation. In this section a brief explanation of the physical hardware layout is presented. A graphical overview of the connectivity is presented in fig. 3.1

In the NVIDIA GPU devices, there are two types of computational components: The stream processor and the special functions unit. The stream processor handles simple operations such as multiplication, addition subtraction and comparison. Advanced operations as reciprocal, reciprocal square root, log2x, 2x, sin and cos are sent to the special functions unit. In contrast to most CPUs, the registers are shared among a number of processors (usually 8) and assigned to the processor by a controlling unit upon execution. The controlling unit is called

[1] E.g. `cudaMemcpy(...)` has built in barrier synchronization.

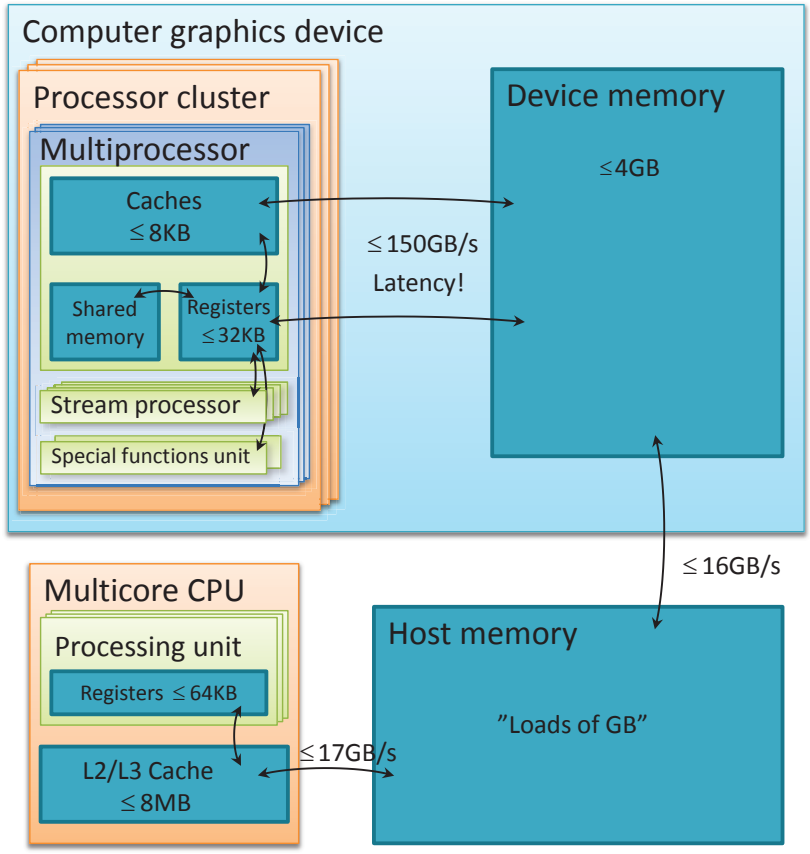

Figure 3.1: Graphical overview of GPU/CPU connectivity.

a multiprocessor and further contains caches for read-only memory and a special shared memory. In general, there is no latency for memory transfers within the controller although the actual transfer takes 4 clock cycles. Transfers to or from the device memory also takes 4 clock cycles but has a high latency on top of that (400-600 clock cycles). The shared memory (often referred to as being 'on chip') is as fast to access as registers but is only in use if the programmer explicitly asks for it. Shared memory can be thought of as manually controlled cache. Further, the multiprocessor is controlled by a thread processor cluster. The number of multiprocessors per thread processor cluster ranges from 2 to 4 within existing devices but is of no significant importance to code optimization. Although the number of multiprocessors can be ignored, the number of active threads should exceed the number of stream processors. [4]

Cache sizes, number of registers and the amount of shared memory, constant memory and cache of a computer graphics device is summarized by the compute capability (CC) of the device. The compute capabilities existing today are versions 1.0, 1.1, 1.2, 1.3 (Tesla hardware architecture) and 2.0 (Fermi hardware architecture). In general it is important to know the compute capability of the target device since some optimizations will rely on the capability of the device. A full description of the compute capabilities is available in the Cuda Programming Guide [4]. The specifications of installed devices are easily obtained with the program `deviceQuery.exe` found in the CUDA SDK.

3.1.1 Device and host memory

Memory transfer between host and device(s) is a basic step in GPU-programming. Allocating the correct type of memory on the host means a lot to the bandwidth obtained in host/device communication.

Ordinary memory allocation

- Normal access behavior for CPU
- Slowest data transfer rates to/from device
- Synchronous data transfers only

Pinned (page locked) memory

- Always resides in system memory (i.e. cannot be paged)
- Bandwidth between device and host thread allocating memory is higher
- Supports asynchronous data transfer (streaming) between host and device
- cudaAllocHost allocates pinned memory

Portable memory

- Extension to pinned memory
- Higher bandwidth available from all host threads to device

Write-combined memory (in extension to pinned)

- Extension to pinned memory
- Not cache-able at host (reading is 'prohibitively slow' [CUP231])
- Fastest data transfer rate between host and device (up to 40% faster than pinned memory)

Mapped memory

- Extension to pinned memory. On some devices, host memory can further be mapped directly onto the device. This means that streaming is unnecessary as memory transfer is implicitly performed as the kernel needs it. As device reads and writes memory asynchronous, the programmer has to be aware of data hazards[2].

The intension of the project is to work with problems which can reside in the GPU memory all at once. The relatively slow host/device communication is therefore omitted if the outputs does not need to be stored very often.

3.1.2 Special device memory and cache

A GPU is a mixed SIMD[3], SPMD[4] processor and has a far more advanced cache layout than on a regular CPU. On the GPU, threads are clustered in execution groups called blocks. The number of threads in a single block as well as the total number of blocks associated with a kernel invocation or grid is assigned dynamically by the programmer at runtime. On the GPU, the smallest active group at the time is a block; a block counts in the scale of hundreds of threads which proceed in lock step groups when scheduled. Although not all lock step groups will be active at the same time, all threads of the entire block will need to have registers assigned.

The number of registers available to a shared multiprocessor is constant but the number of registers used by a single thread depend on the kernel. Hence the number of active blocks will be dependent on the number of registers needed for a single block and the total number of registers available per multiprocessor. For compute capability 1.3, the number of registers per shared multiprocessor is

[2]Write immediately followed by write or read might not be considered safe
[3]Single Instruction, Multiple Data
[4]Single Program, Multiple Data

8192 Bytes. If a kernel requires 32 registers and there are 64 threads in a block, the maximum number of active blocks is $\frac{8192}{32 \cdot 64} = 4$.

Each block also has access to a cache shared among the threads in a block. This is referred to as shared memory. As for the registers, the amount of shared memory in use depend on the program. But where the number of registers is given implicitly by program size, the shared memory is dynamically allocated upon invocation of a kernel. The shared memory is also a limited resource and the available amount is specified by the compute capability. For CC 1.3, the amount of shared memory per multiprocessor is 16384 Bytes. As with registers, this can also be a limiting factor to the number of active blocks on the GPU: For a kernel which uses 5400B shared memory, the maximum number of active blocks is limited to $\frac{16384}{5400} = 3$.

For the Fermi architecture, the size of shared memory and L2 cache[5] can be adjusted by the programmer. Current configurations allow for 48/16 (default), 32/32 or 16/48 division of the 64KB in the combined memory.

3.1.3 Execution order and synchronization

Each kernel is associated with a grid which has in the scale up to billions of blocks. The blocks are executed in no particular order but each block is associated with a block index. The block indices are either 1D or 2D depending on the launch configuration. The same applies to threads; there is no particular order in which the lock step groups proceed and each thread is also associated with an id within the block. The thread indices are either 1D, 2D or 3D, again depending on launch configuration.

As mentioned, the threads proceed in lock step groups. Such groups are called warps and are currently of size 32 regardless of compute capability. The maximum number of active warps is determined by the compute capability of the device. For compute capability 1.3 the maximum is 32 active warps and for compute capability 2.0, 48 warps. The total number of ready[6] warps depends on block size and the number of active blocks; if 3 blocks are active and they have 64 threads each, the number of ready warps must then be $(64 \cdot 3)$ 192.

Section 3.1 describe how local memory access is associated with high latency. The GPU tries to hide that latency by simply putting the warp on hold until the global memory is ready. While the warp is paused, the multiprocessor switches

[5] L2 cache is not available in Tesla devices

[6] A ready process will be either active, paused or waiting for the processor to work on the process

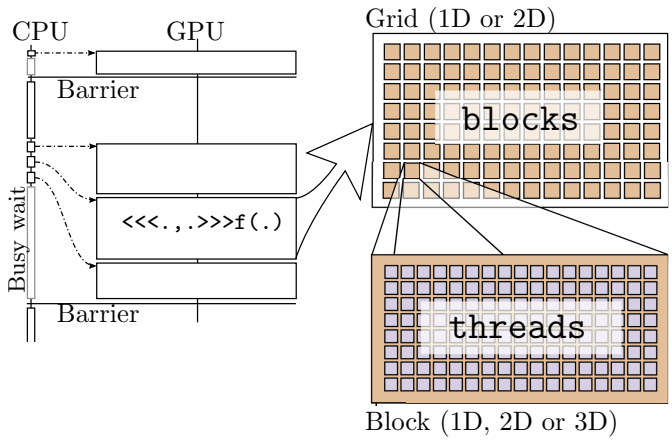

Figure 3.2: CPU kernel invocation is asynchronous to GPU execution and synchronization is done using busy wait. Upon kernel invocation, the number of blocks and threads per block is specified by the special CUDA parameter list `<<<blocks, threads>>>func(...)`.

context to another warp waiting to be scheduled. Contrary to CPUs, a GPU has a low cost of switching thread context. If enough warps are ready, it is possible for the GPU to hide the latency enough to not suffer too much from the global memory latency. Global memory access is not the only operation associated with high latency. Latency is also introduced when a warp is waiting for other warps (thread barriers, atomic operations) or for some advanced computation to complete (cos, sin etc.). Whenever possible, the GPU automatically tries to hide the latency so having enough ready warps is generally important to obtain fast code. Preferably, the block size should be chosen such that the number of warps exceed the maximum number of active warps supported by hardware.

The global memory latency is associated with the transfer of memory - not the size of the transfer. In other words, one large transfer is to prefer rather than a bunch of small transfers. The number of transfers needed to transfer all requested memory to the warp depend on the match of memory alignment which is 128B. When neighbor threads request memory which is 'near by', the transfer is coalesced (see fig. 3.3).

Every 32 consecutive threads belong to the same warp. For any possible block layout, the thread number is found following way: A block layout of size $T_x \times T_y \times T_z$ is chosen as launch configuration. For a given thread, the id (t_x, t_y, t_z)

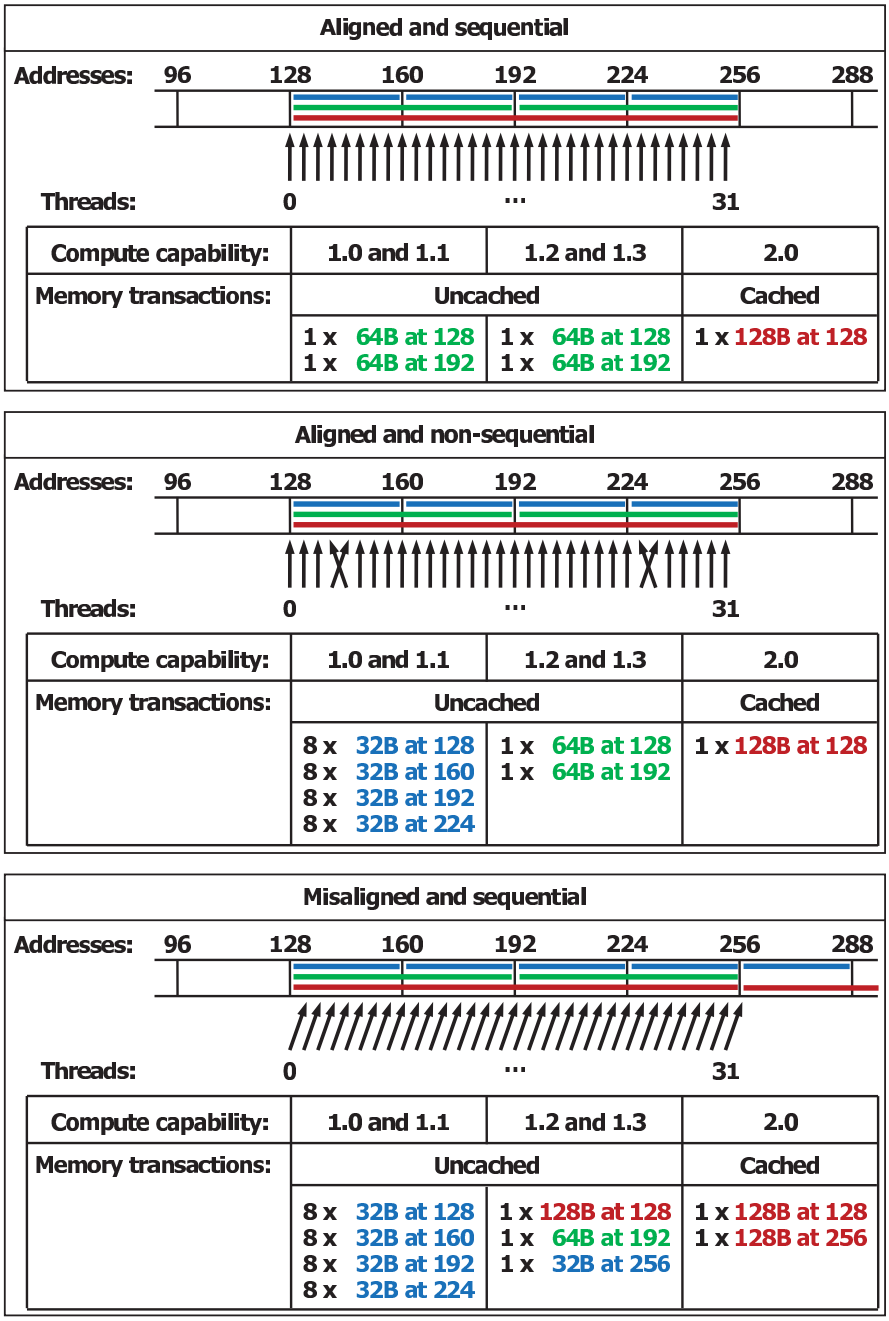

Figure 3.3: Examples of global memory accesses by a warp, 4-Byte word per thread, and associated memory transactions based on compute capability. Courtesy of NVIDIA Corporation

is given. The thread number is then given by

$$t_n = t_x + (t_y + t_z T_y)T_x \tag{3.1}$$
$$0 \leq t_x < T_x, \quad 0 \leq t_y < T_y, \quad 0 \leq t_y < T_y$$

Thread number 0-31 belong to warp 0, thread 32-63 to warp 1 etc.

Many algorithms has a certain overlap of memory requirement for neighboring elements. As explained in section 3.1.2, the GPU features a shared memory which is basically a set of shared registers. When neighboring elements need the same pieces of memory, the use of shared memory is very often beneficial to reduce the number of global memory transfers required. Although faster, the shared memory is aligned to and accessed through 16 banks/gates. One bank can serve only one thread at the time. Although threads in a warp proceed as one lock step group, the shared memory is accessed by a half-warp[7] at the time. If several threads access memory from the same bank, the hardware splits the request into as many separate conflict-free requests as necessary. A bank conflict can occur only within a half-warp.

[7]First half-warp consist of threads 0-14 and the second, threads 15-31

Chapter 4

Implementation

The intension of the project is to work with problems which can reside in the GPU memory all at once. Disregarding the need for storing data at certain time steps, the essential data transfers are therefore only kernel parameters (which is asynchronous) and transfer of the calculated defect norm $||\boldsymbol{d}||$ which is only a few bytes. The host is thus reduced to being a GPU task scheduler which for large problems will spend most of the time waiting for the GPU. A kernel cannot be a particularly large program, so the number of kernels will be in the scale of 10-20.

In order to solve the problem, there are two parts which must be dealt with: The transformed Laplace problem and the IVP. It is not possible to solve the IVP without solving the Laplace problem and it is therefore logical to begin with the implementation of Defect Correction and its preconditioner (Coarse Grid Correction). Coarse Grid Correction again depend on other components as a smoother, restriction and prolongation. The prioritized order of implementation will thus be

1. Choose a memory layout
2. Implement and validate the components of Coarse Grid Correction
3. Implement and validate Coarse Grid Correction

4. Implement and validate Defect Correction
5. Implement and validate RK4 method
6. Implement and validate advanced CGC components
7. Optimize bottle neck components

Validation of the procedures is important due to the complexity of the algorithm; even simple bugs will take an unreasonable amount of time to fix if they can not be isolated to a certain part of the code. To reduce the debugging time, the initial implementation should also address only basic requirements for the algorithm; it is far easier to improve a working implementation rather than try to construct the optimized program from the beginning. Optimizations will be addressed in chapter 5.

In order to validate the procedures, it is a good idea to implement the same algorithms in a mathematical inspired environment such as Matlab or Octave. Although time consuming to implement the algorithm more than once, the benefit of having a mathematically correct version is not to underestimate. Debugging a parallel program is in itself non trivial and adding additional layer of complex mathematical algorithms will not make it any easier. With a few exceptions, the entire program was therefore implemented in Matlab prior to implementation in C for CUDA. The implementation in Matlab will not be discussed further.

In order to ensure correctness, the resulting system matrix was compared to previous works of A. Engsig-Karup . Further, the result of a few time steps were compared to the previously verified program to ensure that the algorithms produce similar results. In section 6.3 the Laplace solver is verified and 6.2 and the total algorithm validated.

4.1 Definitions, utility functions and execution safety

The CUDA programming model encourage to use thread indices to determine which element should be processed. In chapter 3 it is described how threads are launched in blocks. The threads index and current block index is given by the hardware. Often the global thread index will be needed rather than the local block thread index. Throughout the program we will therefore use the preprocessor definitions

```
#define tx threadIdx.x
#define ty threadIdx.y
#define Tx threadIdx.x + blockIdx.x*blockDim.x
#define Ty threadIdx.y + blockIdx.y*blockDim.y
```

For the most cases, a 2D launch configuration is chosen. It is therefore convenient to have a general way to calculate the number of blocks needed as a function of the block configuration; the block configuration is an optimization parameter. When the total number of threads should be $N_x \times N_y$, the minimal number of blocks required is calculated by

```
inline dim3 fitblock(dim3 threads, int Nx, int Ny){
  ASSERT(threads.z == 1);
  dim3 blocks;
  blocks.x = Nx/threads.x + (Nx % threads.x > 0 ? 1:0);
  blocks.y = Ny/threads.y + (Ny % threads.y > 0 ? 1:0);
  blocks.z = 1;
  return blocks;
}
```

Since threads are launced in blocks of a fixed size $B_x \times B_y$, the maximum global thread id will thus be

$$\max T_x = t_x + B_x \cdot b_x$$

where t_x and b_x are local thread and block indicies. Since N_x/B_x will not very often be an integer, there will be excess threads. The very first thing to do for all procedures presented in the remainder of the chapter is therefore a safety check.

Listing 4.1: Safety check; is current thread inside the physical domain $N_x \times N_y$?

```
if(!(Tx < N.x && Ty < N.y)){
  return; //Stop! Outside physical domain
}
```

There should be a safety check for EVERY kernel implemented. The condition may vary but there should be a safety check never the less. E.g. the ghost update, the limits will be determined by the total grid size `M` including ghost points rather than `N` (grid size without ghost points).

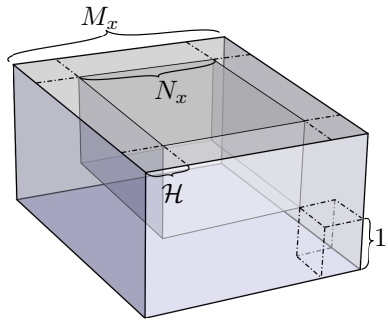

Figure 4.1: Grid size is given by its physical/inner $N_x \times N_y \times N_z$ grid points and the number of ghost layers in the horizontal directions. There is always only 1 layer of ghost points in the vertical direction.

4.2 Memory Layout and access

In section 1.3 we conclude that only the vector of the matrix vector product should be stored. Since the vector and the grid of the physical domain represent the same, we will from here think of the vector as the underlying 3D grid. The grid sizes will generally be given by

$$M_x \times M_y \times M_z \tag{4.1}$$

$$M_x = N_x + 2\mathcal{H}, \quad M_y = N_y + 2\mathcal{H}, \quad M_z = N_z + 1 \tag{4.2}$$

$$\mathcal{H} = \left\lfloor \frac{\text{stencilwidth}}{2} \right\rfloor \tag{4.3}$$

$\mathcal{H}$ will also be referred to as the halo width or the number of ghost layers in the horizontal direction. When a particular grid size is mentioned, only the number of inner grid points are specified. E.g. $257 \times 21 \times 6$ reffers to the a grid with total size $(257 + 2\mathcal{H}) \times (21 + 2\mathcal{H}) \times (6 + 1)$. A grid is illustrated in fig. 4.1.

Although convenient working only with the 3D grid, storage is 1D. In order to access the grid in an intuitive (x, y, z)-kind of manner, a 3D $\leftrightarrow$ 1D mapping is therefore needed. The Laplace problem is defined by its velocity potential at the surface and will thus be needed when evolving the IVP. Since storage is linear, the easiest (and for CUDA also the fastest) way to access the velocity potential at the surface is to let the surface be defined as $M_x \times M_y$ consecutive grid values. The z-coordinate will thus have to be the slowest varying in the mapping of $(x, y, z) \to n$ where n is the linear memory index. Whether x or y coordinate should be the fast index is not important since the Laplace problem is rotation invariant. The x coordinate is chosen to vary faster than y such a C/C++ array access style is adopted. For practical reasons, the surface values should be allocated in the beginning of the array such that no offset is needed

to get the surface values. The $3D \to 1D$ mapping is thus given by

$$(x, y, z) \to n : n = x + yM_x + zM_xM_y \tag{4.4}$$

In a CUDA program, elements benefit from being accessed according to the thread id. Typical 2D access could be $(x + o_x, y + o_y, z)$ where x and y relate to the thread id. It is therefore convenient to introduce offset indexing denoted $(o_x, o_y, z)_{x,y}$. The conversion from offset indexing to regular indexing will depend on how the threads are associated with the grid. For a 2D block layout with one thread per surface grid point (including ghost points), offset indexing is given by

$$(o_x, o_y, z)_{x,y} \to n : n = (T_x + o_x) + (T_y + o_y)M_x + zM_xM_y \tag{4.5}$$

Listing 4.2: Implementation of (4.5)

```
__device__ inline
int memoryIdxXY(int ox, int oy, int z, int3 M){
  return (Tx + ox) + (Ty + oy)*M.x + (Tz + z)*M.x*M.y;
}
```

For a 2D block layout with one thread per surface grid point in the physical domain (ie. ghost points excluded), offset indexing is given by

$$(x, y, z) \to n : n = (x + G_x) + (y + G_y)M_x + zM_xM_y \tag{4.6}$$

Rather than $\mathcal{H}$, G_x and G_y are used as the number of ghost points in the horizontal directions.

Listing 4.3: Implementation of (4.6)

```
__device__ inline
int memoryIdxXY(int ox, int oy, int oz, int3 M){
  return (Tx + Gx + ox) + (Ty + oy)*M.x + oz*M.x*M.y;
}
```

Having decided on the memory layout it is time to move on to implementation of the methods.

4.3 Finite difference estimates

To estimate derivatives we use the finite difference sums calculated by `fdcoeffF`. The stencils will be pre-calculated by the CPU since they do not change. Since

the GPU processes the data by a SPMD approach, all stencils will need to be available to all threads at all times. We will therefore store the stencils in the same chunk of memory since the number of arguments must be static. The stencil used will assume a step-length of size 1; the same stencil can therefore be used for both x, y and z-direction as long as it is scaled with the appropriate step length.

For a size n stencil, we will create all n possible stencils and store them consecutively in memory. In particular, for the stencil approximating the first derivative, we get

$$S^{[1]} = [\ \underline{c_1} \quad c_2 \quad \cdots \quad c_n\ ] = \texttt{fdcoeffF(1,1,1:n)} \tag{4.7}$$

$$S^{[2]} = [\ c_1 \quad \underline{c_2} \quad \cdots \quad c_n\ ] = \texttt{fdcoeffF(1,2,1:n)} \tag{4.8}$$

$$\vdots$$

$$S^{[n]} = [\ c_1 \quad c_2 \quad \cdots \quad \underline{c_n}\ ] = \texttt{fdcoeffF(1,n,0:n)} \tag{4.9}$$

$$\mathit{stored} : [\ S^{[1]} \quad S^{[2]} \quad \cdots \quad S^{[n]}\ ] \tag{4.10}$$

A brief summary of the CUDA implementation of the finite difference estimates is available in section D.1. The stencils are stored in global device memory and the a pointer is passed as a function argument.

4.4 Basic components of Coarse Grid Correction

4.4.1 Updating ghost points

In section 1.2.1 we introduce ghost points. The purpose of ghost points is to ensure that the Neumann boundary conditions are fulfilled. Most of the finite difference operators will be preceded by making sure that the ghost points actually do fulfill the boundary conditions for the current values. Since the ghost points will need to be updated for many grid operators, it makes good sense to treat the ghost updates separately and call the procedure when needed.

There are two general cases of grids which will have ghost points. One is the 3D transformed Laplace equation. The other is the 2D surface elevation prior to approximation of spatial derivatives. For the 3D non linear case, the bottom corners of the domain will need special attention as described in section 1.2.1. A sample code for the 2D and 3D ghost update method is presented in section D.2.

4.4.2 Low order residual

The low order residual is a required component of the Coarse Grid Correction algorithm. Implementing the low order residual serve a second purpose as well: The difference between the residual calculation and the damped Jacobi is very little. The low order residual method should calculate

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{\Phi}, \quad \boldsymbol{b} = [\tilde{\phi}\ \ \boldsymbol{0}]^T$$

where $\boldsymbol{A}$ is the 2^{nd} order approximation of the linear transformed Laplace equation. Recall that the vectors $\boldsymbol{\Phi}$, $\boldsymbol{b}$ and $\boldsymbol{r}$ are represented as 3D variables and $\boldsymbol{A}$ is generated when needed using the stencil methods presented in section 4.3.

In the residual calculation, the still water depth $\boldsymbol{h}$ is needed. It varies over the domain but is the same for all elements in the a vertical column. From [4] it is experienced that global memory transfers are relatively slow and it is therefore considered the only option to let a thread process an entire column at the time; $\boldsymbol{h}$ cannot be reused otherwise. We will therefore adopt a 2D launch configuration. The initial kernel is presented in section D.3.

4.4.2.1 Validation

The best way to evaluate the residual is to ensure that $\boldsymbol{A}$ is correct. This is a bit tricky since $\boldsymbol{A}$ is not given directly. It can be calculated though; the residual is given by

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$$

where $\boldsymbol{A}$ is a 2^{nd} order linear approximation to the transformed Laplace equation. Calculating the residual when $\boldsymbol{b} = \boldsymbol{0}$ and a single element in $\boldsymbol{x}$ is -1 and the remaining 0 will extract a single column of $\boldsymbol{A}$:

$$\boldsymbol{r}_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{1,2} \\ a_{2,2} \\ a_{3,2} \end{bmatrix} \tag{4.11}$$

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{r}_1 & \boldsymbol{r}_2 & \boldsymbol{r}_3 \end{bmatrix} \tag{4.12}$$

The resulting matrix can then be inspected manually or compared to the result of another validated matrix generation procedure.

4.4.3 Damped Jacobi method

Damped Jacobi is a component of the Coarse Grid Correction algorithm. The damped Jacobi procedure should implement either of

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \lambda \boldsymbol{D}^{-1}\boldsymbol{r}, \quad \boldsymbol{x} \equiv \boldsymbol{\Phi} \tag{4.13}$$

$$\boldsymbol{x}^{k+1} = (1-\lambda)\boldsymbol{x}^k + \lambda \boldsymbol{D}^{-1}(-(\boldsymbol{L}+\boldsymbol{U})\boldsymbol{x}^k + \boldsymbol{b}) \tag{4.14}$$

The easiest way to implement the Jacobi method is by reusing the low order residual procedure. The obvious choice is therefore to implement (4.13).

Although the damped Jacobi is convergent regardless of the initial guess to $\boldsymbol{x}$, we see no point in not to use that the value of $\boldsymbol{x}$ at the surface simply is the value of $\boldsymbol{b}$. The implementation of the damped Jacobi method is therefore modified a bit:

$$\boldsymbol{x}^{k+1} = \begin{cases} \boldsymbol{b} & \text{for surface values} \\ \boldsymbol{x}^k + \lambda \boldsymbol{D}^{-1}\boldsymbol{r} & \text{elsewhere} \end{cases} \tag{4.15}$$

4.4.3.1 Validation

Given that the residual computation procedure is correct, it is very difficult to get the damped Jacobi method wrong. To validate the procedure, convergence must be established:

$$\boldsymbol{e} = \boldsymbol{x} - \hat{\boldsymbol{x}} \tag{4.16}$$

$$|\boldsymbol{e}| \to 0 \quad \text{for increased iteration count} \tag{4.17}$$

where $\boldsymbol{x}$ is the analytic solution to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ and $\hat{\boldsymbol{x}}$, the Jacobi iterate. The convergence should have asymptotic $O(n^{-1})$ behavior where n is the number of iterations.

4.4.4 Restriction

The restriction procedure should implement the method introduced in section 2.1.1. Restriction is done by calculation of an average in a neighborhood of every second fine grid element in the direction(s) of the coarsening and store the result in a coarse grid. [4] recommend to make several small programs rather than one big. The coarsening procedure will therefore consist of a series of kernels which will be invoked for the particular coarsening strategy. This

gives totally 7 coarsening strategies; 3 for coarsening in one direction, 3 for coarsening in 2 directions and finally the one for coarsening in all directions. A simple way to construct the kernel is by a gather scheme. A brief overview of the implementation of the 2D and 3D coarsening implementation is available in section D.5.

4.4.4.1 Validation

It should be checked if the numeric integral of the coarse grid variable approximate the integral of the fine grid variable; (2.1) must be true. A good debugging tool for the restriction procedure is to simply just visualize the restricted grid.

4.4.5 Coarse Grid Correction

Coarse Grid Correction is a method using alot of different components. This component implement the invocation strategy of coarse grid correction; the μ-cycle procedure should implement algorithm 3. Implementation provided in section D.6.

4.5 Advanced Coarse Grid Correction components

4.5.1 Red Black Gauss Seidel

RBGS is no doubt a method worth including; the method is a very efficient smoother. RBGS can be implemented by replacing of all $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ with just $\boldsymbol{x}$. For the red/black part, lines 2-8 of the code below should be merged into the Jacobi method as well.

```
1 for(int Z = 1; Z<N.z;Z++){
2   bool colored = (Tx % 2 == 0) ^ (Ty % 2 == 0) ^ (Z % 2 == ↩
      0);
3
4   if(mode == ISRED){
5     if(colored) continue;
6   }else{
```

```
7       if(!colored) continue;
8     }
9
10    //...
11    //remaining Jacobi algorithm
12    //...
13  }
```

In order to perform both 'red' and 'black' update, the method should be invoked twice;

```
1  rbgs(x, ISRED, ...);
2  ghostupdate(x, ...);
3  rbgs(x, ISBLACK, ...);
```

4.5.2 Line smoother

A line smoother is basically an extension to a point smoother. The line smoother should implement algorithm 4 presented in section 2.1.5.3. We will implement the algorithm in two steps: In step one we will collect the contents of $\boldsymbol{a}_{-1}$, $\boldsymbol{a}_0$, $\boldsymbol{a}_1$ and $\tilde{\boldsymbol{b}}$ and in step two we will solve the system. In order to solve the local system $\widehat{\boldsymbol{G}}^{+}\boldsymbol{u}_\sigma = \tilde{\boldsymbol{b}}$ on the GPU, we will have to allocate memory for the $\boldsymbol{a}$-arrays. We will do so with local arrays. Unfortunately this has the disadvantage that (slow) local memory will be used; only if the arrays are accessed in a monotonic increasing fashion, the arrays can be stored in (fast) registers. On the other hand, of the arrays are stored in registers, the register use will then increase with $8N_\sigma$ since we need 4 arrays (one double use 2 registers). For single point precision only half the number of registers will be needed though. The largest discomfort of using local arrays is that they must be allocated compile time thus exceed the largest number of vertical grid elements ever used. By experiment it was found that the array size could be set to 128 double precision elements with no drop in performance so this is not considered a problem.

Implementation notes on the σ-line xy-Jacobi smoother is available in section D.8. An xy-RBGS variant can be created subsequently in the same way as the ordinary RBGS smoother described in section 4.5.1.

4.5.2.1 Validation

A line smoother can be validated in the same way as any other smoother; see section 4.4.3.1. The convergence properties of the line smoother is expected to be the same as for the underlying point smoother although the error is expected to decrease faster for the first few iterations, even if the vertical step length is smaller than the horizontal.

4.6 Defect Correction

The defect correction algorithm in itself is not very complex. It uses basically four components: Calculation of the high order residual, invocation of the preconditioner, calculating a sum and finally the norm of the defect. The problematic component here, is calculation of the norm; the norm is a (transformed) reduction[1] of the data and therefore not necessarily embarrassing parallel. Introduction of a non embarrassingly parallel algorithm can potentially ruin the $O(n)$ scaling properties. This is not a problem in practice though: Given N_t threads, N_e total elements, the execution time is determined by

$$\text{reduction time} = \max(0, N_e - N_t)O(1) + O(\log N_t) \tag{4.18}$$

In general N_e will be much (at least hundreds of times) larger than N_t and the algorithm is therefore expected to be $O(N_e)$ thus still scale. The parallel reduction sum was implemented using Thrust which is a library containing various efficient algorithms implemented in CUDA.

It has come clear that the Thrust method has Rendezvous invocation style; the methods do not return until the kernel has done its work due to the call of synchronous methods somewhere down the line. It is of no importance for this program since the CPU is not intended to do work concurrent with the GPU but for general usage of the Thrust library, this is a rather important detail.

4.6.1 Various order residual

The various order residual is a required component of the Coarse Grid Correction algorithm and should calculate

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A\Phi}, \quad \boldsymbol{b} = [\tilde{\phi}\ \ \boldsymbol{0}]^T$$

[1] Many elements are reduced to just one.

where $\boldsymbol{A}$ is the n^{nd} order approximation of the transformed Laplace equation. Recall that the vectors $\boldsymbol{\Phi}$, $\boldsymbol{b}$ and $\boldsymbol{r}$ are represented as 3D variables and $\boldsymbol{A}$ is generated when needed.

Calculating the non linear various order residual is much more complicated than the linear low order residual; various stencil lengths are needed to be taken into account and surface variables considered as well. Since the surface variables (η, h and derivatives hereof) for an entire vertical column at the time, it make sense to process the high order residual column by column..

The validation procedure for the non linear high order residual procedure is the same as for the linear low order residual procedure; see section 4.4.2.1.

4.7 Surface evolution and model validation

In section 1.1.1 it was argued that a 4 stage Runge Kutta solver is sufficiently stable. Prior to the evaluation of (1.1) and (1.2) the transformed Laplace equation will need to be solved for the current surface velocity potential such $\tilde{\omega}$ can be estimated. Since the derivatives of the surface variables are needed but only surface elevation is known, they should be estimated first. The procedure is sketched in algorithm 6. The initial guess for the Laplace solver (Defect Correction) should be the last available solution to the problem, here denoted $\boldsymbol{\Phi}^*$. A brief summary of the implementation is given in section D.10.

Algorithm 6 Evaluation of temporal derivatives $\partial_t \eta$ and $\partial_t \tilde{\phi}$

1: Estimate spatial derivatives of h and η
2: Solve transformed Laplace equation $\rightarrow \boldsymbol{\Phi}$ (initial guess $\boldsymbol{\Phi}^*$)
3: Estimate $\tilde{\omega} \leftarrow \partial_z \sigma \boldsymbol{\Phi}_\sigma|_{\sigma=1}$
4: Estimate spatial derivatives of $\tilde{\phi}$
5: Use estimations to evaluate (1.1) and (1.2)

Implementing the RK4 method is from here trivial; letting $\boldsymbol{k}$ being a tensor with $\boldsymbol{\eta}_t$ and $\boldsymbol{\phi}_t$, the standard 4 step procedure presented in section 1.1.1 can be used.

4.7.1 Validation

We are in the fortunate situation to have a complete code to compare to and supportive solid knowledge about the topic; a good but informal way to validate

not just the Runge Kutta method but the entire program is to reconstruct results from physical results. One such test is the Whalin shoal problem where waves are generated in one end of a bounded domain and absorbed in the other using relaxation zones as described in [6]. Halfway through the domain, the bottom rises such the water depth is lowered with a resulting amplification of the wave magnitude closely behind the underwater hill. The Runge Kutta method thus include a relaxation step after each estimate of the next surface elevation and velocity potential. The altered RK4 method is described by algorithm 7.

Algorithm 7 RK4 with relaxation zone update

$\boldsymbol{x} \equiv \{\eta, \tilde{\phi}\}$
$\boldsymbol{\Phi}^* = \boldsymbol{0}$
repeat
    $\boldsymbol{k}_1 \leftarrow \Delta t \boldsymbol{f}(\boldsymbol{x}, t)$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \frac{\boldsymbol{k}_1}{2}$
    Update relaxation zones of η, and $\tilde{\phi}$.
    $\boldsymbol{k}_2 \leftarrow \Delta t \boldsymbol{f}(\boldsymbol{x}, t + \frac{\Delta t}{2})$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \frac{\boldsymbol{k}_2}{2}$
    Update relaxation zones of η, and $\tilde{\phi}$.
    $\boldsymbol{k}_3 \leftarrow \Delta t \boldsymbol{f}(\boldsymbol{x}, t + \frac{\Delta t}{2})$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{k}_3$
    Update relaxation zones of η, and $\tilde{\phi}$.
    $\boldsymbol{k}_4 \leftarrow \Delta t \boldsymbol{f}(\boldsymbol{x}, t + \Delta t)$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \frac{1}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)$
    Update relaxation zones of η, and $\tilde{\phi}$.
until end of time

Chapter 5

Code optimization

After the initial version of a program is established, bottleneck procedures should be optimized in order to utilize the hardware platform better. The difference between a naive GPU implementation and a fully optimized program can be several factors. Since the program is quite complex (there are approximately 20 different individual components) it will be out of scope of this project to take all aspects of the program into consideration. For optimization, we will target Compute Capability 1.3 devices and more specifically a Quadro FX 5800 GPU will be used for benchmarking. The device differs from Tesla C1060 by having a graphical output unit and a different price but basically it has the same specifications.

Optimizing kernel code should be done in a series of steps. A prioritized list of general optimization strategies is presented below

(i) Reduce algorithmic complexity; i.e. check if expensive operations can be removed by altering the algorithm

(ii) Minimize the use of global memory. Prefer reusing and shared the variable through shared memory where possible

(iii) Ensure global memory accesses are coalesced whenever possible

(iv) Avoid different execution paths within the same warp

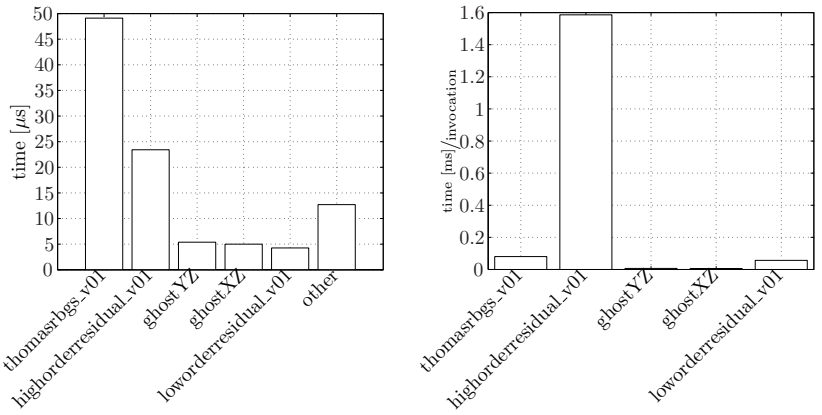

Figure 5.1: Initial profiling of the CUDA implementation. 10 time steps for the Whalin shoal (gridsize $257 \times 21 \times 6$) are benchmarked using the CUDA profiler. The smoother is called 3700 times and the high order residual only 89 times

(v) Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts

(vi) To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy).

(vii) Optimize the launch configuration

(viii) Prefer faster, more specialized kernels over slower, more general ones when possible.

Item (iv) in particular has been taken into account from an early stage of the project: Avoiding to eliminate the ghost points from the equations, all low order finite difference sums can be calculated without branching.

Prior to optimization, we will use the CUDA Profiling tool to get a quick overview of the current bottlenecks. The CUDA Profiler provide an easy and precise measuring of the kernel execution time. Kernel executions are not slowed down by the profiler although the hosting program is. A profiling of the initial program using a point smoother or a line smoother is provided in fig. 5.1. The average execution time for the line smoother is low compared to the high order residual procedure. The line smoother is called totally 3700 times on various sized grids and the high order residual is called 89 times and only on the fine grid.

5.1 Optimization strategy

Obviously, the smoothers and the high order residual are bottleneck problems of the algorithm. Although not directly a bottleneck, we will start with the low order residual; the high order residual and the line smoother are much more costly, but they are unfortunately also very much more complex. In order to understand how the shared memory can be used, we will therefore begin with optimization of the low order residual. The optimizations done and conclusions on this procedure can further be directly applied to the smoothers with only little additional effort.

In order to increase the effective bandwidth of any algorithm the global memory transfers should be minimized (item (ii) on the prioritized list). This might involve use of the shared memory which can be rather tricky to get right for complex problems. It is therefore a good idea to make an estimate of the scale of an eventual speedup of the method prior to implementation rather than head straight on to implementation. The listings below gives an example of such practice.

Listing 5.1: Some unoptimized code

```
1 _ftype sum = 0;
2 for(int i = 0; i<somenumber;i++){
3   sum += global[n+i];
4 }
5 sum = some_other_global[n]*lengthy_calculation(sum);
```

Listing 5.2: Optimal read redundancy

```
1 _ftype sum = 0;
2 _ftype g = global[n];
3 shared_memory[0] = g;
4 __syncthreads();
5 for(int i = 0; i<somenumber;i++){
6   sum += shared_memory[0];
7 }
8 sum = some_other_global[n]*lengthy_calculation(sum);
```

Using a fixed element of the shared memory bank conflicts are always avoided (list item (v)). Generally, the memory access should reflect how the optimized version is expected to be. For this particular code it is relevant to use the shared memory only for one of the two global memory transfers, which the dummy code reflects. It is important that the shared memory is used rather

than a register whenever relevant; although NVIDIA claims shared memory to be as fast as registers, it is in practice slower in cases when the access latency cannot be hidden well. The number of times some element in global memory is accessed redundantly we will refer to as the read or write redundancy. For the above code, the read redundancy of the `global` variable is `somenumber` for the unoptimized code.

Regardless of the code, the kernel launch configuration should be optimized as well (list item (i)). Even bad code gain from using an optimized launch configuration. The maximum number of threads in a launch configuration cannot exceed 512 threads[1] and the block dimension can vary freely[2] within this scope. The 2D and 3D grid variables have C/C++ access style and neighboring threads will thus benefit from coalesced memory transfers when accessing global memory collectively in this direction. To reduce the search space we will therefore seek an optimized solution in the domain

$$(B_x \times B_y) = (8 \cdot m \times n), \quad B_x B_y \leq 512 \tag{5.1}$$

$$(m, n) : \mathbb{N}^+ \times \mathbb{N}^+ \tag{5.2}$$

In appendix F we present a generalized procedure to benchmark any kernel regardless of parameters and thus be used to find an optimized kernel launch configuration. The best and worst launch configurations for the non linear various order residual, linear low order residual and RBGS line smoother on 3 different grid sizes are specified in tables 5.1, 5.3 and 5.4 for double precision and also for the various order residual also in single precision in 5.2.

The general trend is that when the grid size goes up, the configurations gets less important. As an example, the top 18 configurations for the $257 \times 17 \times 6$ grid are less than 5% from the best configuration. For the $3073 \times 257 \times 17$ grid, the top 68 configurations are within the 5% limit. Some configurations appear more often than others within the 5% level, but there is no general best choice although a total number of 32 threads is not too bad in general. Due the to the coalesced memory transfers, blocks elongated in the y-direction never appear among the best configurations. Optimally, the solver should be configured prior to launch when the grid sizes are known. Particularly important is it to get the fine grid operations configured correctly since a suboptimal configuration at that level will be expensive, particularly for large scale problems. As an example, fig. 5.7 shows that a (16,4) configuration is an okay general choice but for e.g. a $3073 \times 257 \times 17$ grid, the configuration ranks no. 75 and is 5% slower than the optimal configuration.

[1] For Compute Capability 2.0, 1024 (Fermi architecture)
[2] Threads in the z-direction cannot exceed 64

Kernel table - 6. order residual

Double precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$	
1	(88,2)	100.0%	(32,2)	100.0%	(32,2)	100%
2	(96,2)	+2.9%	(16,4)	+0.3%	(64,1)	+0.27%
3	(32,2)	+3.0%	(64,1)	+0.8%	(16,4)	+0.76%
4	(32,6)	+3.1%	(16,12)	+1.6%	(32,6)	+2.0%
5	(64,3)	+3.2%	(32,6)	+1.7%	(64,3)	+2.5%
	Bottom 5					
5	(8,15)	+145.7%	(8,3)	+105.1%	(16,5)	+83.0%
4	(8,16)	+149.0%	(8,9)	+108.0%	(8,10)	+88.0%
3	(16,1)	+168.1%	(16,1)	+161.2%	(72,1)	+100.4%
2	(8,2)	+172.9%	(8,2)	+163.6%	(24,3)	+102.4%
1	(8,1)	+367.8%	(8,1)	+413.2%	(8,9)	+108.7%

Table 5.1

Kernel table - 6. order residual

Single precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$	
1	(32,3)	100.0%	(64,1)	100.0%	(64,1)	100.0%
2	(88,1)	+0.2%	(312,1)	+0.5%	(32,2)	+0.4%
3	(96,1)	+0.3%	(320,1)	+0.6%	(320,1)	+1.1%
4	(16,6)	+0.5%	(32,2)	+1.0%	(312,1)	+1.5%
5	(40,1)	+0.6%	(160,2)	+1.1%	(160,2)	+2.0%
	Bottom 5					
5	(224,1)	+105.6%	(8,17)	+111.7%	(8,17)	+116.3%
4	(232,1)	+106.0%	(8,3)	+119.5%	(8,3)	+122.2%
3	(256,1)	+108.5%	(16,1)	+204.6%	(16,1)	+208.5%
2	(248,1)	+108.8%	(8,2)	+213.3%	(8,2)	+214.3%
1	(8,1)	+277.4%	(8,1)	+499.2%	(8,1)	+510.7%

Table 5.2

Kernel table - Low order residual

Double precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$	
1	(16,3)	100.0%	(32,1)	100.0%	(160,1)	100.0%
2	(88,2)	+0.2%	(16,3)	+1.2%	(32,1)	+0.1%
3	(48,2)	+1.0%	(144,1)	+2.1%	(144,1)	+0.5%
4	(48,1)	+1.5%	(32,7)	+2.1%	(192,1)	+0.9%
5	(16,5)	+1.9%	(16,2)	+2.2%	(80,1)	+1.0%
	Bottom 5					
5	(200,1)	+80.5%	(8,41)	+42.5%	(8,62)	+44.1%
4	(168,3)	+80.9%	(8,42)	+43.5%	(8,63)	+44.7%
3	(56,9)	+82.3%	(8,63)	+45.5%	(8,51)	+45.1%
2	(256,2)	+82.7%	(8,64)	+46.3%	(8,64)	+45.8%
1	(64,8)	+174.0%	(8,1)	+105.2%	(8,1)	+99.5%

Table 5.3

Kernel table - RBGS line smoother

Double precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$	
1	(24,2)	100.0%	(64,1)	100.0%	(32,2)	100.0%
2	(16,3)	+0.1%	(32,2)	+1.5%	(64,1)	+1.9%
3	(8,6)	+2.5%	(16,4)	+3.0%	(16,4)	+2.5%
4	(16,2)	+4.0%	(64,2)	+3.7%	(16,8)	+2.7%
5	(48,2)	+4.1%	(112,1)	+4.1%	(224,2)	+2.8%
	Bottom 5					
5	(72,6)	+147.4%	(376,1)	+44.1%	(24,1)	+42.4%
4	(88,5)	+149.7%	(24,1)	+44.5%	(8,3)	+43.3%
3	(112,4)	+153.3%	(8,3)	+44.8%	(8,2)	+47.2%
2	(24,18)	+160.3%	(8,2)	+45.9%	(16,1)	+48.0%
1	(40,11)	+161.7%	(8,1)	+169.7%	(8,1)	+174.6%

Table 5.4

5.2 Benchmarking

A kernel is either memory bound or compute bound; a memory bound kernel is limited by the memory bandwidth and a compute bound kernel, by the instruction throughput of the GPU. The expectation to at least smoothing and low order residual procedures is that they are memory bound since the number of instructions per global memory transfer is relatively low for those kernels. In order to measure the effectiveness of a kernel, it is therefore relevant to consider how effective the memory bandwidth is used. The memory bandwidth depend on the device and for the Tesla C1060/Quadro FX 5800, it is 102.4 $^{\mathrm{GB}}/_{s}$. In the world of computers, GB (10^6) is often mistaken for GiB (1024^3) and it is therefore emphasized that whenever GB is used, it is in the sense of 10^6 and not 1024^3.

Although the bandwidth is fixed, full utilization can be obtained only when global memory transfers are coalesced as described by fig. 3.3. The number to measure is therefore the effective memory bandwidth utilization. The effective bandwidth will depend on the problem and should be calculated from the number of elements processed compared to execution time.

$$\text{eff. bandwidth} = \frac{\text{memory read} + \text{memory written}}{\text{execution time}} \tag{5.3}$$

For the initial implementations of the finite difference sums, the number of elements read from global memory will depend on the stencil sizes. Instead of counting elements read, we will count the number of elements touched by a stencil during the calculation of the finite difference sum. The total transfer size also depends on the number of elements written. The number of transfers for calculation of a finite difference sum and storing in an output variable is thus given by

$$N_{read} = N_{physical} + N_{bottom} + N_{sides} \tag{5.4}$$
$$N_{physical} = N_x N_y N_z \tag{5.5}$$
$$N_{bottom} = N_x N_y \tag{5.6}$$
$$N_{sides} = 2\mathcal{H} N_z (N_x + N_y) \tag{5.7}$$
$$N_{write} = N_{physical} \tag{5.8}$$
$$\tag{5.9}$$

The effective memory transfer is thus given by

$$M_{eff.tr.} = (N_{read} + N_{write}) \cdot \texttt{sizeof(\_ftype)} \tag{5.10}$$

where `_ftype` is either `double` (8B) or `float` (4B).

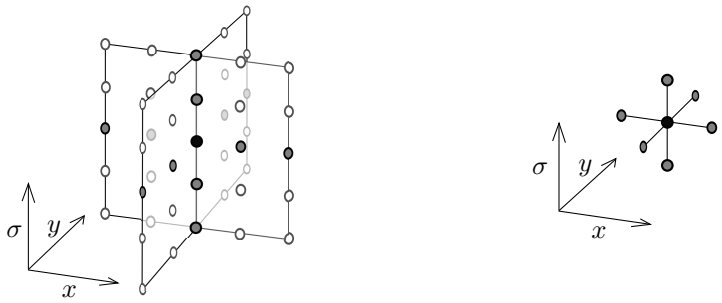

Figure 5.2: The 3D non linear various order stencil (left) touches points in the σx- and σy-planes. The 2^{nd} order linear stencil (right) touches only the nearest neighbors.

Grid size	Surface elm.s	Domain elm.s	Degrees of freedom
$257 \times 17 \times 6$	4,369	26,214	42,343
$1537 \times 129 \times 9$	198,273	1,784,457	2,083,050
$3073 \times 257 \times 17$	789,761	14,215,698	14,575,986

Table 5.5: Some numbers regarding grid sizes. The degrees of freedom is the total number of points, including ghost points ($N_x + 2 \cdot 3 \times N_y + 2 \cdot 3 \times N_x + 1$).

In section 5.1 it is seen that the execution time depends on selection of kernel launch configuration. The benchmark should represent the throughput of the algorithm given optimal conditions. Any benchmark provided is therefore measured using optimal kernel launch configuration found using the method presented in (5.1).

It is relevant to investigate whether the performance of a method depend on the elongation of the domain and benchmarks on the best method developed is therefore tested with grid sizes specified by

$$N_z = 6N_x = a \cdot N_y, a \in 1, 2, 4, 8, 16 N_y N_x \leq N_{max} \tag{5.11}$$

The halo size depend on the method in question.

5.3 Low order residual

Before considering shared memory, we will improve the procedure by reduction of register usage and also remove unnecessary operations; the low order residual always use the same stencil and it is therefore most likely wasted clock cycles to perform the computation using a loop. Fixing the kernel to just one particular order, the dynamic stencils which are in global memory are stripped from the

Low order residual
Double precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(48,1)	98 ns	(64,1)	5.4 ms	(32,1)	43 ms	x1
Est. limit	(16,3)	48 ns	(16,3)	1.9 ms	(16,4)	14 ms	x2-3
A	(16,3)	49 ns	(32,6)	1.9 ms	(32,1)	12 ms	x2-3.5
B	(32,8)	110 ns	(32,6)	2.8 ms	(32,6)	17 ms	x0.9-2.5

Table 5.6: Est. limit: What can be done with shared memory? A: Reduced kernel size, reuse of vertical grid components and removal of stencils. B: Shared memory

procedure. To increase occupancy on the GPU, the calculation of the total grid size (ghost points has to be accounted for) was moved to the CPU. Since the calculation has to be done for all threads, this is a sequential part of the algorithm no matter what. The improved kernel is presented in section E.1.

For the improved kernel, an upper limit for the shared memory optimizations should be estimated using the method described in section 5.1. The benchmarking shows that the expected upper limit for kernel speedup through code optimizations is times 2 for double precision (table 5.6) and times 11 for single (table 5.6).

Contrary to the expectation, the improved procedure is at least as fast as the estimated upper limit even though shared memory is not used to reduce the read redundancy. If the kernel is executed using single precision (5.7) rather than double precision, there is a significant speedup. Shared memory can thus be used to improve the efficiency of the kernel.

For the shared memory optimization, the grid element central to the stencil is copied to shared memory to reduce redundant data transfers. In order to also copy into shared memory also from the halos, the entire border of the block will only copy into shared memory and do no residual computations such each $B_x \times B_y$ sized block will process the central $B_x - 2 \times B_y - 2$ elements of the block (fig. 5.3, dots without cups). Generally the method resembles that of [11] although their method is for high order finite difference computations.

The procedure is described by algorithm 8. Safety checks are added to avoid segmentation errors (out of bounds memory access - line 6 and 15) and to prevent threads, which are intended to only update the halo in performing other computations (line 14). Another important detail is that all threads of the blocks must reach the barriers; system behavior is undefined otherwise. For a more in depth code, we will refer to section E.1.

Low order residual
Single precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(88,2)	72 ns	(272,1)	4.60 ms	(272,1)	37.0 ms	x1
Est. limit	(16,2)	22 ns	(176,1)	0.49 ms	(64,1)	3.4 ms	x3-11
Method 1	(16,9)	25 ns	(96,1)	0.99 ms	(32,1)	7.5 ms	x2.9-4.9
Method 2	(8,23)	71 ns	(32,12)	0.80 ms	(32,12)	4.9 ms	x1-7.5

Table 5.7: Low order residual. Est. limit: What can be done with shared memory? A: Reduced kernel size, reuse of vertical grid components and removal of stencils. B: Shared memory

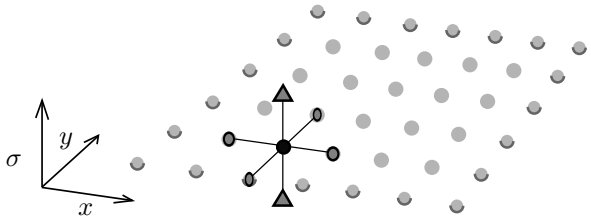

Figure 5.3: Each dot represent a thread in a thread block. For the shared memory approach chosen, only the internal threads (dots without cups) can perform work. The vertical element should be reused through registers (illustrated for one thread - triangles)

Algorithm 8 Computation of low order residual

Block size: $B_x \times B_y$. Local thread indicies: $t_x = 0, B_x - 1$ and $t_y = 0, B_y - 1$.
Halo size: $\mathcal{H}$. Shared memory variable: $S_{i,j}$. Registers: $\mathtt{x}_u$, $\mathtt{x}_c$ and $\mathtt{x}_d$
$\mathtt{x}_c \leftarrow U_{t_x,t_y,0}$
$\mathtt{x}_d \leftarrow U_{t_x,t_y,1}$
for $z \leftarrow 1, N_z$ **do**
    if $t_x < M_x \wedge t_y < M_y$ **then**
        $\mathtt{x}_u \leftarrow \mathtt{x}_c$
        $\mathtt{x}_c \leftarrow \mathtt{x}_d$
        $\mathtt{x}_d \leftarrow U_{t_x,t_y,z+1}$
    end if
    `__syncthreads()`
    $S_{t_x,t_y} \leftarrow \mathtt{x}_c$
    `__syncthreads()`
    if $t_x \geq \mathcal{H} \wedge t_x < B_x - \mathcal{H} \wedge t_y \geq \mathcal{H} \wedge t_y < B_y - \mathcal{H}$ **then**
        if $t_x < M_x \wedge t_y < M_y$ **then**
            Calculate U_{xx} and U_{yy} using S
            Calculate U_{ss} using $\mathtt{x}_u$, $\mathtt{x}_d$ and $\mathtt{x}_c$
            $r = b - (U_{xx} + U_{yy} + \sigma_z U_{ss})$
        end if
    end if
end for

According to the benchmarks in table 5.6 and fig. 5.5, the improved kernel is without shared memory is the superior method for the double precision implementation. For single precision (table 5.7), the method using shared memory is the better choice when the grid size is large: For grid size $257 \times 17 \times 6$, the method is not competitive although it is for grids of size $1537 \times 129 \times 9$ and $3073 \times 257 \times 17$.

5.3.1 Launch configuration and benchmarking

A larger set of benchmarks for the improved version without shared memory are presented in fig. 5.4. For systems with less than some 10,000 internal grid points, the initialization overhead dominate the solution time. The benchmarks also point out that for optimal launch configurations, the scalability does not depend on the anisotropy the grid: Recall that the grid anisotropy vary from $N_x = N_y$ to $N_x = 16N_y$.
Performing the benchmarking with a fixed 16×4 launch configuration show only little difference from the benchmarking using optimal launch configurations. 8×8 on the other hand does not give steady performance for small grids. 16×4 is hence an okay general choice for the low order residual launch configuration. For the shared memory version it is seen that the method has a break even with the method without shared memory at approximately grid sizes of 10,000 internal grid points (fig. 5.5).

5.4 Jacobi and RBGS smoother

The optimization from the low order residual can be applied directly to the Jacobi smoother. Since the methods are practically identical, the Jacobi method will not be discussed further. The RBGS smoother can contrary to the Jacobi smoother be optimized further; for the initial kernel, every other thread is idle. Instead of letting every other thread idle, the threads will process the elements using an offset. Therefore the $(o_x, o_y, z)_{xy}$-utility function is altered to access every second element:

Listing 5.3: Altered memory access function

```
1 inline int RBmemoryIdxXY(int ox, int oy, int oz, int3 M, ↩
      int isRed){
2   return z*M.x*M.y
3         + (Ty + Gy + offsety)*M.x
4         + Tx*2 + isRed + Gx + ox;
```

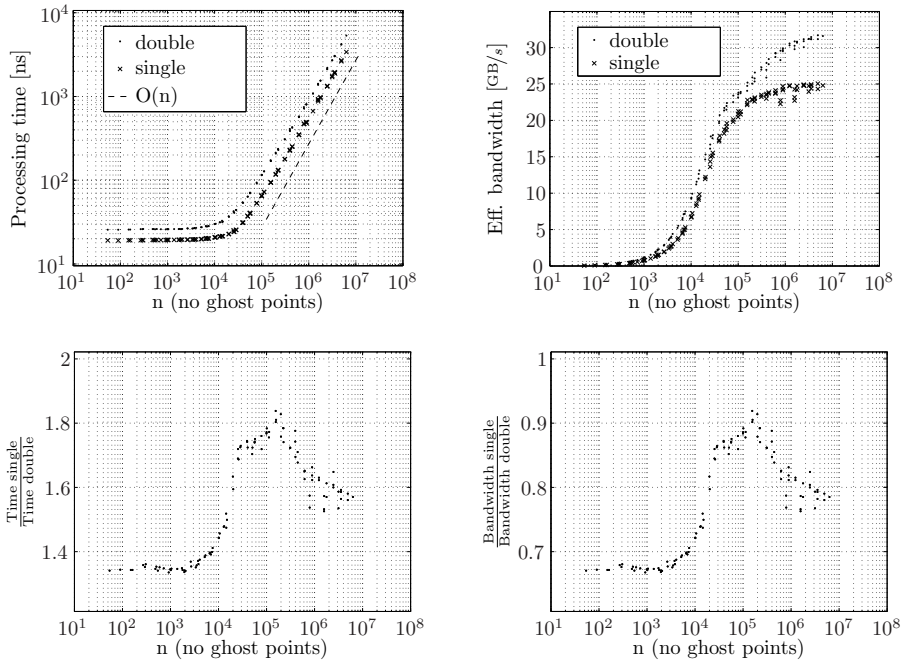

Figure 5.4: Benchmarks for the optimized linear 2. order residual procedure **without** shared memory. Effective bandwidth calculated using (5.10) with $\mathcal{H} = 1$

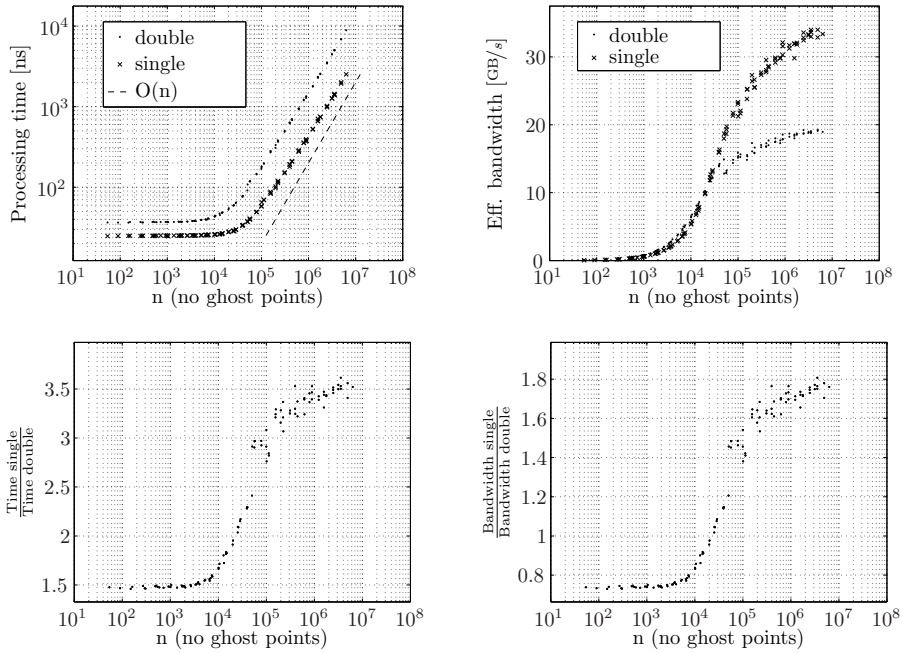

Figure 5.5: Benchmarks for the linear 2. order residual procedure **with** shared memory (only optimized launch configurations used). Effective bandwidth calculated using (5.10) with $\mathcal{H} = 1$

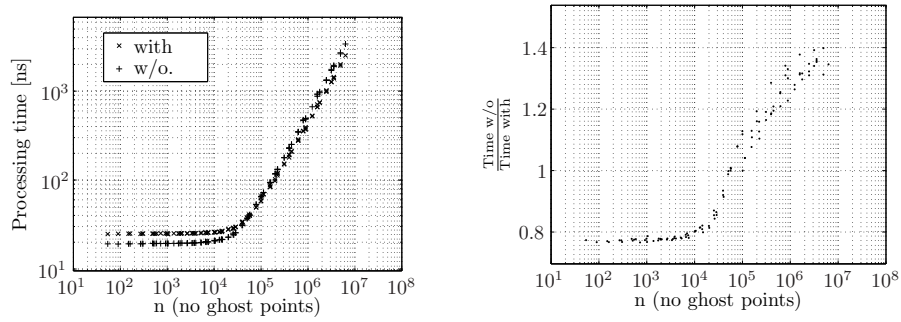

Figure 5.6: For single precision, it is faster to use the shared memory version of the low order residual for $n > 50,000$. Up to 40% is gained from switching!

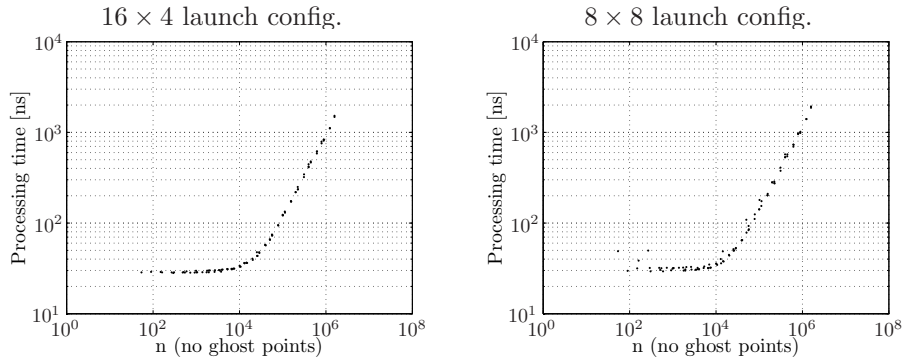

Figure 5.7: The performance of the double precision low order residual procedure seems to be more or less optimized at 16 × 4 threads.

```
5 }
```

Listing 5.4: Optimized RBGS with and without line smoothing

```
1  __global__
2  void RBGSsmoother(...)
3    //...
4    for(Z = 1; Z < N.z; Z++){
5      if(mode == ISBLACK && Z % 1 == 0){
6        if(Y%2 == 1) isRed = 1;
7      }else{
8        if(Y%2 == 0) isRed = 1;
9      }
10     //...
11     Ucenter = RBmemoryIdxXY(0,0,0, M,isRed);
12     //...
13   }
14 }
15
16 __global__
17 void RBGSLinesmoother(...)
18   //...
19   for(Z = 1; Z < N.z; Z++){
20     if(mode == ISBLACK){
21       if(Y%2 == 1) isRed = 1;
22     }else{
23       if(Y%2 == 0) isRed = 1;
24     }
25     //...
26     Ucenter = RBmemoryIdxXY(0,0,0, M,isRed);
```

Red Black Gauss Seidel
Double precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(24,2)	146 ns	(16,11)	5.47 ms	(32,2)	39.8 ms	x1
A	(24,2)	124 ns	(16,12)	3.12 ms	(16,4)	21.8 ms	x1.2-1.8

Table 5.8: Naive: Starting point, improved low order residual. A: Idle time reduced by 50%

Red Black Gauss Seidel
Single precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(32,2)	73.63 ns	(32,1)	2.02 ms	(32,1)	15.16 ms	1x
A	(16,1)	71.97 ns	(80,3)	1.60 ms	(32,1)	11.58 ms	x1-1.3

Table 5.9: Naive: Starting point, improved low order residual. A: Idle time reduced by 50%

```
27      //...
28    }
29  }
```

where the mode is set to `ISBLACK` every other time the kernel is called. One important detail about this setting is that the block dimension should be even for both dimension. Otherwise, the Gauss-Seidel method will not be a Red Black-variant. This rather simple optimization renders an almost double as fast algorithm for large double precision grids (table 5.8). In depth code of the line smoother variant is presented in section E.2. The shared memory variant is not implemented since it is found that the strategy is suboptimal for the similar low order residual procedure procedure.

Execution time and algorithm bandwidths for the best kernel (reduced idle time, no shared memory, optimized launch configuration) are presented in fig. 5.8. The bandwidth of the single and double precision program are the same. The execution time for the single precision kernel is therefore 50% of the double precision kernel since only half the amount of memory needs to be transfered. Notice that the effective bandwidth is approximately 50% of that of the low order residual method. The cause is that RBGS reads all global memory variables twice - once for red, once for black.

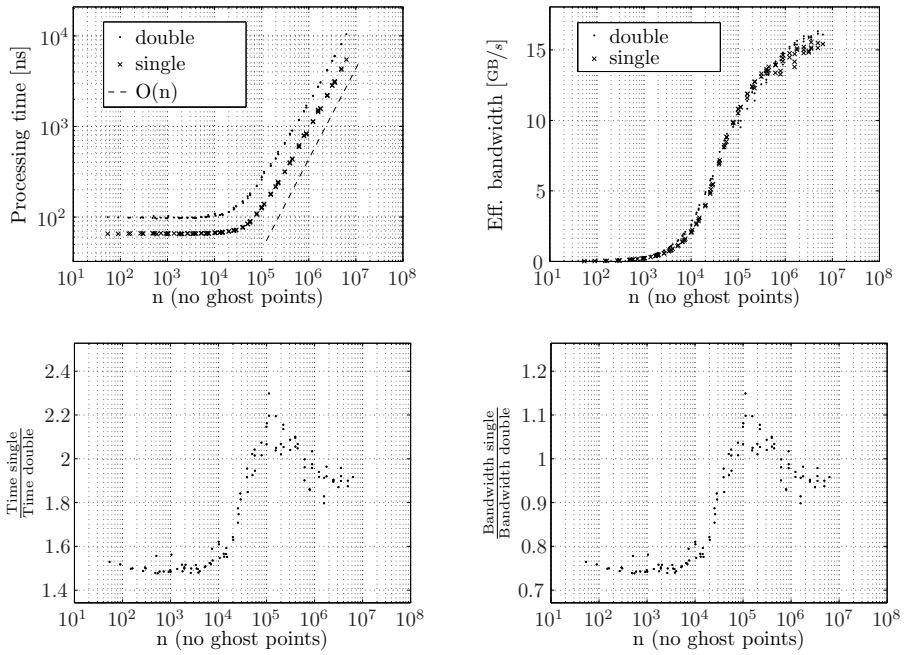

Figure 5.8: Benchmark of the optimized Red Black Gauss Seidel method (only optimized launch configurations used). The effective bandwidth is calculated using (5.10) with $\mathcal{H} = 1$

5.5 Line smoother

The optimizations presented in section 5.3 and 5.4 can be applied directly to the line smoothers as well. Details about evaluation of the finite difference sum are therefore omitted.

The line smoothers have a large usage of local memory which is as slow as global memory. The local memory usage occur due to kernel array allocation which can only be held in registers if they (1) are sufficiently small and (2) always are accessed by monotonic increasing indexing. Since the Thomas Algorithm access elements in both directions (forward and backward elimination), the arrays will regardless of size exist in local memory. In order to reduce the array usage, the kernel complexity is reduced by altering the algorithm such the forward elimination is included in the calculation of the off center x- and y-direction contributions to (1.57) (denoted $\widehat{\boldsymbol{G}}^{-}\boldsymbol{x}_{\sigma}$). The vectors $\boldsymbol{a}_{-1}$ and $\boldsymbol{a}_0$ are thus reduced to $\mathbf{0}$ and $\mathbf{1}$ respectively and therefore superfluous to store (algorithm 9). $\boldsymbol{a}_1$ and $\boldsymbol{b}$ still need to be stored. The resulting local system matrix is therefore given by

$$\widehat{\boldsymbol{G}}^{+} = \begin{bmatrix} 1 & \hat{a}_{1,1} & & & & \\ 0 & 1 & \hat{a}_{1,2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & 0 & 1 & \hat{a}_{1,N_\sigma} & \\ & & -1 & 0 & 1 & \hat{0} \end{bmatrix} \tag{5.12}$$

In order to save memory, the values of $\boldsymbol{a}_1$ can be calculated recursively as well. Although possible to do, it should not be done since the computation looses accuracy. It might be possible to store only every second or third value but it has not been investigated further. $\boldsymbol{a}_1$ must therefore also be saved.

As an alternative to store $\hat{\boldsymbol{a}}_1$, it can be recomputed when needed since it does not need any global memory elements. The strategy is not investigated fully but initial tests with $N_z = 6$ have shown an improvement of 5-10% in computation time. It has already been shown that using shared memory for the finite difference sum is sub-optimal in double precision.

As for the ordinary Gauss-Seidel method, the shared memory variant of the finite difference sum was not implemented since it was seen for the low order residual to be a good strategy only for large problems in single precision. Since the shared memory is free anyways, it might be beneficial to substitute the local memory arrays with an array in shared memory. It is done fairly easy by dereferencing the shared memory at an appropriate offset.

Algorithm 9 Modified forward elimination

$\boldsymbol{d} \leftarrow \boldsymbol{b}$ ▷ Right hand side of (1.57)
$\boldsymbol{a} \equiv \boldsymbol{a}_{-1}, \quad \boldsymbol{b} \equiv \boldsymbol{a}_0, \quad \boldsymbol{c} \equiv \boldsymbol{a}_1$ ▷ in place variables for improved readability

$c_i \leftarrow 0$
$c_1 \leftarrow \frac{c_1}{b_1}$ ▷ For this case, $c_1 = 0$
$b_1 \leftarrow \frac{b_i}{b_i}$ ▷ Effectively $b_i \leftarrow 1$
for $i \leftarrow 2, N_\sigma$ **do**
▷ Assemble small equation system and rhs
    $a_i \leftarrow S^+_{(up)}$
    $b_i \leftarrow S^+_{(center)}$
    $c_i \leftarrow S^+_{(down)}$
    $d_i \leftarrow d_i - \boldsymbol{G}^-_\sigma \boldsymbol{x}_\sigma$
▷ Combine with forward elimination
    $a_i \leftarrow a_i - b_{i-1} \cdot \frac{a_i}{b_{i-1}}$ ▷ No need to store; effectively $a_i \leftarrow 0$
    $b_i \leftarrow b_i - c_{i-1} \cdot \frac{a_i}{b_{i-1}}$
    $d_i \leftarrow d_i - d_{i-1} \cdot \frac{a_i}{b_{i-1}}$
▷ Scale row such $b_i \leftarrow 1$
    $b_i \leftarrow \frac{b_i}{b_i}$ ▷ No need to store; effectively $b_i \leftarrow 1$
    $c_i \leftarrow \frac{c_i}{b_i}$
    $d_i \leftarrow \frac{d_i}{b_i}$
end for

RBGS Line smoother
Single precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(32,6)	147 ns	(160,3)	6.03 ms	(128,4)	45.0 ms	x1
A	(16,3)	88 ns	(128,3)	2.65 ms	(512,1)	20.2 ms	x1.7-2.3
B	(48,2)	84 ns	(320,1)	1.88 ms	(256,2)	14.2 ms	x1.8-3.2
C	(24,2)	79 ns	(16,2)	1.41 ms	(120,1)	13.19 ms	x1.9-3.4

Table 5.10: Naive: Starting point, naive RBGS. A: Algorithm complexity reduced. B: Idle time reduced by 50%. C: Substitute local array with smem array.

RBGS Line smoother
Double precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(32,2)	263 ns	(32,2)	10.10 ms	(64,2)	74.06 ms	x1
A	(24,2)	206 ns	(32,2)	7.28 ms	(64,1)	51.83 ms	x1.3-1.4
B	(24,2)	173 ns	(64,1)	4.38 ms	(32,2)	29.67 ms	x1.5-2.4
C	(8,6)	167 ns	(16,1)	5.92 ms	(56,1)	68.83 ms	x1.1-1.6

Table 5.11: Naive: Starting point, naive RBGS. A: Algorithm complexity reduced. B: Idle time reduced by 50%. C: Substitute local array with smem array.

Listing 5.5: Substitution of array with shared memory

```
1 //_ftype a3[MEMSIZE];
2 _ftype *a3 = &smem[  (tx + ty*blockDim.x)*2*N.z];
3 //_ftype d[MEMSIZE];
4 _ftype *d  = &smem[2*(tx + ty*blockDim.x)*2*N.z];
```

The remainder of the algorithm is unchanged! Unfortunately, there is no benefit in using the shared memory for double precision and only little benefit for single precision. The optimal implementation of the RBGS line smoother is thus the kernel with reduced complexity and reduced idle time.

Benchmarks for the optimal version shows that the difference between single precision and double precision is a factor of two (fig. 5.9). The maximum measured effective bandwidth for the current implementation is 10-12 $^{GB}/_{s}$ regardless of single or double precision is used.

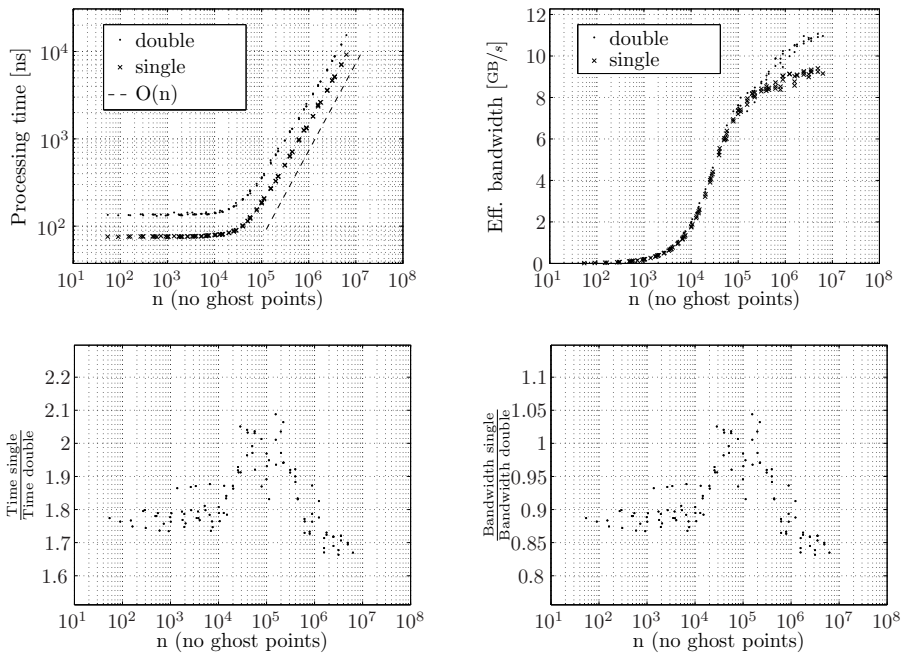

Figure 5.9: Benchmarks for the optimized RBGS line smoother (only optimized launch configurations used). The effective bandwidth is calculated using (5.10) with $\mathcal{H} = 1$

5.6 High order residual

In the previous section, it is seen that calculation of the high order residual is an expensive procedure of the program and unfortunately it is also the most complex method. The first and easiest improvement is to move the various order stencils to constant memory rather than placing them in global memory. Constant memory also reside in global memory but is cached; if the same elements are read only and accessed over and over, it is therefore beneficial to store the variables in global memory. The kernel size for the naive version uses 78 registers for a double precision kernel and the block size is therefore limited to a total of $(\frac{16384}{78})$ 210 threads per multiprocessor[3]. By trial and error, it was possible to reduce the number of registers to 70 which allows for 234 threads totally.

The high order residual method uses mixed derivatives which increase the demands to the amount of shared memory compared to that of the low order residual. Instead of sharing only a single plane, a number of planes should be shared in order to hold all elements touched by all stencils. Processing a plane of size $P_x \times P_y$, the total number of shared memory elements is

$$\min N_{shared} = \left[N_x N_y + 2\mathcal{H}(N_x + N_y) + \mathcal{H}^2\right] \mathcal{W} \tag{5.13}$$

$$\min N_{shared} = \left[N_x N_y + 2\mathcal{H}(N_x + N_y) + \mathcal{H}^2\right] (2\mathcal{H} + 1) \tag{5.14}$$

$$M_{shared} = N_{shared} \cdot \texttt{\_sizeof(\_ftype)} \tag{5.15}$$

where $\mathcal{W}$ is the stencil width.

For the low order residual procedure it was possible to instantiate threads also over halo points which decreased the complexity of the algorithm. This is not possible for the high order residual; there are neither enough registers or shared memory to do that. Using a kernel with no 'wasted' threads to load a variable number of halo points into shared memory was not implemented due to a very high complexity of such a method. Allocating threads over halo points is possible if the method is split into two passes where pass 1 calculate the $U_{xx}, U_{x\sigma}, U_{\sigma}$ and $U_{\sigma\sigma}$-contribution and pass 2 the U_{yy} and $U_{y\sigma}$ contributions. The problem of splitting the method into two passes is that the read redundancy goes up for both 3D grid and 2D surface variables. It was found that the method is not efficient (see table 5.13 and 5.12). The best version is thus the procedure with reduced kernel size and stencils moved to constant memory.

The benchmarks for the best 6. order residual method shows a big difference in bandwidth for the single and double precision program. It is therefore likely that

[3] CC 1.1, 1.2: Max 768 threads. CC 1.3: Max 1024 threads. CC 2.0: 1536 threads per multiprocessor

High order residual
Single precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(64,5)	0.88 ms	(64,3)	57.7 ms	(96,2)	461 ms	x1
Limit	(16,3)	0.14 ms	(16,4)	6.3 ms	(64,1)	48 ms	x6.3-9.6
A	(32,3)	0.45 ms	(64,1)	18.4 ms	(64,1)	142 ms	x2.0-3.3
B, pass1	(64,1)	0.22 ms	(320,1)	11.1 ms	(288,1)	85 ms	-
B, pass2	(32,12)	0.15 ms	(16,25)	6.8 ms	(8,51)	52 ms	-
B, total	-	0.37 ms	-	17,97 ms	-	227 ms	x2-2.4

Table 5.12: Naive: Stencils are in global memory. Est. limit: What can be done with shared memory? A: Stencils moved to constant memory, kernel size reduced. B: Two pass method using shared memory.

High order residual
Double precision

	$257 \times 17 \times 6$		$1537 \times 129 \times 9$		$3073 \times 257 \times 17$		Relative
	Conf.	Time	Conf.	Time	Conf.	Time	to Naive
Naive	(16,4)	1.05 ms	(192,1)	64.41 ms	(96,2)	518 ms	x1
Est. limit	(88,2)	0.56 ms	(16,4)	31.00 ms	(32,2)	242 ms	x1.9-2.1
A	(88,2)	0.79 ms	(32,2)	45.09 ms	(32,2)	348 ms	x1.3-1.5
B, pass1	(64,3)	0.57 ms	(64,1)	33.55 ms	(192,1)	252 ms	-
B, pass2	(32,9)	0.42 ms	(8,35)	15.93 ms	(8,35)	120 ms	-
B, total	-	0.99 ms	-	49.48 ms	-	372 ms	x1-1.4

Table 5.13: Naive: Stencils are in global memory. Est. limit: What can be done with shared memory? A: Stencils moved to constant memory, kernel size reduced. B: Two pass method using shared memory.

there are not enough threads active to occupy the bandwidth fully; transferring is $102.4^{GB}/_{s}$ but there is a large latency prior to the transfer. If there are not enough active threads, too few memory requests are active at the time thus leaving a gap between single and double precision bandwidths.

5.7 Optimized kernels

Performance improvement plots for the optimizations to the high order residual and line smoother are presented in fig. 5.11.

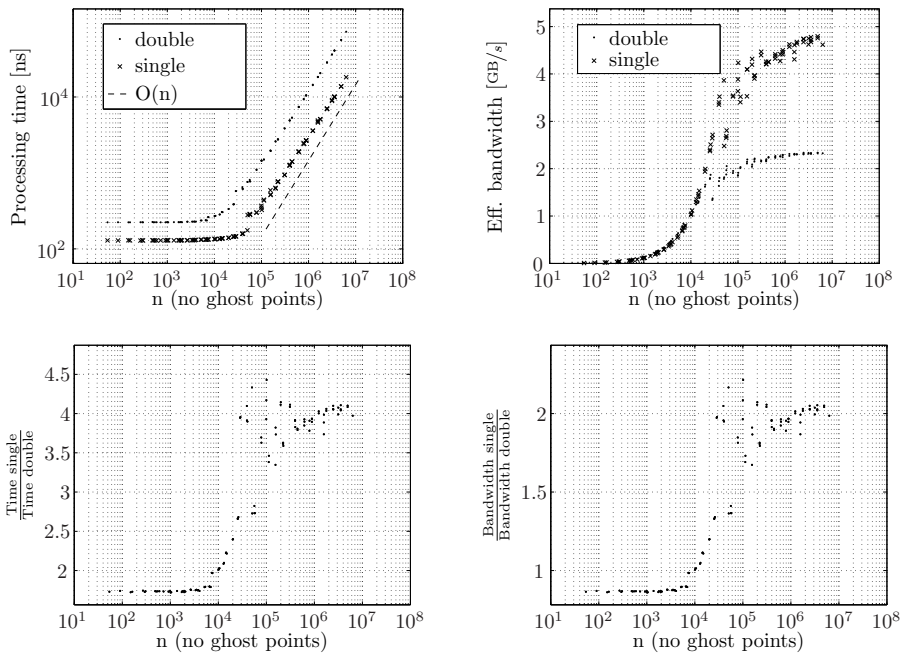

Figure 5.10: Benchmarks for the best implementation of the 6. order non linear residual procedure (only optimized launch configurations used). The effective bandwidth is calculated using (5.10) with $\mathcal{H} = 3$

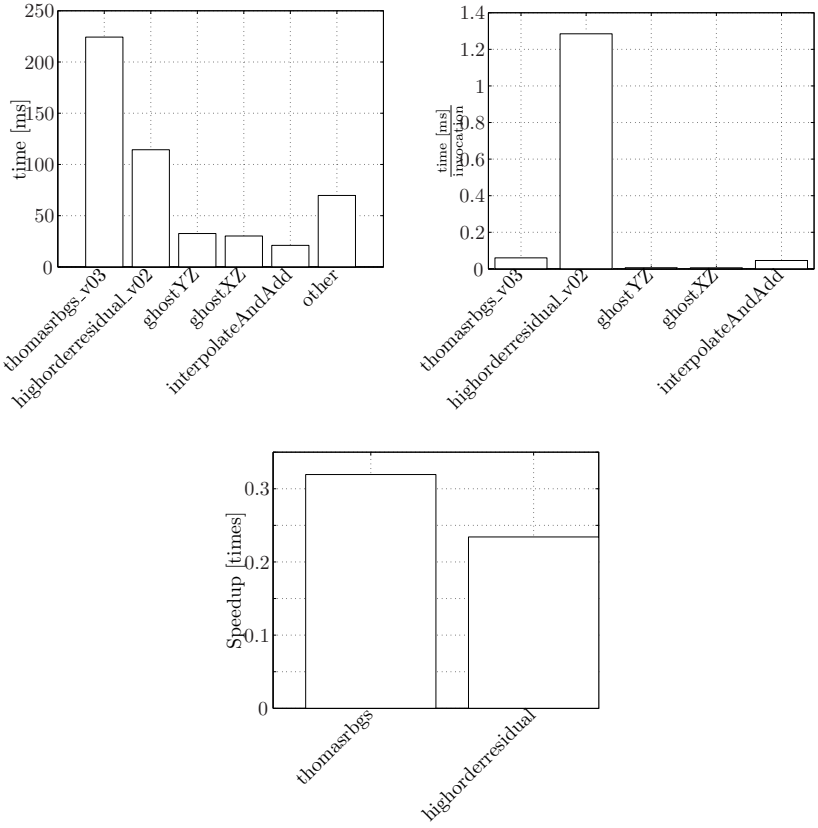

Figure 5.11: Profiling of the optimized CUDA implementation. 10 time steps for the Whalin shoal (gridsize $257 \times 21 \times 6$) are benchmarked using the CUDA profiler. The smoother is called 3700 times and the high order residual only 89 times

Chapter 6

Results

There are two aspects of this project which are interesting. One is the numeric point of view, and the other is parallelization and optimizations.

6.1 Verification

In order for the algorithm to be convergent, the accuracy must be increased for decreased step size. In particular, the error should be described by

$$||\boldsymbol{\epsilon}||_2 \leq O(h^p) \tag{6.1}$$

where $\boldsymbol{\epsilon}$ is the error of the estimate, p is the spatial discretization order and h the smallest step size in the domain. In order to verify the iterative solver, a non linear periodic shallow water wave is used. The special about the wave is that there exist an analytic solution and that the solution can be used to verify the method. The verification cannot cover the entire model since the analytic solution assumes flat bottom. The wave is controlled by the factors $k \cdot h$ (the wave number) and $H \cdot L$ (wave height relative to wave length).

For verification a wave with $H/L = 0.0088 = 30\%$ of breaking and wave number $kh = 0.5$ is used. Regarding the convergence. The actual order of convergence

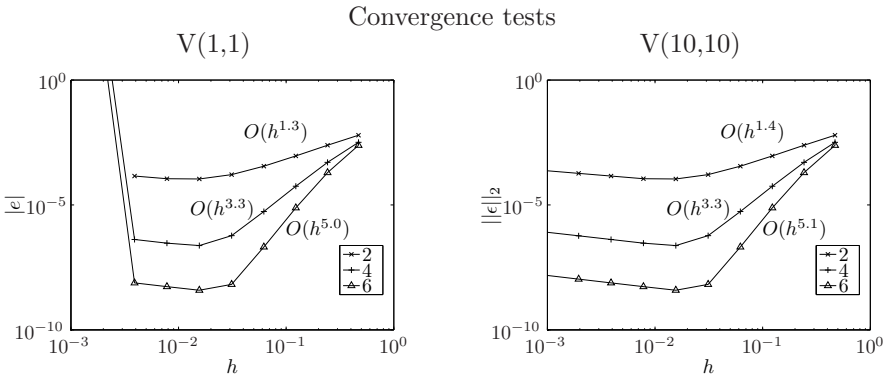

Figure 6.1: Left: The Laplace solver is convergent only when $\Delta x > 10^{-1.9}$ (160 points per wave). The actual order in the convergent region is approximately one lower than the intended order (see legend).

is approximately one lower than the intended order. The reason for the 'lacking order' and missing convergence for $h_x \ll h_z$ was not found at the end of the project.

6.2 Validation

For validation, the Whalin shoal experiment has been used for which the still water depth is given by

$$h(x,y) = \begin{cases} 0.4572, & 0 \le x < 10.67 - \Sigma(y) \\ 0.4572 + 1/25(10.67 - \Sigma(y) - x), & 10.67 - \Sigma(y) \le x < 18.29 - \Sigma(y) \\ 0.1524, & 18.29 \le x \le 35 \end{cases} \tag{6.2}$$

The still water depth is thus described by a flat bottom in two levels with a semicircular transition. One of the original experiments has wave period $T = 2$ and wave height $H = 0.0150$. For the numerical experiment, the waves are generated in the zone $0 \le x < 5$ and absorbed in $30 < x \le 35$. The domain discretization is set to $257 \times 21 \times 6$ in order for the waves to be resolved sufficiently. The experiments were carried out using a formally $O(\Delta x^6, \Delta t^4)$ discretization scheme, an absolute tolerance of 10^{-4} and a relative tolerance of 10^{-6}. For analysis, a temporal Fourier transfer is done over the three wave periods in the time domain $t = 44s - 50s$ along the center of the shoal. The computed results are in good agreement with that of [1] (fig. 6.2, 6.3).

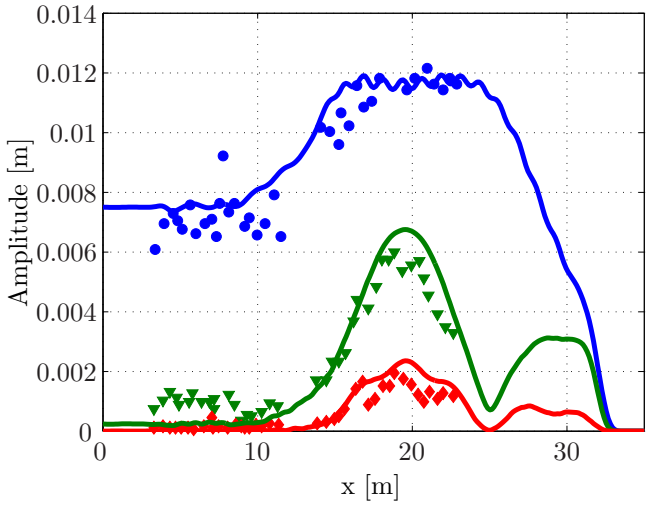

Figure 6.2: Fourier analysis of Whalin shoal case ($T = 2$, $H = 0.039$) in the interval $t = 44s - 50s$ showing first (blue), second (green) and third (red) wave harmonics

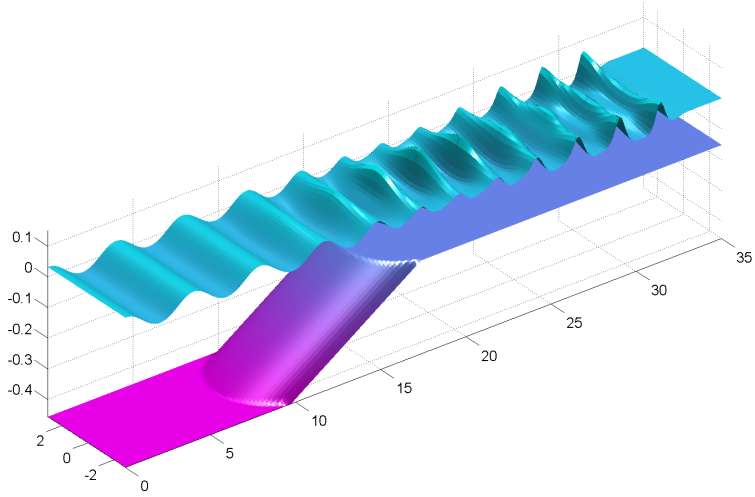

Figure 6.3: Snap shot of Whalin shoal case ($T = 2$, $H = 0.039$, $t = 50s$)

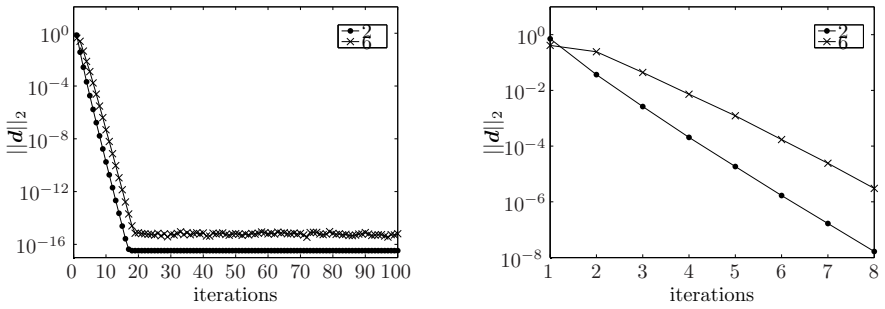

Figure 6.4: The plot shows the convergence curves for various (intended) orders of discretization using a V(5,5) configuration for the preconditioner to ensure effectiveness. The wave resolution is 26 points with $h = 0.077 = 10^{-1.11}$

6.3 Convergence of the Defect Correction method

Defect Correction basically is a very simple algorithm. The effectiveness of the algorithm depends only on

(i) How close $\boldsymbol{\mathcal{M}}^{-1}$ is to $\boldsymbol{A}^{-1}$

(ii) The effectiveness of the preconditioner

The convergence of the solver will be analyzed using the 2-norm of the defect $\boldsymbol{d}$. The dependency on order will be examined first. For analysis, a snapshot of a single shallow water wave will be used. The wave parameters are chosen to $kh = 0.5$, $H/L = 0.0088 = 30\%$ of breaking with an intended 6^{th} order non linear spatial discretization on grid resolutions $N_y = 7, N_z = 6$ and N_x varying from 13 to 1537. The number of ghost points in the horizontal direction is $\mathcal{H} = 3$. The total domain length is set to $L = 2$ and the step length thus varies from $\Delta x = 0.0013 = 10^{-2.8}$ to $\Delta x = 0.25 = 10^{-0.6}$. In order to limit the analysis, only the RBGS line smoother which is also the most effective smoother implemented is applied.

The convergence curve for various orders is provided in fig. 6.4. After the first few iterations, the convergence rate (steepness of the curve) is the same regardless of discretization. The convergence rate for the 2^{nd} order approximation is one digit per iteration.

Fundamental parameters	
Smoothings	$(\nu_1, \nu_2) \in (\mathbb{N}^+, \mathbb{N}^+)$
Smoother type	Jacobi, Gauss Seidel, Line smoother, ...
Cycle type	F, μ (primarily V or W)

Table 6.1: Basic parameters of the Coarse Grid Correction method.

Non fundamental parameters	
Precision	Single, double
Digressiveness of coarsening	$0 \leq \alpha \leq \infty$; see section 2.4.
Number of grids in use	1 to as many possible
Improved initial guess	Potential flow from earlier time steps

Table 6.2: Non fundamental parameters of the Coarse Grid Correction method. Note that the the current implementation does not allow the precision of the preconditioner to differ from that of the DC method.

6.3.1 Effectiveness of the preconditioner

The preconditioner has various parameters which can be adjusted in order to tune it for some particular problem. The basic parameters of the algorithm are listed in table 6.1 and more advanced but non-essential parameters in table 6.2. One important parameter is the number of pre- or post-relaxations. The most commonly used combinations are given by

$$(\nu_1, \nu_2), \quad \nu_2 \in \{\nu_1, \nu_1 - 1\} \tag{6.3}$$

where ν_1 and ν_2 are the number of pre- and post-smoothings respectively. Changing the number of pre- and post-smoothings proves to be vital in order to ensure convergence of the algorithm. The figures 6.5, 6.6 and 6.7 show iteration plots for various resolutions of the problem using V-, F- and W- cycling strategies respectively. For the V- and F-cycle, a (1,1)-configuration ensures convergence only for $\Delta x \geq 10^{-2.6}$. Increasing the number of smoothings ensures convergence also for more fine resolutions. It has not been investigated in detail how many pre- or post-smoothings which will be needed to ensure convergence for particular problem and it is therefore a user controlled parameter of the algorithm.

Increasing the number of smoothing not only ensures convergence. It also control the effectiveness in term of number of outer correction iterations. It is advisable to experiment with the number of smoothings: If the number of outer iterations in the Defect Correction method is high, it might be efficient, especially for large grids to increase the number of smoothings to improve convergence.

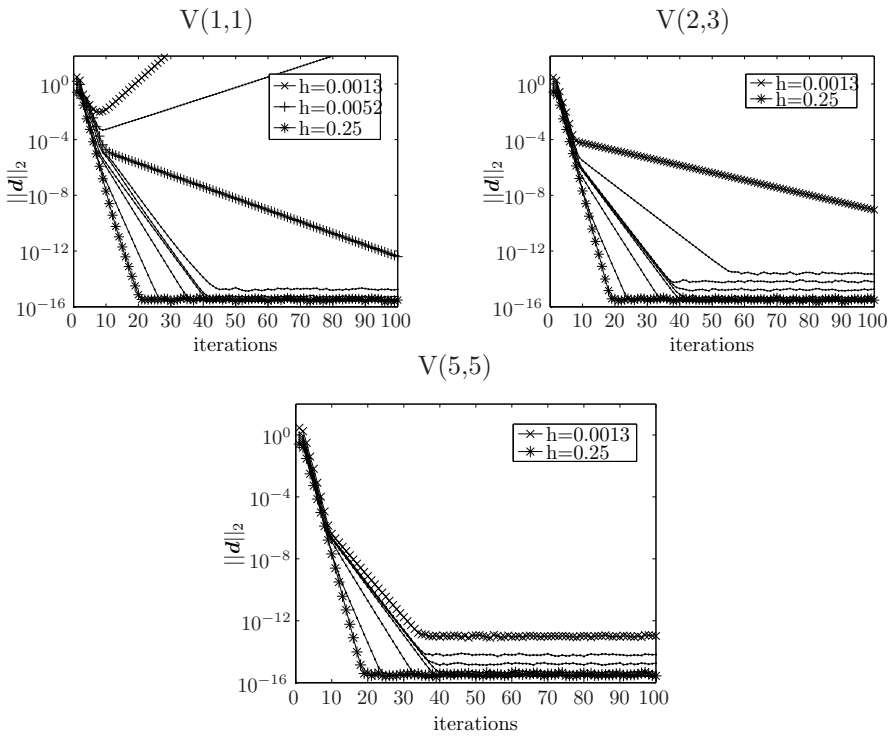

Figure 6.5: Convergence of Defect Correction depend on its configuration. For a V(1,1)-configuration, the smallest step size which ensures convergence for this particular problem is $h \geq 10^{-2.6}$. The setup uses $k \cdot h = 0.5$, $H/L = 0.0088 = 30\%$ of breaking with and intended 6^{th} order discretization.

Requiring an absolute tolerance of e.g. 10^{-7} with a V(1,1)-configuration in a domain with step length $h = 0.0052$ (corresponding to a wave resolution of 385 points) will require approximately 35 outer iterations (fig. 6.5, upper left). If a V(5,5) configuration is chosen instead, only 10 outer iterations are required. By increment of the number of smoothings, the computational effort goes up and it is therefore not given which configuration is the most effective. Although the W(1,1) configuration is convergent already for one iteration, the strategy is *very* time consuming.

In section 2.4 it was hypothesized that aliasing components might cause problem when a large number of grids are used and the apparent conclusion from fig. 6.5, 6.6 is that the hypothesis is correct: If there residual is not smooth enough, the solver is divergent which is most likely caused by aliasing components.

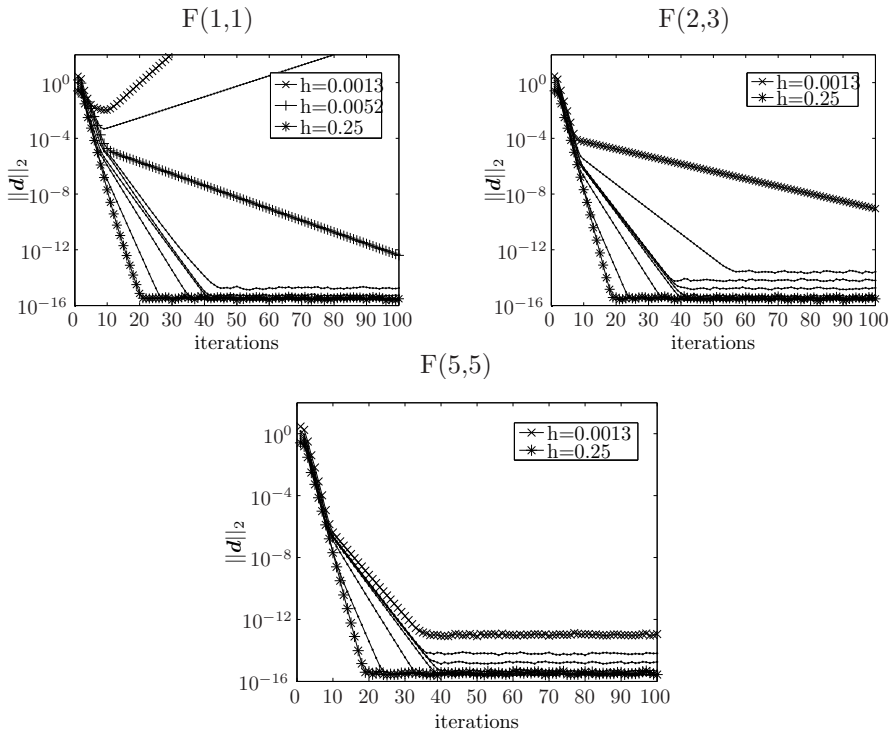

Figure 6.6: Convergence of Defect Correction depend on its configuration. There is practically no difference between the F- and V-cycle for this particular problem. The setup uses $kh = 0.5$, $H/L = 0.0088 = 30\%$ of breaking with and intended 6^{th} order discretization.

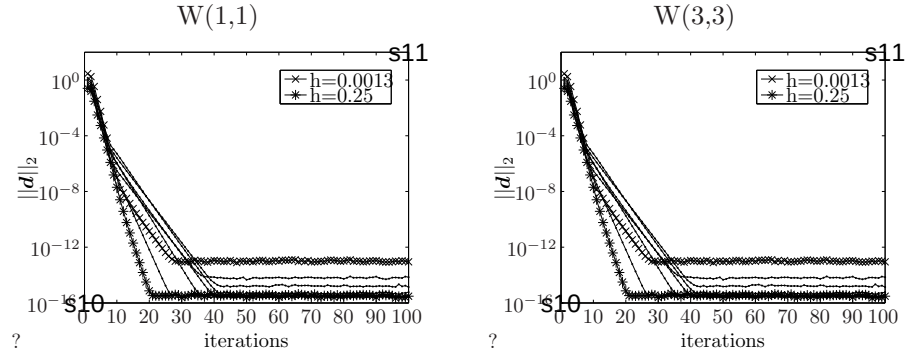

Figure 6.7: Convergence of Defect Correction depend on its configuration. Although it shows good convergence, the W-cycle in this implementation is very slow and it is therefore recommended not to be used. The setup uses $kh = 0.5$, $H/L = 0.0088 = 30\%$ of breaking with and intended 6^{th} order discretization.

6.4 Limiting the number of grid levels

Since the multigrid implementation seem to have problems with aliasing components, it might very well be an idea not to use many grids. If the correcting term is transfered to a coarse grid without being properly damped, aliasing components will add noise to the signal. At some point, the signal to noise (SNR) ratio will be so poor that none of the original signal is left. The result is that the algorithm tries to correct something wrong which in worst case may lead to divergence which was experienced in fig. 6.5 and 6.6.

In the section 6.3 it was found that there is practically no difference between using an F- or a V-cycle. Therefore only the V-cycle is considered. Again a non linear wave with $kh = 0.5$, $H/L = 0.0088 = 30\%$ of breaking with and intended 6^{th} order discretization is used.

In fig. 6.8, $N_x = 1537$. For a V(1,1)-configuration, using 1, 2 or 3 levels of grids, the algorithm is convergent although the convergence is very slow. The reason for the very slow convergence is most likely that too much of the error exist in frequencies which will never propagate into the high frequency area on any of the used grids. Unfortunately, a V(1,1) configuration is not strong enough to remove enough high frequency components to include a 4^{th} level of restriction with a resulting divergent behavior.

Increasing the number of smoothings to a V(2,2) configuration the high frequency components are dampened enough to also use the 4^{th} and the 5^{th} level of grids. Since a number of smoothings is applied rather than a direct solver, only error components in the frequency range $1/32 - 1/1$ can be removed (eventually see table 6.3). When 5 grids are used, the convergence makes a sudden snap after approximately 10 iterations after which the convergence is poor due to either aliasing components or even lower frequency error components existing in the grid. Increasing the number of smoothings even further, it can be seen that the error is caused primarily by aliasing effects rather than very low frequency error components: The error when using 5 grids is now as low as when 6-9 grids are used. Even though alot of smoothings are applied, there are still problems with aliasing and perhaps also even lower frequencies: The algorithm still has a sharp bend which an ideal solver should not have. At this point a direct solver ought to be applied since the smoother apparently is not effective enough to remove the remaining aliasing components, even for a V(5,5) configuration.

What is relevant to notice is for a sufficiently resolved error at a certain level, it does not help to increase the number of smoothings. In particular the convergence curves of using 1-3 grids does not change dramatically from V(1,1) to either V(2,2) or V(5,5). A straight forward thought is therefore to investigate

	Fine grid frequency domains			
	Low		High	
Grid level	From	To	From	To
1	0	$1/2$	$1/2$	1
2	0	$1/4$	$1/4$	$1/2$
3	0	$1/8$	$1/8$	$1/4$
4	0	$1/16$	$1/16$	$1/8$
5	0	$1/32$	$1/32$	$1/16$
n	0	$1/2^n$	$1/2^n$	$1/2^{n-1}$

Table 6.3: Each of the various grids are associated with a frequency range. In an optimal multigrid solver, all high frequency components will be removed from the associated high frequency part of the signal. Using 4 grids and no direct solver on the 4^{th} grid, the multigrid method should thus be able to remove components in the range $1/16 - 1/1$.

whether it will suffice to use a V(1,1) smoother for grids 1-3, V(2,2) for level 4 etc. while not increasing the number of outer iterations.

6.5 Scalability, limitations and speedups

One of the key properties of program, is that it should scale; a problem with n degrees of freedom should take $O(n)$ time to solve, and have $O(n)$ memory usage. The most expensive variables are the fine 3D grids. A piece of consecutive memory large enough to hold all levels of fine and coarse grids used for CGC will be referred to as a multigrid.

Due to the multigrid preconditioner, the memory usage depend on the coarsening strategy. It is possible to provide an upper and a lower bound for the memory usage for multigrids: Every time the grid size is halved along an axis, the problem size is halved. Reducing problem size in 2 direction at the same time, the grid size is reduced to $1/4$ of the original size, and reducing in all 3 directions, the grid size is reduced to $1/8$ of the fine grid. The memory usage of a multigrid will thus have an upper bound determined by[1]

$$\text{Multigrid allocation} < \sum_{k=0}^{\infty} \frac{N}{2^k} = 2N \tag{6.4}$$

$$\tag{6.5}$$

[1]Ghost points neglected.

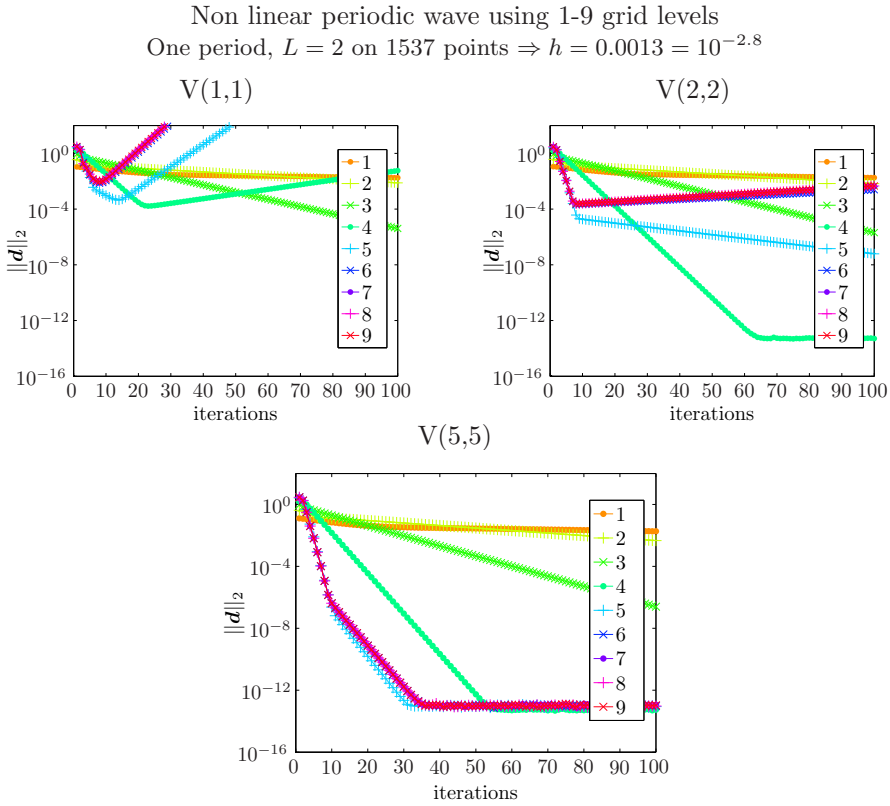

Figure 6.8: Given a convergent setup, it is seldom a bad choice to stop after a 4-5 levels. The setup uses $kh = 0.5$, $H/L = 0.0088 = 30\%$ of breaking with and intended 6^{th} order discretization.

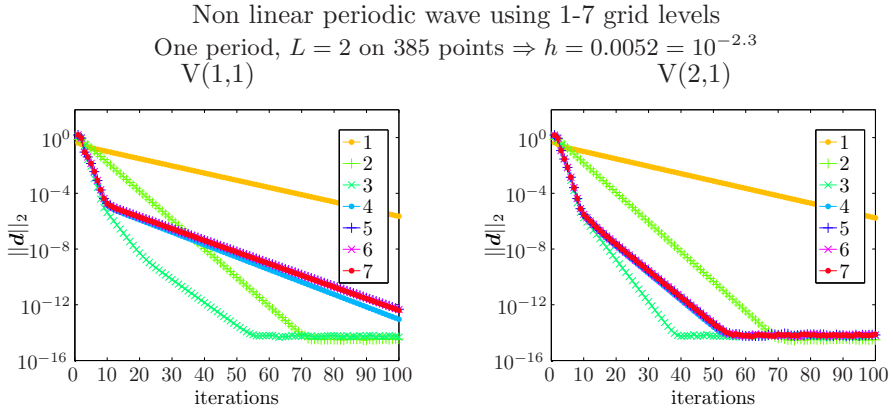

Figure 6.9: Given a convergent setup, it is seldom a bad choice to stop after a 4-5 levels. The setup uses $k \cdot h = 0.5$, $H/L = 0.0088 = 30\%$ of breaking with and intended 6^{th} order discretization.

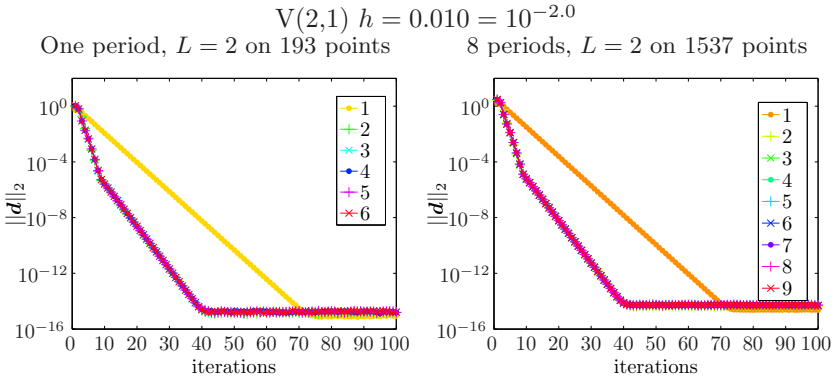

Figure 6.10: Altering the physical domain size does not change convergence dramatically when the physics are note altered.

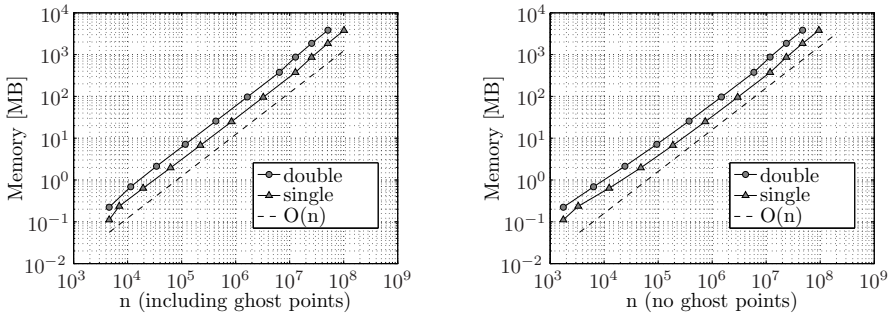

Figure 6.11: Scaling of device RAM use with (left) and without (right) ghost points included.

where $N = N_x N_y N_z$. The grid usage of the program is listed in table D.1, appendix D, and the program memory usage is found to be determined by

$$\text{max elements} = (4 \cdot 2 + 2)N_x N_y N_z + (13 \cdot 2 + 8)N_x N_y \tag{6.6}$$

$$= (10N_z + 44)N_x N_y \tag{6.7}$$

$$\text{max memory} = \text{max elements} \cdot \texttt{sizeof(\_ftype)} \tag{6.8}$$

If a Gauss-Seidel type smoother is used, the 3D multigrid usage is reduced with 1 grid, reducing the memory allocation to

$$\text{max elements} = (8N_z + 42)N_x N_y \tag{6.9}$$

The largest systems which could be in memory on a Tesla device (4GB) has approximately 51 million degrees of freedom for double precision and 101 million degrees of freedom for single precision (10 and 11 levels of grids respectively). In fig. 6.11 it can be seen the memory usage of the algorithm has $O(n)$ scaling, even for small grids.

For the benchmarking of time integration, a non linear periodic wave with $kh = 0.5$, $H/L = 0.088 = 30\%$ of breaking is solved using a V(1,1)-configuration for the Laplace problem. In order to ensure that the Laplace solver is stable, the wave resolution is held approximately constant at 32 points per wave. Thus for the maximum test size, $N_x = 671,745, N_y = 7, N_z = 6$ (3.72 GB) there are totally 20,995 waves in the x-direction of the domain. In order to be independent of the problem, the benchmark provided is created taking 10 time steps and dividing with the total number of Defect Correction applied. The unit is thus time per outer iteration. The algorithm scales nicely for large problems on both the Tesla (fig. 6.12) and Fermi (fig. 6.13) architecture.

The optimizations applied to the low order residual, smoother and high order residual procedures give a V-cycle speedup of approximately times 1.5 in double

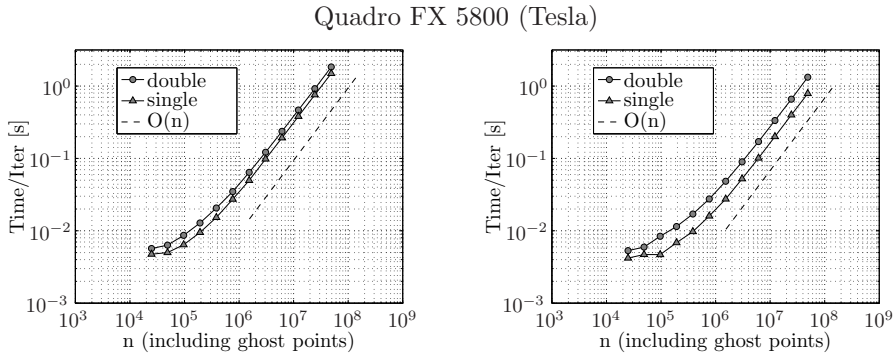

Figure 6.12: Scaling of iteration time on a Tesla (CC 1.3) device for naive implementation (left) optimized implementation (right).

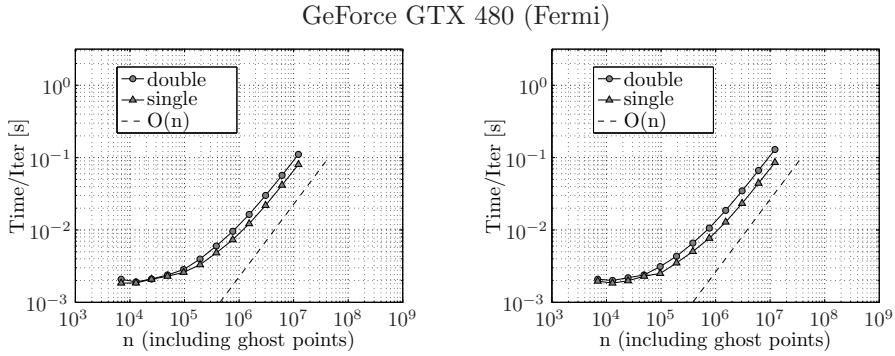

Figure 6.13: Scaling of iteration time on a Fermi (CC 2.0) device for naive implementation (left) Tesla optimized implementation (right).

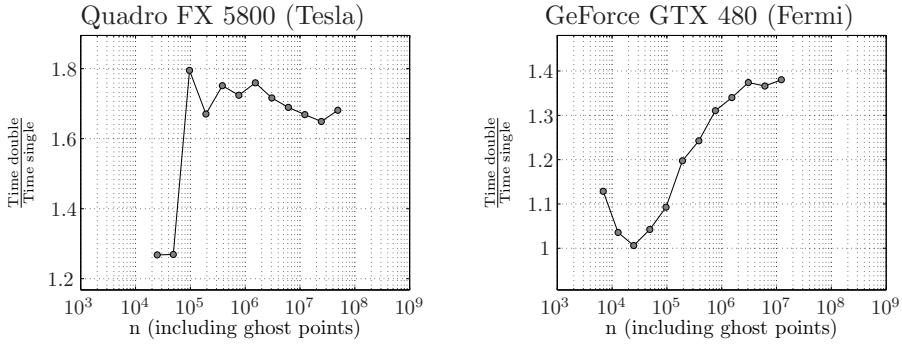

Figure 6.14: Demoting from double to single precision, speeds up the computation.

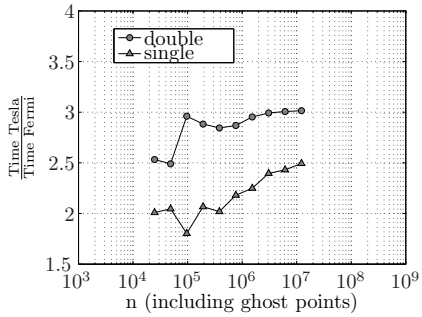

Figure 6.15: Changing from a high end CC 1.3 to a high end CC 2.0, the program is sped up by a factor of 2-3

precision and times 2 in single precision (fig. 6.12). Moving the program to the Fermi architecture speeds the program up by a factor of 2-3 times (fig. 6.15). This is somewhat more than expected since the algorithm is memory bound and the bandwidth is increased 'only' by approximately 70% ($102.4^{GB}/_{s} \rightarrow 177.4^{GB}/_{s}$). Rather interestingly, the applied optimizations seem to be suboptimal on the Fermi; the naive implementation of the program is in fact faster than the program optimized for the Tesla.

The final and very important result is the performance gain from migrating the sequential CPU code to parallel GPU code. The CPU code was executed on a system contemporary to the GPUs. In particular, the CPU is an Intel Core i7 920 processor @2.67GHz, and the CPU-to-RAM-bandwidth is estimated to $11.5^{GB}/_{s}$.
In fig. 6.16 it can be seen that the speedup for small problems is in the scale of 2 for the Quadro FX 5800, and 9 for the GeForce GTX 480. For larger systems, the speedup on the Quadro FX 5800 saturates at about 11 times, and for the GeForce GTX 480, the saturation level is apparently not met within the scale of problems, which can be solved with the current CPU code. The maximum obtained speedup is 42 times faster than the CPU version of the program, which is in the high end of the expectations: In section 1.4 the limiting factor was found to be the bandwidth, rather than the parallel fraction of the code since the problem is memory bound, rather than compute bound. The speedup is a combination of the higher device bandwidth and the decreased memory usage obtained by storing only the necessary vectors and generating matrices on demand.

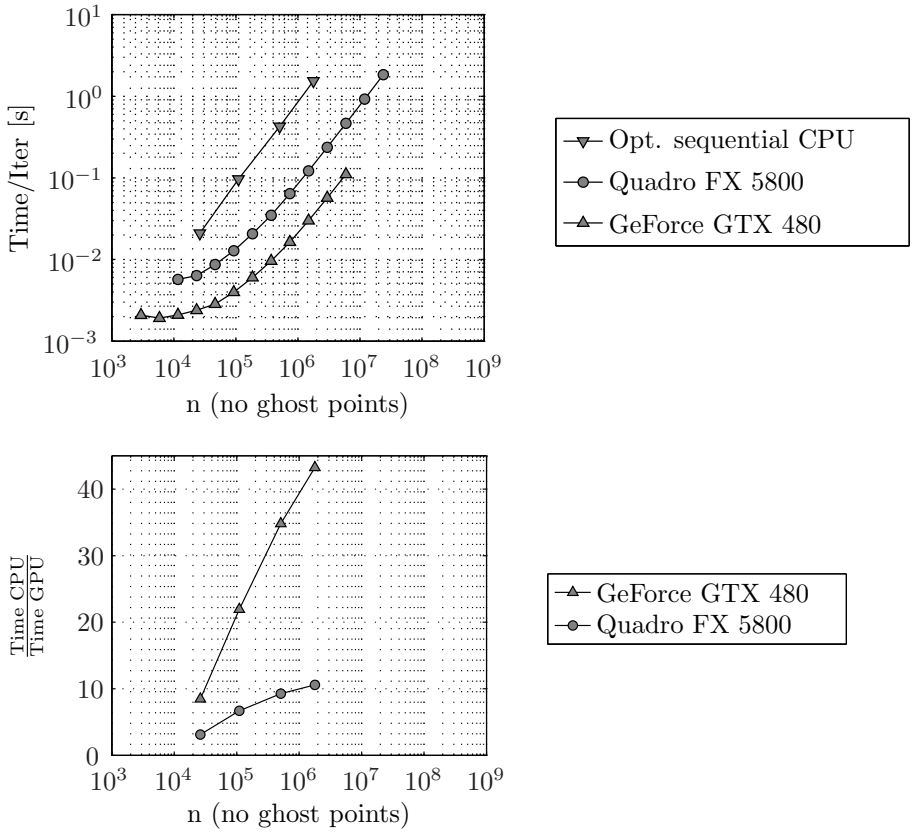

Figure 6.16: GPU program compared to sequential (non OpenMP) CPU program. Iteration time is normalized to be virtually problem independent by dividing the timing of a given program by the number of outer Defect Correction iterations performed.

Chapter 7

Future work

There are two primary aspects of the work with the Defect Correction method with a multigrid preconditioner: One is the algorithmic, and the other is hardware optimizations. It is during this work found that there is reason to investigate certain part of the program further. In section 6.3 it was found that the effectiveness of the multigrid preconditioner depend on the effectiveness of the smoother. In particular, if the step length is small, it was found that it might be beneficial to vary the number of smoothings according grid level. If it is possible to reduce the number of smoothings at the fine level, the computation time will definitely be reduced. The problem is mostly of mathematical interest since the normal usage of this kind of solver would not need that kind of resolution. If a fine grid resolution is to be used, it should also be determined what the cause is to the divergent behavior for small step lengths.

Regarding the CUDA implementation, the bottleneck problems remain the high order residual and line smoother. For the high order residual table 5.13 suggests that it is possible to gain another 40% over the current implementation[1] if a good shared memory approach is found. Prior to this optimization it should be determined whether there is a benefit from applying shared memory on a Fermi GPU. Another 5-10% can be found in detecting the optimal kernel configurations for the particular problem solved: The effectiveness of the bottleneck kernels depend on the combination of grid layout and launch configuration. Since a

[1] Assumed the code is executed on a Quadro FX 5800 device or similar

simulation is in the scale of thousands of time steps, there is a potential speedup by using the correct launch configuration.

Appendix A

σ-transform and derivations

Given is the potential flow formulation.

$$\partial_t \eta = -\nabla\eta \cdot \nabla\tilde{\phi} + \tilde{w}(1 + \nabla\eta \cdot \nabla\eta) \tag{A.1}$$

$$\partial_t \tilde{\phi} = -g\eta - \frac{1}{2}(\nabla\tilde{\phi} \cdot \nabla\tilde{\phi} - \tilde{w}^2(1 + \nabla\eta \cdot \nabla\eta)) \tag{A.2}$$

The σ-transform will affect only the calculation of the velocity potential in the fluid.

$$\phi = \tilde{\phi}, \qquad z = \eta \tag{A.3}$$

$$\nabla^2\phi + \partial_{zz}\phi = 0, \qquad -h \leq z < \eta \tag{A.4}$$

$$\partial_z\phi + \nabla h \cdot \nabla\phi = 0, \qquad z = -h \tag{A.5}$$

As stated, the intention of performing the σ-transform is to create a cuboid time invariant domain. The change of variable should therefore be some mapping in the vertical which scales the horizontal domain accordingly. In order to do so, a linear mapping of the coordinate is introduced.

$$\sigma \equiv \frac{z + h(\boldsymbol{x})}{\eta(\boldsymbol{x}, t) + h(\boldsymbol{x})} \equiv \frac{z + h(\boldsymbol{x})}{d(\boldsymbol{x}, t)}, \quad d(\boldsymbol{x}, t) = \eta(\boldsymbol{x}, t) + h(\boldsymbol{x}) \tag{A.6}$$

Hence the computational domain is located in $0 \leq \sigma \leq 1$. Changing variable will affect the derivatives:

$$\begin{aligned}
\Phi &\equiv \phi(\sigma) && \text{(A.7)}\\
\partial_x\phi &= \partial_x\Phi\partial_x x + \partial_y\Phi\partial_x y + \partial_\sigma\Phi\partial_x\sigma = \partial_x\Phi + \partial_\sigma\Phi\partial_x\sigma && \text{(A.8)}\\
\partial_y\phi &= \partial_y\Phi\partial_x y + \partial_y\Phi\partial_y y + \partial_\sigma\Phi\partial_y\sigma = \partial_y\Phi + \partial_\sigma\Phi\partial_y\sigma && \text{(A.9)}\\
\partial_z\phi &= \partial_z\Phi\partial_x z + \partial_z\Phi\partial_y z + \partial_\sigma\Phi\partial_z\sigma = \partial_\sigma\Phi\partial_z\sigma && \text{(A.10)}
\end{aligned}$$

The second derivative

$$\begin{aligned}
\nabla^2\phi &= \begin{bmatrix} \partial_x & \partial_y \end{bmatrix} \begin{bmatrix} \partial_x\phi \\ \partial_y\phi \end{bmatrix} = \partial_{xx}\phi + \partial_{yy}\phi && \text{(A.11)}\\
\partial_{xx}\phi &= \partial_x\left(\partial_x\phi\right) && \text{(A.12)}\\
&= \partial_x\left(\partial_x\Phi + \partial_\sigma\Phi\partial_x\sigma\right) && \text{(A.13)}\\
&= \partial_x\left(\partial_x\Phi\right) + \partial_x\left(\partial_\sigma\Phi\partial_x\sigma\right) && \text{(A.14)}
\end{aligned}$$

The latter term is transformed using the product rule:

$$\begin{aligned}
\partial_x\left(\partial_\sigma\Phi\partial_x\sigma\right) &= \partial_x\left(\partial_\sigma\Phi\right)\partial_x\sigma + \partial_\sigma\Phi\partial_x\left(\partial_x\sigma\right) && \text{(A.15)}\\
\partial_{xx}\phi &= \partial_x\left(\partial_x\Phi\right) + \partial_x\left(\partial_\sigma\Phi\right)\partial_x\sigma + \partial_\sigma\Phi\partial_{xx}\sigma && \text{(A.16)}
\end{aligned}$$

The chain rule is applied to $\partial_x\left(\partial_\sigma\Phi\right)$ and $\partial_x\left(\partial_x\Phi\right)$.

$$\begin{aligned}
\partial_x\left(\partial_x\Phi\right) &= \partial_{xx}\Phi + \partial_{x\sigma}\Phi\partial_x\sigma && \text{(A.17)}\\
\partial_x\left(\partial_\sigma\Phi\right) &= \partial_{x\sigma}\Phi + \partial_{\sigma\sigma}\Phi\partial_x\sigma && \text{(A.18)}\\
\partial_{xx}\phi &= \left(\partial_{xx}\Phi + \partial_{x\sigma}\Phi\partial_x\sigma\right) && \text{(A.19)}\\
&\quad + \left(\partial_{x\sigma}\Phi + \partial_{\sigma\sigma}\Phi\partial_x\sigma\right)\partial_x\sigma + \partial_\sigma\Phi\partial_{xx}\sigma && \text{(A.20)}\\
\partial_{xx}\phi &= \partial_{xx}\Phi + 2\partial_{x\sigma}\Phi\partial_x\sigma + \partial_{\sigma\sigma}\Phi\partial_x\sigma\partial_x\sigma + \partial_\sigma\Phi\partial_{xx}\sigma && \text{(A.21)}
\end{aligned}$$

The second partial derivative in the y-direction will thus be

$$\partial_{yy}\phi = \partial_{yy}\Phi + 2\partial_{y\sigma}\Phi\partial_y\sigma + \partial_{\sigma\sigma}\Phi\partial_y\sigma\partial_y\sigma + \partial_\sigma\Phi\partial_{yy}\sigma \qquad \text{(A.22)}$$

And finally for the Laplacian, we get

$$\begin{aligned}
\nabla^2\phi &= \begin{bmatrix} \partial_x & \partial_y \end{bmatrix} \begin{bmatrix} \partial_x\phi \\ \partial_y\phi \end{bmatrix} = \partial_{xx}\phi + \partial_{yy}\phi && \text{(A.23)}\\
\nabla^2\phi &= \nabla^2\Phi + 2\nabla\left(\partial_\sigma\Phi\right)\nabla\sigma + \partial_{\sigma\sigma}\Phi\nabla\sigma\cdot\nabla\sigma + \partial_\sigma\Phi\nabla^2\sigma && \text{(A.24)}
\end{aligned}$$

As seen, spatial derivatives of sigma are needed as well.

$$\partial_z\sigma = \partial_z\left(\frac{z + h(\boldsymbol{x})}{d(\boldsymbol{x},t)}\right) = \frac{1}{d(\boldsymbol{x},t)} \qquad \text{(A.25)}$$

$$\nabla\sigma = \begin{bmatrix} \partial_x \\ \partial_y \end{bmatrix}\sigma = \begin{bmatrix} \partial_x \\ \partial_y \end{bmatrix}\left(\frac{z+h\left(\boldsymbol{x},t\right)}{d\left(\boldsymbol{x},t\right)}\right) \tag{A.26}$$

$$\partial_x\sigma = \partial_x\left(\frac{z+h}{d}\right) \tag{A.27}$$

$$= \frac{d\partial_x\left(z+h\right)-\left(z+h\right)\partial_x d}{d^2} \tag{A.28}$$

$$= \frac{d}{d}\frac{\partial_x h}{d} - \frac{z+h}{d}\frac{\partial_x d}{d} \tag{A.29}$$

$$= \frac{\partial_x h}{d} - \sigma\frac{\partial_x d}{d}, \quad \partial_x d = \partial_x\eta + \partial_x h \tag{A.30}$$

$$= (1-\sigma)\frac{\partial_x h}{d} - \sigma\frac{\partial_x\eta}{d} \tag{A.31}$$

The partial derivative in the y-direction is similar due to the symmetry of the problem. Finally, the gradient is given as a vector of partial derivatives.

$$\begin{bmatrix} \partial_x\sigma \\ \partial_y\sigma \end{bmatrix} = (1-\sigma)\begin{bmatrix} \frac{\partial_x h}{d} \\ \frac{\partial_y h}{d} \end{bmatrix} - \sigma\begin{bmatrix} \frac{\partial_x\eta}{d} \\ \frac{\partial_y\eta}{d} \end{bmatrix} \tag{A.32}$$

$$\nabla\sigma = (1-\sigma)\frac{\nabla h}{d} - \sigma\frac{\nabla\eta}{d} \tag{A.33}$$

To find the Laplacian of σ, we will need to find the second partial derivatives since $\nabla^2\sigma = \partial_{xx}\sigma + \partial_{yy}\sigma$.

$$\partial_{xx}\sigma = \partial_x\left((1-\sigma)\frac{\partial_x h}{d}\right) - \partial_x\left(\sigma\frac{\partial_x\eta}{d}\right) \tag{A.34}$$

For the first term

$$\partial_x\left((1-\sigma)\frac{\partial_x h}{d}\right) = -\partial_x\sigma\frac{\partial_x h}{d} + (1-\sigma)\partial_x\left(\frac{\partial_x h}{d}\right) \tag{A.35}$$

$$\partial_x\left(\frac{\partial_x h}{d}\right) = \frac{d\partial_{xx}h - \partial_x h\partial_x d}{d^2} \tag{A.36}$$

$$\partial_x\left((1-\sigma)\frac{\partial_x h}{d}\right) = -\frac{\partial_x\sigma}{d}\partial_x h + \frac{(1-\sigma)}{d}\left(\partial_{xx}h - \frac{\partial_x h\partial_x d}{d}\right) \tag{A.37}$$

For the second term

$$\partial_x\left(\sigma\frac{\partial_x\eta}{d}\right) = \partial_x\sigma\frac{\partial_x\eta}{d} + \sigma\partial_x\left(\frac{\partial_x\eta}{d}\right) \tag{A.38}$$

$$\partial_x\left(\frac{\partial_x\eta}{d}\right) = \frac{d\partial_{xx}\eta - \partial_x\eta\partial_x d}{d^2} \tag{A.39}$$

$$\partial_x\left(\sigma\frac{\partial_x\eta}{d}\right) = \frac{\partial_x\sigma}{d}\partial_x\eta + \frac{(1-\sigma)}{d}\left(\partial_{xx}\eta - \frac{\partial_x\eta\partial_x d}{d}\right) \tag{A.40}$$

Gathering terms and inserting $\partial_x d = \partial_x \eta + \partial_x h$ reveals

$$\partial_{xx}\sigma = -\frac{\partial_x \sigma}{d}\partial_x h \tag{A.41}$$
$$+\frac{(1-\sigma)}{d}\left(\partial_{xx}h - \frac{\partial_x h \partial_x h}{d} + \frac{\partial_x h \partial_x \eta}{d}\right) \tag{A.42}$$
$$-\frac{\partial_x \sigma}{d}\partial_x \eta \tag{A.43}$$
$$-\frac{\sigma}{d}\left(\partial_{xx}\eta - \frac{\partial_x \eta \partial_x \eta}{d} + \frac{\partial_x h \partial_x \eta}{d}\right) \tag{A.44}$$

Rearranging a bit, a usable result is finally obtained

$$\partial_{xx}\sigma = \frac{(1-\sigma)}{d}\left(\partial_{xx}h - \frac{\partial_x h \partial_x h}{d}\right) - \frac{\sigma}{d}\left(\partial_{xx}\eta - \frac{\partial_x \eta \partial_x \eta}{d}\right) - \frac{\partial_x \sigma}{d}(\partial_x h + \partial_x \eta) + \frac{(1-2\sigma)}{d}\left(\frac{\partial_x h \partial_x \eta}{d}\right) \tag{A.45}$$

Since the operation is symmetric in terms of dimensions, $\partial_{yy}\sigma$ is similar to $\partial_{xx}\sigma$. $\partial_{yy}\sigma$ can therefore be concluded to be given by

$$\partial_{yy}\sigma = \frac{(1-\sigma)}{d}\left(\partial_{yy}h - \frac{\partial_y h \partial_y h}{d}\right) - \frac{\sigma}{d}\left(\partial_{yy}\eta - \frac{\partial_y \eta \partial_y \eta}{d}\right) \tag{A.46}$$
$$-\frac{\partial_y \sigma}{d}(\partial_y h + \partial_y \eta) + \frac{(1-2\sigma)}{d}\left(\frac{\partial_y h \partial_y \eta}{d}\right) \tag{A.47}$$

Finally, the Laplacian of σ can also be given in vectorized form which will be

$$\partial_{xx}\sigma + \partial_{yy}\sigma = \frac{(1-\sigma)}{d}\left(\partial_{xx}h + \partial_{yy}h - \frac{\partial_x h \partial_x h + \partial_y h \partial_y h}{d}\right) \tag{A.48}$$
$$-\frac{\sigma}{d}\left(\partial_{xx}\eta + \partial_{yy}\eta - \frac{\partial_x \eta \partial_x \eta + \partial_y \eta \partial_y \eta}{d}\right) \tag{A.49}$$
$$-\frac{\partial_x \sigma}{d}(\partial_x h + \partial_x \eta) - \frac{\partial_y \sigma}{d}(\partial_y h + \partial_y \eta) \tag{A.50}$$
$$+\frac{(1-2\sigma)}{d^2}(\partial_x h \partial_x \eta + \partial_y h \partial_y \eta) \tag{A.51}$$
$$\nabla^2 \sigma = \frac{(1-\sigma)}{d}\left(\nabla^2 h - \frac{\nabla h \cdot \nabla h}{d}\right) \tag{A.52}$$
$$-\frac{\sigma}{d}\left(\nabla^2 \eta - \frac{\nabla \eta \cdot \nabla \eta}{d}\right) \tag{A.53}$$
$$-\frac{\nabla \sigma}{d} \cdot (\nabla h + \nabla \eta) \tag{A.54}$$
$$+\frac{(1-2\sigma)}{d^2}\nabla h \nabla \eta \tag{A.55}$$

Appendix B

Underline Notation

Underline notation is a notation relating to a graphical representation of a discrete function. The notation is often used in DSP to represent discrete filters and has nothing to do with vectors although looking like one. The underlined element shows the center element of the filter/weighted sum.

$$f(n) = \left[\begin{array}{ccc} -\frac{1}{2} & \underline{0} & \frac{1}{2} \end{array} \right] \tag{B.1}$$

Since the center element is underlined, the indexing for this particular function is

$$n = \left[\begin{array}{ccc} -1 & \underline{0} & 1 \end{array} \right] \tag{B.2}$$

The correct interpretation of (B.1) is therefore

$$f(-1) = \tfrac{1}{2} \quad f(0) = 0 \quad f(1) = \tfrac{1}{2} \tag{B.3}$$

and zero elsewhere. For discrete functions of higher dimension, the underlined element again shows the center element

$$f(n,m) = \left[\begin{array}{ccc} \frac{3}{4} & \underline{0} & -\frac{3}{4} \\ -1 & 0 & 1 \\ \frac{1}{4} & 0 & -\frac{1}{4} \end{array} \right] \tag{B.4}$$

The corresponding implicit numbering of elements is then

$$(n,m) = \left[\begin{array}{ccc} (0,-1) & (0,0) & (1,1) \\ (1,-1) & (1,0) & (2,1) \\ (2,-1) & (2,0) & (3,1) \end{array} \right] \tag{B.5}$$

The correct interpretation of (B.1) is therefore

$$\begin{array}{lll} f(0,-1)=\frac{3}{4} & f(0,0)=0 & f(0,1)=-\frac{3}{4} \\ f(1,-1)=-1 & f(1,0)=0 & f(1,1)=1 \\ f(2,-1)=\frac{1}{4} & f(2,0)=0 & f(2,1)=-\frac{1}{4} \end{array} \tag{B.6}$$

and zero elsewhere. In order to make the notation flexible, the stencil is also bound to a direction. The stencil S_x is thus applied in the x-direction of the domain and S_y in the y-direction of the domain:

$$S_x = \begin{bmatrix} 2 & 1 & 3 \end{bmatrix}_x, \quad S_y = \begin{bmatrix} 3 & 1 & 2 \end{bmatrix}_y \tag{B.7}$$

The notation benefits from being compact and allows for 3D stencils to be properly represented on a (2D) piece of paper.

B.0.1 Mixed derivatives

The grid is regular with grid spacing Δx and Δy in the x- and y-dimension respectively. To illustrate the versatility, a centered approximation is applied in the horizontal x-direction and an off-centered approximation is used in the vertical z-direction. The mixed derivative is found by the tensor product of S_x and S_z which in underline notation is simplified to a vector-vector product:

$$S_x = \frac{1}{\Delta x}\begin{bmatrix} -\frac{1}{2} & \underline{0} & \frac{1}{2} \end{bmatrix}_x, \quad S_y = \frac{1}{\Delta z}\begin{bmatrix} \underline{-\frac{3}{2}} & 2 & -\frac{1}{2} \end{bmatrix}_z \tag{B.8}$$

$$S_{xz} = S_z^T S_x \tag{B.9}$$

$$S_{xz} = \frac{1}{\Delta x \Delta z}\begin{bmatrix} \frac{3}{4} & \underline{0} & -\frac{3}{4} \\ -1 & 0 & 1 \\ \frac{1}{4} & 0 & -\frac{1}{4} \end{bmatrix}_{xz} \tag{B.10}$$

The primary advantage of calculating the mixed derivatives using tensor products is that the method is robust and reduce the risk of programming errors. In depth code of implementation of the mixed derivative is available in listing D.4, appendix D.

Appendix C

Platforms

The platforms used are specified by the tables below

HP Z800 workstation:

Operating system	Windows 7 (64-bit)
CPU	Intel Xeon W5590, 3.33GHz
RAM	12GB (PC-10700)
GPU	NVIDIA Quadro FX 5800
Display driver version	257.21
CUDA Runtime Version	2.3

NVIDIA Quadro FX 5800

Compute Capability	1.3
RAM	4.0 GB
Multiprocessors	30
Number of cores	240
Memory bandwidth	$102.4^{\mathrm{GB}/s}$
Clock rate	1.30 GHz

ProConsole workstation:

Operating system	Windows 7 (64-bit)
CPU	Intel Core i7 Extreme 9065, 3.20GHz
RAM	3GB (PC-10700)
GPU	NVIDIA GeForce 480 GTX
Display driver version	260.99
CUDA Runtime Version	2.3

NVIDIA GeForce 480 GTX

Compute Capability	2.0
RAM	1.5 GB
Multiprocessors	60
Number of cores	480
Memory bandwidth	$177.4^{\mathrm{GB}}/_{s}$
Clock rate	1.40 GHz

Appendix D

CUDA implementations

For the sake of completeness this chapter presents either full or pseudo implementation of most of the components used. The total grid/memory usage of the program is specified by table D.1.

D.1 Finite difference estimates

Calculating the finite difference estimate at some given point is then done in three steps: Calculate which stencil to use, then apply stencil and finally, scale according to step length in the given direction. For the x-direction we will calculate the stencil number `snx` by

Listing D.1: Calculate stencil number

```
1 int maxdist = (stencilwidth-1)/2; //NB! stencilwidth is ↩
    always odd
2 int snx = maxdist; //stencil number (x-direction)
3 if(Tx < maxdist){
4   //left asymmetric stencil
5   snx -= maxdist-Tx;
6 }
```

Table D.1: Overview of memory consuming variables

Variable	Type	Memory consumpsion	usage
dc_x	3D Multigrid	$N_x N_y N_z$	Iterate of DC
dc_b	3D Multigrid	$N_x N_y N_z$	Rhs of DC
mg_x0	3D Multigrid	$N_x N_y N_z$ + recursions	Iterate of CGC
mg_x1	3D Multigrid	$N_x N_y N_z$ + recursions	Iterate of CGC
mg_r	3D Multigrid	$N_x N_y N_z$ + recursions	Iterate of CGC
mg_b	3D Multigrid	$N_x N_y N_z$ + recursions	Iterate of CGC
E	2D Multigrid	$2(N_x N_y$ + recursions)	Surface variable
P	2D Multigrid	$2(N_x N_y$ + recursions)	Surface variable
Ex	2D Multigrid	$N_x N_y$ + recursions	Surface derivative
Exx	2D Multigrid	$N_x N_y$ + recursions	Surface derivative
Ey	2D Multigrid	$N_x N_y$ + recursions	Surface derivative
Eyy	2D Multigrid	$N_x N_y$ + recursions	Surface derivative
h	2D Multigrid	$N_x N_y$ + recursions	Bottom variable
hx	2D Multigrid	$N_x N_y$ + recursions	Bottom derivative
hxx	2D Multigrid	$N_x N_y$ + recursions	Bottom derivative
hy	2D Multigrid	$N_x N_y$ + recursions	Bottom derivative
hyy	2D Multigrid	$N_x N_y$ + recursions	Bottom derivative
k_Edt	2D grid	$4N_x N_y$	Runge Kutta stages 1-4
k_Pdt	2D grid	$4N_x N_y$	Runge Kutta stages 1-4

```
7  else if(M.x-1 < Tx + maxdist){
8    //right asymmetric stencil
9    snx += (maxdist+Tx)-(M.x-1);
10 }
```

Knowing the stencil number we can now apply the stencil

Listing D.2: Apply stencil

```
1 for(int i = 0; i<stencilwidth; i++){
2   int offsetx = i+snx*stencilwidth;
3   _ftype sv = stencil1D[i];
4   Ux = sv*x[memoryIdxXY(offsetx,0,Z, M)];
5 }
```

Finally the stencil should be scaled according to the (uniform) step length in the given direction

Listing D.3: Scale finite difference sum

```
1 Ux *= (Lx/(N.x-1)) // 1/hx
```

For mixed derivatives, we will simply expand listing D.1 with a nested loop and apply appropriate scaling for both directions

Listing D.4: Apply stencils

```
1  for(int k = 0; k<stencilwidth; k++){ //inner loop z
2    int sz =   k + snz*stencilwidth;
3    int oz = -(k - snz); //stencil is reversed compared to x-↩
         and y-direction
4
5    for(int i = 0; i<stencilwidth; i++){ //inner loop x
6      int sx = i + snx*stencilwidth;
7      int ox = i - snx;
8
9      //combined stencil value
10     _ftype sv = stencil1[sx]
11                *stencil1[sz];
12
13     Ax_Uxs += sv*igrid[memoryIdxXY(ox,0,Z + oz, memsize)];
14   }
15 }
16 Ax_Uxs *= Lx/(N.x-1)*Ls/(N.z-1); // 1/(hx hsigma)
```

Notice that the offsetting in the vertical direction is reversed (line 3) compared to the horizontal direction. This is due to an increment in `z` imply decrement in σ (remember that surface values are stored at `z=0` and bottom values at `z = Nz-1`).

D.1.1 Low order linear approximations

The entire Coarse Grid Correction part of the program will be using low order linear approximations. Since the approximation order will therefore be fixed to 2 and there will be no mixed derivatives, complexity of the program can be decreased by fixing the loop presented in listing D.2. As a small benifit, the performance will most likely also be a bit better since there will be less register use and one loop overhead less to process. For the x-direction, the fixed function derivatives will be given by

Listing D.5: Fixed 2^{nd}-order derivatives

```
1 Ux  = (0.5*U[memoryIdxXY(-1,...)]
```

```
//        0*U[memoryIdxXY( 0,...)]
       -0.5*U[memoryIdxXY( 1,...)])/hx;
Uxx = (    U[memoryIdxXY(-1,...)]
         -2*U[memoryIdxXY( 0,...)]
         +  U[memoryIdxXY( 1,...)])/(hx*hx)
```

D.2 Updating ghost points

For the 2D case, we will update opposing sides concurrently.

A simple way to implement the procedure in CUDA is to allocate threads according to the length of the side being updated; updating the ghost points $\eta_{x,-h} = \eta_{x,h}$ can be done in with M_x threads.

Listing D.6: Update ghost points along x-axis

```
1  if(!(Tx < M.x))
2    return;
3  int t = total_ghost_points;
4
5  for(int i = 1; i <= t;i++){
6    //gp: ghost point (index)
7    //vp: value to copy - value point (index)
8    int gp = memoryIdxXY( 0, t-i,0, M);
9    int vp = memoryIdxXY( 0, t+i,0, M);
10
11   grid[gp] = grid[vp];
12
13   gp = memoryIdxXY( 0, M.y-1-t+i,0, M);
14   vp = memoryIdxXY( 0, M.y-1-t-i,0, M);
15
16   grid[gp] = grid[vp];
17 }
```

For periodic boundary conditions, lines 9 and 14 are swapped and modified slightly:

Listing D.7: Update ghost points along x-axis (periodic domain)

```
1 int gp = memoryIdxXY( 0, t-i, 0, M);
2 int vp = memoryIdxXY( 0, M.y-1-t-(i-1),0, M);
3 //...
```

```
4  gp = memoryIdxXY( 0, M.y-1-t+i,0, M);
5  vp = memoryIdxXY( 0, t+(i-1),0, M);
```

The ghost update along the y-axis is quite similar since nearly everything is the same. Only the indexing and initial safety check will need to be changed;

Listing D.8: Update ghost points along y-axis (bounded domain)

```
1  if(!(Ty < M.y))
2    return;
3  int t = total_ghost_points;
4
5  for(int i = 1; i <= t;i++){
6    //gp: ghost point (index)
7    //vp: value to copy - value point (index)
8    int gp = memoryIdxXY( t-i,0,0, M);
9    int vp = memoryIdxXY( t+i,0,0, M);
10
11   grid2D[gp] = grid2D[vp];
12
13   gp = memoryIdxXY( M.x-1-t+i,0,0, M);
14   vp = memoryIdxXY( M.x-1-t-i,0,0, M);
15
16   grid2D[gp] = grid2D[vp];
17 }
```

Listing D.9: Update ghost points along y-axis (periodic domain)

```
1  int gp = memoryIdxXY( t-i,0,0, M);
2  int vp = memoryIdxXY( M.x-1-t-(i-1),0,0, M);
3  //...
4  gp = memoryIdxXY( M.x-1-t+i,0,0, M);
5  vp = memoryIdxXY( t+(i-1),0,0, M);
```

Launch configuration should then be 1D with enough threads to cover the entire domain, ghost points included. The block size is set to 32 elements which is the size of a warp. The block size is subject to optimization which is addressed in section 5.1.

```
1  int nt = 32; //number of threads
2
3  //sides
4  if(N.x > 1){
5    dim3 threads(1,nt,1); //update along y-axis
```

```
6     dim3 blocks = fitblock(threads, 1, M.y);
7     if(periodic_x)
8       kernel::ghostupdateX2Dp<<<blocks, threads>>>(grid2D, M)↩
            ;
9     else
10      kernel::ghostupdateX2D<<<blocks, threads>>>(grid2D, M);
11  }
12
13  if(N.y > 1){
14    dim3 threads(nt,1,1); //update along x-axis
15    dim3 blocks = fitblock(threads, M.x, 0);
16    if(periodic_y)
17      kernel::ghostupdateY2Dp<<<blocks, threads>>>(grid2D, M)↩
            ;
18    else
19      kernel::ghostupdateY2D<<<blocks, threads>>>(grid2D, M);
20  }
```

D.2.1 3D ghost update - bottom boundary 1

Updating ghost points in 3D is not so different from updating in 2D. Rather than updating with 1D launch configuration, it should be a 2D configuration. For starters, the ghostupdate at the border for a low order linear problem is given by mirroring. There is always only one layer of ghost points so the program is very small.

Listing D.10: Update bottom ghost points for 2^{nd} order linear approximation

```
1  if(!(Tx < M.x && Ty < M.y))
2    return;
3
4  int gp = memoryIdxXY(0,0,M.z-1-1+1, M);
5  int vp = memoryIdxXY(0,0,M.z-1-1-1, M);
6  grid3D[gp] = grid3D[vp];
```

The initial launch configuration is set to 8×8.

```
1  if(linear && order == 2){
2    dim3 threads(8,8,1);
3    blocks = fitblock(threads, nx,ny);
4    kernel::ghostXY_linear<<<blocks, threads>>>(grid3D, M);
5  }
```

D.2.2 3D ghost update - bottom boundary 2

Updating the bottom boundary in the various order non linear case is somewhat more complex; all surface variables are needed, and larger stencils must be applied to the grid. The non linear bottom ghost layer update should be isolated similarly to the example given by (1.91).

```
1  int stencilNumberZ = (stencilwidth-1)-1; //not last but second last
2  #define snz stencilNumberZ
3
4  //calculate Us_hat, Ux, Uy, sigmax, sigmay
5  //...
6  //isolate ghost point
7
8  int sz = k+snz*stencilwidth; //index to pull from stencil
9  _ftype sv = stencil1[sz];//stencil value
10 sv /= hz;
11
12 int n = memoryIdxXY( 0, 0,M.z-1-1+1, M);
13 U[n] = -(Us_hat +
14           (hx_*Ux + hy_*Uy)/(sigmaz + (hx_*sigmax + hy_*sigmay))
15         )/sv;
```

where the values `Us_hat`, `Ux` and `Uy` are the finite difference sums approximating $\hat{U}_\sigma$, U_x and U_y.

Depending on the previous code, the value of the ghost point prior to the update might be undefined. It is therefore advisable to avoid including the ghost point value in the computation (line 5 of the code below).

```
1  _ftype Us_hat = 0;
2  for(int k = 0; k<stencilwidth; k++){
3    int oz = -(k-snz); //z-offset in stencil
4
5    if(oz == 1) //ghostlayer! Do not include in computation
6      continue;
7
8    int sz = k + snz*stencilwidth; //index to pull from stencil
9    _ftype sv = stencil1[sz];
10
11   Us_hat += sv*U[memoryIdxXY(0,0,Z + oz, memsize)];
```

```
12 }
13 Us_hat *= Ls/(N.z-1);
```

D.3 Low order residual

Recall that $\boldsymbol{x}$ and $\boldsymbol{b}$ all are 3D grid variables.

Listing D.11: Calculation of residual

```
1  _ftype d = h[n]; //linear Laplace; eta = 0
2  _ftype sigmaz = 1/d;
3
4  //calculate residual
5  int n = memoryIdxXY(0,0,0, M);
6  r[n] = b[n]-x[n]; //surface residual
7
8  for(int Z = 1; Z<Nz;Z++){
9    n = memoryIdxXY(0,0,Z, M);
10   //...
11   //calculate Uxx, Uyy, Uss, sigmaz
12   //...
13   _ftype Ax = Uxx + Uyy + (sigmaz*sigmaz)*Uss;
14   r[n] = b[n]-Ax;
15 }
```

The finite difference sums `Uxx`, `Uyy` and `Uss` are calculated using the procedures introduced in section 4.3 (code: section D.1).

D.4 Damped Jacobi

The vectors $\boldsymbol{x}^{[t]}$ and $\boldsymbol{x}^{[t+1]}$ are represented by the 3D grid variables `x0` and `x1` respectively. With an exception of a few lines of code in listing D.11, the kernels are identical.

Listing D.12: Implementation of damped Jacobi relaxation

```
1  _ftype d = h; //linear Laplace: eta = 0
2  _ftype sigmaz = 1/d;
3
```

```
4  //Surface element
5  int n = memoryIdxXY(0,0,0, M);
6  x1[n] = b[n]; //surface residual
7
8  for(int Z = 1; Z<Nz;Z++){
9    n = memoryIdxXY(0,0,Z, M);
10   //...
11   //calculate Uxx, Uyy, Uss
12   //...
13   _ftype Ax = Uxx + Uyy + (sigmaz*sigmaz)*Uss;
14   _ftype r = b[n]-Ax;
15   x1[n] = x0[n] + lambda*r;
16 }
```

The launch will be the same as for the residual:

```
1  updateghostpoints(x0, ...);
2
3  dim3 threads;
4  threads.x = 8; threads.y = 8;
5  blocks = fitblock(threads, domainsize.x, domainsize.y);
6  dampedJacobi<<<blocks, threads>>>(x1, x0, b, ...)
```

D.5 Restriction

A simple way to construct the kernel is by a gather scheme; let there be one thread per coarse grid surface element (without ghost points) and let each thread process all elements in a single vertical grid row. The update is then given by looping through the entire domain of the stencil. Since we use C++ array access style, grid elements $y_{i,j,k}$ and $x_{2i,2j,2k}$ will be on top of each other when $\boldsymbol{y}$ and $\boldsymbol{x}$ are coarse and fine grid variables respectively. We can therefore use the utility function `memoryIdxXY`, also to access the fine grid elements by just adding the thread index as an offset to the lookup function (lines 2 and 10 of the code below).

D.5.1 3D coarsening - initial kernel

```
1  _ftype s1D[3] = {0.25, 0.5, 0.25};
2  int n = memoryIdxXY(X, Y, 2*K+(k-1), Mf);
```

```
cgrid[memoryIdxXY(0,0,K, Mc)] = fgrid[n];

for(K = 1; K != cgridsize.z; K++){
  sum = 0;
  for(int k = 0; k < 3; k++){
    for(int j = 0; j < 3; j++){
      for(int i = 0; i < 3; i++){
        memoryIdxXY(X+(i-1), Y+(j-1), 2*K+(k-1), Mf):
        //multiply with stencil tensor product.
        sum += s1D[i]*s1D[j]*s1D[k]*fgrid[n];
      }
    }
  }
  cgrid[memoryIdxXY(0,0,K, Mc)] = sum;
}
```

Here `Mc` and `Mf` are domain sizes of the fine and coarse grid respectively (ghost points included). Recall that the thread indicies are used to identify which grid element to process and load.

D.5.2 2D coarsening - initial kernel

The strategy is the same as for the 3D case although the kernel alters a bit. For xz-coarsening, we get

```
_ftype s1D[3] = {0.25, 0.5, 0.25};
int n = memoryIdxXY(X, 0, 2*K, Mf);
cgrid[memoryIdxXY(0,0,K, Mc)] = fgrid[n];

for(K = 1; K != cgridsize.z; K++){
  sum = 0;
  for(int k = 0; k < 3; k++){
    for(int i = 0; i < 3; i++){
      memoryIdxXY(X+(i-1), 0, 2*K+(k-1), Mf);
      //multiply with stencil tensor product.
      sum += s1D[i]*s1D[k]*fgrid[n];
    }
  }
  cgrid[memoryIdxXY(0,0,K, Mc)] = sum;
}
```

Notice here everything regarding the y-direction is simply omitted.

D.5.3 1D coarsening - initial kernel

Once again, we just ommit the directions of the irrelevant directions. For x-coarsening we get

```
int n = memoryIdxXY(X, 0, K, Mf);
cgrid[memoryIdxXY(0,0,K, Mc)] = fgrid[n];

for(K = 1; K != cgridsize.z; K++){
  sum = 0;
  for(int i = 0; i < 3; i++){
    memoryIdxXY(X+(i-1), 0, K, Mf);
    //multiply with stencil tensor product.
    sum += s1D[i]*fgrid[n];
  }
  cgrid[memoryIdxXY(0,0,K, Mc)] = sum;
}
```

D.5.4 Launch configuration

Since the coarsening strategy depend on which coarsening strategy is applied, the correct kernel should be executed. Prior to excecution, the ghost points should be up to date.

```
ASSERT(ghost points up to date); //pseudo assert...

bool cx = Nc.x < Nf.x, cy = Nc.y < Nf.y, cz = Nc.z < Nf.z;
bool cxy =  cx &  cy & !cz,   cxz  = cx & !cy &  cz;
bool cyz = !cx &  cy &  cz,   cxyz = cx &  cy &  cz;

cx = !(Nc.y < Nf.y) & !(Nc.z < Nf.z);
cy = !(Nc.x < Nf.x) & !(Nc.z < Nf.z);
cz = !(Nc.x < Nf.x) & !(Nc.y < Nf.y);

blocks = fitblock(threads, Mc.x, Mc.y);
if(cx){
  kernel::coarseX<<<blocks,threads>>>(cgrid,Nc, fgrid,Nf);
}else if(cy)
  ...
```

D.6 μ-cycle

By convesion, `x1` is an output pointer; it should always point to the updated grid variable. Upon invokation, `x1` might, might not be an instance. This will be known to the smoother only. Notice that the ghost update are inserted here although most sample code suggest to update the ghost points as part of the given procedure. The reason to not put the ghost update inside the procedure is that performance measurement will be easier if only one kernel is associated to each of the components.

```
1  void vcycle(_ftype *&x1, _ftype *&x0, _ftype *b, int3 N, ↩
       ...){
2    ASSERT(order == 2 && linear);
3
4    int3 cN = coarsen(options, N);
5    if( cN == N ){
6        //...
7        //Coarse Grid Solver
8        //...
9
10       return;
11   }
12   for(int i = 0; i < options.v1 ;i++){
13     device::smooth(x1, x0, b, N, ...);
14     std::swap(x0, x1); //swap data
15   }
16   std::swap(x0, x1); //unswap data
17
18   device::ghostupdate(x1, N, ...);
19   device::loworderresidual(r, x1, b, N, ...);
20
21   device::ghostupdate(r, N, ...);
22   device::coarsen3D(cb, cN, r, N);
23
24   //initial guess for coarse grid
25   device::fill(cx0, cx0 + grid3Dmem(cN), (_ftype)0);
26
27   //proceed recursively...
28   for(int u = 0; u < options.ucycle; u++){
29     vcycle(cx1, cx0, cb, ...);
30     std::swap(cx1, cx0);
31   }
32   std::swap(cx1, cx0);
33
34   device::ghostupdate(cx1, cN, ...);
```

```
35    device::interpolateAndAdd(x1, N, cx1, cN);
36
37    for(int v2 = 0; v2<options.v2;v2++){
38      std::swap(x0, x1);
39      device::smooth(x1, x0, b, N, ...);
40    }
41  }
```

D.7 Various order non linear residual

Most of the elements of the various order residual is described by other components. The surface variables (`E`, `Ex`, `Exx`, `Ey`, `Eyy`, `h`, `hx`, `hxx`, `hy`, `hyy`) are precalculated, thus loaded from global memory when prior to calculation of the derivatives in σ.

Listing D.13: Various order non linear residual

```
1  int n = memoryIdxXY(0,0,0, memsize);
2  r[n] = b[n]-U[n];
3  for(int Z = 1; Z<nz;Z++){ //outer loop z
4    _ftype sigmax, sigmaxx, sigmay, sigmayy;
5
6    //...
7    //Calculate
8    //sigmax, sigmaxx, sigmay and sigmayy
9    //Uxx, Uyy, Uss, Uxs, Uys, Us
10   //...
11
12   int n = memoryIdxXY(0,0,Z, memsize);
13   _ftype Ax =
14      Uxx + Uyy
15     +(sigmax*sigmax + sigmay*sigmay + sigmaz*sigmaz)*Uss
16     +(2*sigmax)*Uxs + (2*sigmay)*Uys
17     +(sigmaxx + sigmayy)*Us;
18
19   r[n] = b[n]-Ax;
20 }
```

The mentioned two pass algorithm will need shared memory. Both pass 1 and pass 2 shared memory is overwritten when the stencil does not touch an element any more. A very easy way to do that is by using the modulus operator.

Listing D.14: Shared memory access

```
__device__
inline int smemoryIdxXY(int ox, int oy, int z){
  return  (z % stencilwidth)*blockDim.x*blockDim.y
        + (threadIdx.y + oy)*blockDim.x
        +  threadIdx.x + ox;
}
```

Global memory on the other hand a bit more complex.

Listing D.15: Global memory access when using shared memory

```
#define Xh (-halo + (int)threadIdx.x + (int)blockIdx.x*(-2*halo+(int)blockDim.x))

__device__
inline int memoryIdxXY(int ox, int oy, int z, int3 M){
  int t = z*M.x*M.y
        + (Ty + Gy + oy)*M.x
        +  Xh + Gx + ox;
  return t;
}
```

Do notice that the thread indicies are explicitly casted to `int`. If not added, depending on the declaration of halo, the definition `Xh` might suffer from underflow.

Listing D.16: Safety checks

```
if(Xh < M.x && Y < N.y){
  //safe to read, also from halo zones
}
if(threadIdx.x >= halo && threadIdx.x < blockDim.x-halo){
  if(Xh < N.x && Y < N.y){
    //safe to write (i.e. avoid writes in halo zones)
  }
}
```

Finally, for the launch configuration there must be accounted for that only a part of the threads are performing any calculations. Also the configuration should cover the entire domain, including ghost points.

```
ASSERT(threads.x > 2*halo);
effectivethreads.x = threads.x-2*halo;
```

```
effectivethreads.y = threads.y;
blocks = fitblock(effectivethreads, M.x,N.y);
smemsize = sizeof(_ftype)*threads.x*threads.y*stencilwidth;
hres_pass1<<< blocks, threads, smemsize >>>(r,x,b, ...);
```

D.8 Line smoother

We will use the aliases a_0, a_1 and a_2 for the code rather than $\boldsymbol{a}_{-1}$, $\boldsymbol{a}_0$, $\boldsymbol{a}_1$ respectively.

Listing D.17: Step 1: Collect $\boldsymbol{a}_0$, $\boldsymbol{a}_1$, $\boldsymbol{a}_2$ and $\tilde{\boldsymbol{b}}$

```
for(int Z = 1; Z<N.z;Z++){
  //...
  //calculate off-center contributions of Uxx, Uyy//
  //...

  b[Z] = bgrid[memoryIdxXY(0,0,Z, memsize)];
  b[Z] -= offcenter_contrib; //now b tilde

  //calculate matrix values
  a1[Z] =  1* sigmaz*sigmaz/(hs*hs);
  a2[Z] = -2*(sigmaz*sigmaz/(hs*hs)
              + 1/(hx*hx) + 1/(hy*hy));
  a3[Z] =  1* sigmaz*sigmaz/(hs*hs);
}//Z-loop
```

Listing D.18: Step 2: Solve $\widehat{\boldsymbol{G}}^{+}\boldsymbol{x}_\sigma = \tilde{\boldsymbol{b}}$

```
int Z = 1;
_ftype scale;
for(   ;Z < N.z; Z++){
  scale = a1[Z]/a2[Z-1];
  a1[Z] = a1[Z]-a2[Z-1]*scale;
  a2[Z] = a2[Z]-a3[Z-1]*scale;
  b[Z]  = b[Z] - b[Z-1]*scale;
}

a1[Z] = a3[Z-2]/a2[Z-2];
a2[Z] = 1;
b[Z]  = b[Z-2]/a2[Z-2];

scale = a1[Z]/a2[Z-1];
```

```
15 a1[Z] = a1[Z]-a2[Z-1]*scale;
16 a2[Z] = a2[Z]-a3[Z-1]*scale;
17 b[Z]  = b[Z] - b[Z-1]*scale;
18
19 //backward substitution
20 b[Z] = b[Z]/a2[Z];
21
22 for(Z = N.z-1; Z >= 0; Z--){
23   b[Z] = (b[Z] - b[Z+1]*a3[Z])/a2[Z];
24   x[memoryIdxXY(0,0,Z, memsize)] = b[Z];
25 }
```

D.9 Defect Correction

Notice that no initial guess is set here; using the previous solution as initial guess is better than **0**. The RK4 integration algorithm sets the initial guess of the Defect Correction method to **0** for the at the very first time step.

```
1  do{
2    k++;
3
4    //set initial guess for preconditioner
5    device::fill(d0, d0 + grid3Dmem(gridsize), 0.0);
6    device::fill(d1, d1 + grid3Dmem(gridsize), 0.0);
7
8    //update bouding conditions
9    device::highorderresidual(r, x, b, N, ...);
10
11   //preconditioning problem/ low order error estimate
12   vcycle(d1, d0, r, N, ...);
13
14   //defect correction/update
15   device::plusequals(grid, grid + grid3Dmem(gridsize), d1);
16
17   //convergence test
18   norm = calc2norm(d1, gridsize);
19   std::swap(d0, d1); //for gauss-seidel type smoothers...
20
21   //test for convergence
22   if(k == 1){//first iteration
23     normfirst = norm;
24   }
25
```

```
26    converged = norm <= options.rtol*normfirst + options.atol;
27  }while(!converged && k < options.maxdciter);
```

D.10 Surface evolution

Evaluating the governing equations of the fluid motion problem can be split into a series of steps as described in section 1.1.1

```
1  device::ghostupdate2D(E, N);
2  device::ghostupdate2D(Phi_tilde, N);
3
4  //Update derivatives
5  calcderivatives(E, ...);
6
7  //Solve transformed Laplace equation
8  device::fill(b, b + grid3Dmem(N), (_ftype)0); //rhs
9  cudaMemcpy(b, Phi_tilde, grid2Dmem(gridsize)*sizeof(_ftype), D2D);
10
11 //Solve Laplace equation
12 defectcorrection(Phi, b, ...);
13
14 //2) evaluate eta and phi
15 kernel::calcEdtPdt<<<blocks, threads>>>(Edt, Pdt,
16   E, dmem.Ex, dmem.Ey,
17   dmem.h, dmem.dcgrid,
18   gridsize.x, gridsize.y, gridsize.z,
19   dmem.constants.stencils1_on, dmem.constants.stencilWidth_on,
20   Lx, Ly, options.linear);
```

The standard 4 step Runge Kutta method is presented below with a few modifications; in order to be able to apply tests of various kinds, a relaxation zone system is introduced. Inside a relaxation zone, the surface variables are set to follow some well known behaviour with typically one zone generating waves and another absorbing waves.

```
1  //...
2  //set initial guess of Defect Correction = 0
3  //...
4
```

```
5  while(t < tend){
6
7    //k1 = [Edt1, Pdt1]
8    evalDiffEq(Edt1, Pdt1, E0, P0, ...);
9
10   //k2 = [Edt2, Pdt2]
11   advanceEP(E1, P1, E0, P0, Edt1, Pdt1, 0.5*dt, ...);
12   update_relaxationzones(E1, P1, t + 0.5*dt, ...);
13   evalDiffEq(Edt3, Pdt3, E1, P1, ...);
14
15   //k3 = [Edt3, Pdt3]
16   advanceEP(E1, P1, E0, P0, Edt2, Pdt2, 0.5*dt, ...);
17   update_relaxationzones(E1, P1, ...)
18   evalDiffEq(Edt3, Pdt3, E1, P1, ...);
19
20   //k4 = [Edt4, Pdt4]
21   advanceEP(E1, P1, E0, P0, Edt3, Pdt3, 1.0*dt, ...);
22   update_relaxationzones(E1, P1,...);
23   evalDiffEq(Edt4, Pdt4, E1, P1, ...);
24
25   //gather [E P] = 1/6*(k1+2*k2+2*k3+k4)
26   advanceEP4(E1, P1, E0, P0,
27              Edt1, Pdt1, Edt2, Pdt2,
28              Edt3, Pdt3, Edt4, Pdt4,
29              dt, ...);
30   update_relaxationzones(E1, P1, t + 1.0*dt, ...);
31
32   t += dt;
33 }
```

Appendix E

Optimized code

In this chapter we present bits and pieces used for optimizing some of the procedures of appendix D. Be carefull not to confuse the original memory access function `memoryIdxXY` used for the optimized code; optimization once in a while introduce additional threads update the halo.

E.1 Low order residual

For the low order residual xy-slices are processed one at the time as descrived in section 5.3. In order to reduce the global memory access, the dynamic stencils are swapped with fixed estimates and registers `r0`, `r1` and `r2` reuses the global memory along the vertical direction. The number of registers in the kernel could for example be reduced by precalculating `M` on the CPU. The essential parts of the low order residual method is presented below

Listing E.1: Improved kernel

```
1 if(X < N.x && Y < N.y)
2   return;
3 sigmaz = 1/h[memoryIdxXY(0,0,0, M)];
4 r1   = U[memoryIdxXY(0,0,0, M)];
```

```
5  r2  = U[memoryIdxXY(0,0,1, M)];
6  for(int Z = 1; Z<N.z;Z++){
7    r0 = r1; r1 = r2;
8    r2 = U[memoryIdxXY(0,0,Z+1, M)];
9    Uxx = U[memoryIdxXY(-1, 0,Z, M)] - 2*r1
10       + U[memoryIdxXY( 1, 0,Z, M)];
11   Uyy = U[memoryIdxXY( 0,-1,Z, M)] - 2*r1
12       + U[memoryIdxXY( 0, 1,Z, M)];
13   Uss = r0 - 2*r1 + r2;
14   Ax = Uxx/(hx*hx) + Uyy/(hy*hy) + Uss*sigmaz/(hs*hs);
15
16   int n = memoryIdxXY(0,0,Z, M);
17   r[n] = b[n]-Ax;
18 }
```

For the shared memory optimization, the central grid element is copied to shared memory (`smem`) to completly avoid redundant data transfers. In order to also copy into shared memory also from the halos, the entire border of the block will only copy into shared memory and do no residual computations.

Remember a barrier both before and after `smem` access; we have a race condition otherwise. The method primarily differ from the original residual method by the barriers which are also the reason to the additional safety checks; all threads must process the loop rather than just exit.

Listing E.2: Improved kernel using shared memory

```
1  if(Xh < N.x && Yh < N.y){
2    sigmaz = 1/h[memoryIdxXY(0,0,0, M)];
3  }
4  r1  = U[memoryIdxXY(0,0,0, M)];
5  r2  = U[memoryIdxXY(0,0,1, M)];
6  for(int Z = 1; Z<N.z;Z++){
7    if(Xh < M.x && Yh < M.y)
8    {
9      r0 = r1;
10     r1 = r2;
11     r2 = U[memoryIdxXY(0,0,Z+1, M)];
12   }
13   __syncthreads(); //all threads must reach the barrier!
14   smem[smemoryIdxXY(0,0)] = r1;
15   __syncthreads();
16   //only threads which are not used for halo updates
17   if(tx >= halo && tx < blockDim.x-halo &&
18      ty >= halo && ty < blockDim.y-halo){
19     //and only threads
```

```
20      if(Xh < N.x && Yh < N.y){
21        Uxx = smem[smemoryIdxXY(-1, 0)] - 2*r1
22            + smem[smemoryIdxXY(1,0)];
23        Uyy = smem[smemoryIdxXY( 0,-1)] - 2*r1
24            + smem[smemoryIdxXY(0,1)];
25        Uss = r0 - 2*r1 + r2;
26        Ax = Uxx/(hx*hx) + Uyy/(hy*hy) + Uss*sigmaz/(hs*hs);
27
28        int n = memoryIdxXY(0,0,Z, M);
29        r[n] = b[n]-Ax;
30      }
31    }
32  }
```

The utility functions for the shared memory version of the low order residual differ from the general memory access function. `Gx` and `Gy` are the number of ghost points in the horizontal direction in case the number of ghost layers is greater than the halo needed. `Yh` and `Xh` are used rather than `Y` and `X`:

Listing E.3: Memory access for low order residual

```
1  __constant__ int halo; //set to 1
2  #define Xh (-halo + tx + blockIdx.x*(-2*halo + blockDim.x))
3  #define Yh (-halo + ty + blockIdx.y*(-2*halo + blockDim.y))
4  //Xh ranges from -halo to (Nx-1)-halo (+nearest block end)
5
6  __device__
7  inline int memoryIdxXY(int ox, int oy, int z, int3 M){
8    return (Xh + Gx + ox) + (Yh + Gy + oy)*M.x + z*M.x*M.y;
9  }
10
11 int smemoryIdxXY(int ox, int oy){
12   return (tx + ox) + (ty + oy)*blockDim.x;
13 }
```

And finally, the kernel launch configuration should fit the entire domain (size `M`), including ghost points:

```
1  dim3 threads(8, 8, 1);
2  int2 effective;
3  effective.x = threads.x-2*halo;
4  effective.y = threads.y-2*halo;
5  dim3 blocks = fitblock(effective, M.x, M.y);
6  int smemlem = threads.x*threads.y;
```

```
7  kernel::loworderresidual<<< blocks, threads , smemelm*↩
       sizeof(_ftype) >>>(r, x, b, ...);
```

E.2 Improved RBGS Line Smoother

The difference from the original RBGS line smoother is that the x-index of the thread is multiplied by two and shifted by one for every other thread. The initial security check is therefore a bit different and the launch configuration *must* thus be even numbers in both x- and y-direction.

Listing E.4: The altered RBGS line smoother

```
1  __device__
2  inline int RBmemoryIdxXY(int ox, int oy, int z, int3 M, int↩
       isRed){
3    return z*M.x*M.y
4           + (Y + Gy + oy)*memsize.x
5           + X*2 + isRed + Gx + ox;
6  }
7
8  __global__
9  void rbgslinesmoother(_ftype *U, _ftype *b, ...){
10
11   if(!(2*Tx+isRed < N.x && Ty < N.y)) //Altered!
12     return;
13
14   //...
15
16   //central stencil element
17   _ftype a2x = -2*(1 + (ch1x*ch1x + ch1y*ch1y)/((sigmaz*↩
         sigmaz)*ch1z*ch1z));
18
19   for(int Z = 1; Z<N.z;Z++){
20     ////////////////////////////////////
21     //calculate off-center contributions//
22     ////////////////////////////////////
23     a = grid[RBmemoryIdxXY(-1,0,Z, M,isRed)];
24     b = grid[RBmemoryIdxXY( 1,0,Z, M,isRed)];
25     offcenter_contrib += (a+b)/(hx*hx);
26
27     a = grid[RBmemoryIdxXY(0,-1,Z, M,isRed)];
28     b = grid[RBmemoryIdxXY(0, 1,Z, M,isRed)];
29     offcenter_contrib += (a+b)/(hy*hy);
```

```
30
31      //Forward elimination//
32      //...
33    }
34
35    //Backward elimination//
36    //...
37  }
```

Listing E.5: Calculating launch configuration

```
1  ASSERT(threadsx % 2 == 0 && threadsy % 2 == 0);
2  dim3 threads(threadsx, threadsy, 1);
3  nx = N.x/2;
4  if(N.x % 2 == 1)
5    nx += 1;
6  dim3 blocks = fitblock(threads, nx,ny);
```

Appendix F

Automated kernel tuning

A poor kernel configuration can be much slower than a correctly configured kernel. In order to easily interface with any kernel regardless of paramerters an autotuning method is developed with the purpose of benchmarking a kernel. The basic idea is to decouple the benchmarking procedure from the executed kernel through a user defined wrapper which take a (kernel) function pointer and a paramter structure. The basic layers of abstraction needed perform a single timing is given in listing F.1-F.3.

Listing F.1: General template timing method

```
1  template<typename Kernel, typename Params>
2  _timing getTiming(int2 threads,
3    void(*wrapper)(Kernel, Params, int2),
4    Kernel method, Params params){
5
6    cudaRecordEvent(start);
7
8    wrapper(method, params, config);
9
10   cudaRecordEvent(stop);
11   cudaEventSynchronize(stop);
12   cudaError_t e = cudaGetLastError();
13
```

```
14    if(e != cudaSuccess){
15      return _timing::badconfig;
16    }else{
17      _timing t(threads);
18      cudaEventElapsedTime(&t.time, start, stop);
19      return t;
20    }
21  }
```

The method `wrapper` is a function pointer given to `getTiming` as a parameter.

Listing F.2: Kernel wrapper

```
1  struct _pstruct{
2    int nx; int a;
3    int ny; bool b;
4  };
5
6  typedef __device__ void(_funcdef*)(int, bool);
7
8  void myWrapper(_funcdef method, _pstruct p, int2 config){
9    dim3 threads(config.x, config.y);
10   dim3 blocks = fitblock(threads, p.nx, p.ny);
11   method<<<blocks, threads>>>(p.a, p.b);
12 }
13
14 __device__ void some_kernel(int parameterA, bool parameterB↩
       ){
15   //...
16 }
17
18 __device__ void some_other_kernel(int parameterA, bool ↩
       parameterB){
19   //...
20 }
```

Notice that the header of `funcdef` and `somekernel` are the same.

Listing F.3: Benchmarking two similar kernels

```
1
2  int2 threads;
3  threads.x = nx;    threads.y = ny;
4
5  _pstruct p;
6  p.nx = N.x;  p.a = my_important_parameter;
```

```
7 p.ny = N.y;  p.b = my_other_important_parameter;
8 _timing bestConfigA = getTiming(threads, myWrapper, ↩
      some_kernel, p);
9 _timing bestConfigB = getTiming(threads, myWrapper, ↩
      some_other_kernel, p);
```

The template parameter list of `getTiming` are automatically derived from its the paramters. The amount of coding to get the performance timings of a method with another header, now reduce to write a new parameter structure and kernel wrapper.

Bibliography

[1] O. L. Allan P. Engsig-Karup, Harry B. Bingham, "An efficient flexible-order model for 3d nonlinear water waves," *Jurnal of Computational Physics*, vol. 228, 2009.

[2] W. L. Briggs, *A Multigrid Tutorial*. SIAM, 1987.

[3] N. Corporation, "The cuda compiler driver nvcc," NVIDIA Corporation, Tech. Rep., August 2010, available from doc folder of CUDA Toolkit installation.

[4] ——, "Nvidia cuda programming guide," NVIDIA Corporation, Tech. Rep., August 2010, available from doc folder of CUDA Toolkit installation.

[5] A. P. Engsig-Karup, "Efficient low-storage solution of unsteady fully nonlinear water waves using a defect correction method."

[6] ——, *Unstructured Nodal DG-FEM Solution of High-order Boussinesq-type Equations*. Technical University of Denmark, Section of Coastal, Structural and Maritime Engineering, August 2006, phD Thesis.

[7] J. D. O. et al., "A survey of general-purpose computation on graphics hardware," *COMPUTER GRAPHICS forum*, vol. 26, pp. 80–133, 2007.

[8] B. Fornberg, "Calculations of weights in finite different formulas," *SIAM*, vol. 40, no. 3, pp. 685–691, September 1998.

[9] D. G. M. John G. Proaksis, *Digital Signal Processing - Principles, Algorithms, and Applications - 4. th edition*. Pearson Prentice Hall, 2007.

[10] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, 2007.

[11] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *ACM International Conference Proceeding Series*, vol. 383. ACM, March 2009.

[12] A. S. Ulrich Trottenberg, Cornelis Oosterlee, *Multigrid*. Elsevier Ltd, 2001.