Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

PROJECTE FINAL DE CARRERA

Parallelization of Finite Difference Methods:
Nodal and Mimetic solutions of the wave equation.

Estudis: Enginyeria de Telecomunicació

Autor: Ferran Moya Arrayás

Codirectors: Beatriz Otero, Otilio Rojas

Any: 2016

Collaborations

Departament of Computer Architecture – Barcelona School of Telecommunications Engineering (ETSETB)

Thanks

I would first like to thank my project director Dr. Beatriz Otero of the Departament of Computer Architecture at Barcelona School of Telecommunications Engineering. She gave me total independence and constantly encouraged me to push my limits and keep on improving my work.

I would also like to thank the Dr. Otilio Rojas for his help and guidance in understanding compact finite difference methods and their subtleties as well as to MSc. Luis J. Córdova whose MATLAB algorithms were seminal to this work.

Last but not least, I would like to thank my family –my parents and my brother– and my friends for encouraging and supporting me throughout the development and writing of this work and my life in general.

Ferran Moya Arrayás

Contents

Abstract

In the present work we analyse two finite difference methods for the propagation of acoustic waves proposed by Córdova *et al.* [1] to optimize its performance. The computational domain is rectangular. The first of these methods uses a nodal grid and traditional compact finite differences that imply solving tridiagonal systems for numerical differentiation. The second method applies mimetic differential operators in explicit form in a center-distributed grid, avoiding the need to solver linear systems of equations.

Starting from a basic MATLAB implementation of both methods, we study their algorithmic complexity and attempt a first optimization of the reference code. Then several optimization strategies are explored. We develop a single–threaded C++ version and then build an OpenMP version to exploit multithread parallelism. Massive parallelization strategies are implemented in a GPU accelerated version using CUDA. Finally, we perform a comparative study of the methods and their behaviour towards different optimization strategies.

Each optimization technique used is optimal for a certain range of problem sizes. By combining the options developed in this work, it is possible to obtain an execution time speedup between 10–28× (nodal method) and 18–50× (mimetic method) for the whole range of usable problem sizes.

Resum

En el present treball s'analitzen dos mètodes en diferències finites per a la propagació d'ones acústiques proposats per Córdova *et al.* [1] amb l'objectiu d'optimitzar-ne el rendiment. El domini computacional és rectangular, i el primer d'aquests mètodes fa servir una malla nodal i diferències finites compactes tradicionals que impliquen la resolució de sistemes tridiagonals per la diferenciació numèrica. El segon mètode aplica en forma explícita operadors mimètics de diferenciació en una malla centre-distribuïda, evitant d'aquesta manera la resolució de sistemes lineals.

Partint d'una implementació bàsica per MATLAB dels mètodes proposats, s'analitza la complexitat numèrica i s'intenta optimitzar el codi original. A partir d'aquesta implementació de referència explorem diverses estratègies d'optimització. Primerament es desenvolupa una versió en C++ per a una sola CPU i a partir d'aquesta implementació s'explota el paral·lelisme mitjançant una versió multifil emprant la tecnologia OpenMP. Seguidament s'exploren les possibilitats de paral·lelització massiva mitjançant el càlcul accelerat per GPU amb CUDA. Finalment es presenta un estudi comparatiu dels diferents mètodes i el seu comportament enfront de les diferents optimitzacions emprades.

Cada tècnica d'optimització resulta òptima per a un rang de mides del problema. Combinant les diferents opcions desenvolupades en el present treball, es possible obtenir una acceleració del temps d'execució entre 10–28× (mètode nodal) i 28–50× (mètode mimètic) per a tot el rang de mides usables.

Resumen

En el presente trabajo se analizan dos métodos en diferencias finitas para la propagación de ondas acústicas propuestos por Córdova *et al.* [1] con el objeto de optimizar su rendimiento. El domino computacional es rectangular, y el primero de estos métodos emplea una malla nodal y diferencias finitas compactas tradicionales que implican la resolución de sistemas tridiagonales para la diferenciación numérica. El segundo método aplica en forma explicita operadores miméticos de diferenciación en una malla centro-distribuida, y así evita la solución de sistemas lineales.

Partiendo de una implementación básica para MATLAB de los métodos propuestos, se analiza la complejidad numérica y se intenta optimizar el código original. A partir de esta implementación de referencia se exploran diversas estrategias de optimización. Primeramente se desarrolla una versión en C++ para una sola CPU y a partir de esta implementación se explota el paralelismo mediante una versión multihilo usando la tecnología OpenMP. Seguidamente se exploran las posibilidades de paralelización masiva mediante el cálculo acelerado por GPU con CUDA. Finalmente se presenta un estudio comparativo de los distintos métodos y su comportamiento con las diferentes optimizaciones usadas.

Cada técnica de optimización utilizada resulta óptima para un rango de tamaños de problema. Combinando las diferentes opciones desarrolladas en el presente trabajo, es posible obtener una aceleración del tiempo de ejecución de entre 10–28$\times$ (método nodal) y 28–50$\times$ (método mimético) para todo el rango usable de tamaños de problema.

Chapter 1

Introduction

1.1 Project context

Acoustic waves are longitudinal waves which propagate through the adiabatic compression and decompression of the particles of the medium. For instance, sound is an acoustic wave where the particles of either a gas or a liquid vibrate according to an acoustic source. Seismic waves are another example of acoustic waves and a common application for the algorithms discussed in this work.

The acoustic wave problem is defined as a system of differential equations which has no general closed solution. For real world application, numerical methods are required to simulate acoustic waves. The numerical methods discretize the problem constructing a grid of nodes over the domain. Nodal methods are defined in the seminal work by Lele [2]. The first nodal grid methods [3] [4] were low order approximations. Center distributed methods [5] employed a grid where the scalars are offset by $\frac{h}{2}$ to center the variables into their dependent nodes to increase accuracy. Those methods were improved to 4th order [6] [7] [8] and even applied to solve some elastic problems. The increase of method order gave rise to stability problems –which where most significant on the domain boundary due to the potential discontinuities– so methods where devised to reduce the order [9] [10].

The methods studied in this work employ two types of differential operators. The nodal method employs implicit operators (they require solving a system of equations in the forward-backward direction) whereas the ones used by the mimetic method are explicit (they compute the forward value directly from the current one). It is expected that an implicit method will be more stable than an explicit one, albeit slower due to the increased computational cost.

In [11], Aboulai and Castillo generalize and reparametrize the 4-th order operators proposed by Castillo and Grone [12]. This gives rise to a decomposition in smaller (compact) stencils which is the one studied in this work. Aside from the perfomance increase given by the proposed methods, more improvement can be gained with traditional software optimization techniques. We also expect to be able to increase performance with some degree of parallelization.

From the utilization of current multiprocessor systems to exploiting massive parallelism in GPU hardware, parallelization is a very active topic in today's high performance computing and several technologies exist to streamline the development of parallel software. In this work we will employ *OpenMP*, which is a multiplatform C/C++ API for concurrent programming on multi-CPU architectures and *CUDA*, which is a set of extensions to the C/C++ language to allow heterogeneous parallel programming on *NVIDA* GPUs.

Modern commercial computer architectures have 2-8 CPUs [1]. A commercial GPU usually has hundreds to thousands of processing units with an architecture which is totally unlike a general purpose CPU. The use of *OpenMP* and *CUDA* helps to speedup the development of the software but the radical differences between CPU and GPU oriented parallelization pose a serious challenge to the effective parallelization of these finite difference methods, which are mostly sequential in nature.

The optimization and search for parallelization opportunities in the methods proposed in [1] is the main objective of this work.

1.2 Objectives

There are three main work objectives.

1. Implement the finite difference methods for solving wave problems proposed by [1].
2. Test several optimization techniques to improve its performance.
3. Develop a software to test the different implementations, automating the performance exploration over the wide range of parameters and variations.

To ensure the results correctness, a test system will be developed for automated implementation validation.

To ensure reproducibility, a system will be developed to define the implementations to be tested for each method and the parameter range to explore. Results will be presented in a unified format for their automatic processing.

The work in [1] is recent. This adds an exploratory nature to this work due to the anticipated need of modifying or widening its scope. Each optimization will be implemented in an incremental fashion, aiming for a high level of code reusability and thus providing a high degree of scalability to allow for agile exploration of new solutions.

For this reason, even though obtaining the maximum performance will be a high priority for this work, it also becomes essential to design a modular and reusable software, being this the most prioritary design objective.

1.3 Document structure

This work begins by summarising the fundamentals of acoustic wave propagation and reviewing some concepts of finite difference methods that lay out a more in depth analysis of the nodal and mimetic methods as proposed by [1].

A brief summary of linear algebra libraries is included along an analysis of the reference MATLAB implementation as introductory material to the application description.

[1]In the case of supercomputing systems, the CPU count can be much higher.

The main application is conceptually divided into a general application structure – which is mostly a high level abstraction of the theoretical ideas – and the low level implementation of the optimization/parallelization strategies (C++, OpenMP and CUDA).

In the last part of the work, results for all implementations and variations are presented and their performance is compared. Finally, the most important results are analysed in the conclusions.

Three annexes are included, which contain relevant MATLAB and CUDA source code, along with a guide on matrix naming for an easier source review.

Chapter 2

Theoretical background

2.1 Acoustic wave propagation

In this section, we review the formulation of the acoustic wave equation subject to Dirichlet boundary conditions and adequate initial conditions. Acoustic waves have several properties:

- Propagation is achieved by compression and decompression of medium particles.
- Acoustic waves are longitudinal waves, i.e., the displacement of the medium is parallel to to the travelling direction.
- Acoustic propagation is assumed as an adiabatic process: no energy is transferred in or out of the system.

The propagation of acoustic waves can be described by the pressure-velocity formulation of the wave equation

$$\begin{cases} \dfrac{1}{k}\dfrac{\partial u}{\partial t} = -\nabla \cdot \mathbf{v} + f \\ \rho \dfrac{\partial \mathbf{v}}{\partial t} = -\nabla u \end{cases} \tag{2.1}$$

where ρ is the medium density, k is the adiabatic medium compression modulus, $\mathbf{u}$ is the pressure field and vector $\mathbf{v} = (v, w)$ the particle velocity vector.

This work concerns with the finite difference simulation of acoustic wave propagation under the conditions described below. However, the numerical methods we use can be readily extended to the solution of more general acoustic problems.

2.2 Boundary conditions

In this work, we consider as propagation domain $\Omega = [0,1] \times [0,1]$ whose boundary is $\partial\Omega$, as used in [1]. In addition, we impose Dirichlet boundary conditions on $\partial\Omega$, thus the unknown solution to 2.1 is specified on this boundary

$$u(x, y, t) = u_o(x, y, t) \quad \forall x, y \in \partial\Omega \tag{2.2}$$

Finally, we here also provide initial conditions with the form $u(x, y, t = 0)$ and $\mathbf{v} = 0$.

2.3 A review of finite difference methods

Finite Difference Methods (FDM) are a family of numerical methods for solving differential equations based on replacing each continuous derivative by a finite difference approximation.

Thus, the discrete computational version of the original differential equation is a set of difference equations [13]. In the case of time dependent problems, the finite difference discretization is performed at two main levels:

- The time interval is discretized in time steps which usually need to be small for stability constraints and to avoid using of long stencils that may increase memory costs.
- The spatial domain is discretized by using a grid of points where the difference equations are evaluated and finally solved. The distance between each two points of the grid is adjusted according to the accuracy tolerance required on discrete solutions.

A common class of finite difference methods discretize time derivatives by implicit strategies and then require solving a system of linear equations to update the discrete solution. Similarly, standard compact finite differences employ implicit stencils for spatial differentiation and arising linear systems are typically banded, and therefore efficiently solved [2] [14]. This work focuses on the two high-order compact finite difference methods described in [1], one of them is spatially implicit, while the other one is a recent explicit scheme. Next, we briefly introduce some basic finite difference concepts to provide some context to the development of these compact methods.

2.3.1 Finite differences on nodal grids

Let us use the Taylor expansion of a smooth one-dimensional function $f(x)$ to construct a finite difference approximation for the first derivative. On a grid with spacing h that comprises the point x_o, we can write

$$f(x_o+h) = f(x_o) + \frac{f'(x_o)}{1!}\cdot h + R_1(x) \tag{2.3a}$$

$$f'(x_o) \approx \frac{f(x_o+h) - f(x_o)}{h} \tag{2.3b}$$

Which is the *forward difference* form of a finite difference divided by the grid size plus an $R_1(x)$ error term which is the first order reminder of the Taylor Series Expansion. We can improve the approximation by using more series terms, building a higher order finite difference operator. Depending on which grid points we use to compute the finite difference we will get the three most used forms of finite differences [15]:

- **Forward difference**: the forward (next) point of the function is used to compute the difference.
$$f'(x_o) \approx \frac{f(x_o+h) - f(x_o)}{h} \tag{2.4}$$
- **Backward difference**: the backward (previous) point of the function is used to compute the difference.
$$f'(x_o) \approx \frac{f(x_o) - f(x_o-h)}{h} \tag{2.5}$$
- **Central difference**: the central (in–between) points of the function are used to compute the difference.
$$f'(x_o) \approx \frac{f(x_o+\frac{h}{2}) - f(x_o-\frac{h}{2})}{h} \tag{2.6}$$

From this relations one can define discrete differential operators that compute the derivative approximation from the function evaluations according to the selected criteria.

2.3.2 Stencils

The set of points used for computing a finite difference around a central point is called a *stencil*. Stencils are an easy way to visualize the spatial-time dependencies in a finite difference algorithm. By only considering the time dependencies, an implicit simple stencil will be 'T' shaped, with the current value (j, n) at the base and the future values $(j-1, n+1), (j, n+1), (j+1, n+1)$ at the top. On the other hand, its explicit version will be a reversed stencil, with three points (the current values) at the base and one at the top (the next step value).

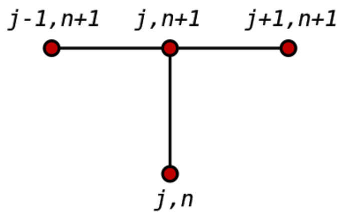

Figure 2.1: Implicit Method Stencil

To achieve higher precision on a finite difference approximation a larger stencil that accounts for several neighbouring points can be used. However, wide implicit stencils are not desirable because thay imply high computation costs due to the complexity of the linear system arising from the stencil application.

2.3.3 Popular methods for parabolic problems

We can obtain the second derivatives with the succesive application of the finite diferences along each dimension. Several methods are possible:

- **FTCS** [14]: The *Forward-Time Central-Space* or *explicit* method uses a *forward difference* for space and a *central difference* for space.
- **BTCS**: The *Backward-Time Central-Space* or *implicit* method uses a *backward difference* for space and a *central difference* for space.
- **CTCS** [16]: The *Crank-Nicolson* method uses a central difference at time $t_{n+\frac{1}{2}}$ and a second-order central difference for the space derivative.

Each method has different computational requeriments, stability properties and error rates:

Method	Computation	Stability	Error
Explicit	Direct	Conditional	$O(h_t) + O({h_x}^2)$
Implicit	Equation system	Unconditional	$O({h_t}^2) + O(h_x)$
Crank-Nicolson	Equation system	Unconditional	$O({h_t}^2) + O({h_x}^2)$

Let us assume the following 1D Dirichlet problem defined on a region Ω:

$$\begin{cases} u_t = \alpha u_{xx}, \quad x \in \Omega, \quad t > 0 \\ u(x,0) = \phi(x), \quad x \in \Omega \cup \partial\Omega \\ u(x,t) = \gamma(x,t), \quad x \in \partial\Omega, \quad t > 0 \end{cases} \tag{2.7}$$

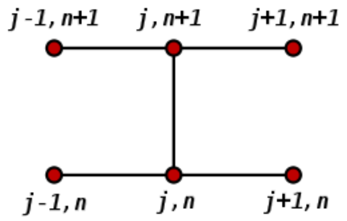

Figure 2.2: 1D Crank-Nicholson Stencil

Let U_j^n denote the finite difference approximation $u_j^n = u(j\Delta x, n\Delta t)$. Then, applying the 1D Crank-Nicholson stencil we obtain the following discretization:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = \alpha \frac{\delta_x^2}{\Delta x^2} \frac{U_j^{n+1} + U_j^n}{2} \tag{2.8}$$

Where the second differential operator in space $\frac{\delta_x^2}{\Delta x^2}$ is computed as a second order central difference from 2.3.1. We are effectively applying the *trapezoidal rule* in time, so convergence is second order. The method is also implicit, to obtain a value of u at the next timestep a system of linear equations must be solved.

2.3.4 Alternating Direction Implicit (ADI) method

The application of the *Crank-Nicolson* method to a time dependent differential equation usually leads to a banded system. In the case of parabolic problems, the Peaceman-Rachford decomposition [17] transforms a banded system into a more easily solvable tridiagonal systems by incorporating a half-step iteration, and solving along one spatial direction at a time. For a brief review, let us the system of ordinary differential equations as given in [18]:

$$\frac{dy}{dt} = f(t, y), \qquad \text{where} \tag{2.9a}$$

$$f(t, y) = f_1(t, y) + f_2(t, y) \tag{2.9b}$$

where the splitting function f_1 corresponds to a one-dimensional differential operator with a associated tridiagonal finite difference operator. Then, the following equations define the ADI method of Peaceman and Rachford:

$$y^* = y_n + \frac{1}{2}\Delta t f_1(t_n + \frac{\Delta t}{2}, y^*) + \frac{1}{2}\Delta t f_2(t_n, y_n) \tag{2.10a}$$

$$y_{n+1} = 2y^* - y_n + \frac{1}{2}\Delta t f_2(t_n + \Delta t, y_{n+1}) - \frac{1}{2}\Delta t f_2(t_n, y_n) \tag{2.10b}$$

This splitting scheme is the base of the implicit compact method on nodal grids in [1], and allows reducing computation time.

2.3.5 Staggered grids

When using a finite difference method, we need to define a grid where the differences are computed. The most natural option seems to be an equispaced grid. This is true for the main function grid, but not for the differences grid.

For each two points of the grid we will get a difference. This value 'belongs' to the point between the two used for the difference but using the same grid for the function and the differences we can only assign it to the left or right point.

A much better solution [2] can be obtained by using an staggered grid, that is, a regular h–spaced grid for the function values and a h–spaced with $\frac{h}{2}$–offset grid for the differences.

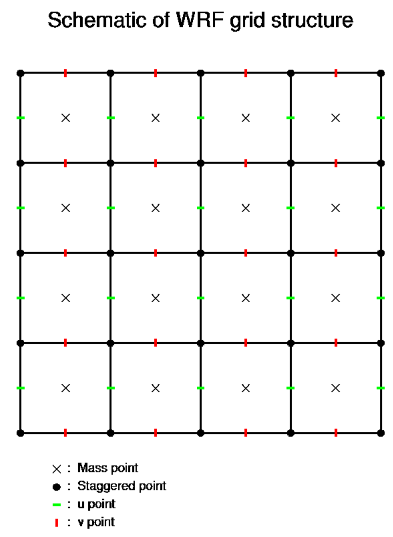

Figure 2.3: Staggered grid example

Chapter 3

The Nodal method

3.1 Description

The nodal method we refer in this work is the one implemented by *Córdova et al* [1] to model acoustic wave propagation. The method applies a high order compact spatial discretization on a nodal grid. Time integration uses a Crank-Nicolson discretization which is efficiently solved using a Peaceman-Rachford ADI 2.3.4 [19] decomposition.

A system of linear equations is constructed around the derivative and the function value to achieve a third order central difference while using only compact stencils. In matrix notation a CFD application can be written as:

$$\mathbf{P}\mathbf{U}' = \mathbf{Q}\mathbf{U} \tag{3.1}$$

where $\mathbf{U}$ is the matrix of function over the $N \times N$ lattice grid, $\mathbf{U}'$ is the nodal approximations to the derivative values on same grid and $\mathbf{P}$, $\mathbf{Q}$ are $N \times N$ stencil matrices implementing the implicit third order CFD explained above.

$$\mathbf{P} = \begin{pmatrix} 2 & 4 & 0 & & \dots & & 0 \\ 1 & 4 & 1 & 0 & \dots & & 0 \\ & & & \vdots & & & \\ 0 & & \dots & 0 & 1 & 4 & 1 \\ 0 & & \dots & 0 & 0 & 4 & 2 \end{pmatrix}, \mathbf{Q} = \frac{1}{h} \begin{pmatrix} -5 & 4 & 1 & 0 & 0 & \dots & 0 \\ -3 & 0 & 3 & 0 & 0 & \dots & 0 \\ & & & \vdots & & & \\ 0 & \dots & 0 & 0 & -3 & 0 & 3 \\ 0 & & \dots & 0 & 1 & -4 & 5 \end{pmatrix} \tag{3.2}$$

The apparent complexity of 3.1 is offset by the fact that we already have the values at $\partial\Omega$ due to the Dirichlet condition imposed so the inner systems defined by $\mathbf{P}, \mathbf{Q}$ are tridiagonal and easily solvable using the Thomas Algorithm. With this scheme we can achieve a high convergence rate so less iterations are needed.

The derivative approximations are computed along each direction ($\mathbf{U_x}$ for x axis and $\mathbf{U_y}$ for y axis).

$$\mathbf{U_x}\mathbf{P}^T \approx \mathbf{U}\mathbf{Q}^T, \mathbf{P}\mathbf{U_y} \approx \mathbf{Q}\mathbf{U} \tag{3.3}$$

Applying the Dirichlet condition to $\partial\Omega$ in 2.1 we can find which values are fixed by the boundary condition:

$$-\frac{1}{k}\frac{\partial u}{\partial t} = \nabla \cdot \mathbf{v} \approx v_x + w_y \tag{3.4}$$

$$\rho\frac{\partial \mathbf{v}}{\partial t} = (\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}) \approx (u_x, u_y) \tag{3.5}$$

From 3.5 we can see that the first and last column of $\mathbf{V}$ and the first and last row of $\mathbf{W}$ are fixed by the boundary condition so $\bar{\mathbf{V}}$ will be an $(N-2) \times N$ matrix and $\bar{\mathbf{W}}$ will be an $N \times (N-2)$. From 3.4 we can see that we only need to compute the internal values of $\mathbf{Vx}, \mathbf{Wy}$ and they effectively are $(N-2) \times (N-2)$ matrices.

We define the $(N-2)\times(N-2)$ $\bar{\mathbf{P}}$ matrix and the $(N-2)\times N$ $\bar{\mathbf{Q}}$ matrix which are reductions of the $\mathbf{P}, \mathbf{Q}$ matrices where redundant equations have been removed. Thus, the rest of the system has the following form:

$$\bar{\mathbf{V}}_{\mathbf{x}}\bar{\mathbf{P}}^T \approx \bar{\mathbf{V}}\bar{\mathbf{Q}}^T, \bar{\mathbf{P}}\bar{\mathbf{W}}_{\mathbf{y}} \approx \bar{\mathbf{Q}}\bar{\mathbf{W}} \tag{3.6}$$

Then *Córdova et al* introduce the operators:

$$A_1 \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} -v_x \\ -u_x \\ 0 \end{bmatrix} \tag{3.7}$$

$$A_2 \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} -w_y \\ 0 \\ -u_y \end{bmatrix} \tag{3.8}$$

$$\tag{3.9}$$

along with their discrete versions

$$A_{1h} \begin{bmatrix} \bar{\mathbf{U}} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix} = - \begin{bmatrix} \mathbf{V}\bar{\mathbf{Q}}^T(\bar{\mathbf{P}}^T)^{-1} \\ \dots \\ \mathbf{U}\mathbf{Q}^T(\mathbf{P}^T)^{-1} \\ \dots \\ 0 \end{bmatrix} \tag{3.10}$$

$$A_{2h} \begin{bmatrix} \bar{\mathbf{U}} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix} = - \begin{bmatrix} \bar{\mathbf{P}}^{-1}\bar{\mathbf{Q}}\mathbf{W} \\ \dots \\ 0 \\ \dots \\ \mathbf{P}^{-1}\mathbf{Q}\mathbf{U} \end{bmatrix} \tag{3.11}$$

Note that only rows $i = 2...N-1$ are computed for $\mathbf{W_y}$ and only columns $j = 2...N-1$ are computed for $\mathbf{V_x}$.

Then we use the following Crank-Nicolson time discretization:

$$\left(I - \frac{\Delta t}{2}A_{1h} - \frac{\Delta t}{2}A_{2h}\right)\mathbf{U}_{\mathbf{CN}}^{\mathbf{m+1}} = \left(I + \frac{\Delta t}{2}A_{1h} + \frac{\Delta t}{2}A_{2h}\right)\mathbf{U}_{\mathbf{CN}}^{\mathbf{m}} \tag{3.12}$$

where $\mathbf{U}_{\mathbf{CN}}^{\mathbf{m}}$ is a new vector holding the nodal discretization $\left[\bar{\mathbf{U}}, \mathbf{V}, \mathbf{W}\right]^T$ at time $t = m\Delta t$.

The ADI Peaceman-Rachford algorithm is used to get a two stage solution where each stage involves solving along one direction:

$$\left(I - \frac{\Delta t}{2}A_{1h}\right)\begin{bmatrix} \tilde{\mathbf{U}} \\ \tilde{\mathbf{V}} \\ \tilde{\mathbf{W}} \end{bmatrix} = \left(I - \frac{\Delta t}{2}A_{2h}\right)\begin{bmatrix} \bar{\mathbf{U}} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix}^m \tag{3.13}$$

$$\left(I - \frac{\Delta t}{2}A_{2h}\right)\begin{bmatrix} \mathbf{U} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix}^{m+1} = \left(I - \frac{\Delta t}{2}A_{1h}\right)\begin{bmatrix} \tilde{\mathbf{U}} \\ \tilde{\mathbf{V}} \\ \tilde{\mathbf{W}} \end{bmatrix} \tag{3.14}$$

The first stage involves two coupled sets of linear systems:

$$\begin{cases} \tilde{\mathbf{U}}\bar{\mathbf{P}}^T + \frac{\Delta t}{2}\tilde{\mathbf{V}}\bar{\mathbf{Q}}^T = \bar{\mathbf{U}}^m\bar{\mathbf{P}}^T - \frac{\Delta t}{2}\left(\bar{\mathbf{P}}^{-1}\bar{\mathbf{Q}}\mathbf{W}^m\bar{\mathbf{P}}^T\right) \\ \tilde{\mathbf{V}}\mathbf{P}^T + \frac{\Delta t}{2}\tilde{\mathbf{U}}\mathbf{Q}^T = \mathbf{V}^m\mathbf{P}^T \end{cases} \tag{3.15}$$

Now, by using the following known terms at $t = m\Delta t$

$$\begin{aligned} \mathbf{A} &= \bar{\mathbf{U}}^m\bar{\mathbf{P}}^T - \frac{\Delta t}{2}\left(\bar{\mathbf{P}}^{-1}\bar{\mathbf{Q}}\mathbf{W}^m\bar{\mathbf{P}}^T\right) \\ \mathbf{B} &= \mathbf{V}^m\mathbf{P}^T \end{aligned}$$

system 3.15 can be rewritten as

$$\begin{bmatrix} \tilde{\mathbf{U}} & \tilde{\mathbf{V}} \end{bmatrix} \begin{bmatrix} \bar{\mathbf{P}}^T & \frac{\Delta t}{2}\mathbf{Q}^T \\ \frac{\Delta t}{2}\bar{\mathbf{Q}}^T & \mathbf{P}^T \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \end{bmatrix} \tag{3.16}$$

and solved through *Block Thomas Algorithm* exploiting $\mathbf{P}$ tridiagonality:

$$\begin{cases} \tilde{\mathbf{U}}_{k+1}\bar{\mathbf{P}}^T = \mathbf{A} - \frac{\Delta t}{2}\tilde{\mathbf{V}}_{\mathbf{k}}\bar{\mathbf{Q}}^T \\ \tilde{\mathbf{V}}_{k+1}\mathbf{P}^T = \mathbf{B} - \frac{\Delta t}{2}\tilde{\mathbf{U}}_{\mathbf{k+1}}\mathbf{Q}^T \end{cases}, \quad \tilde{\mathbf{V}}_{\mathbf{0}} = \mathbf{V}^{\mathbf{m}} \tag{3.17}$$

until $||\tilde{\mathbf{U}}^i_{k+1} - \tilde{\mathbf{U}}^i_k|| < \varepsilon_u$ and $||\tilde{\mathbf{V}}^i_{k+1} - \tilde{\mathbf{V}}^i_k|| < \varepsilon_v$.

Note that this is slightly different to the version proposed in [1]. *Córdoba et al* solve the system row by row whereas in the above description we solve all the systems in one step (i.e.: $\mathbf{A}, \mathbf{B}$ are full matrices whereas in the original paper the row vectors $\mathbf{a_i}, \mathbf{b_i}$ are used).

This has no impact to the solution, the row systems are decoupled so the operations implied are exactly the same, but note that this is crucial to achieve the complete parallelization of the algorithm.

No matrix inversion is needed as the solved systems are always tridiagonal. The same process is performed to the second stage of the Peaceman-Rachford decomposition to obtain $\mathbf{W}^{m+1}$ and $\mathbf{U^{m+1}}$.

$$\begin{cases} \bar{\mathbf{P}}\mathbf{U}^{m+1}_{k+1} &= \mathbf{C} - \frac{\Delta t}{2}\bar{\mathbf{Q}}\mathbf{W}^{\mathbf{m+1}}_{\mathbf{k}} \\ \mathbf{P}^T\mathbf{W}^{m+1}_{k+1} &= \mathbf{D} - \frac{\Delta t}{2}\mathbf{Q}^T\mathbf{U}^{\mathbf{m+1}}_{\mathbf{k+1}} \end{cases}, \quad \mathbf{W}^{\mathbf{m+1}}_{\mathbf{0}} = \mathbf{W}^{\mathbf{m}} \tag{3.18}$$

$$\begin{aligned} \mathbf{C} &= \bar{\mathbf{P}}\tilde{\mathbf{U}} - \frac{\Delta t}{2}\left(\bar{\mathbf{P}}\tilde{\mathbf{V}}\bar{\mathbf{Q}}^T(\bar{\mathbf{P}}^T)^{-1}\right) \\ \mathbf{D} &= \mathbf{P}\tilde{\mathbf{W}} \end{aligned}$$

3.1.1 Physical parameters

We will now modify the presented equations to incorporate the physical parameters k and ρ by redefining the spatial continuous operators A_1 and A_2 in 3.7.

$$A_1 \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} -kv_x \\ -\frac{1}{\rho}u_x \\ 0 \end{bmatrix} \tag{3.19}$$

$$A_2 \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} -kw_y \\ 0 \\ -\frac{1}{\rho}u_y \end{bmatrix} \tag{3.20}$$

$$\tag{3.21}$$

along with their discrete versions

$$A_{1h} \begin{bmatrix} \bar{\mathbf{U}} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix} = - \begin{bmatrix} k\mathbf{V}\bar{\mathbf{Q}}^T(\bar{\mathbf{P}}^T)^{-1} \\ \cdots \\ \frac{1}{\rho}\mathbf{U}\mathbf{Q}^T(\mathbf{P}^T)^{-1} \\ \cdots \\ 0 \end{bmatrix} \tag{3.22}$$

$$A_{2h} \begin{bmatrix} \bar{\mathbf{U}} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix} = - \begin{bmatrix} k\bar{\mathbf{P}}^{-1}\bar{\mathbf{Q}}\mathbf{W} \\ \cdots \\ 0 \\ \cdots \\ \frac{1}{\rho}\mathbf{P}^{-1}\mathbf{Q}\mathbf{U} \end{bmatrix} \tag{3.23}$$

The following Crank-Nicholson time discretization and the Peaceman-Rachford decomposition remain without any change 3.12 and 3.13.

In the light of a more general media, the first stage of the PR-ADI algorithm 3.16 becomes:

$$\begin{cases} \tilde{\mathbf{U}}\bar{\mathbf{P}}^T + \frac{\Delta t}{2}k\tilde{\mathbf{V}}\bar{\mathbf{Q}}^T = \bar{\mathbf{U}}^m\bar{\mathbf{P}}^T - \frac{\Delta t}{2}k\left(\bar{\mathbf{P}}^{-1}\bar{\mathbf{Q}}\mathbf{W}^m\bar{\mathbf{P}}^T\right) \\ \tilde{\mathbf{V}}\mathbf{P}^T + \frac{\Delta t}{2}\frac{1}{\rho}\tilde{\mathbf{U}}\mathbf{Q}^T = \mathbf{V}^m\mathbf{P}^T \end{cases} \tag{3.24}$$

For an easy parametrization, we assume that $\rho = 1$ so the wave speed is just given by k, i.e. $c^2 = k$ given that $c^2 = k/\rho$. A further simplification is introduced to solve problems with harmonic solutions such as in [1, equation 20] with periods λ and T. In this case, $c^2 = ({}^{\lambda}/_{T})^2$.

3.2 Algorithm

The algorithm of the nodal method described in the previous section given in Algorithm 1. For clarity, the two steps of the *Alternate Direction Implicit* method are broken down into two procedures given in Algorithm 2. The pseudocode given here solves the whole set of simultaneous tridiagonal systems in one step, as opposed to the original work, were row–by–row and column–by–column solution is used. The algorithms are the same except for the Algorithm 2 which solves one column/row at a time in the original work.

The algorithm employs the following constants:

- N: Number of nodal points used along each axis for the $[0,1] \times [0,1]$ region.
- cfl_{max}: Stability time step factor. The more oscillatory the solution is, the smaller the time step needs to be. Low values of cfl_{max} produce finer time steps ($cfl_{max} = 0.915$ is used in this work as found stable in [1]).
- ε: Required tolerance for tridiagonal system solution ($\varepsilon = 1e - 5$ is used in this work).
- k_{max}: Maximum number of iterations for convergence of ADI solver ($k = 12$ is used in this work).

The following indexing notations are used:

- $\bar{\mathbf{X}}$: refers to the reduced form of matrix $\mathbf{X}$ according to the method description.
- $:$ refers to every element along the dimension where it is used.
- $\bar{:}$ refers to elements $n = 2, ..., (N - 1)$ along the dimension where it is used.

3.3 Complexity analysis

In this section we will study the complexity of the nodal method as a function of the grid density N. N is not the only complexity factor for this algorithm. Problems whose solution is hard will require not only a finer grid (which is included in the analysis) but also a smaller cfl_{max}. Complexity is proportional to cfl_{max} as it linearly increases the outer loop iteration count, so we will only consider solving a fixed complexity problem with variable sized grids.

There are two additional sequential loops: 1.15 and 1.20. This is not readily apparent in the fully vectorized version as solving is accomplished in a parallel fashion, but for sequential implementations, *row–by–row* and *column–by–column* solving is used, so N iterations are performed. For complexity analysis we will only consider the fully vectorized version as the number of operations is effectively the same[1] To simplify the complexity analysis we will consider every matrix $\in \mathbb{R}^{N \times N}$, $N \gg 1 \implies (N - 2) \approx N$.

[1]The fully vectorized version performs 1 operation on an $N \times N$ matrix while the sequential code performs N operations on N sized vectors. All the operations used in this work are decoupled in the sense that both implementations have equal complexity aside from parallelization or hardware considerations such as *cache* performance.

Algorithm 1 The Nodal method

```
Require: $\mathbf{U_o} \in \mathbb{R}^{N\times N}, \Delta t > 0$
Ensure: $\mathbf{U} \in \mathbb{R}^{N\times N}, \mathbf{V} \in \mathbb{R}^{N\times N}, \mathbf{W} \in \mathbb{R}^{N\times N}$
 1: procedure NODAL($\mathbf{U_o}, \mathbf{cfl_{max}}$)
 2:     $\Delta t \leftarrow \frac{cfl_{max}}{N-1}$
 3:     $\mathbf{U^0} \leftarrow \mathbf{U_o}$                                   ▷ Initial Values
 4:     $\tilde{\mathbf{U}} \leftarrow 0$
 5:     $\mathbf{V^0} \leftarrow 0$
 6:     $\mathbf{W^0} \leftarrow 0$
 7:     $\bar{\mathbf{P}} \leftarrow \mathbf{P}(\bar{:}, \bar{:})$                          ▷ Reduced Matrices
 8:     $\bar{\mathbf{Q}} \leftarrow \mathbf{Q}(\bar{:}, :)$
 9:     $\mathbf{H} \leftarrow \bar{\mathbf{P}} \setminus \bar{\mathbf{Q}}$                          ▷ Precompute value
10:     for $m \in 0 \ldots \frac{t_{end}}{\Delta t}$ do
11:         $\mathbf{F} \leftarrow \mathbf{HW}(:, \bar{:})\bar{\mathbf{P}}^T$                       ▷ ADI first stage
12:         $\tilde{\mathbf{V}} \leftarrow \mathbf{V}^m$
13:         $\tilde{\mathbf{W}} \leftarrow \mathbf{W}^m$
14:         $\tilde{\mathbf{W}}(:, \bar{:}) \leftarrow \mathbf{W}^m(:, \bar{:}) - \frac{\Delta t}{2}\mathbf{P} \setminus \mathbf{QU}^m(\bar{:}, \bar{:})$
15:         $\left(\bar{\tilde{\mathbf{U}}}, \bar{\tilde{\mathbf{V}}}\right) \leftarrow$NODALROWSOLVER($\mathbf{U}^m, \mathbf{V}^m, \mathbf{F}^m$)
16:         $\mathbf{G} \leftarrow \bar{\mathbf{P}}\tilde{\mathbf{V}}(\bar{:}, :)\mathbf{H}^T$                      ▷ ADI second stage
17:         $\mathbf{V}^{m+1} \leftarrow \tilde{\mathbf{V}}$
18:         $\mathbf{W}^{m+1} \leftarrow \tilde{\mathbf{W}}$
19:         $\mathbf{V}^{m+1}(\bar{:}, :) \leftarrow \tilde{\mathbf{V}}(\bar{:}, :) - \frac{\Delta t}{2}\tilde{\mathbf{U}}(\bar{:}, \bar{:})\mathbf{Q}^T/\mathbf{P}^T$
20:         $\left(\bar{\mathbf{U}}^{m+1}, \bar{\mathbf{W}}^{m+1}\right) \leftarrow$NODALCOLUMNSOLVER($\tilde{\mathbf{U}}, \tilde{\mathbf{W}}, \mathbf{G}$)
21:         $t \leftarrow t + \Delta t$
22:     end for
23:     return $\mathbf{U}, \mathbf{V}, \mathbf{W}$
24: end procedure
```

Algorithm 2 Nodal ADI Solver

```
Require: $\mathbf{U} \in \mathbb{R}^{N\times N}, \mathbf{V} \in \mathbb{R}^{N\times N}, \mathbf{F} \in \mathbb{R}^{(N-2)\times(N-2)}, k > 0, \varepsilon > 0$
Ensure: $\tilde{\mathbf{U}} \in \mathbb{R}^{N\times N}, \tilde{\mathbf{V}} \in \mathbb{R}^{N\times N}$
 1: procedure NodalRowSolver($\mathbf{U}, \mathbf{V}, \mathbf{F}$)
 2:     $\mathbf{A} \leftarrow \mathbf{U}(\bar{:}, \bar{:})\bar{\mathbf{P}}^T - \frac{\Delta t}{2}\lambda^2 \mathbf{A}(1 \ldots N-2, :)$
 3:     $\mathbf{B} \leftarrow \mathbf{V}(\bar{:}, :)\mathbf{P}^T$
 4:     $\mathbf{U}_k \leftarrow \mathbf{U}(\bar{:}, \bar{:})$
 5:     $\mathbf{V}_k \leftarrow \mathbf{V}(\bar{:}, :)$
 6:     do                                          ▷ Thomas Algorithm Tridiagonal Solve
 7:         $\mathbf{U}_{k+1} \leftarrow \mathbf{A} - \frac{\Delta}{2}\lambda^2 \left(\mathbf{V}_k \bar{\mathbf{Q}}^T\right) / \bar{\mathbf{P}}^T$
 8:         $\mathbf{V}_{k+1} \leftarrow \mathbf{B} - \frac{\Delta}{2} \left(\mathbf{U}_{k+1} \mathbf{Q}^T\right) / \mathbf{P}^T$
 9:         $test \leftarrow ||\mathbf{U}_{k+1} - \mathbf{U}_k|| + ||\mathbf{V}_{k+1} - \mathbf{V}_k||$
10:         $\mathbf{U}_k \leftarrow \mathbf{U}_{k+1}$
11:         $\mathbf{V}_k \leftarrow \mathbf{V}_{k+1}$
12:         $k \leftarrow k + 1$
13:     while $test > \varepsilon$ & $k < k_{max}$         ▷ Tolerance/non-converging stop criteria
14:     return $\tilde{\mathbf{U}} \leftarrow \mathbf{U}_k, \tilde{\mathbf{V}} \leftarrow \mathbf{V}_k$
15: end procedure
Require: $\mathbf{U} \in \mathbb{R}^{N\times N}, \mathbf{W} \in \mathbb{R}^{N\times N}, \mathbf{G} \in \mathbb{R}^{(N-2)\times(N-2)}, k > 0, \varepsilon > 0$
Ensure: $\mathbf{U^{m+1}} \in \mathbb{R}^{N\times N}, \mathbf{W^{m+1}} \in \mathbb{R}^{N\times N}$
16: procedure NodalColumnSolver($\mathbf{U}, \mathbf{W}, \mathbf{G}$)
17:     $\mathbf{C} \leftarrow \bar{\mathbf{P}}\mathbf{U}(\bar{:}, \bar{:}) - \frac{\Delta t}{2}\lambda^2 \mathbf{G}(:, 1 \ldots N-2)$
18:     $\mathbf{D} \leftarrow \mathbf{P}\mathbf{W}(:, \bar{:})$
19:     $\mathbf{U}_k \leftarrow \mathbf{U}(\bar{:}, \bar{:})$
20:     $\mathbf{W}_k \leftarrow \mathbf{W}(:, \bar{:})$
21:     do                                          ▷ Thomas Algorithm Tridiagonal Solve
22:         $\mathbf{U}_{k+1} \leftarrow \bar{\mathbf{P}} \setminus \left(\mathbf{C} - \frac{\Delta}{2}\lambda^2 \bar{\mathbf{Q}}\mathbf{W}_k\right)$
23:         $\mathbf{W}_{k+1} \leftarrow \mathbf{P} \setminus \left(\mathbf{D} - \frac{\Delta}{2}\mathbf{Q}\mathbf{U}_{k+1}\right)$
24:         $test \leftarrow ||\mathbf{U}_{k+1} - \mathbf{U}_k|| + ||\mathbf{W}_{k+1} - \mathbf{W}_k||$
25:         $\mathbf{U}_k \leftarrow \mathbf{U}_{k+1}$
26:         $\mathbf{W}_k \leftarrow \mathbf{W}_{k+1}$
27:         $k \leftarrow k + 1$
28:     while $test > \varepsilon$ & $k < k_{max}$         ▷ Tolerance/non-converging stop criteria
29:     return $\mathbf{U}^{m+1} \leftarrow \mathbf{U}_k, \mathbf{W}^{m+1} \leftarrow \mathbf{W}_k$
30: end procedure
```

3.3.1 Operations

The operations considered in this analysis will be:

Operation	Type	Cost	Order
Assignment	Dense Matrix	$T_=$	N^2
Addition/Substraction	Full Matrix	T_+	N^2
Scale	Full Matrix	T_λ	N^2
Product	Dense $\times$ Dense	$T_\times$	N^3
	Dense $\times$ Sparse	$T_{\times s}$	BN^2
System Solve[2]	Dense $\times$ Tridiagonal	T_{ss}	N^2
Norm	Dense	T_n	N^2

3.3.2 Analysis

The nodal method is composed of three nested iterations:

- The outer iteration performs the time stepping. Finer grids require smaller time steps (see 1.2). Its iteration count is $I_{\Delta t} = \frac{cfl_{max}}{N-1}$.
- We have two sequential inner loops on each of the ADI solver stages in 2.6 and 2.21. As we will show later, the iteration count of the loops (from now on I_{ADI}) is roughly constant with N. For worst-case analysis I_{ADI} is limited to k_{max}.

Section	Loop Count	$T_=$	T_+	T_λ	$T_\times$	$T_{\times s}$	T_{ss}	T_n
Initialization	1	6					1	
Pre row solver	$I_{\Delta t}$	4	1	1	1	2	1	
Row solver – Pre	$I_{\Delta t}$	4	1	1		2	1	
Row solver – Loop	$I_{\Delta t} \cdot I_{ADI}$	4	4	2		2	2	2
Row solver – Post	$I_{\Delta t}$	2						
Pre column solver	$I_{\Delta t}$	4	1	1	1	2	1	
Column solver – Pre	$I_{\Delta t}$	4	1	1		2		
Column solver – Loop	$I_{\Delta t} \cdot I_{ADI}$	4	4	2		2	2	2
Column solver – Post	$I_{\Delta t}$	2						

$$
\begin{aligned}
T_{init} &= 6T_= + T_{ss} \\
T_{step} &= I_{\Delta t}\left(20T_= + 4T_+ + 4T_\lambda + 2T_\times + 8T_{\times s} + 2T_{ss}\right) \\
T_{ADI} &= I_{\Delta t} I_{ADI}\left(8T_= + 8T_+ + 4T_\lambda + 4T_{\times s} + 4T_{ss} + 4T_n\right) \\
T_{nodal} &= T_{init} + T_{step} + T_{ADI}
\end{aligned}
$$

$$
\begin{aligned}
I_{\Delta t} T_\times \in O(N^4) &\implies T_{step} \in O(N^4) \\
I_{\Delta t} I_{ADI}(T_=, T_+, T_\lambda, T_{\times s}, T_{ss}, T_n) \in O(N^3) &\implies T_{ADI} \in O(N^3) \\
T_{step} \in O(N^4), T_{ADI} \in O(N^3) &\implies T_{nodal} \in O(N^4)
\end{aligned}
$$

[2]For simplicity, we assume right–divide operator cost is equal to left–divide operator cost.

In practice T_{nodal} is nonetheless $O(N^3)$ due to the constant factor being much higher for T_{ADI} than T_{step} for most of the N useful range.

The asymptotic complexity of the method is defined by the cost of the dense-by-dense matrix multiplication at 11. The constant factor is low for this operation, so it will only dominate the total complexity for high N.

Chapter 4

The Mimetic method

4.1 Description

The mimetic method we refer in this work is the one by *Córdoba et al* [1]. Spatial differentiation uses the explicit fourth-order compact mimetic operators designed on staggered grids, which are well suited for the discretization of the pressure-velocity formulation of the wave equation 2.1. As in the nodal method, time integration is carried out by the implicit Crank-Nicolson scheme that is efficiently solved by using the Peaceman-Rachford ADI (2.3.4) decomposition.

Mimetic operators are a special family of finite differences that preserve or "mimic" certain mathematical properties satisfied by the continuous gradient and divergence differential operators. In this work, we are interested in the discrete Divergence D and Gradient G proposed by [12] on 1D staggered grids that were later reformulated on compact forms in [20].The second order accurate mimetic Divergence and Gradient correspond to

$$h\mathbf{D}_2 = \begin{pmatrix} -1 & 1 & & & & \\ & -1 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & -1 & 1 & \\ & & & & -1 & 1 \end{pmatrix} \in \mathbb{R}^{N\times(N+1)}, \tag{4.1}$$

$$h\mathbf{G}_2 = \begin{pmatrix} -\frac{8}{3} & 3 & -\frac{1}{3} & & & \\ 0 & -1 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & -1 & 1 & 0 \\ & & & -\frac{1}{3} & -3 & \frac{8}{3} \end{pmatrix} \in \mathbb{R}^{N\times(N+1)} \tag{4.2}$$

where sub indexes denote the nominal accuracy. Fourth order D and G can be constructed by taking the product of the second order versions with auxiliary operators R_4^D and R_4^G

$$\underbrace{\begin{pmatrix} \frac{1045}{1142} & \frac{492}{2291} & -\frac{418}{2371} & \frac{328}{6821} & -\frac{25}{15576} & 0 & \dots \\ -\frac{1}{24} & \frac{13}{12} & -\frac{1}{24} & 0 & 0 & 0 & \dots \\ 0 & -\frac{1}{24} & \frac{13}{12} & -\frac{1}{24} & 0 & 0 & \dots \end{pmatrix}}_{\mathbf{R}^4_D}, \underbrace{\begin{pmatrix} \frac{503}{399} & -\frac{1234}{2003} & \frac{551}{1217} & -\frac{719}{7198} & \frac{25}{9768} & 0 & \dots \\ -\frac{2}{35} & \frac{941}{840} & -\frac{29}{420} & \frac{1}{168} & 0 & 0 & \dots \\ 0 & -\frac{1}{24} & \frac{13}{12} & -\frac{1}{24} & 0 & 0 & \dots \end{pmatrix}}_{\mathbf{R}^4_G}. \tag{4.3}$$

The mimetic method in [1] uses the fourth–order operators D_4 and G_4 given by

$$\mathbf{G}_4 = \mathbf{R}_G^4 \mathbf{G}_2, \tag{4.4}$$

$$\mathbf{D}_4 = \mathbf{R}_D^4 \mathbf{D}_2 \tag{4.5}$$

Actually, $\mathbf{R}_D^4$ and $\mathbf{D}_2$ are differentiation matrices with a smaller bandwidth than the original operator $\mathbf{D_4}$, so compact differentiation is achieved by successive application of the reduced operators. Similarly, the application of $\mathbf{R}_G^4$ and $\mathbf{G}_2$ replaces the differentiation by means of $\mathbf{G_4}$.

One important difference between the *mimetic* and the *nodal* methods used by *Córdoba et al* is the grid distribution of discrete wavefields 4.1. The discretization of u field correspond to $(N+1)\times(N+1)$ grid evaluations, and values are located at the center of each cell, the central point of boundary edges, and grid corners. The discretization of vector $\mathbf{v}$ is different for each component: $(N+1)$ values of v are located along each grid column where two of these vales belong to border cells, while $(N+1)$ values of w are placed in a similar fashion along each grid row. According to the described grids, the matrix dimensions of discrete wavefields are $\mathbf{U} \in \mathbb{R}^{(N+1)\times(N+1)}$, $\mathbf{V} \in \mathbb{R}^{(N+1)\times N}$ and $\mathbf{W} \in \mathbb{R}^{N\times(N+1)}$. Due to the boundary conditions, we do not need to compute the first and last row of $\mathbf{V}$ and the first and last column of $\mathbf{W}$, those inner matrices will be represented by $\bar{\mathbf{V}} \in \mathbb{R}^{(N-1)\times N}$ and $\bar{\mathbf{W}} \in \mathbb{R}^{N\times(N-1)}$, respectively.

By discretizing 2.1 with the mimetic operators in 4.4 and 4.5 we obtain the mimetic approximation of the problem.

$$\mathbf{U}_x = \mathbf{U}(\mathbf{R}_G^4\mathbf{G}_2)^T, \quad \mathbf{U}_y = (\mathbf{R}_G^4\mathbf{G}_2)\mathbf{U} \tag{4.6}$$

$$\bar{\mathbf{V}}_x = \bar{\mathbf{V}}(\mathbf{R}_D^4\mathbf{D}_2)^T, \quad \bar{\mathbf{W}}_y = (\mathbf{R}_D^4\mathbf{D}_2)\bar{\mathbf{W}} \tag{4.7}$$

Thus, the operators used for the *mimetic method* are explicit whereas the ones for the *nodal method* are implicit and require solving an additional system of equations.

As with the nodal method, the $\mathbf{A}_{1h}$ and $\mathbf{A}_{2h}$ operators from 4.6 and 4.7.

$$A_{1h}\begin{bmatrix}\bar{\mathbf{U}}\\ \mathbf{V}\\ \mathbf{W}\end{bmatrix} = -\begin{bmatrix}\bar{\mathbf{V}}\left(\mathbf{R}_D^4\mathbf{D}_2\right)^T\\ \mathbf{U}\left(\mathbf{R}_G^4\mathbf{G}_2\right)^T\\ 0\end{bmatrix} \tag{4.8}$$

$$A_{2h}\begin{bmatrix}\bar{\mathbf{U}}\\ \mathbf{V}\\ \mathbf{W}\end{bmatrix} = -\begin{bmatrix}(\mathbf{R}_D^4\mathbf{D}_2)\bar{\mathbf{W}}\\ 0\\ (\mathbf{R}_G^4\mathbf{G}_2)\mathbf{U}\end{bmatrix} \tag{4.9}$$

Now we apply the Crank-Nicolson and Peaceman-Rachford decomposition as in 3.12 and 3.13 to obtain the intermediate approximation system:

$$\left\{\begin{array}{l}\tilde{\mathbf{U}} + \frac{\Delta t}{2}\left(\tilde{\mathbf{V}}\mathbf{D}_2^T\right)\mathbf{R}_D^T = \bar{\mathbf{U}}^m - \frac{\Delta t}{2}\mathbf{R}_D\left(\mathbf{D}_2\bar{\mathbf{W}}^m\right)\\ \tilde{\mathbf{V}} + \frac{\Delta t}{2}\left(\tilde{\mathbf{U}}\mathbf{G}_2^T\right)\mathbf{R}_G^T = \mathbf{V}^m\\ \tilde{\mathbf{W}} = \mathbf{W}^m - \frac{\Delta t}{2}\mathbf{R}_G\left(\mathbf{G}_2\mathbf{U}^m\right)\end{array}\right. \tag{4.10}$$

To solve this system, where $\tilde{\mathbf{W}}$ can be explicitly calculated, we define appropriate $\mathbf{A}$ and $\mathbf{B}$ matrices analogous to the ones in 3.16 and then iterate as in 3.17 until $\tilde{\mathbf{U}}$ and $\tilde{\mathbf{V}}$ converge up to a sufficiently small value ε.

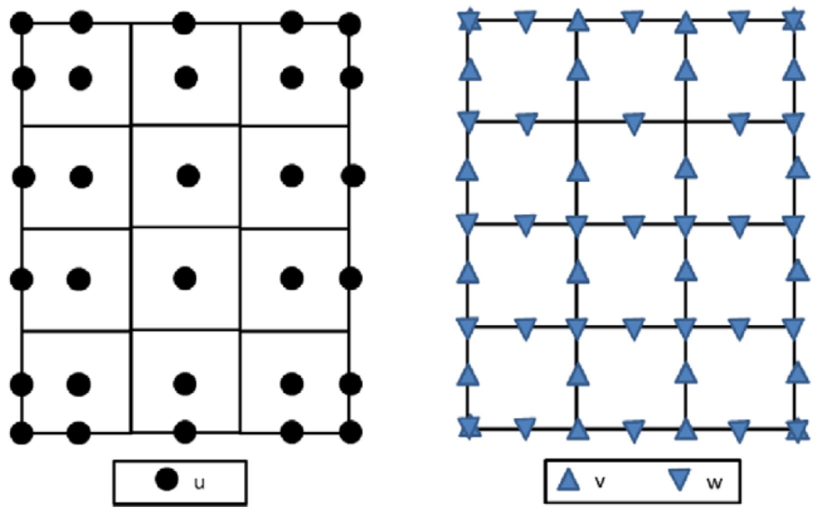

Figure 4.1: Mimetic finite difference discretization on a 2-D staggered grid.

4.2 Algorithm

The mimetic method described in the previous section algorithm is given in Algorithm 3. For clarity, the two steps of the *Alternate Direction Implicit* method are broken down into two procedures given in Algorithm 4. The pseudocode given here solves the whole set of simultaneous systems in one step, as opposed to the original work, were row–by–row and column–by–column solving is used. The algorithms are the same except for the Algorithm 4 which solves one column/row at a time in the original work.

The algorithm employs the following constants:

- N: Number of nodal points along each axis used for the $[0, 1] \times [0, 1]$ region.
- cfl_{max}: Time step factor. The more oscillatory the solution is, the smaller the time step needs to be. Low values of cfl_{max} produce finer time steps ($cfl_{max} = 0.915$ is used in this work).
- ε: Required tolerance for system solution ($\varepsilon = 1e - 5$ is used in this work).
- k_{max}: Maximum number of iterations for convergence of ADI solver ($k = 12$ is used in this work).

The following indexing notations are used:

- $\bar{\mathbf{X}}$: refers to the reduced form of matrix $\mathbf{X}$ according to the method description.
- : refers to every element along the dimension where it is used.
- $\bar{:}$ refers to elements $n = 2...(N - 1)$ along the dimension where it is used.

Algorithm 3 The Mimetic method

```
Require: $\mathbf{U_o} \in \mathbb{R}^{N\times N}, \Delta t > 0$
Ensure: $\mathbf{U} \in \mathbb{R}^{N\times N}, \mathbf{V} \in \mathbb{R}^{N\times N}, \mathbf{W} \in \mathbb{R}^{N\times N}$
 1: procedure MIMETIC($\mathbf{U_o}, \mathbf{cfl_{max}}$)
 2:     $\Delta t \leftarrow \frac{cfl_{max}}{N-1}$
 3:     $\mathbf{U^0} \leftarrow \mathbf{U_o}$                                         ▷ Initial Values
 4:     $\tilde{\mathbf{U}} \leftarrow \mathbf{U_o}$
 5:     $\mathbf{V^0} \leftarrow 0$
 6:     $\mathbf{W^0} \leftarrow 0$
 7:     $\mathbf{D}_4 \leftarrow \mathbf{R}_D^4 \mathbf{G}_2$                                   ▷ Gradient operator
 8:     $\mathbf{G}_4 \leftarrow \mathbf{R}_D^4 \mathbf{D}_2$                                   ▷ Divergence operator
 9:     $\mathbf{H} \leftarrow \mathbf{R}_D^4 \mathbf{D}/h$
10:     $\mathbf{H}_1 \leftarrow \mathbf{R}_G^4 \mathbf{G}/h$
11:     for $m \in 0 \ldots \frac{t_{end}}{\Delta t}$ do
12:         $\mathbf{F}^m \leftarrow \mathbf{H}\mathbf{W}^m(:, \bar{:})$                          ▷ ADI first stage
13:         $\tilde{\mathbf{V}} \leftarrow \mathbf{V}^m$
14:         $\tilde{\mathbf{W}} \leftarrow \mathbf{W}^m$
15:         $\tilde{\mathbf{W}}(:, \bar{:}) \leftarrow \mathbf{W}^m(:, \bar{:}) - \frac{\Delta t}{2}\mathbf{H}_1\mathbf{U}^m(:, \bar{:})$
16:         $\left(\tilde{\bar{\mathbf{U}}}, \tilde{\bar{\mathbf{V}}}\right) \leftarrow$ MIMETICROWSOLVER($\mathbf{U}^m, \mathbf{V}^m, \mathbf{F}^m$)
17:         $\mathbf{G}^m \leftarrow \tilde{\mathbf{V}}(\bar{:}, :)\mathbf{H}^T$                          ▷ ADI second stage
18:         $\mathbf{V}^{m+1} \leftarrow \tilde{\mathbf{V}}$
19:         $\mathbf{W}^{m+1} \leftarrow \tilde{\mathbf{W}}$
20:         $\mathbf{V}^{m+1}(\bar{:}, :) \leftarrow \tilde{\mathbf{V}}(\bar{:}, :) - \frac{\Delta t}{2}\tilde{\mathbf{U}}(\bar{:}, \bar{:})\mathbf{H}_1^T$
21:         $\left(\bar{\mathbf{U}}^{m+1}, \bar{\mathbf{W}}^{m+1}\right) \leftarrow$ MIMETICCOLUMNSOLVER($\tilde{\mathbf{U}}, \tilde{\mathbf{W}}, \mathbf{G}^m$)
22:         $t \leftarrow t + \Delta t$
23:     end for
24:     return $\mathbf{U}, \mathbf{V}, \mathbf{W}$
25: end procedure
```

Algorithm 4 Mimetic ADI Solver

```
Require: $\mathbf{U} \in \mathbb{R}^{N\times N}, \mathbf{V} \in \mathbb{R}^{N\times N}, \mathbf{F} \in \mathbb{R}^{(N-2)\times(N-2)}, \mathbf{H} \in \mathbb{R}^{(N-2)\times(N-1)}, \mathbf{H}_1 \in \mathbb{R}^{(N-1)\times N}, k > 0, \varepsilon > 0$
Ensure: $\tilde{\mathbf{U}} \in \mathbb{R}^{N\times N}, \tilde{\mathbf{V}} \in \mathbb{R}^{N\times N}$
 1: procedure MimeticRowSolver($\mathbf{U}, \mathbf{V}, \mathbf{F}$)
 2:     $\mathbf{A} \leftarrow \mathbf{U}(\bar{:},\bar{:}) - \frac{\Delta t}{2}\lambda^2\mathbf{F}(1\dots N-2,:)$
 3:     $\mathbf{B} \leftarrow \mathbf{V}(\bar{:},:)$
 4:     $\mathbf{U}_k \leftarrow \mathbf{U}(\bar{:},\bar{:})$
 5:     $\mathbf{V}_k \leftarrow \mathbf{V}(\bar{:},:)$
 6:     do                                          ▷ Thomas Algorithm Tridiagonal Solve
 7:         $\mathbf{U}_{k+1} \leftarrow \mathbf{A} - \frac{\Delta}{2}\lambda^2\mathbf{V}_k\mathbf{H}^T$
 8:         $\mathbf{V}_{k+1} \leftarrow \mathbf{B} - \frac{\Delta}{2}\mathbf{U}_{k+1}\mathbf{H}_1^T$
 9:         $test \leftarrow ||\mathbf{U}_{k+1} - \mathbf{U}_k|| + ||\mathbf{V}_{k+1} - \mathbf{V}_k||$
10:         $\mathbf{U}_k \leftarrow \mathbf{U}_{k+1}$
11:         $\mathbf{V}_k \leftarrow \mathbf{V}_{k+1}$
12:         $k \leftarrow k + 1$
13:     while $test > \varepsilon$ & $k < k_{max}$               ▷ Tolerance/non-converging stop criteria
14:     return $\tilde{\mathbf{U}} \leftarrow \mathbf{U}_k, \tilde{\mathbf{V}} \leftarrow \mathbf{V}_k$
15: end procedure
Require: $\mathbf{U} \in \mathbb{R}^{N\times N}, \mathbf{W} \in \mathbb{R}^{N\times N}, \mathbf{G} \in \mathbb{R}^{(N-2)\times(N-2)}, k > 0, \varepsilon > 0$
Ensure: $\mathbf{U^{m+1}} \in \mathbb{R}^{N\times N}, \mathbf{W^{m+1}} \in \mathbb{R}^{N\times N}$
16: procedure MimeticColumnSolver($\mathbf{U}, \mathbf{W}, \mathbf{G}$)
17:     $\mathbf{C} \leftarrow \mathbf{U}(\bar{:},\bar{:}) - \frac{\Delta t}{2}\lambda^2\mathbf{G}(:,1\dots N-2)$
18:     $\mathbf{D} \leftarrow \mathbf{W}(:,\bar{:})$
19:     $\mathbf{U}_k \leftarrow \mathbf{U}(\bar{:},\bar{:})$
20:     $\mathbf{W}_k \leftarrow \mathbf{W}(:,\bar{:})$
21:     do                                          ▷ Thomas Algorithm Tridiagonal Solve
22:         $\mathbf{U}_{k+1} \leftarrow \mathbf{C} - \frac{\Delta}{2}\lambda^2\mathbf{H}\mathbf{W}_k$
23:         $\mathbf{W}_{k+1} \leftarrow \mathbf{D} - \frac{\Delta}{2}\mathbf{H}_1\mathbf{U}_k$
24:         $test \leftarrow ||\mathbf{U}_{k+1} - \mathbf{U}_k|| + ||\mathbf{W}_{k+1} - \mathbf{W}_k||$
25:         $\mathbf{U}_k \leftarrow \mathbf{U}_{k+1}$
26:         $\mathbf{W}_k \leftarrow \mathbf{W}_{k+1}$
27:         $k \leftarrow k + 1$
28:     while $test > \varepsilon$ & $k < k_{max}$               ▷ Tolerance/non-converging stop criteria
29:     return $\mathbf{U}^{m+1} \leftarrow \mathbf{U}_k, \mathbf{W}^{m+1} \leftarrow \mathbf{W}_k$
30: end procedure
```

4.3 Complexity analysis

In this section we will study the complexity of the mimetic method as a function of the grid density N. N is not the only complexity factor for this algorithm. Problems whose solution is hard will require not only a finer grid (which is included in the analysis) but also a smaller cfl_{max}. Complexity is proportional to cfl_{max} as it linearly increases the outer loop iteration count, so we will only consider solving a fixed complexity problem with variable sized grids.

There are two additional sequential loops: 3.16 and 3.21. This is not readily apparent in the fully vectorized version as solving is accomplished in a parallel fashion, but for sequential implementations, *row–by–row* and *column–by–column* solving is used, so N iterations are performed. For complexity analysis we will only consider the fully vectorized version as the number of operations is effectively the same[1] To simplify the complexity analysis we will consider every matrix $\in \mathbb{R}^{N\times N}$.[2]

4.3.1 Operations

The operations considered in this analysis will be:

Operation	Type	Cost	Order
Assignment	Dense Matrix	$T_=$	N^2
Addition/Substraction	Full Matrix	T_+	N^2
Scale	Full Matrix	T_λ	N^2
Product	Dense $\times$ Dense	$T_\times$	N^3
	Dense $\times$ Sparse	$T_{\times s}$	BN^2
	Sparse $\times$ Sparse	$T_{\times ss}$	B^2N
Norm	Dense	T_n	N^2

4.3.2 Analysis

The mimetic method is composed of three nested iterations:

- The outer iteration performs the time stepping. Finer grids require smaller time steps (see 3.2). Its iteration count is $I_{\Delta t} = \frac{cfl_{max}}{N-1}$.
- We have two sequential inner loops on each of the ADI solver stages in 4.6 and 4.21. As we will show later, the iteration count of the loops (from now on I_{ADI}) is roughly constant with N. For worst-case analysis I_{ADI} is limited to k_{max}.

[1]The fully vectorized version performs 1 operation on an $N \times N$ matrix while the sequential code performs N operations on N sized vectors. All the operations used in this work are decoupled in the sense that both implementations have equal complexity aside from parallelization or hardware considerations such as *cache* performance.

[2]$N >> 1 \implies (N-2) \approx N$

Section	Loop Count	$T_=$	T_+	T_λ	$T_\times$	$T_{\times s}$	T_{ss}	T_n
Initialization	1	8		2			2	
Pre row solver	$I_{\Delta t}$	4	1	1		2		
Row solver – Pre	$I_{\Delta t}$	4	1	1				
Row solver – Loop	$I_{\Delta t} \cdot I_{ADI}$	4	4	2		2		2
Row solver – Post	$I_{\Delta t}$	2						
Pre column solver	$I_{\Delta t}$	4	1	1		2		
Column solver – Pre	$I_{\Delta t}$	4	1	1				
Column solver – Loop	$I_{\Delta t} \cdot I_{ADI}$	4	4	2		2		2
Column solver – Post	$I_{\Delta t}$	2						

$$\begin{aligned}
T_{init} &= 8T_= + 2T_{ss} \\
T_{step} &= I_{\Delta t}\left(2T_= + 4T_+ + 6T_\lambda + 4T_{\times s}\right) \\
T_{ADI} &= I_{\Delta t} I_{ADI}\left(8T_= + 8T_+ + 4T_\lambda + 4T_{\times s} + 4T_n\right) \\
T_{mimetic} &= T_{init} + T_{step} + T_{ADI}
\end{aligned}$$

$$\begin{aligned}
I_{\Delta t} T_{\times s} \in O(N^3) &\implies T_{step} \in O(N^3) \\
I_{\Delta t} I_{ADI}(T_=, T_+, T_\lambda, T_{\times s}, T_{ss}, T_n) \in O(N^3) &\implies T_{ADI} \in O(N^3) \\
T_{step} \in O(N^3), T_{ADI} \in O(N^3) &\implies T_{nodal} \in O(N^3)
\end{aligned}$$

Comparing this result to the nodal method, a significant speedup is achieved due to the removal of *dense-by-dense* matrix multiplications.

Even though this asymptotic behaviour is only observed for high values of N, the mimetic method requires less operations to complete (no system solving) which translates in a lower constant factor for the whole algorithm in comparison to the nodal method.

This speedup comes at a cost, as the mimetic method is expected to be less stable, and may require more iterations when solving *harder* problems. This subjective difficulty is given by the solution *roughness* (faster oscillating solutions will be harder to compute as the derivative approximation between grid nodes becomes more inaccurate). Nonetheless, the effective roughness of the problem may be reduced by employing a finer grid, and thus taking advantage of the mimetic method lower complexity.

Chapter 5

Linear algebra libraries

In this chapter we introduce the linear algebra libraries that will be used in this work. These are highly optimized implementations of standard algebra algorithms which have been throughly tested and tuned for optimal performance. On this work we will rely on them whenever possible.

5.1 Basic Linear Algebra Subprograms (BLAS)

The *Basic Linear Algebra Subprograms* [21] are a set of FORTRAN routines that perform basic algebra functions. The library is organized by *levels*:

Rank 1 Mostly vector and scalar–vector operations such as vector sum, dot product, norm, copy, plane rotations, ...

Rank 2 Mostly matrix–vector operations such as multiply–add, inverse multiply, vector outer product, ...

Rank 3 Mostly variations of matrix–matrix produdcts.

Multiple versions of each function exist, distinguished by a prefix and specialized for:

$\mathbb{R}$ numbers *single* precision (prefix s) or *double* precision (prefix d).

$\mathbb{C}$ numbers *single* precision (prefix c) or *double* precision (prefix z).

For this work only the *double* precision $\mathbb{R}$ versions are used. Single precision is not considered for CPU implementations due to native hardware support four 64–bit double precision present in any modern machine.

The main functions of interest for this work are:

dscal $\mathbf{y} \leftarrow \alpha\mathbf{x}$

dadxpy $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$

dnrm2 $||\mathbf{y}||_2$

dgemv $C \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$

dgemm $C \leftarrow \alpha op(\mathbf{A}) op(\mathbf{B}) + \beta\mathbf{C}$

where $op(\mathbf{X})$ performs an optional transposition or conjugate-transposition of the operand $\mathbf{X}$.

Several implementations exist for the BLAS library:

ATLAS The *Automatically Tuned Linear Algebra Software* [22] is an open–source highly optimized single–threaded BLAS library tuned for optimum perfomance on each supported architecture. It will be the library of reference for this work,

openBLAS [23] An open source multi–threaded BLAS library. While tested, its performance was almost equal to the ATLAS library.

MKL The *Intel® Math Kernel Library* is a proprietary library optimized for Intel® processors. Due to their restrictive license and field of application it will not be considered for this work.

ATLAS version *3.10.1-4* will be the library used trough this work as its performance is roughly equal to openBLAS and is fully single-threaded. This is important because we will explore CPU parallelism at the algorithm level, and we do not want any other multithread code in the application.

Being a FORTRAN library has certain implications for the implementation. We will use a FORTRAN to C interface to the library which slightly departs from certain C practices:

- Most arguments, even scalar values, are passed by reference.
- The matrix order is column-major, whereas in C is row-major. This impacts the memory layout of the matrix coefficients. In C, advancing a memory position usually results in obtaining the next element for the row, but in FORTRAN the next column element is the one obtained.[1].

5.2 Linear Algebra PACKage (LAPACK)

LAPACK [24] is a linear algebra library mainly focused on the resolution of simultaneous systems of linear equations, eigenvalue and singular value problems. LAPACK builds on the BLAS library to implement these higher level routines and is written in FORTRAN as well[2]. LAPACK was originally designed to optimize memory access by grouping operations into blocks. This architecture makes it highly amenable to modern multilayer memory hierarchies.
Our main interest in this library is the *dgtsv* function which solves the following equation:

$$\mathbf{AX} = \mathbf{B} \tag{5.1}$$

where A is an $N \times N$ tridiagonal matrix. The system is solved by Gaussian Elimination with partial pivoting. For well conditioned tridiagonal matrices, the routine should effectively implement a Thomas Algorithm which is expected to run in $O(N)$.
We will use *dgtsv* to implement two operations

$$\mathbf{X} = \mathbf{A} \setminus \mathbf{B} \implies \mathbf{AX} = \mathbf{B} \tag{5.2}$$
$$\mathbf{X} = \mathbf{B}/\mathbf{A} \implies \mathbf{XA} = \mathbf{B} \tag{5.3}$$

[1] We will not employ double indexing for matrix memory in our code, explicit flat array indexing will be employed instead to avoid introducing errors due to conflicting language conventions

[2] We will use the C interfaces of both libraries.

For left division, 5.2 direct application of *dgtsv* is adequate. For right division 5.3, one has to transform the equation:

$$\mathbf{XA} = \mathbf{B} \implies \mathbf{A}^T\mathbf{X}^T = \mathbf{B}^T$$
$$\mathbf{X} = \mathbf{A}/\mathbf{B} \implies \mathbf{X} = \left(\mathbf{B}^T \setminus \mathbf{A}^T\right)^T$$

Only *dgtsv* is needed to implement both operations. The transposition of tridiagonal matrix **A** is easily accomplished by simply swapping upper and lower diagonal pointers.

5.3 CUDA BLAS (cuBLAS)

CUDA [25] is an extension of the C language to support massive parallel computation by introducing the CUDA kernels, which are C routines that run on grids of parallel threads. cuBLAS [26] is a proprietary implementation by NVIDIA of the *BLAS* library in CUDA for use with the company GPUs. It is mostly equivalent to other *BLAS* implementations such as *ATLAS*, but it cannot be used as such if one is to exploit the massive parallel computation capabilities of the hardware platform.

Due to the GPU residing on a external bus without direct access to the CPU memory space, transfers become a bottleneck involving DMA requests over a limited bandwidth bus. This is an extremely important consideration for the very high volume of data transfers required by massive parallel computation algorithms. This means that when using *cuBLAS* one has to work with separate address spaces (CPU and GPU). To accomplish this, CUDA provides a special primitive *cudaMemcpy* for transferring data between the two memory spaces.

cuBLAS operations are designed as an abstraction to eliminate the need to write special CUDA kernels to perform parallel computing. As such, its interface is quite similar to any other *BLAS* implementation and we only need to take care of moving the data between the CPU and the GPU when needed. More details will be given in 10.

5.4 Matrix storage

The problem we are trying to solve makes heavy use of sparse matrices. Three types of matrices are defined:

- Banded: Matrices whose non-zero elements are to be found on the main diagonal and its upper and lower diagonals.
- Tridiagonal: Matrices where the non-zero elements are found on the main diagonal and the immediate upper and lower diagonals.
- General Sparse: Matrices where most of the elements are zero.

In this work most of the matrices are banded or tridiagonal. Some general sparse matrices can also be found but they are mostly banded where some bands have few non-zero elements.

For the sake of simplicity only banded matrices will be implemented. Tridiagonal matrices make a special case of banded matrices which will be considered whenever there are optimization opportunities. Banded dominant sparse matrices will be considered as banded. This will impose a performance penalty on some situations as the bandwith of these sparse matrices will be higher than the optimal.

The standard C/C++ matrix storage scheme is *row-major*: each row is stored contiguously. The C/C++ index to memory mapping in array elements is:

$$(i, j) \to j + i \cdot ld \tag{5.4}$$

where ld is the *leading dimension* (the length of the major dimension, row length in this case).

The redundancy found in these special matrices makes the standard C/C++ dense matrix memory layout very inefficient. The libraries used define specific storage schemes for each matrix type:

- Dense matrix: The *column-major* FORTRAN matrix layout is used for general dense matrices. Each column is stored contiguously so the index to memory becomes

  $$(i, j) \to i + j \cdot ld \tag{5.5}$$

  note the switch between i, j in relation with 5.4.

- Banded matrix: A given $N \times M$ banded matrix with KU upper , KL lower diagonals is stored on a $(KU + KL + 1) \times min(N, M)$ *column-major* matrix diagonal by diagonal, starting from the uppermost non-zero one down to the lowest. Some padding is introduced as the length of each diagonal is variable, these elements are not accessed by the library. The layout then becomes:

  $$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix} \to \begin{pmatrix} * & a_{12} & a_{23} & a_{34} \\ a_{11} & a_{22} & a_{33} & a_{44} \\ a_{21} & a_{32} & a_{43} & * \end{pmatrix}$$

  which is mapped by the relation:

  $$(i, j) \to (KU + 1 + i - j, j) \to KU + 1 + i + j \cdot (ld - 1) \tag{5.6}$$

  where $ld = min(N, M)$.

- Tridiagonal matrix: The matrix is represented as three independent vectors, one for each diagonal.

The banded matrix layout introduces some implementation problems when trying to modify an element outside the band, as no storage is reserved for it. We have chosen to raise an exception whenever such situation occurs, making these matrices mostly read-only. This is not problematic as these matrices are defined at initialization and used as constants through the code.

Chapter 6

Reference MATLAB implementation

6.1 Optimization strategies

Several attempts to optimize the base code have been performed so as to have the most optimal baseline implementation. Success has been limited in this regard, as the original code had been thoroughly optimized by its authors.

6.1.1 Matrix inversion

At first glance, the most obvious idea is trying to optimize away the costly matrix inversions that seem to be required by the nodal method. The optimization attempts where:

- Precomputation of the inverse: $O(n^3)$ for inversion and multiplication by inverse.
- Cholesky or LU factorization: $O(n^3)$ for factorization and $O(n^2)$ for solving.

None of these were successful as the systems to be solved are tridiagonal and thus no inversion was being performed in the first place. The cost of solving a tridiagonal system is $O(n)$ which is much better than the two other methods.

6.1.2 Optimized BLAS

Debian update-alternatives command provides an easy way to switch between different implementations of a library. We switch the BLAS implementation with:

```
$ update-alternatives --config libblas.so.3
```

and the LAPACK implementation with:

```
$ update-alternatives --config liblapack.so.3
```

ATLAS and OpenBLAS implementations where tested, the results were mostly equivalent.

6.1.3 Multiprocessing

Multiprocessing seems difficult at first due to the clearly sequential nature of both algorithms. We used the *parallel* package (2.2.1) for Octave [27] to distribute the computation along rows/-columns between several processes.

At first, one process was used for each row/column. This solution performed extremely bad. Little work is assigned to each process resulting in most of the computing power being wasted in managing the processes and their communication.

A second attempt was made where only a small number of processes where used (up to 8 in an 8 core CPU), and lists of columns/rows where assigned to each process. The results were slightly better, but the single-threaded implementation is faster for all but the biggest grid sizes.

The results of using the *parallel* package were poor in general because the need of data replication due to the use of processes instead of threads. No mechanism is available at the time of writing for fast memory sharing/synchronization between processes in the *parallel* package.

6.1.4 Vectorization

The original reference MATLAB implementation in [1] computes the solution in two stages (row-by-row and column-by-column). We call this approach *vector oriented* due to the fact that most operations involve vector arithmetic.

In 1 and 3 we extend the *vector oriented* approach into a *matrix oriented* approach, were we compute the solution across the whole grid for each step with matrix operations.

Our code is mathematically equivalent due to the separable nature of the algorithm. Most of the time we are only proposing a change of notation (i.e. the sum of an array of vectors column by column is equivalent to the sum of matrices composed of the same column vectors).

The only significant difference with the reference code lies in the convergence check step controlled by the ϵ parameter. The reference code uses an error limit relative to a single column whereas the *matrix oriented* approach error limit is relative to the whole matrix.

We add a correction factor to the converge criterion to make sure that the same ϵ value yields comparable results to both approaches. For a vector/matrix with N elements:

$$\epsilon_N > |\mathbf{X} - \mathbf{X_o}| = \sqrt{\sum_{i,j} (x^{ij} - x_o^{ij})^2} = \sqrt{N}\sigma_x \tag{6.1}$$

It follows then that, assuming we want the element error standard deviation σ_x to be the same for two different matrix sizes M and N, we need to compensate for the square root factor in ϵ.

$$\epsilon_M = \sqrt{\frac{M}{N}}\epsilon_N \tag{6.2}$$

In our case $N = 1 \times S$ (*vector oriented*) and $M = S \times S$ (*matrix oriented*), S being the *grid size*, so the final scale factor is:

$$\epsilon_M = \sqrt{\frac{M}{N}}\epsilon_N = \sqrt{\frac{S^2}{S}}\epsilon_N = \sqrt{S}\epsilon_N \tag{6.3}$$

Our novel *matrix oriented* approach presents itself quite useful for optimization purposes. Grouping operations can be advantageous in some situations (i.e. minimizing the number of function calls or reusing redundant operands).

The effect is notable for the MATLAB implementation due to the high cost of function calls and loops in an interpreted language. The *matrix oriented* implementation performs faster for all but the higher grid sizes.

On the one hand, for small grid sizes, small vector operations are so fast that the interpreter becomes the bottleneck. The *matrix oriented* approach removes the innermost loop and increases operation size, improving performance significantly.

On the other hand, when the grid becomes so large that the whole matrix cannot fit into the CPU cache, the *vector oriented* approach becomes faster as the smaller memory footprint of vectors minimizes cache misses.

Chapter 7

General application structure

7.1 Language

The language chosen for the project is C++11[1] for several reasons:

- While being a modern object oriented language, it has lower level features such as pointers or dynamic memory management for a fine-grained control of the generated code.
- Highly expressive language with features such as polymorphism, template metaprogramming and operator overloading together with smart pointers, lambda functions, move semantics,... enable the construction of an easy readable *MATLAB*–like syntax.
- Seamless integration with algebra libraries (C interfaces are available) and CUDA (which is a C++ extension).

Templates have been used to avoid code duplication in single–precision / double–precision implementations. Another advantage of template usage is that classes are specialized to the used types prior to its usage, so the object code generated is usually more efficient.

A major drawback of templates is that instances of templated classes share no link in the class hierarchy. This is makes it hard to implement useful templated polymorphism so we have used the *Curiously Recurrent Template Pattern*[2] (CRTP) [28] to implement static polymorphism in templated classes.

7.2 Architecture

The main design goals for the implementation is to provide a flexible system for testing different parallelization strategies in a way that enables reliable performance comparison.

For this reason, we will decouple the algorithm implementation from the computational routines by abstracting them into several classes.

- The *Matrix* class performs all the low level matrix operations. This is an abstract class which will be specialized depending on the optimization strategy.
- The *Problem* class defines the problem to solve: parameters, grid, initial values and exact solution.

[1]C++14 is also available, but we will not employ any of the new features from it.

[2]The CRTP is a C++ idiom in which a class is derived from a class template instance which uses the derived class as template argument

- The *Nodal/Mimetic* classes implement each of the numeric methods relaying on the *Matrix* class to perform any calculation.

This architecture leads to a natural parallelization at two levels. We have already mentioned that the algorithm can be computed in a *vector oriented* fashion (row–by–row and column–by–column) or in a *matrix oriented* fashion where full matrices are used to solve the whole set of systems in one step.

- The *vector oriented* approach lends itself nicely to parallelization as row (or column) work is completely separable.
- The *matrix oriented* approach is a great opportunity to parallelize at the matrix operation level, where the dimension of the matrices involved allows for a high levels of parallelism.

7.2.1 *Matrix* objects

The Matrix class is a templated interface for the any matrix implementation. It provides the following functions:

- The *constructors* and the *Create()* method are used to set the matrix size and initialize the values.
- Most unary operators are implemented as class methods. Some unary operators are implemented as friend functions to improve readability such as $tr(X)$[3] or $sin(X)$.
- Binary operators such as product, addition,... are implemented as friend functions. The default implementation for this operators is to use the unary form to perform the operation.
- Matrix elements can be accessed and modified through the *X(i,j)*[4] operator. Matrix subsetting is also possibly by range specification $X(i_{min}, i_{max}, j_{min}, j_{max})$.[5]

Several versions of each function have been implemented to leverage move–semantics and minimize memory allocation and copying, specially in constructors and assignment operators.

Two categories of *Matrix* objects have been implemented in this work: dense and banded matrices5.4.

7.2.2 *MatrixTester* objects

The *MatrixTester* template class implements a lightweight unit testing module. The *Matrix-Tester* performs two test types:

- *Matrix* derived class implementation (creation, assignment, operators,. . .).
- Interaction between *Matrix* classes (assignment and binary operators mixing classes).

[3]For the transposition operator, the $tr(X)$ form is used to get a new matrix object whereas the $X.tr()$ form overwrites X contents.

[4]Rows and columns are 0–indexed. Reverse indexing is also possible by specifying negative indices where $-n$ indexes the nth–last element.

[5]MATLAB equivalent notation would be $X(i_{min} + 1 : i_{max} + 1, j_{min} + 1 : j_{max} + 1)$.

7.2.3 *Problem* objects

The problem object is used to store the problem specific information:

- Initial and boundary conditions for solver.
- Problem-specific parameters like solution wavelength λ.
- Exact solution over solver grid and solver solution error.

7.2.4 *Grid* objects

They are helper objects used to define the grid for the solver. There are three types of *Grid* objects:

- *LinearGrid* objects construct a N point equally spaced grid.
- *StaggerGrid* objects construct a grid on cell-center points along with boundary and edge points such as the one used in mimetic methods 4.1.
- *FunctionGrid* objects apply a function over a given grid.

Similar to MATLAB, grid objects can also be used to generate meshes with the *meshgrid* member function. Meshes are returned as a pair of matrices with the x, and y coordinates over the mesh.

7.2.5 *Nodal* and *Mimetic* objects

Nodal and *Mimetic* classes compute the solution to the given *Problem* by different methods. This objects perform the following functions:

- Generate grids with a given density.
- Initialize internal solver data structures according to some parameters.
- Solve the given problem.

7.3 Program flow

The main program flow performs a testbench of the implemented solvers over several parameters and implementations.

- *Problem*: The topmost loop iterates over values of *labmda* (the problem wavelength). A high *lambda* makes the problem more difficult for the solver, specially over coarse grids, as the solution oscillates faster.
- *Implementation*: different implementations are tested: single-threaded, multi-threaded and GPU.
- *Implementation variations*: floating point precision, number of threads, ...
- *Solver*: multiple grid sizes are tested.

The results are stored in a vector of *test_summary* structs:

Listing 7.1: test_summary and timing structures

```
struct test_summary{
        int n;                    // Grid size
        std::string strategy;     // Strategy description
        std::string details;      // Strategy details
        double lambda;            // Problem wavelength
        double epsilon;           // Solver epsilon
        int threads;              // Threads used
        std::string precision;    // Precision used
        // Timings for each method
        std::map<std::string, struct timing> methods;
};

struct timing{
        double user_time;         // User ellapsed time
        double wall_time;         // Wall ellapsed time
        double error[3];          // Final Errors (U,V,W)
        double iterations[2];     // Average iterations (rows,cols)
        int minit,maxit;          // Iteration limit used
};
```

The configuration used for this tests con be altered via the following constants:

- NLAMBDAS: The number of wavelengths to be tested.
- (epsilons,lambdas): A list with the pairs of epsilon/wavelength to be used.
- NCASOS: The number of grid sizes to be tested.
- CASOS: A list with the grid sizes that will be tested for each wavelength.

Several implementations will be used for each configuration:

- Single Threaded: vector and matrix oriented single threaded implementation,
- Multi Threaded: vector oriented multi threaded OMP implementation,
- GPU: Single and Double precision tests with *min_it* ranging from 0 to 3.

Each implementation is selected according to:

- The underlying implementation used in the Matrix class and precision used when instantiating the solver template.
- The *epsilon* and *lambda* passed to the solver constructor.
- The solver strategy (*strategy_single*, *strategy_matrix*, *strategy_matrix*) passed to the solver constructor.
- The minimum iteration count passed to the solver constructor.

Full information of the method, implementation strategy, precision, threading and hardware used are stored for each run.

For each test, both *user time* (the CPU time used by the process) and *wall time* (the real time ellapsed) are recorded. The ratio is used to compute the achieved ocupation of each method.

The study of the number of iterations required to solve the rows/columns tridiagonal system in the inner loop of both methods is also relevant for optimization purposes, as we will use it to limit the number of iterations performed (as stored in minit, maxit).

Full information of each run is stored and printed to *stdout* in CSV format for reproducible postprocessing. R scripts are provided for reading in the CSV results and reproduce the plots and tables found in this document.

Chapter 8

C++ implementation

The C++ implementation is the base implementation for the *OpenMP* and *CUDA* ones. In this chapter we are referring to the single threaded C++ approach and as such we won't be using any paralletization techniques.

8.1 Programming model

For the single threaded implementation the hardware elements to consider are CPU, memory and cache.

8.1.1 CPU

For the CPU we are interested in computing power in GFLOPS. This will be determined by CPU clock frequency, but also by CPU internals such as instruction set, pipeline and ability to execute more than one instruction per cycle. In this work we will be mainly using an *Intel(R) Core(TM) i7-4770 CPU at 3.40GHz*. CPU specifications can be found on table 8.1.1.

To evaluate the CPU computing power we use the *MKL linux LINPACK benchmark 11.3.3.0.11*. This tests evaluates the computing power of the CPU by solving a large system of linear equation. The results are shown in table 8.1.1

This test is optimized for large arrays, as can be seen by the increasing GFLOPS performance with system size. On the one hand the biggest system (10000 equations) requires 763 MB of RAM and is solved in 12.062 seconds at 55.2 GFLOPS, on the other hand the smallest system (8 equation) requires 512 bytes of RAM and only achieves 0.3 GFLOPS.

In our problem, we will be solving relatively small systems (less than 512 equations) so we won't be able to achieve peak performance from the libraries.

8.1.2 Memory

As the grid size becomes larger, more memory is needed to store the matrices and the memory bandwidth will become an important performance factor. The memory clock frequency used in this work is the fastest supported by the CPU (1600 MHz), with a theoretical peak transfer rate of 12800 MB/s. This rate is only achievable in ideal conditions when the access pattern is regular.

CPU	
Release Date	Q2'2013
Processor Number	I7-4770
Cache Size	8 MB SmartCache
Bus Speed	5 GT/s DMI2
Instruction Set	64 bit
Instruction Set Extensions	SSE4.1/4.2, AVX 2.0
Floating point speed	99.72 GFLOPS [1]
Lithography	22 nm
Cores / Subprocesses	4/8
Clock frequency (base/turbo)	3.4 GHz / 3.9 GHz
Power	84 W

RAM	
Memory Type	DDR3
Memory Speed	1600 MHz
Memory Configuration	2×4096 MB + 2×2048 MB
Latency	36 cycle + 57 ns

Cache	
L1 Instruction cache	32 KB, 64 B/line, 8-WAY
L1 Data cache	32 KB, 64 B/line, 8-WAY, 4-5 cycle latency
L2 cache	256 KB, 64 B/line, 8-WAY, 12 cycle latency
L3 cache	8 MB, 64 B/line, 36 cycle latency

Table 8.1: Test system specs

System Size	Leading Dimension	Alignment (kB)	Average GFLOPS	Maximal GFLOPS
8	8	4	0.2737	0.3086
64	64	4	6.0871	7.4562
128	128	4	15.7027	16.2122
256	256	4	22.3036	23.7091
512	512	4	29.4485	29.8342
1024	1024	4	33.6905	34.2111
4096	4096	4	46.5588	46.7024
10000	10000	4	55.2482	55.2852

Table 8.2: Single-threaded test system LINPACK performance

To evaluate this scenario at first approximation, one can assume each floating point operation consuming on average two 64-bit operand, then the required memory bandwidth to exploit the full computing power of a single CPU will be:

$$55.29\ GFLOPS \times 2\frac{operand}{FLO} \times 8\frac{byte}{operand} = 884.64\ GBPS \qquad (8.1)$$

This is well above the ideal maximum RAM bandwidth. To overcome this problem we need to:

- Make sure our code is optimized to exploit locality and reuse operands.
- Make sure that cache is used in an optimal way.

8.1.3 Cache

Cache is an essential part the CPU to achieve high performance computing. It becomes clear from 8.1 that RAM bandwidth is effectively a bottleneck and cache is the most obvious solution to it.

Cache specifications for this CPU can be found in 8.1.1. This is a three level cache, with a segregated data/instruction L1 cache. The fastest cache level is L1, which is also the smallest to reduce CPU cost. As cache levels grow bigger they also get slower: L2 cache is up to three times slower than L1 and L3 cache is also three times slower than L2.

The L1 cache is 32 KB and it is able to store a full 64×64 double precision matrix. The L2 cache is 256 KB and it is able to store a full 180×180 double precision matrix. We expect performance to drop around these values for *matrix oriented* implementations.

8.2 Optimization strategy

Due to hardware dependence, single-threaded CPU optimizations are very low level in nature.

Our goal in this work is to build a reusable framework to explore the performance gains of diverse optimization techniques in acoustic PDE solvers. We want our results to be:

- Comparable in the sense of requiring a similar effort level.
- Reproducible across a range of hardware.

This rules out resorting to low level optimizations for a specific CPU as it both will require a great deal of work (much higher than the required for parallelization) and it will make our work much less portable.

In fact, aiming for this level of optimization can be seen as a form of reinventing the wheel, as much of this low level work has been done by the authors of the different optimized versions of BLAS and LAPACK libraries. More specifically, we will be using the ATLAS BLAS library, which is an Automatically Tuned Linear Algebra Library.

We create two new template classes to encapsulate this functionality:

- *dgMatrix*: for Double precision General Matrix. A regular dense matrix of double precision elements.
- *dbMatrix*: for Double precision Banded Matrix. An arbitrary banded sparse matrix of double precision elements.

The prefix used for the matrix names is akin to the one used for standard BLAS functions: first character for operation precision and second character for matrix storage.

Listing 8.1: Dense matrix multiplication operator (C

```
dgMatrix dgMatrix::operator*(const dgMatrix& b) const{
        assert(cols()==b.rows());

        dgMatrix ret(rows(),b.cols());

        cblas_dgemm(blasorder_,transposed_,b.transposed(),
                                rows(),b.cols(),cols(),
                                1.0,data_.get()+offset(),stride(),
                                b.data_.get()+b.offset(),b.stride(),
                                0,ret.data_.get()+ret.offset(),
                                ret.stride());

        return ret;
}
```

Now that we don't need to specifically focus on the low level aspects, we are free to look deeper at algorithmic level optimizations which will be portable once built on the top of the BLAS library.

The operations offloaded to the library are the ones where low level considerations are most critical: instruction ordering will have a big impact on maximizing pipeline usage, different memory access patterns will generate different cache hit rates, ... There are however some relevant aspects of the algorithm that are not taken in consideration with this approach alone.

Consider the example of matrix-by-matrix product operator given in listing 8.1 performing the operation $\mathbf{C} = \mathbf{AB}$ through an underlying BLAS *cblas_dgemm* call.

It is clear that using the dgMatrix template operator will be much readable and less error prone than the original BLAS C call given the high number of parameters required for each call (matrix dimensions, strides, data pointers and so on).

Memory management for object $\mathbf{C}$ is an important part of the operation. It is implemented in the *dgMatrix* constructor. Memory deallocation is accordingly performed at destruction time.

However, this simple strategy is sub-optimal when $\mathbf{C}$ is a temporary result such as in the expression $\mathbf{E} = \mathbf{D} + \mathbf{AB}$ where we can reuse the memory reserved for the temporary operation $\mathbf{C} = \mathbf{AB}$ in the final step $\mathbf{E} = \mathbf{D} + \mathbf{C}$, not only saving a memory allocation call, but also avoiding the copy of the partial result elements.

To overcome this performance pitfall without losing readability we have employed two techniques[2]:

- Move semantics: we implement *r-value* aware versions of operations where the reuse of allocated memory can be exploited.
- Shared pointers: we manage memory through the *C++* standard library *std::shared_ptr* template, which is a multithread aware reference counted *smart pointer*.

An example of this is given in the listing 8.2 where the reuse of the temporary *r-value* is implemented in the addition operator. We call the move constructor listed in 8.3 for *ret* trough *std::move* operator. The move constructor will check whether the temporary is a submatrix reference (in which case it cannot be reused as it refers to a portion of a bigger matrix) and if possible, move the ownership of the underlying smart pointer to the instance being constructed. This is done with a simple pointer assignment, no memory allocation or transfer is performed.

Listing 8.2: *r-value* version of the matrix addition operator

```
T operator+(const T& m,T&& b){
        if(b.is_submatrix())
                return m+b;

        T ret(std::move(b));
        ret+=(m);
        return ret;
}
```

Listing 8.3: *dgMatrix* move constructor

```
dgMatrix::dgMatrix(dgMatrix&& m){
        // Copy m attrs
        rows_=m.rows_;
        cols_=m.cols_;
        stride_=m.stride_;
        offset_=m.offset_;
        transposed_=m.transposed_;
        submatrix_=m.submatrix_;

        if(!m.is_submatrix()){
                // Reuse m memory when not from a submatrix
                data_=m.data_;
                m.data_=nullptr;
        }
```

[2]This features are new additions to the C++11 standard.

```
          else{
                    // Allocate memory for new matrix and copy
                    data_=shared_ptr<double>(new double[cols_*rows_],std
                        ::default_delete<double[]>());
                    memcpy(data_.get(),m.data_.get(),cols_*rows_*sizeof(
                        double));
          }
}
```

Chapter 9

OpenMP implementation

OpenMP is an API supporting multiprocessing programming through the use of shared memory. It supports C/C++ and FORTRAN programming languages and is available for a wide range of architectures and operating systems (Linux, OS X, Windows, AIX, HP-UX, Solaris, ...). We will use the OpenMP API to optimize the single threaded C/C++ algorithm described in the previous chapter through CPU multithreading.

9.1 Programming model

OpenMP becomes a portable platform for the development of parallel software, which is easy integrable in C/C++ software as it is implemented a series of compiler extensions mostly trough the use of *pragma directives* which are implemented through calls to the OpenMP runtime library.

9.1.1 Core elements

There are five core elements to the OpenMP computing approach:

- Parallel control structures: when using the *parallel* directive, the compiler will fork the process.
- Work sharing: to distribute the work among threads, OpenMP provides directives such as *parallel for* which it will assign each iteration of the loop to a thread, or the *omp section* directive which marks sections of the code which will be run by only one thread.
- Data environment: in OpenMP data is shared by default between the threads. One can use the *private* clause to define *thread local* variables.
- Data synchronization: as data is *shared* by default, there are several synchronization clauses such as *critical, atomic, ordered, ...* to control the access to shared data.
- Runtime control: some auxiliary library functions are provided to control the behaviour of the runtime execution. The most important ones are the *omp_set_num_threads* and *omp_get_num_threads* to set and get the number of running threads. If no specific number is given, it can also be controlled at execution time through the *OMP_NUM_THREADS* environment variable.

9.1.2 Memory model

OpenMP distributes work between threads with a common address space in a *shared memory model*. Thus, any non private variables are shared between threads with no implicit synchronization mechanisms.

To ensure *coherence* and allow the *communication* of shared data between threads OpenMP provides the *flush* directive, which serves as a sequence point at which the thread is guaranteed to see a consistent view of memory with respect to the *flush set* (the explicitly defined group of variables to be flushed at the sequence point).

OpenMP has a *weak consistency* memory model, that is, consistency of reads and writes is only guaranteed relative to the synchronization points (*flushes*). A flush is always consistent with another flush.

In practice, this means that even though memory is shared between threads, to update a shared value one must *flush* the relevant value to ensure the change becomes visible to the other threads. Locks can also be used to synchronize data and operations across threads.

9.2 Optimization strategy

For CPU parallelization we will be using the *vector oriented* approach to the problem.

When assigning each row/column vector to a thread, this approach lends itself nicely to the parallelization as the work to be performed by each thread has no dependence on the other ones, so no synchronization is needed.

The advantages in this case are substantial:

- The values precomputed in initialization and the common part of each iteration are performed by a single thread and shared automatically.
- Most of the workload is found in the row/column computation, so we are optimizing where it is needed most.
- No overhead is introduced by locking or synchronization mechanisms.
- Very little modification of the current code is needed for this kind implementation.
- Introducing parallelism at the algorithm level makes unnecessary the parallelization of the lower level algebra libraries, which could be an extremely complex and costly task. We are able to reuse the already optimized single thread libraries instead.

There is one significant drawback though, and it is that the most costly operation of the algorithm in terms of complexity is performed in the single thread region, and thus not parallelized.

This is of little importance for most of the grid sizes used for our algorithm, as the operation is fast enough to be hidden by the cost of the parallel section, but it can hinder full parallelization of very large grids due to its larger complexity factor.

As has already been said, the implementation of the parallelization is extremely simple. This is shown in listing 9.1.

Listing 9.1: Main Nodal Solver Routine

```
// Nodal problem solver
int solve(problema<M>& p,enum solver_strategy strategy){
        // Initialize shared matrices
        init(p.params());

        // Define initial conditions
        U_ = p.Uo();
        V_ = p.Vo();
        W_ = p.Wo();

        M A,C;
        M Utilde,Vtilde,Wtilde;

        Utilde = U_;
        for(int iter=0;iter<niter_;iter++)
        {
                A = M_ * W_(0,-1,1,N_-2)*tr<bM>(P_);
                Vtilde =V_;
                Wtilde =W_;

                Wtilde(0,-1,1,N_-2) = W_(0,-1,1,N_-2)-dts2_*
                                      ldivide(S_,T_*U_(1,N_-2,1,N_-2));

                switch(strategy){
                        case strategy_matrix:
                                row_solver_matrix(A,Utilde,Vtilde);
                                break;
                        case strategy_omp:
                                row_solver_parallel(A,Utilde,Vtilde);
                                break;
                        case strategy_single:
                        default:
                                row_solver_single(A,Utilde,Vtilde);
                }


                C  = P_*Vtilde(1,N_-2,0,-1)*tr<M>(M_);
                V_ = Vtilde;
                W_ = Wtilde;
                V_(1,N_-2,0,-1) = Vtilde(1,N_-2,0,-1)-dts2_*
                                  rdivide(Utilde(1,N_-2,1,N_-2)*
                                  tr<bM>(T_),tr<bM>(S_));
                switch(strategy){
                        case strategy_matrix:
                                column_solver_matrix(Utilde,Wtilde,C);
                                break;
                        case strategy_omp:
                                column_solver_parallel(Utilde,Wtilde,C);
                                break;
                        case strategy_single:
                        default:
                                column_solver_single(Utilde,Wtilde,C);
                }
        }
        return 0;
}
```

The optimization strategy to be used is defined by the *enum solver_strategy* which has three values:

- strategy_single: where the single threaded *vector approach* is used.
- strategy_matrix: where the *matrix oriented* approach is used.
- strategy_omp: where the multithreaded *vector approach* is used.

The implementation of the single and parallel row solver is then identical with the only difference being the *omp parallel for* pragma directive, used to indicate to the OpenMP library that the workload of each iteration should be split among available threads.

The synchronization is performed implicitly at the end of the *parallel for* section so only one line is needed to transform our single threaded code to a multithreaded implementation. The other parallel solvers (column nodal, row mimetic, column nodal, ...) are implemented with the same technique.

OpenMP automatically manages the thread creation. By the output of the *ps* command we believe it is using an efficient thread pool implementation where one thread is created for each available core and then reused when finished, limiting unnecessary thread creation and destruction.

Listing 9.2: Single and parallel row solvers

```
int row_solver_single(const M& A, M& Utilde, M& Vtilde){
        for(unsigned int i=1;i<N_-1;i++)
                row_solve(i,A,Utilde,Vtilde);
        return 0;
}

int row_solver_parallel(const M& A, M& Utilde, M& Vtilde){
        #pragma omp parallel for
        for(unsigned int i=1;i<N_-1;i++)
                row_solve(i,A,Utilde,Vtilde);
        return 0;
}
```

It is worth noting that this implementation solves the problem found when we tried to apply the same strategy to the Octave implementation 6.1.3. In that parallelization attempt we lacked the flexibility of a shared memory multithread model, and we found that the cost of process spawning and inter–process communication greatly defeated the optimization attempt.

This is not the case anymore, as memory is shared implicitly. The shared address space removes the need to transfer data between threads, and the join is nicely performed by OpenMP at the end of the *parallel* section.

Chapter 10

CUDA implementation

CUDA is an parallel computing platform and Application Programming Interface designed for heterogeneous parallel programming. CUDA is created and maintained by NVIDIA to allow the use of their GPUs (graphics processing unit) for general computation algorithms. Due to wide adoption of GPUs as parallel computing platforms in several fields from science to financials, NVIDIA developed specialized versions of their GPUs specifically for this usage, the TESLA family.

Nonetheless, consumer GPU cards, traditionally used as hardware accelerators for video games, have become an ubiquitous and extremely affordable solution due to the high volumes of the consumer market. Technologies such as CUDA or their non-propietary counterparts such as *openACC* or *openCL* have profoundly changed the industry, as resources and techniques reserved to those with access to supercomputing infrastructure are now available at a fraction of a price and, most importantly, providing a scalable model that goes from prototyping on regular consumer GPUs to massive parallelization in a cluster of dedicated TESLA GPUs.

10.1 Programming model

CUDA is an heterogeneous computing environment, that is, more than one kind of processor is involved in the computation. CUDA distinguishes between two types of processors:

- The host is the CPU and its memory.
- The device is the GPU and its memory.

This distinction is important, as the memory spaces for the host and device are separate and thus, special attention must be paid to its management and the communication between the two. The PCI Express (PCIe) is used to connect the CPUs and the GPUs and PCIe switches are used in cluster environments to build a tree-like hierarchy of GPU.

Memory transfers can only be performed across a PCIe bus such as host–device or device–device when in the same PCIe bus. Even though this architecture achieves a high throughput, it is limited by the relatively small bandwidth of the PCIe link (6 GB/s)[1].

[1]In real world applications there can be more than one host or device. Multihost programming is achieved trough conventional multi-threading techniques such as the ones in the previous chapter. Multidevice programming is enabled by specific CUDA functions for working with a specific device and transfer information between them. In our application only one host and device is considered as the size of the problem is small.

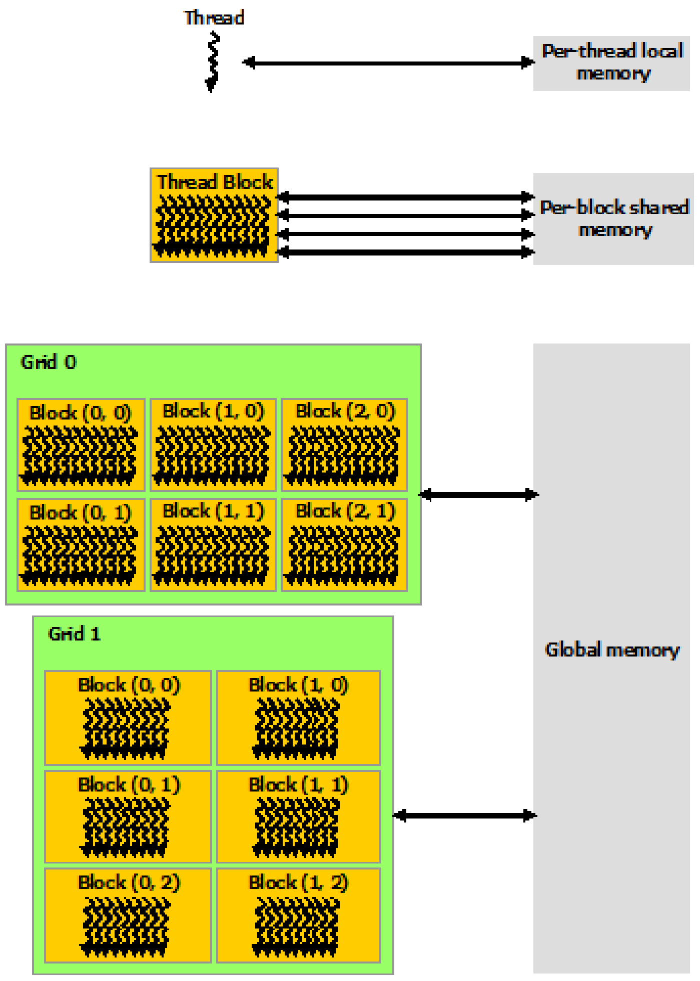

Figure 10.1: CUDA memory hierarchy [29]

To minimize this IO bound limit, CUDA applications usually are designed around the following structure.

1. Copy input data from CPU memory to GPU memory.

2. Load GPU program and execute, caching data on chip for performance.

3. Copy results from CPU memory to GPU memory.

Memory transfers are performed through the *cudaMemcpy* function, and GPU code is organized in *kernels*. A *kernel* is a function that runs on the device, it is written in C/C++ using the *__global__* keyword and called (launched) through a special triple angle brackets syntax:

```
mykernel<<<blocks , threads_block , shared_memory ,  stream_number>>>(params );
```

Due to hardware limitations, not all threads of the kernel are executed in parallel.

- The thread runs the kernel code and has access to a thread local memory.

- The smallest concurrency unit is the warp which is a group of 32 threads. Usually each processor in the device runs a warp at any given time.
- Threads are also grouped into blocks. Threads in a block may or may not run in parallel, but they have access to a per–block shared memory and can synchronize to the other threads in the block.
- The blocks [2] are part of a grid, which contains all the threads that will be launched. Threads from separate blocks only can communicate through global memory.
- Newer GPUs can run different kernels at the same time. CUDA provides the *stream* abstraction to accommodate more than one *execution flow*: kernels launched on a given stream will be scheduled sequentially whereas kernels launched on different streams are launched asynchronously.

The execution and memory hierarchy defined by this model emerges as it is physically impossible to run an arbitrary large number of threads in a fully parallel way. This is the result of hardware constraints that pose a big impact on how memory is organized, it's latency, availability of registers for active threads, etc...

10.2 Hardware architectures

A CUDA device is a hierarchical multiprocessing device:

1. Each CUDA platform is composed of one or more devices connected through PCIe.
2. Each device is composed of *streaming multiprocessors* which are *Single Instruction Multiple Data* processors.
3. Each *streaming multiprocessor* is composed of CUDA cores that execute the same instruction for different data concurrently.

One can see that each CUDA core will run a kernel thread, several interleaved warps will run on each streaming multiprocessor, and blocks will be assigned to one or more streaming multiprocessors as required. The block hierarchy level plays a key role in limiting memory/register requirements as no synchronization is possible between blocks. For this same reason, the number of threads per block is limited to 512/1024 on most devices, whereas block count limit is much higher.

This hierarchy means that a clever distribution of the threads through the device will be critical in obtaining maximum parallelism as several bottlenecks emerge.

- Kernel divergence will be severely penalized, as the SIMD design implies idle cores when the same instruction cannot be issued to all of them.
- Different levels of memory will have different speeds, thread register bank being the fastest and global memory being the slowest.

[2]CUDA seamlessly allows for 1D, 2D or 3D block and grid geometries, which is useful for matrix algebra or 3D applications.

- A regular memory access pattern for each warp is critical to ensure maximum memory bandwidth usage and cache performance.
- A high number of warps is required to hide the memory access latency.
- Inter–block communication is not possible.
- Very complex kernels will limit the ability to parallelize as not enough registers will be available in the streaming multiprocessor for the full thread count.

10.2.1 Fermi

Fermi is a CUDA microarchitecture used in the NVIDIA GeForce 400/500 and TESLA. Chip specifications can be found on table 10.2.1.

As shown in figure 10.2.1 the chip is composed of a series of SMs connected to the DRAM trough a L2 cache.

Each SM (see 10.3) is composed of:

- A instruction cache
- A dual warp scheduler, which can issue instructions to any of the two groups of 16 threads in the warp.
- A separate dispatch unit which can dispatch instructions even when an SFU is occupied.
- A register file
- A series of CUDA cores, composed of an FPU and an ALU.
- A series of load/store units.
- A series of SFU units, for the execution of trascendental functions.
- Unified shared memory and L1 cache.

Streaming Multiprocessors (SM)	32 CUDA cores
Host interface	PCIe v2
DRAM	DDR5 (up to 6GB)
Peak performance	1.5 TFlops
Global memory clock	2 GHz
DRAM bandwidth	192 GB/s
Registers per SM	32K 32 bit registers
L1 cache / shared memory	48 kB shared + 16 kB L1 or 16 kB shared / 48 kB L1
L2 cache	768 KB per chip

Table 10.1: Fermi specifications

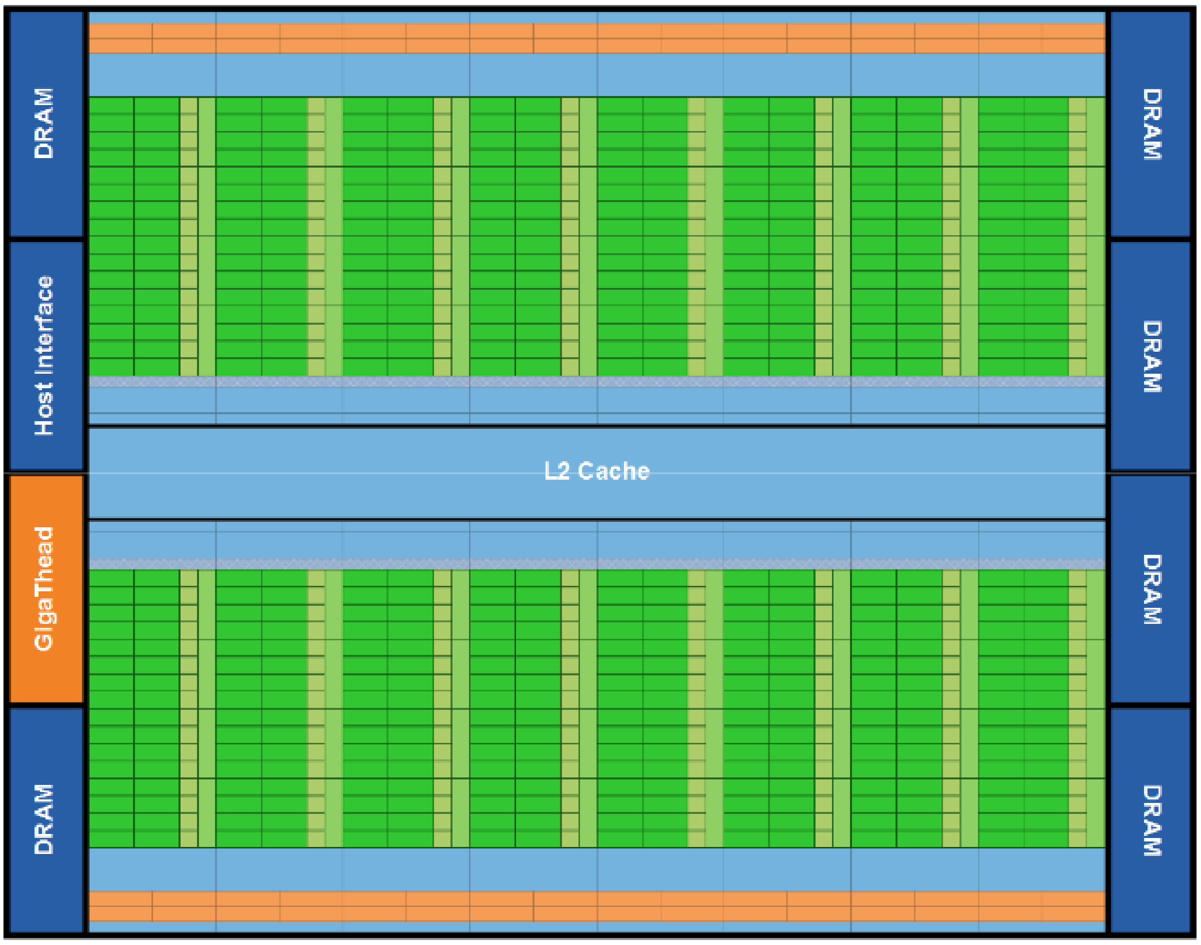

Figure 10.2: Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache) [29]

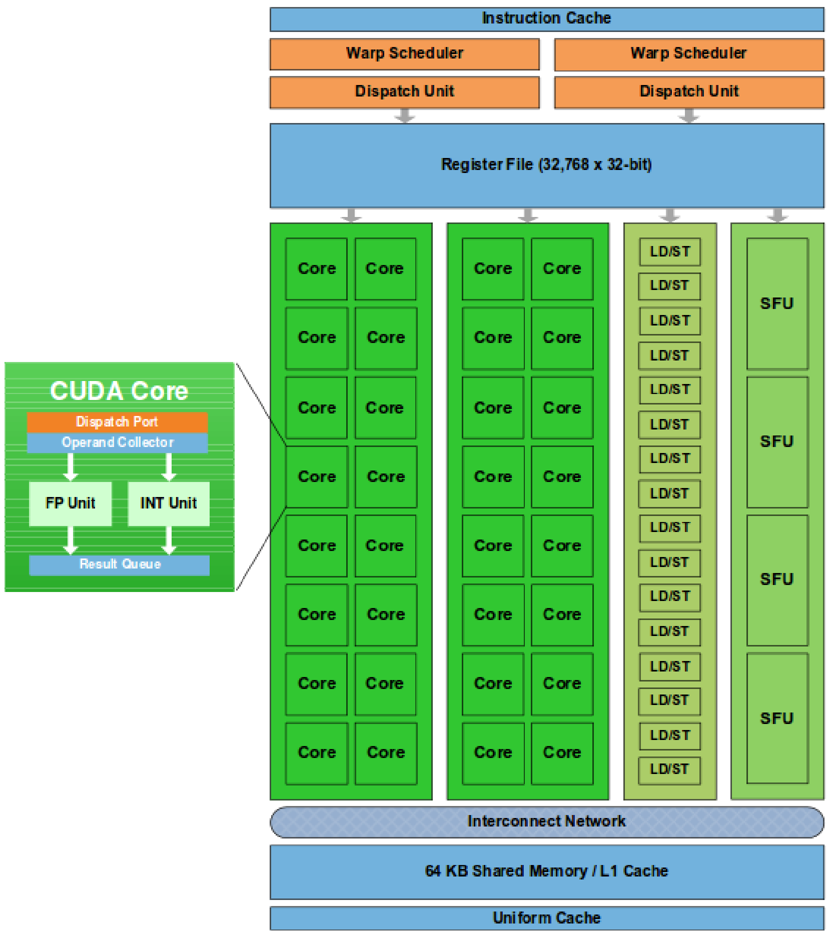

Figure 10.3: Fermi Streaming Multiprocessor (SM) [29]

Streaming Multiprocessors (SM)	32 CUDA cores
Host interface	PCIe v2
DRAM	DDR5 (up to 6GB)
Peak performance	4.6 TFlops
Global memory clock	7 GHz
DRAM bandwidth	224 GB/s
Registers per SM	64K 32 bit registers
L1 cache / shared memory	48 kB shared + 16 kB L1 or 16 kB shared / 48 kB L1
L2 cache	2048 KB per chip

Table 10.2: Maxwell specifications

10.2.2 Maxwell

The Maxwell architecture is two generations ahead the Fermi architecture. Its block structure is quite similar to the Fermi (see 10.4 and 10.5), but with several improvements:

- Reduced latency for the integer arithmetic instructions.
- Dedicated shared memory independent of L1 cache.
- Reduced number of CUDA cores, but higher efficiency due to improved scheduling and occupancy.[3]
- Lower power consumption and higher speed clocks due to the 28 nm manufacturing process.

The Maxwell generation targeted low cost power optimized solutions, so there are changes that may hurt performance:

- No dedicated double precision FPU means high latency for this kind of instructions.
- Memory bus size is reduced from 192 bit to 128 bit.

10.3 Optimization strategy

It becomes evident that performance will be heavily conditioned by the ability of our code to fully exploit the hardware resources. Due to the small grid sizes, achieving a high occupancy will be our topmost priority.

As stated in 8.2 we won't to rewrite the whole algorithm as a specific CUDA kernel. On the one hand, this would achieve the best performance, as every operation could be fine tuned resulting in highly optimized kernels.

[3]NVIDIA claims that a 128 core Maxwell has 90 % of the performance of the 192 previous generation core

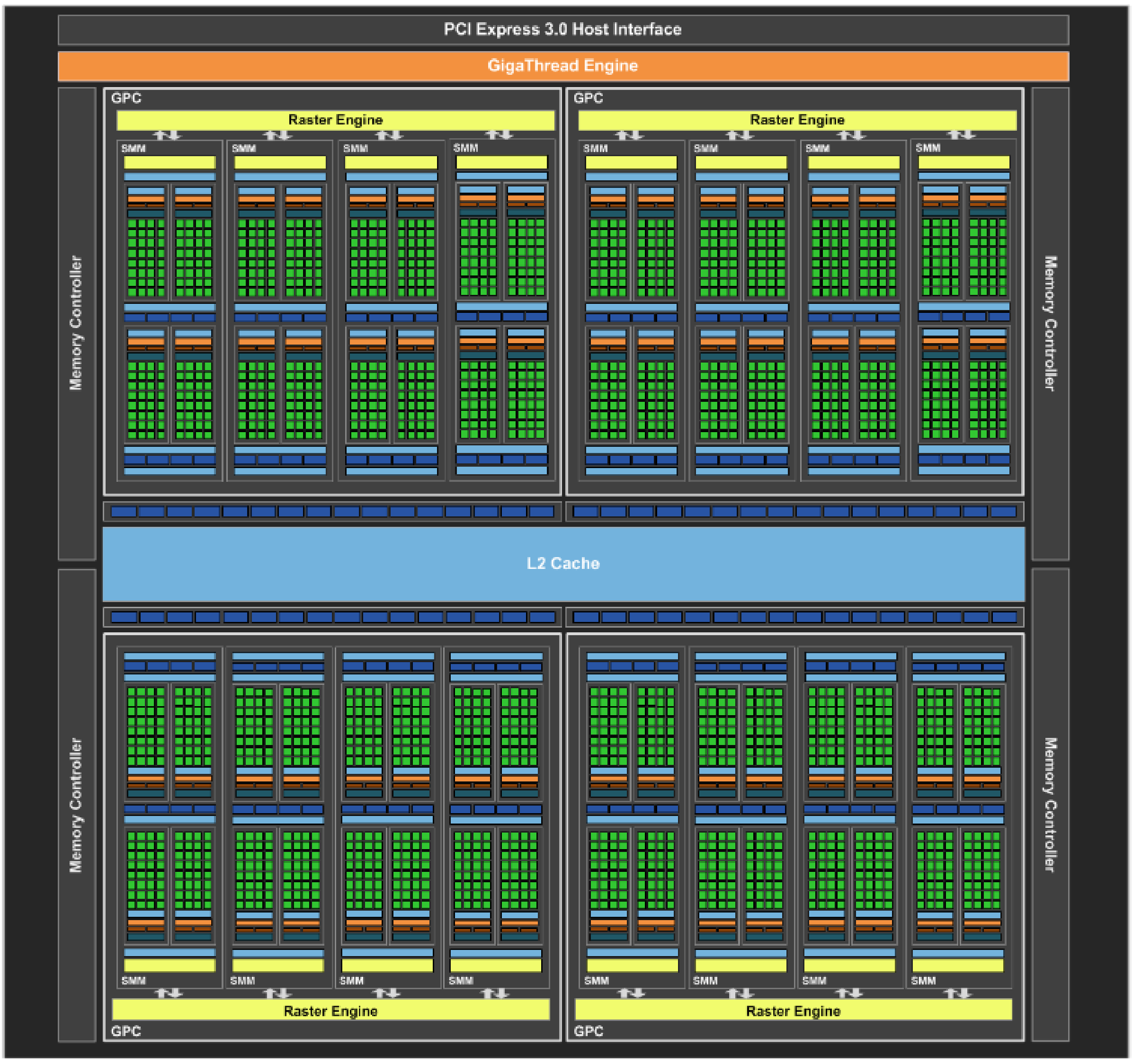

Figure 10.4: GM204 Full chip block diagram [30]

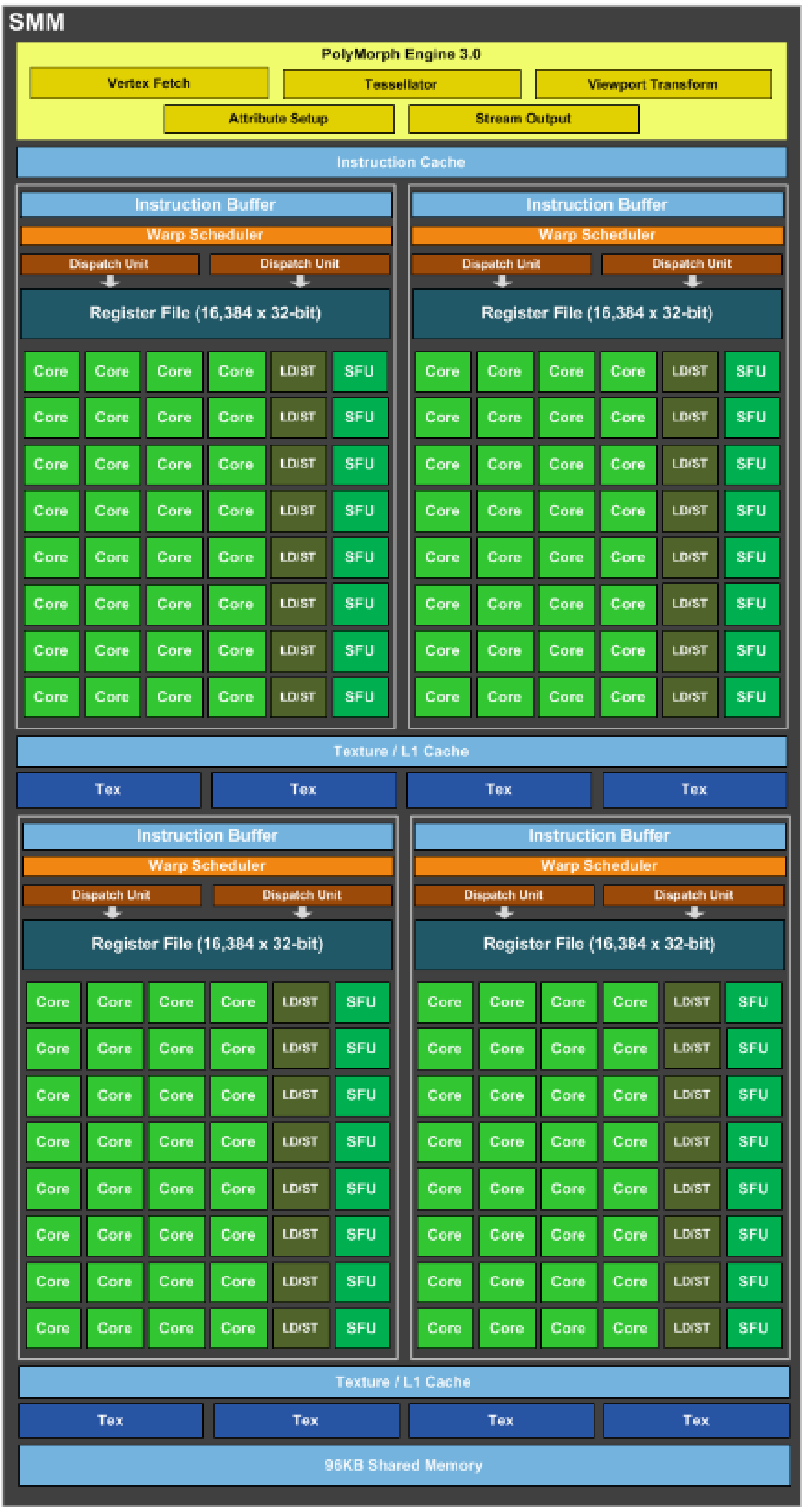

Figure 10.5: Maxwell Streaming Multiprocessor (SM) [30]

On the other hand, these very specific kernels present some drawbacks:

- The higher complexity of the kernels can hurt performance due to high register or shared memory usage, limiting achievable concurrency.
- Due to the higher resource usage, performance will be hardware dependent as different warp schedulers or shared memory/cache availability become critical.
- The code would not be reusable with a high cost in terms of development and maintenance time.

In practice we have found that, for large grid sizes, the occupancy of relatively simple kernels such as the Thomas algorithm for tridiagonal systems becomes register count limited. It can be expected then that, by aiming for the highest performance and developing overly complex kernels, we could be degrading performance for very large problems, which are the ones that really require the use of the GPU computing power.

For this reasons we have decided to opt for a library based implementation, using cuBLAS and developing specific kernels only when cuBLAS is missing a required functionality.

10.4 Implementation

10.4.1 *cudaEnvironment* class

One of the main implementation objectives is writing a drop–in replacement for the family of *Matrix* templates built over the standard linear algebra libraries.

Due to the low level considerations introduced by the CUDA API such as driver connection and grid layout, we must find a way of transparently managing those aspects.

The *cudaEnvironment* class is a helper class usually instantiated once at program startup and connected by default to the other CUDA classes trough a constant static member reference assigned by default at object construction.[4].

The environment is then internally used by all CUDA class templates when needed:

- When calling driver functions such as *cudaMalloc* or *cudaFree*.
- When calling cuBLAS functions which require a driver handler.
- When launching kernels to obtain a grid configuration.
- When switching or syncing streams.

[4]Although the *cudaEnvironment* instance usually functions as a singleton, it does not need to (i.e: when using systems with multiple devices)

Aside from driver initialization and handler management, the main task of the *cudaEnvironment* object is to determine the optimal kernel call parameters regarding grid size. At initialization, we define a *target_tpb* [5] parameter which tells the environment the desired number of threads per block. Through the *cudaEnvironment::assign()* function, the *cudaEnvironment* object returns a suggested grid configuration consisting of blocks where the given number of threads have been distributed according to the target[6].

Streams are managed through the *cudaEnvironment* instance in a LIFO style where one uses *cudaEnvironment::stream_begin()* to create and switch to a new stream and *cudaEnvironment::stream_end()* to dispose the last created stream. Synchronization between running streams is achieved by calling *cudaEnvironment::stream_sync()*.

This interface is very easy to use, and we do not consider the restrictiveness of the LIFO mechanism to be a problem in most real world situations, as one will launch as many streams as desired (which are independent by design), sync to obtain the result and then destroy them.

10.4.2 *cuMatrix* template class

The *cuMatrix* template class plays the same role as the Matrix class for the C++ implementation.

Two template classes are derived from it: *cudgMatrix* for dense CUDA matrices and *cudbMatrix* for sparse banded CUDA matrices. These templates can be specialized for single or double precision and present the same programmer interface as the *dgMatrix* and *dbMatrix* template classes respectively.

One essential difference of this templates lies in the memory management. The storage for these template classes is always allocated in device memory for maximum speed. Explicit conversions from *dgMatrix* and *dbMatrix* are provided for transferring data from one domain to another [7]. As in 8.2, the *cuMatrix* classes employ extensively the move semantics of C++11 to minimize memory transfers and optimize performance.

Access by reference to individual matrix elements is tricky as the reference is pointing to a non–cpu address space. The template class *cuElementReference* is a proxy that works as an *smart reference* to enable this functionality. Element–wise matrix manipulation on a foreign address space is extremely inefficient an should be only used for debug purposes.

The calls to linear algebra libraries have been replaced by cuBLAS calls, and supplemented with custom kernels where functionality was not available.

[5]By default $target\_tpb = ENV\_DEFAULT\_TPB = 256$

[6]This is not always possible (i.e: when there are not enough threads to fill the block) nor mandatory (the code has no obligation to use the kernel launch parameters provided by the *cudaEnvironment* instance)

[7]This is provided for convenience. For the reasons detailed above, high performance can only be obtained when all the data and computation is performed on the GPU

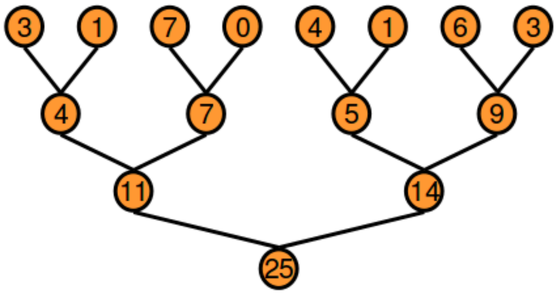

Figure 10.6: Visualization of the parallel reduction algorithm [31]

10.4.3 CUDA kernels

A range of kernels [8] has been written to perform simple functionality which is more less trivial to implement in CPU, but no CUDA version is available in cuBLAS [9]. The source for this kernels is listed in C.1:

- The *sin()* and *cos()* trigonometric functions.
- Several copy functions that take into account matrix transformations such as transpositions or submatrix selection.
- The *scale()* function to scale a matrix by an scalar.
- The *linspace()* function to generate a linearly spaced vector of elements.
- The *dlaswp()* function to perform row interchange in gaussian elimination algorithms.

10.4.4 Parallel reduction

The parallel reduction algorithm[10] is the base of the norm and distance kernels. Parallelization of the reduction process is performed in a hierarchical fashion. Each thread adds a pair of elements. Then, half of the threads will add the partial results computed. This will be repeated for $log_2 N$ iterations, until the whole list is reduced to a single value.

Several practical considerations must be considered to achieve optimal performance: minimal divergence, array indexing to optimize data locality, unrolling some operations, ...

One limitation of this technique is that, as the threads per block count is limited to 1024, lists bigger than 2048 elements cannot be reduced [11]. This is an important limitation for computing matrix norms and distances, as matrices as small as 45×45 already reach the limit. Bigger reductions are computed in two steps. First one reduces in parallel all the columns of the matrix

[8]To enhance modularity all the kernels have been included in the *kernel* namespace. Some of the kernels are templates themselves for a seamless transition between single and double precision arithmetic.

[9]cuBLAS version might be available but it does not take into account matrix transformations becoming inefficient for our usage.

[10]Our parallel reduction algorithm, listed in appendix C.2, is an adaptation of the work in [31].

[11]One thread is required for every pair of elements in the list.

to a vector in shared memory. Then the vector is reduced again to a single value. This technique is used in the *nrm2()* kernel.

The reduction algorithm has been split in two parts, the first iteration is done by the caller to allow for input data transformation, and then the recursive reduction is performed by calling the *reduce()* kernel. This is the case of the *norm()* and *dist()* kernels, where the square and the square of the difference is respectively computed in the fist iteration.

10.4.5 Banded matrix kernels

We had to implement some missing banded matrix functionality from cuBLAS. For instance, Level 3 banded matrix by dense matrix multiply was not provided, but Level 2 banded matrix by vector multiply was. We built the L3 operation over the L2 one by decomposing a matrix–matrix multiply into independent matrix–vector multiply and accumulate operations.

Our first approach, using streams to overlap the vector–matrix products, obtained a very low occupancy due to the cost of kernel launching relative to the workload of each thread. We found that best results are obtained when the whole matrix is computed with a single launch, as the work per thread increases and a higher number of threads are effectively launched, increasing the opportunities to obtain better occupancy.

We had the same results with stream based distance and matrix norm computation. We found ourselves implementing full matrix kernels as building over the provided vector oriented kernels was extremely inefficient.

Another important operation missing from cuBLAS was the tridiagonal system solver, which we implemented from scratch. The Thomas algorithm [32] is mostly sequential for each right–hand side, so achieving high occupancy was tricky. More parallelizable alternatives to the Thomas algorithm exist, but they have higher computational complexity and are only justified with extremely large systems. In our implementation, the kernel occupancy became register usage limited for the largest systems, implying that the solution is adequate for the range of sizes of the problem.

10.4.6 Solver convergence

At the inner loop of both algorithms, an iterative method is used to solve a linear system of equations. The number of steps required until convergence is limited by a constant.

This poses a problem for CUDA performance as one floating point value must be transferred form device to host to test for convergence. This is wasteful as the PCIe bus is slow and has a high latency.

To solve this problem we study the average number of steps until convergence. Lucky for us, this number is small and roughly constant so we use this information to implement a *minimum* number of iterations (2) until convergence is tested, saving most of the unneeded device to host transfers.

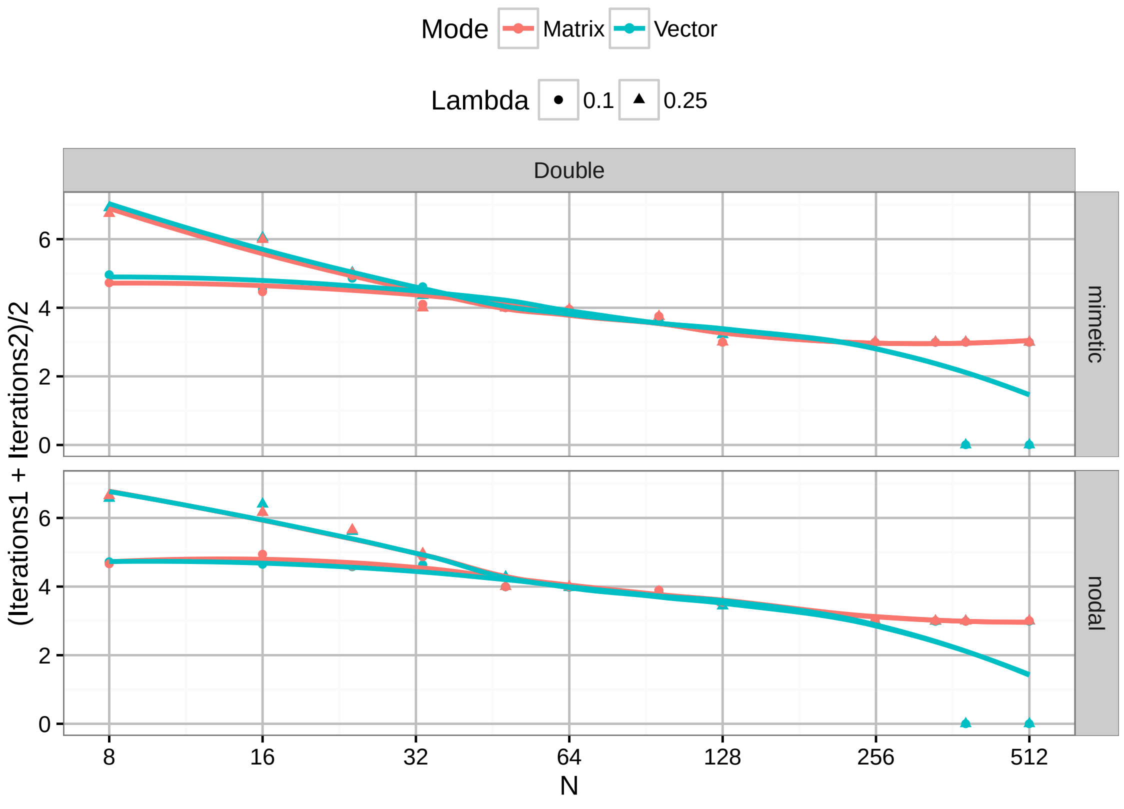

Figure 10.7: Solver iterations until convergence

Chapter 11

Results

11.1 Test systems

Two systems where used for testing. System 1 (S1) mounts an Intel core-i7 CPU and a NVIDIA Maxwell GPU, System 2 (S2) mounts an AMD Phenom-II X6 processor and a NVIDIA Fermi GPU. Aside from S1 being newer and more powerful than S2, these systems have selected because there are some architectural differences that may affect performance.

- S1 CPU clock speed is $\approx 20\%$ faster than S2's.
- S1 has 4 physical CPUs but HyperThreading technology allows to run up to 8 threads whereas S2 has 6 physicals CPU for a total of 6 threads.
- S1 cache L1, L2 cache is half the size of the S2 L1,L2 cache. S1 L3 cache is bigger than S2 L3 cache. S1 bus speed is 25% higher than S1.
- S1 GPU is more $2.5\times$ more powerful than S2's but S1 GPU lacks native double precision floating point support.
- S1 TDP is 25% lower than S2 TDP.

Full specifications for S1 and S2 can be found in table 11.1 and 11.1 respectively.

11.2 Output

Figures 11.1,11.2,11.3,11.4,11.5,11.6,11.7,11.8 illustrate the output and error of nodal and mimetic methods for an homogeneous problem for characteristic wavelengths of $\lambda = \frac{1}{4}$ and $\lambda = \frac{1}{8}$.

For this grid size the resulting error is quite low ($< 10^{-3}$) but has a different pattern for each method. The resulting error from applying the nodal method is much *noiser* than the produced by the mimetic method.

We believe this behaviour is due to the higher numerical complexity of the nodal method, which introduces a higher rounding error due to the finite accuracy of the floating point representation. This becomes more evident for $\lambda = \frac{1}{8}$, where the higher rate of variation of the solution challenges the nodal method accuracy, while the mimetic method is able to stay within a more regular error pattern.

11.3 MATLAB / Octave

The single threaded MATLAB / Octave implementation shows that both methods have a similar complexity of approximately $O(N^{3\ldots4})$.

CPU	
Commercial name	Intel(R) Core(TM) i7-4770 CPU at 3.40GHz
Release Date	Q2’2013
Processor Number	I7-4770
Cache Size	8 MB SmartCache
Bus Speed	5 GT/s DMI2
Instruction Set	64 bit
Instruction Set Extensions	SSE4.1/4.2, AVX 2.0
Floating point speed	99.72 GFLOPS [1]
Lithography	22 nm
Cores / Subprocesses	4/8
Clock frequency (base/turbo)	3.4 GHz / 3.9 GHz
Power	84 W

RAM	
Memory Type	DDR3
Memory Speed	1600 MHz
Memory Configuration	2×4096 MB + 2×2048 MB
Latency	36 cycle + 57 ns

Cache	
L1 Instruction cache	4×32 KB, 64 B/line, 8-WAY
L1 Data cache	4×32 KB, 64 B/line, 8-WAY, 4-5 cycle latency
L2 cache	4×256 KB, 64 B/line, 8-WAY, 12 cycle latency
L3 cache	8 MB, 64 B/line, 36 cycle latency

GPU	
Model	GeForce GTX 960 (Maxwell)
Launch Date	January 22, 2015
Code Name	GM206
Lithography	28 nm
Bus Interface	PCIe 3.0 x16
CUDA	5.2 / 1024 cores
Memory	2048 MB
Memory Bus	128bit GDDR5 @ 3505 MHz
Memory Bandwidth	112 GB/s
Processing Power (single/double)	2308/72.1 GFLOPS
TDP	120 W

Table 11.1: Test system 1 specs

CPU	
Commercial name	AMD Phenom(tm) II X6 1055T Processor at 3.40GHz
Release Date	Q2’2010
Processor Number	1055T
Bus Speed	4 GT/s 2000 MHz HyperTransport
Instruction Set	64 bit
Instruction Set Extensions	SSE4a, AMD-V
Floating point speed	54.34 GFLOPS
Lithography	45 nm
Cores / Subprocesses	6/6
Clock frequency (base/turbo)	2.8 GHz / 3.3 GHz
Power	95 W

RAM	
Memory Type	DIMM
Memory Speed	1600 MHz
Memory Configuration	2 × 2048 MB
Latency	157 cycle

Cache	
L1 Instruction cache	6 × 64 kB 2-way set associative, 1 cycle latency
L1 Data cache	6 × 64 kB 2-way set associative, 1 cycle latency
L2 cache	6 × 512 kB 16-way set associative exclusive, 13 cycle latency
L3 cache	Shared 6 MB 48-way set associative, 48 cycle latency

GPU	
Model	GeForce GTX 460 (Fermi)
Launch Date	July 12, 2010
Code Name	GM104
Lithography	40 nm
Bus Interface	PCIe 2.0 x16
CUDA	2.1 / 336 cores
Memory	1023 MB
Memory Bus	256 bit GDDR5 @ 1800 MHz
Memory Bandwidth	115.2 GB/s
Processing Power	907.2 GFLOPS
TDP	160 W

Table 11.2: Test system 2 specs

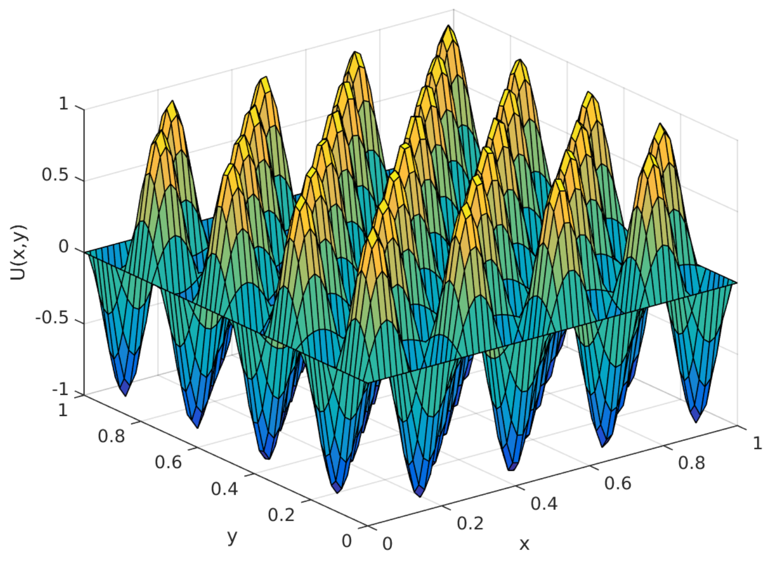

Figure 11.1: Nodal output waveform: $N = 64, \lambda = 1/4$.

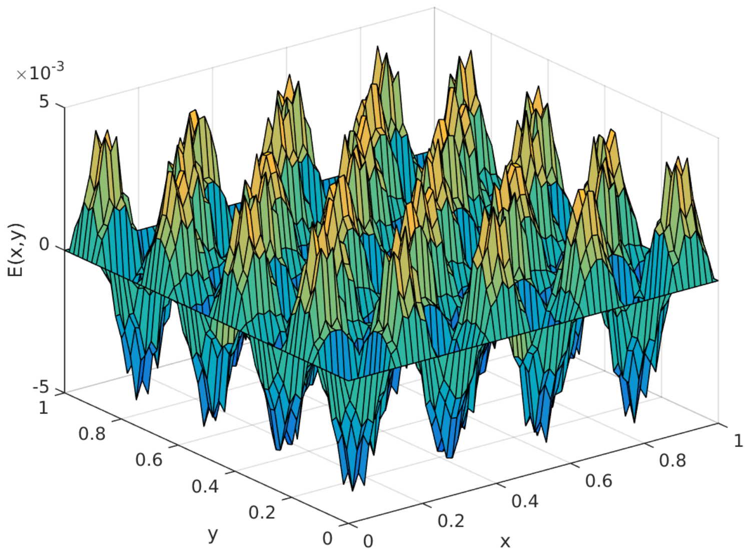

Figure 11.2: Nodal output error: $N = 64, \lambda = 1/4$.

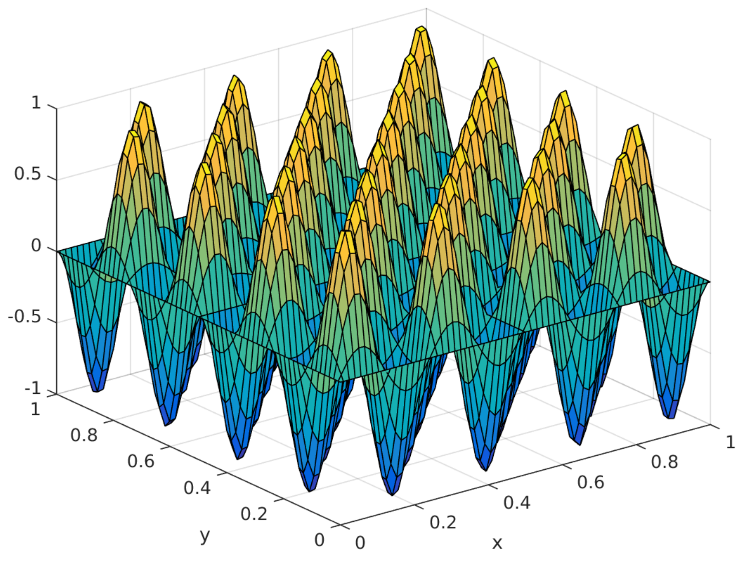

Figure 11.3: Mimetic output waveform: $N = 64, \lambda = 1/4$.

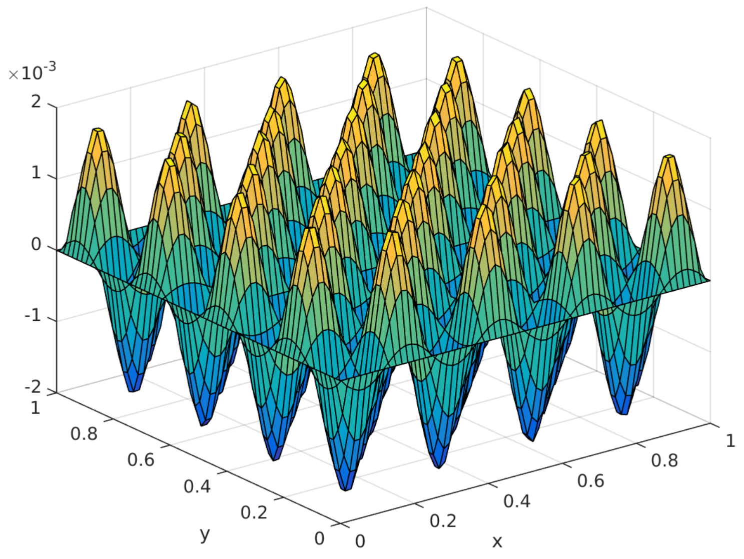

Figure 11.4: Mimetic output error: $N = 64, \lambda = 1/4$.

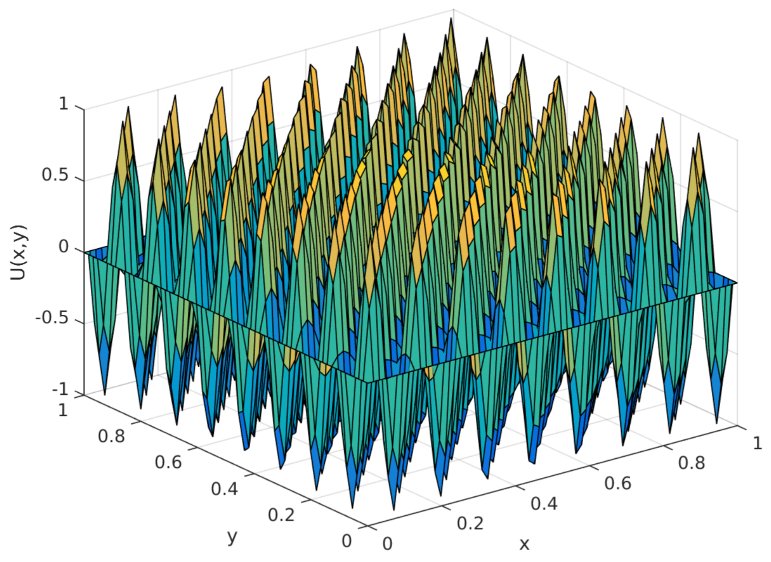

Figure 11.5: Nodal waveform: $N = 64, \lambda = 1/8$.

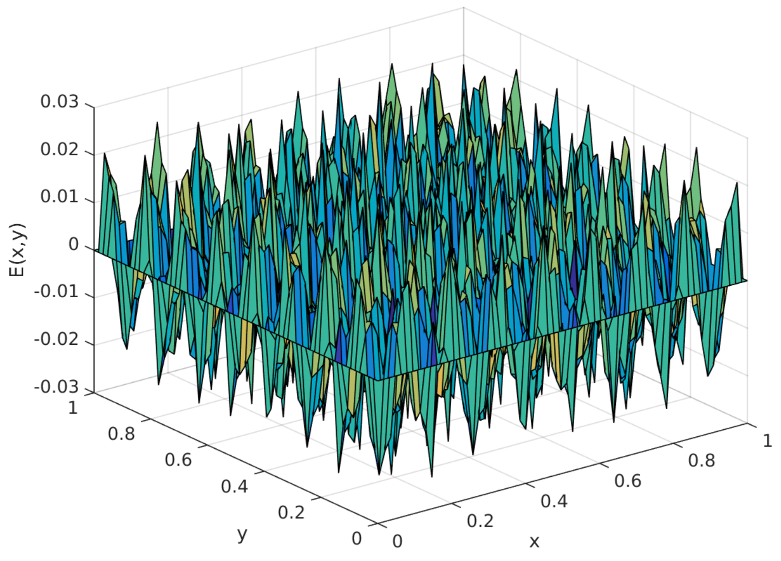

Figure 11.6: Nodal output error: $N = 64, \lambda = 1/8$.

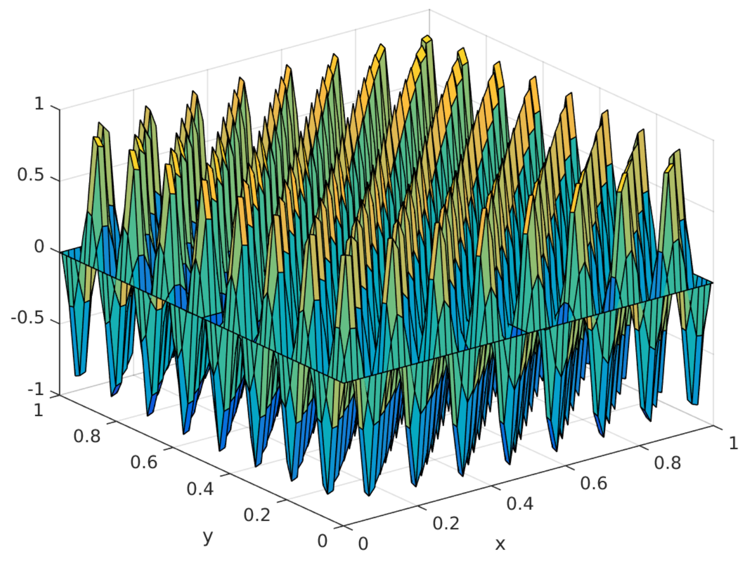

Figure 11.7: Mimetic waveform: $N = 64, \lambda = 1/8$.

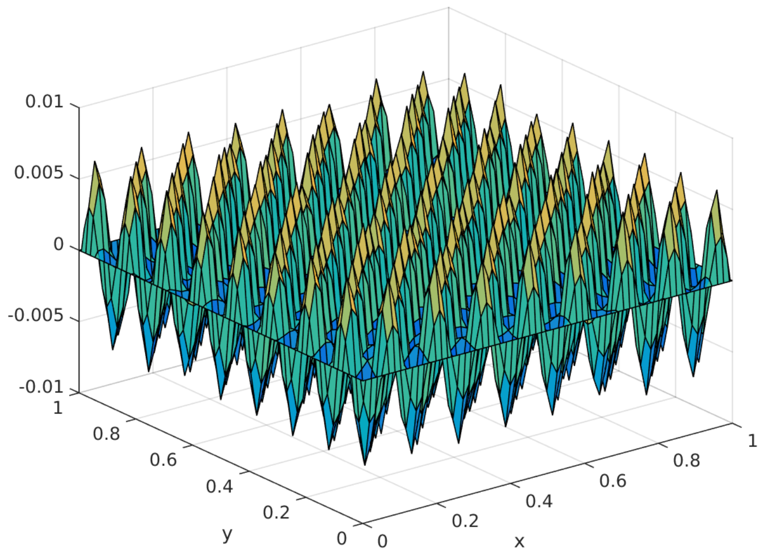

Figure 11.8: Mimetic output error: $N = 64, \lambda = 1/8$.

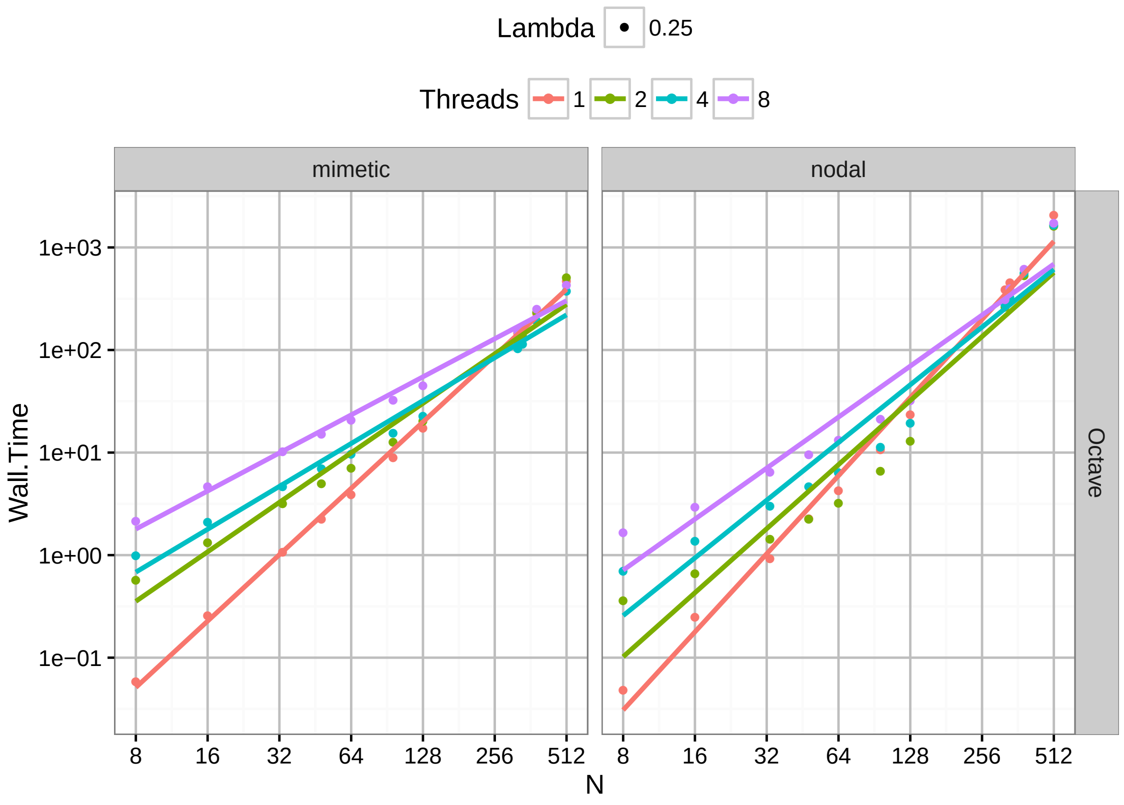

Figure 11.9: Octave: Method timing as function of grid size.

We see now why the parallelization of the Octave code did not work. The overhead introduced by the *parallel* package is so big that it effectively flattens the slope and no practical speedup is achieved until $N > 300$. This overhead increases with the number of processes, as more data needs to be copied over. In fact, the slope is flattening to $O(N^2)$ which is the complexity of copying a matrix. For high N, the speedup obtained is small even when using 8 threads.

For the nodal method the we have $\approx O(N^3)$ and for the mimetic case $\approx O(N^{3.26})$. This linear estimation is to be taken with a grain of salt as the nodal method complexity is not really linear, being flatter for small N and getting steeper as N increases.

This effect could be related to Octave changing the compute strategy for matrices of different sizes. Another factor at play could be cache misses, as the knee is located at the expected L1-L2 cache exhaustion range8.1.3.

Note that the complexities computed are *better* than the theoretical bound found. This is due to the assumption of complexity $\approx O(N^3)$ for matrix product when it is probably $O(N^{2.8})$[2] and a more efficient implementation of quasi-banded sparse matrices.

[2]Using the Strassen algorithm [33]. It could be even $O(N^{2.375477})$ if using the Coppersmith–Winograd algorithm described in [34], although impractical for the small matrices allowed by the methods.

11.4 Single–threaded C++

In figure 11.10 we see how the *matrix oriented* dominates the *vector oriented* implementation. This is expected as the *matrix orientation* is able to exploit row/column redundancies and optimize at the loop level, thereby obtaining a small speedup.

However, as N increases, the *matrix oriented* strategy losses its lead as this relatively small optimization becomes less important relative to the rest of the computations. This optimizations seem to play a bigger role in the nodal method.

The complexity of the methods is now at $O(N^{3.6})$ for the nodal method and $O(N^4)$ for the mimetic method. This values are more in line with the theoretical complexity estimated in this work, losing the edge obtained in MATLAB/Octave.

The times obtained for the C++ implementation are much lower than the reference MATLAB/Octave implementation. This is most certainly due to the removal of the interpreter overhead and compile time optimizations.

The complexity of the MATLAB/Octave code had two terms (interpreter and BLAS). Interpreter complexity is basically O(N) so it is effectively flattening the timing curve and thus giving the false impression of a more efficient algorithm when times are really much better in the C++ implementation.

The BLAS term however is essentially the same for Octave and C++ implementation so, the speedup of C++ code will fade as N gets bigger and the BLAS library calls take all the runtime.

This figure also contains multiple wavelengths for each method which result in an overlap of data points as no significant timing difference exists in practice.

11.5 Multi–threaded C++ (OpenMP)

In the case of multithreading we observe a relatively small speedup. It is noticeable ($2\times$ for 8 core) when compared to the *vector oriented* implementation, but almost no gain is obtained compared to the *matrix oriented* method for small N. Sometimes the parallel version performs worse for very small N due to the overhead of thread creation.

The ($2\times$) gain over the *matrix oriented* implementation is obtained for high N as this implementation converges with the *vector oriented* as N increases. This could be related to an inefficient cache utilization when the matrices become large in the *matrix oriented* implementation.

The OpenMP implementation, being effectively a multithreaded *vector oriented* implementation seems to deal better with large problems. As expected, complexity is the same as in the single thread case.

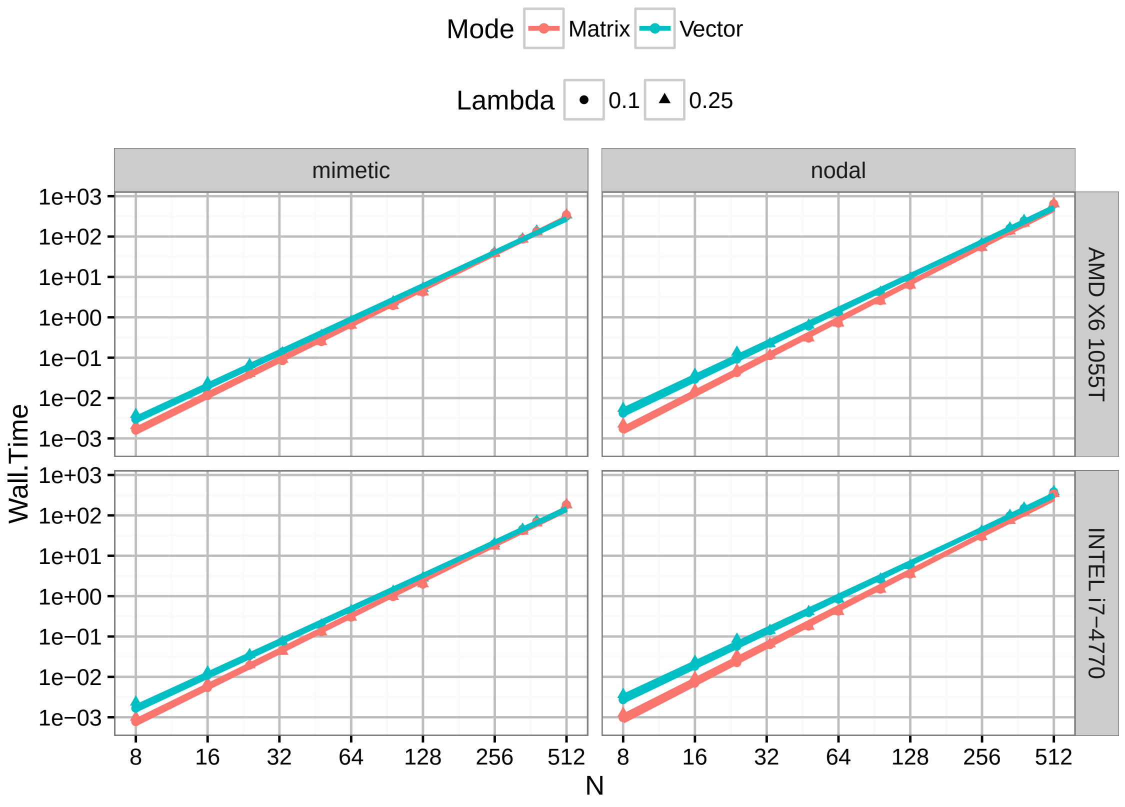

Figure 11.10: C++: Method timing as function of grid size.

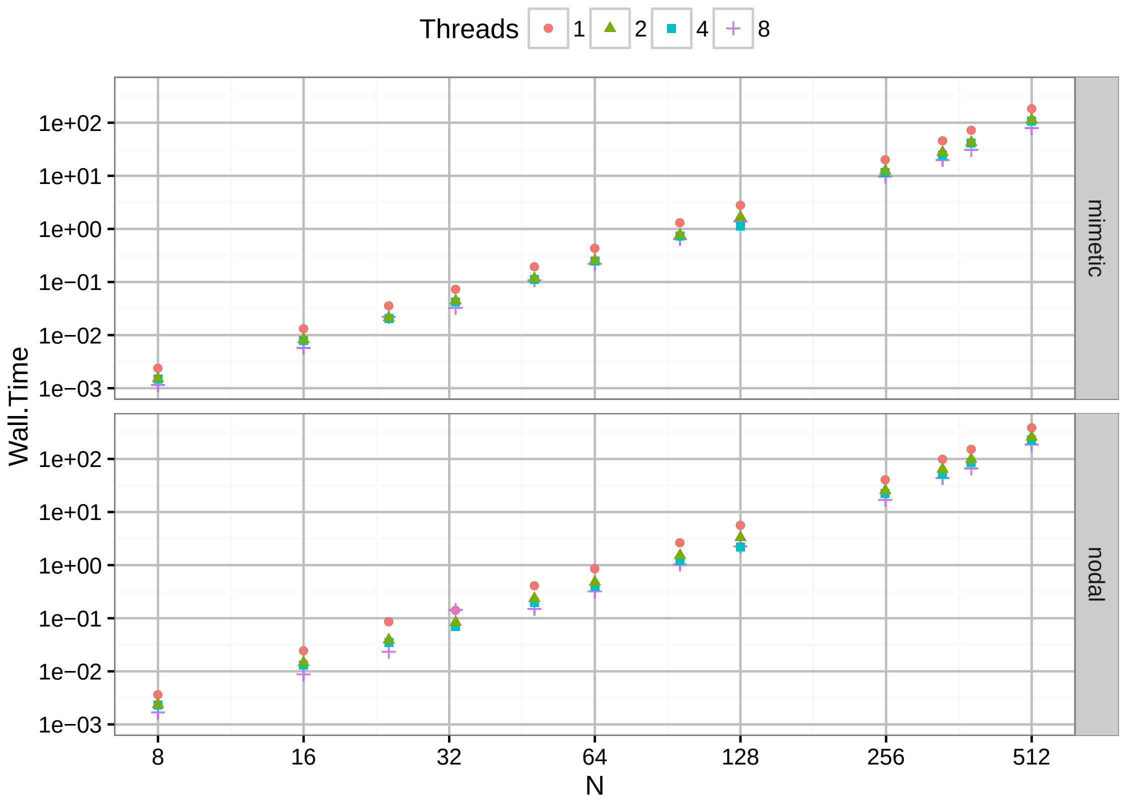

Figure 11.11: OpenMP: Method timing as function of grid size.

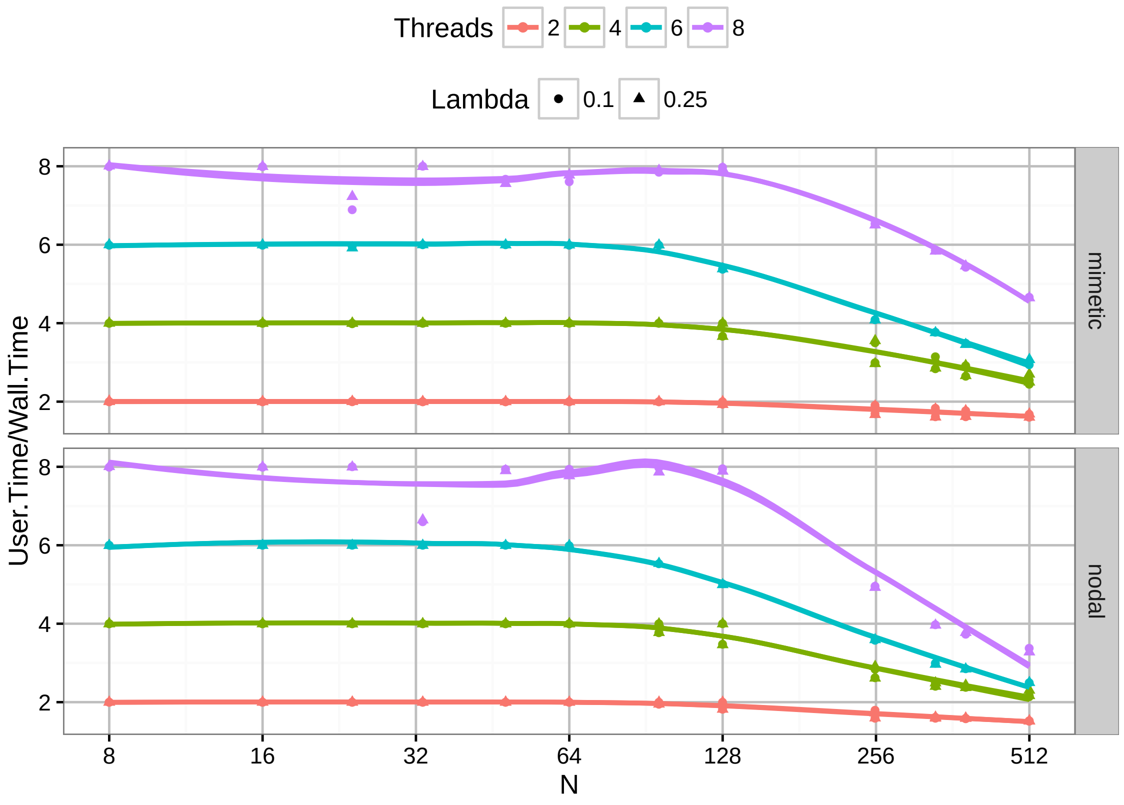

Figure 11.12: Method timing complexity as function of grid size.

Figure 11.12 shows the ocuppation for the methods as a function of N and the number of threads used [3]. As expected, we see that utilization falls as we get out of the L1-L2 cache exhaustion range.

The mimetic method, which uses computations simpler than the nodal method, seems to be able to achieve better utilization, as its curve falls slower for high N than the nodal method.

An anomaly seems to appear near the point N=32 for the case $N = 8$. That point is in fact $N = 33$, which is problematic to the parallelization as it is not divisible by the number of threads used, resulting in an uneven workload and idle threads.

11.6 CUDA C++

With the CUDA implementation with had our first chance of analysing the behaviour of the method error relative to the floating precision used.

Figure 11.13 shows a crucial difference between the mimetic and the nodal method. The mimetic method, with it's simpler arithmetic is able to function either with single or double

[3]The 6 thread data comes form an S2 test

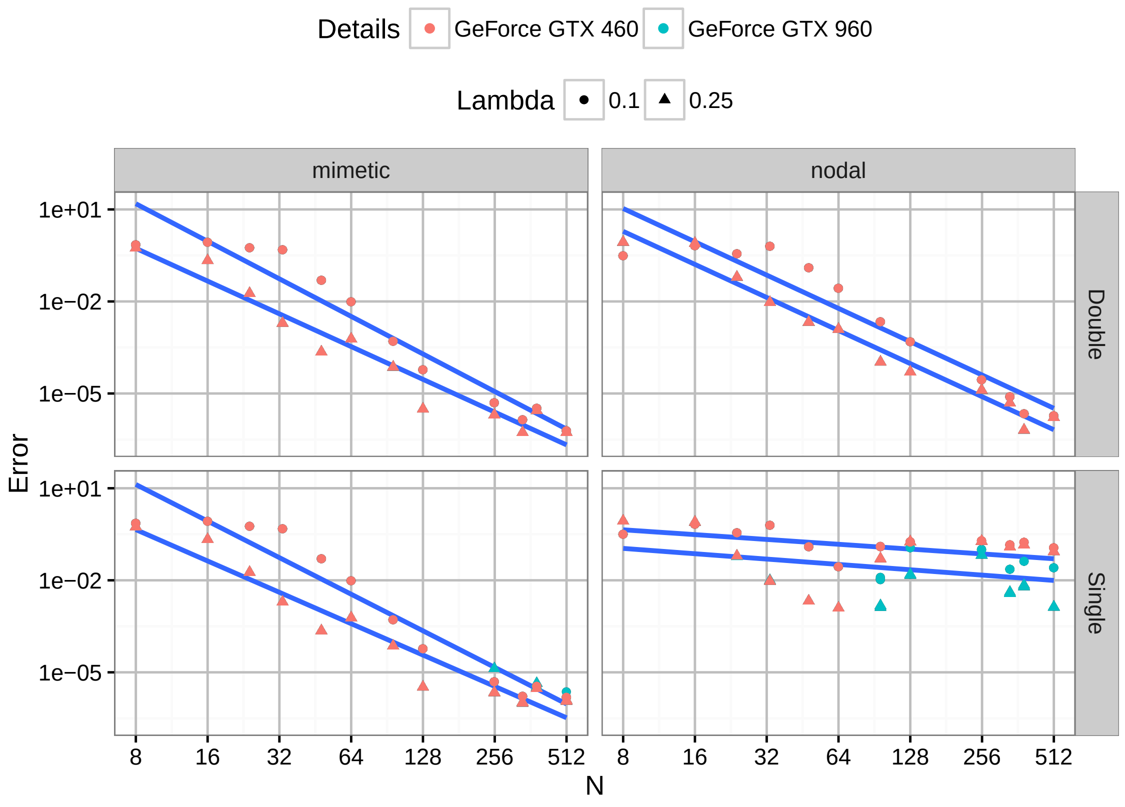

Figure 11.13: CUDA: Method error as function of grid size.

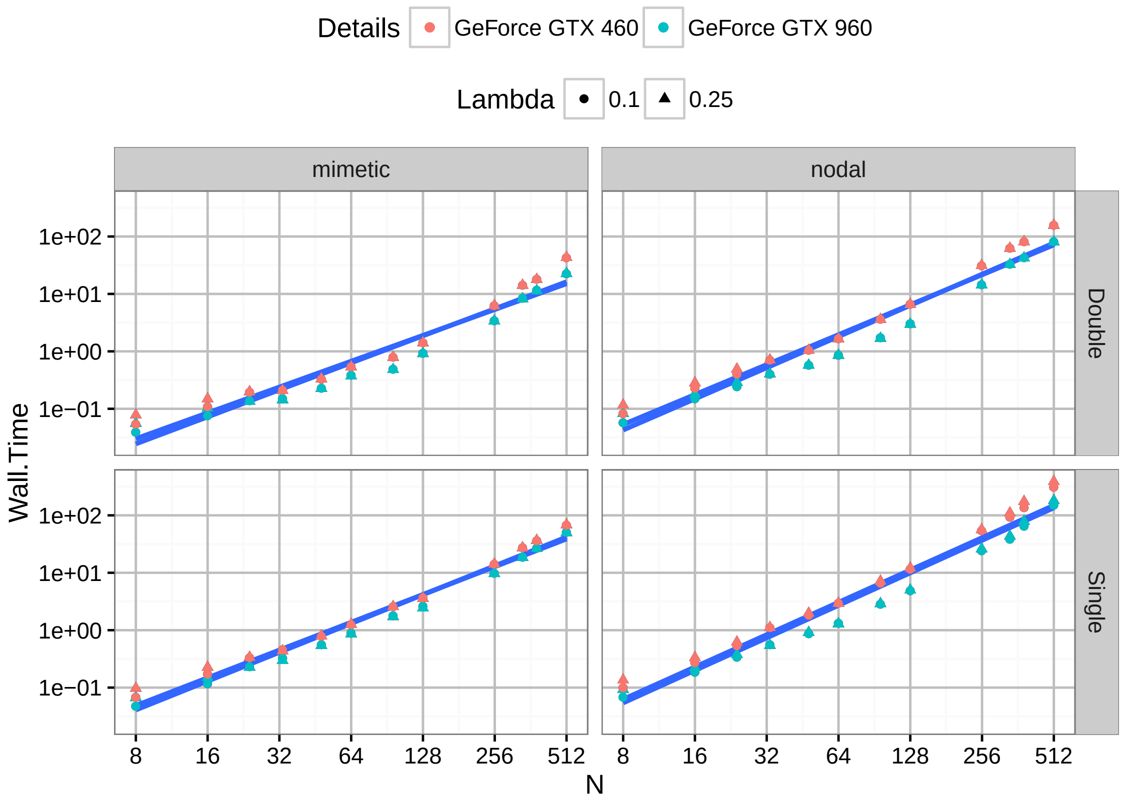

Figure 11.14: CUDA: Method timing as function of grid size.

precision, but the nodal method is able to work with double precision arithmetic.

The most probable reason behind this behaviour is that the nodal method is an implicit method, which means that some systems must be solved at each step.

The systems to be solved are tridiagonal so we use the Thomas Algorithm. The Thomas algorithm performs a forward substitution on the whole matrix followed by a backward substitution. This is a long operation where the error at each step adds up, resulting in degraded accuracy when single precision arithmetic is used.

In this case, floating point precision is not only relevant to the total error, but also the GPU performance as the Fermi chipset on S2 has native floating point support which S2 is lacking.

This fact is already evident in the different level of error obtained for the same single precision implementation of the nodal method on both GPUs.

The CUDA speedup is $\approx 20\times$ for large N, but worse than the C++/OpenMP implementation for $N \leq 96$. This is expected, as the costs of data transfer and kernel launch are too much when the problem is small. This produces flatter curves, specially for small N, as can be seen in the non linear shapes in 11.14.

Interestingly enough, the single precision version is not faster. This is caused by the need of more iterations for the solution to converge as the reduced accuracy induces a slower convergence.

The late check of converge described in 10.4.6 proves valuable to achieve a good performance when using small grid sizes where the cost of the convergence check time is high relative to the light computational workload.

11.7 Compared performance

Obtained speedups relative to the reference implementation are presented in tables 11.3 for the nodal method and 11.4 for the mimetic method.

The results for the MATLAB/Octave parallel implementation are bad. A small amount of improvement is obtained (less than 2×) only for very high N.

The C++ implementation is best for small problems. No interpreter overhead is present and the whole data fits into cache, resulting in impressive performance (almost 40× for $N = 8$). Of the two approaches, *matrix* and *vector* oriented, the first clearly dominates the second as more opportunities for optimization are given to the algebra libraries.

The OpenMP, being based in the inferior *vector oriented* approach has a performance slightly better than the *matrix oriented* approach for $N \leq 128$, but as soon as the L1-L2 cache is exhausted, it becomes faster by a factor of 2×, reaching a speedup of 20× relative to the reference implementation.

We observe different tendencies for the nodal and mimetic GPU implementations. On the one hand, the speedup obtained with the nodal implementation seems to grow with N, probably because the parallization achieved by the Thomas Algorithm is low for the range of sizes where the methods are applicable.

On the other hand, the mimetic method seems to converge to a speedup of about 20×, probably because a high GPU occupancy is achieved limiting further parallelization.

In fact, even the largest problem is relatively small for a GPU, where matrices are usually in the range of millions of elements and we are only using hundreds of thousands at best.

To sum up, a speedup of 10 . . . 28× (nodal) and 18 . . . 50× (mimetic) relative to the reference implementation is achievable over the whole range of grid sizes by employing the correct implementation. CPU based implementations are best for small problems and GPU based solutions shine when the problem is large or very large.

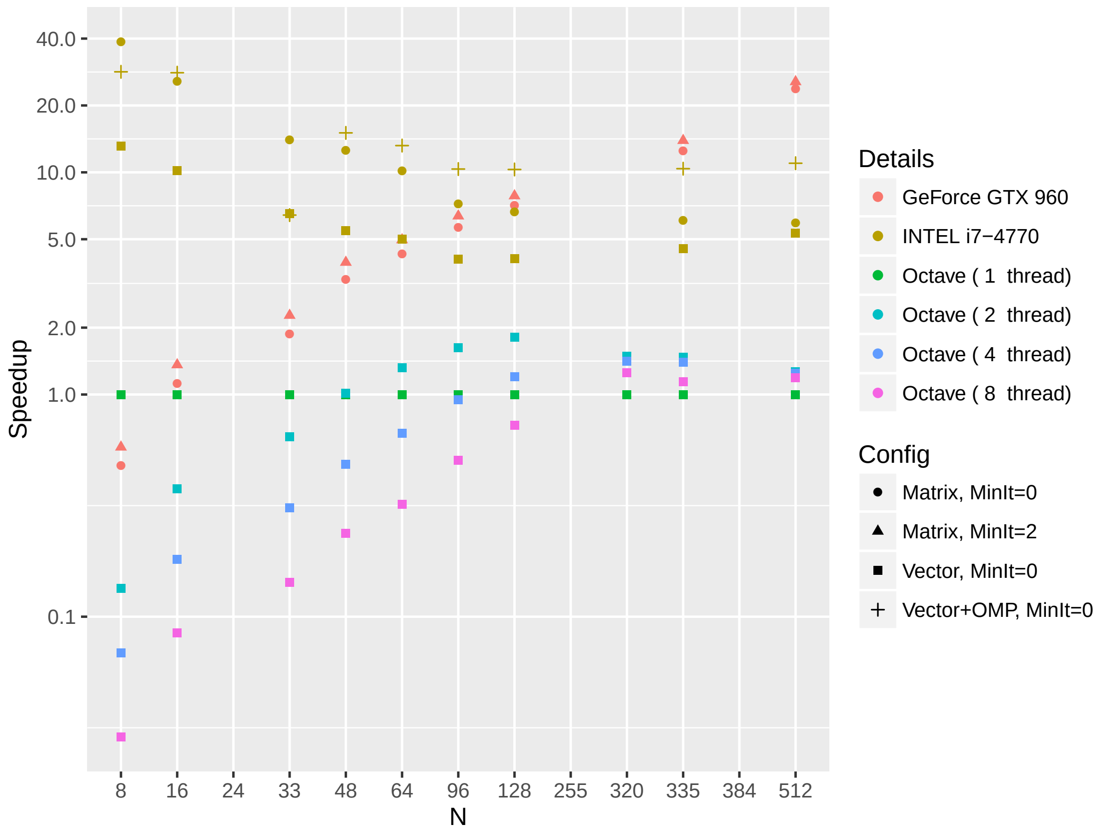

Figure 11.15: Mimetic method timings compared.

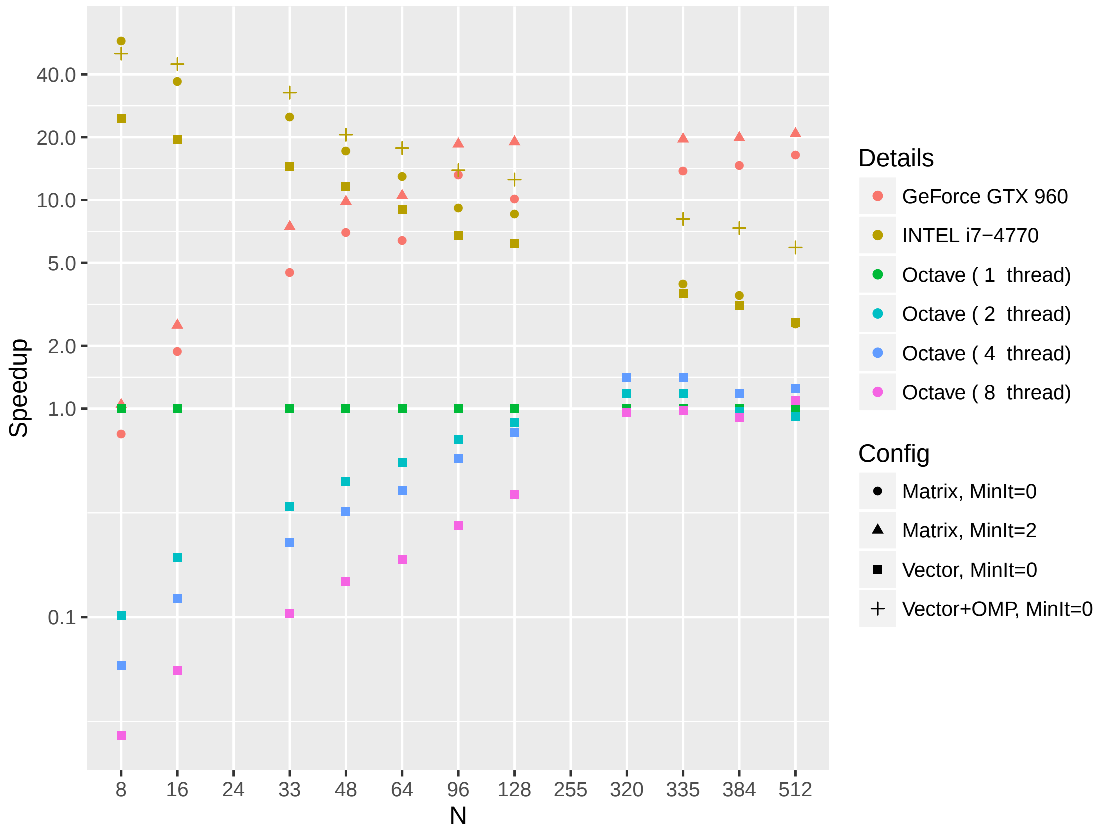

Figure 11.16: Mimetic method timings compared.

Details	Threads	MinIt	8	16	33	48	64	96	128	320	335	512
Octave	1	0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Octave	2	0	0.13	0.38	0.64	1.01	1.32	1.63	1.81	1.49	1.47	1.27
Octave	4	0	0.07	0.18	0.31	0.49	0.67	0.95	1.21	1.41	1.40	1.24
Octave	8	0	0.03	0.08	0.14	0.24	0.32	0.51	0.73	1.25	1.14	1.19
C++	1	0	13.10	10.19	6.51	5.47	5.00	4.06	4.08		4.54	5.31
C++	8	0	**28.34**	**28.08**	**6.42**	**15.05**	**13.20**	**10.35**	**10.30**		10.39	10.99
CUDA	256	0	0.48	1.12	1.87	3.30	4.29	5.65	7.10		12.53	23.80
CUDA	256	2	0.58	1.36	2.27	3.93	4.97	6.36	7.84		**13.91**	**25.57**

Table 11.3: Speedups obtained for each implementation of the nodal method relative to Octave reference implementation. Optimal implementation in bold.

Details	Threads	MinIt	8	16	33	48	64	96	128	320	335	384	512
Octave	1	0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Octave	2	0	0.10	0.19	0.34	0.45	0.55	0.71	0.86	1.18	1.18	0.97	0.92
Octave	4	0	0.06	0.12	0.23	0.32	0.41	0.58	0.76	1.41	1.42	1.19	1.25
Octave	8	0	0.03	0.06	0.10	0.15	0.19	0.28	0.39	0.96	0.98	0.91	1.10
C++	1	0	24.62	19.50	14.45	11.58	8.96	6.79	6.17		3.55	3.14	2.57
C++	8	0	**50.35**	**44.83**	**32.72**	**20.56**	**17.74**	13.89	12.53		8.11	7.34	5.92
CUDA	256	0	0.76	1.88	4.49	6.98	6.39	13.19	10.13		13.80	14.63	16.46
CUDA	256	2	1.04	2.51	7.44	9.83	10.49	**18.54**	**18.99**		**19.59**	**19.88**	**20.77**

Table 11.4: Speedups obtained for each implementation of the mimetic method relative to Octave reference implementation. Optimal implementation in bold.

Chapter 12

Conclusions

The nodal method uses implicit differential operators which means higher numerical stability at the cost of a higher running time. For small N the limiting factor is the number of operations required, specially for system solving. For high N a single dense-by-dense matrix multiplication becomes the main cost due to its high complexity compared to the other operations.

The mimetic method uses explicit differential operators which means lower running time at the cost of a lower numerical stability. This method does not require linear system solving nor dense-by-dense matrix multiplication, which makes it very fast. Due to constant factor and asymptotic complexity being lower for the mimetic method than the one for the nodal method, one could use a denser grid to improve numerical complexity when needed.

In our test cases, using well behaved solutions, the mimetic method is more stable numerically than the nodal method. The mimetic method supports single precision arithmetic (which the nodal method does not); this is due to a lower operation count producing a lower rounding error.

To perform optimally, both methods require some fine-tuning of the cfl_{max} parameter and the solver convergence iteration limit.

The reference MATLAB implementation was hard to optimize, being the main optimization an algorithmical one. We derived a *matrix oriented* approach from the *vector oriented* approach (rows/columns) in the original code. This allowed for a reduction of loops and better utilization of the MATLAB code.

Our main efforts with the C++ implementation were focused on:

- Build an expressive and *algebra friendly* C++ dialect to make the development easier and at the same time allow for multiple algebra implementations under the hood.
- Decouple the physical problem from the algebra/optimization, thus allowing for a much more readable, testable and less error prone code.
- Ensure correctness by building an automated test system for our algebra implementation.

Some C++11 features such as shared pointers and move semantics where essential in achieving expressivity without losing performance.

The resulting C++ code was much faster than the MATLAB one, however the complexity was higher (but more inline with the theoretical estimation). This could be attributed to the higher constant factor of MATLAB operations hiding the cost of the most complex ones, a cost which is revealed when this constant factor drops, as is the case of our C++ implementation.

When we began with the optimization, the first *problem* found was that the matrix size for the methods (8–384) is quite small for a modern computer, making it hard to achieve high perfomance with standard linear algebra libraries, which are optimized for much larger problems.

The *matrix/vector* approaches are the building foundations for our subsequent parallelization attempts as they allow for parallelization at fine (matrix element) or coarse (whole vector) levels. This idea is crucial as it allows us to adapt to the different levels of parallelism provided by different architectures (multi–CPU vs GPU).

The *OpenMP* implementation was almost trivial to develop from the *vector oriented* approach as vectors decouple and no synchronization is needed between threads. This solution works best for small/medium sized problems, but performance drops with small systems when N is not a multiple of the available CPUs, as some CPUs become idle for a significant amount of time.

cuBlas was used for most CUDA algebra implementation, except for some missing or performance critical kernels[1]. Keeping matrices in GPU memory and minimizing transfers between host and device was essential in obtaining good performance. The usage of *streams* was problematic due to the launch time being too high in relation to the kernel working time. In any case GPU occupation was not very high due to the small size of the matrices used in this work. The check for convergence in the ADI solver stage was a performance limiting factor as it required several small *device–host* DMA transfers. As the number of iterations until convergence is roughly constant, we minimized this bottleneck by delaying the convergence check until a fixed number of iterations is performed.

Our results show that the *OpenMP* implementation is better for small N whereas the CUDA implementation is faster for higher N. Switching between implementations depending on the required N would yield the best performance overall.

[1]See appendix C

Appendix A

Matrix naming

Due to the imposibility of using a consistent nomenclature for reduced matrices (i.e.: $\bar{\mathbf{X}}$) in C/C++ or MATLAB code, some matrix renaming has been introduced. The text in this work attempts to be consistent with published work whenever possible.

A.1 The Nodal method

Attached Code	This work
$\mathbf{P}$	$\bar{\mathbf{P}}$
$\mathbf{Q}$	$\bar{\mathbf{Q}}$
$\mathbf{S}$	$\mathbf{P}$
$\mathbf{T}$	$\mathbf{Q}$
$\mathbf{A}$	$\mathbf{F}$
$\mathbf{C}$	$\mathbf{G}$
ai	rows of $\mathbf{A}$
bi	rows of $\mathbf{B}$
cj	columns of $\mathbf{C}$
dj	columns of $\mathbf{D}$
$\mathbf{M}$	$\mathbf{H}$

A.2 The Mimetic method

Attached Code	This work
$\mathbf{A}$	$\mathbf{F}$
$\mathbf{C}$	$\mathbf{G}$
ai	rows of $\mathbf{A}$
bi	rows of $\mathbf{B}$
cj	columns of $\mathbf{C}$
dj	columns of $\mathbf{D}$
$\mathbf{M}$	$\mathbf{H}$
$\mathbf{M}_1$	$\mathbf{H}_1$

Appendix B

MATLAB sources[1]

B.1 Description

The MATLAB implementation described in this chapter is used as a reference to assess the validity of all the other implementations.

B.2 The Nodal method

Initialization of common constants is performed in lines 8-14 of .

- tfin: Simulation end time.
- epsilon: Precision required for convergence in the simultaneous tridiagonal solver.
- cflmax: Used with the grid size to determine the coarseness of the time step (heuristically tuned for the method).
- lamb: Wavelength of the solution.
- casos: List of grid sizes to test.

The for loop in Lines 16-121 performs the computation for each grid size given in *casos*.

1. 26-39 Computes the method matrices **P**,**Q**,**M**,**S**,**T**.
2. 41-50 Compute the grid which will be used along with the exact problem solution for error calculation.
3. 52-102 The for loop solves the acoustic problem for one time step:
   (a) 53-56 Update of the intermediate matrices $\mathbf{A}, \tilde{\mathbf{V}}, \tilde{\mathbf{W}}$.
   (b) 58-76 The for loop solves the first stage of the algorithm row by row.
   (c) 78-81 Update of the matrices $\mathbf{C}, \mathbf{V}, \mathbf{W}$ using the intermediate result from the previous step.
   (d) 83-101 The for loop solves the first stage of the algorithm column by column.
   (e) 104-120 The error is computed and various stats are printed.
4. 123-127 The result from the last computation is plotted along with the solution error.

[1]Code has been adapted for enhance printability. Original file available on digital accompanying material.

```
% UNIVERSIDAD CENTRAL DE VENEZUELA
% POSTGRADO EN CIENCIAS DE LA COMPUTACION
% TUTOR: Dr. OTILIO ROJAS
% ESTUDIANTE: LUIS JOAQUIN CORDOVA DIAZ
%%% Estudio convergencia Compacto nodal 2D un cfl varios
%%% casos = [8 16 33 48 64 96 128 320 335]

tfin = 5/sqrt(2);
epsilon = 1e-5;
dospi = 2*pi;
casos = [8 64 16 33 48 64 96 128 320 335];
s = length(casos);
cflmax=0.915;
lamb=0.25;

for caso = 1:s
    N = casos(caso);
    h = 1/(N-1);
    dt=cflmax*h;
    dts2 = .5*dt;
    niter = round(tfin/dt);
    r = (2:N-1);

    unos = ones(N-2,1);
    dts2_lamb2 = dts2*lamb*lamb;
    P = spdiags([unos 4*unos unos],[-1 0 1],N-2,N-2)/3;
    Q = spdiags([-unos unos],[0 2],N-2,N)/h;
    M = P\Q;

    T = spdiags([-unos unos],[-2 0],N,N-2);
    T(1,1:2) = [4 1]/3;
    T(N,N-3:N-2) = [-1 -4]/3;
    T = T/h;

    unos = ones(N,1);
    S = spdiags([unos 4*unos unos],[-1 0 1],N,N);
    S(1,1:2) = [2 4];
    S(N,N-1:N) = [4 2];
    S = S/3;

    x = linspace(0,1,N);
    y = x';
    V = zeros(N,N);
    W = V;
    Utilde = V;


    U = sin(dospi/lamb*y)*sin(dospi/lamb*x);
    % SOLUCION DE LA U EXACTA
    Uexacta = U*cos(sqrt(2)*dospi*tfin);
    tstart = cputime;
    for n = 1:niter
        A = M*W(:,r)*P';
        Vtilde = V;
        Wtilde = W;
        Wtilde(:,r) = W(:,r)-dts2*(S\T*U(r,r));
        % RESOLUCION DE LA PRIMERA ETAPA FILA POR FILA
        for i = r
            ai = U(i,r)*P'-dts2_lamb2*A(i-1,:);
            bi = V(i,:)*S';
            uviejo = U(i,r);
            vviejo = V(i,:);

            test = 1;
            k = 0;
            while (test > epsilon) && (k < 12)
                unuevo = (ai-dts2_lamb2*vviejo*Q')/P';
                vnuevo = (bi-dts2*unuevo*T')/S';
                test = norm([unuevo-uviejo vnuevo-vviejo]);
                uviejo = unuevo;
                vviejo = vnuevo;
                k = k+1;
            end
            Utilde(i,r) = unuevo;
            Vtilde(i,:) = vnuevo;
        end   % FIN DE LA PRIMERA ETAPA

        C = P*Vtilde(r,:)*M';
        V = Vtilde;
        W = Wtilde;
        V(r,:) = Vtilde(r,:) - dts2*Utilde(r,r)*T'/S';
        % RESOLUCION DE LA SEGUNDA ETAPA COLUMNA POR COLUMNA
        for j = r
            cj = P*Utilde(r,j)-dts2_lamb2*C(:,j-1);
            dj = S*Wtilde(:,j);
            uviejo = Utilde(r,j);
            wviejo = Wtilde(:,j);

            test = 1;
            k = 0;
            while (test > epsilon) && (k < 12)
                unuevo = P\(cj-dts2_lamb2*Q*wviejo);
```

```
                    wnuevo = S\(dj-dts2*T*unuevo);
                    test = norm([unuevo-uviejo; wnuevo-wviejo]);
                    uviejo = unuevo;
                    wviejo = wnuevo;
                    k = k+1;
               end
               U(r,j) = unuevo;
               W(:,j) = wnuevo;
          end   % FIN DE LA SEGUNDA ETAPA
     end

     E = U-Uexacta;
     errornuevo = h*norm(E,'fro');
     telapsed = cputime-tstart;
     if caso == 1
          disp(['ORDEN OBSERVADO CUANDO cfl=0.91..errores  para U',...
                 '--->tfin = 7/sqrt(2)--->epsilon=1e-5'])
          disp(['             N               h               error', ...
                 '             orden      Tiempo'])
          fprintf('%16.1f %16.3f %19.2e---------------%11.3e  \n',...
                   N,h,errornuevo,telapsed)
     else
        orden = log2(errorviejo/errornuevo);
        fprintf('%16.1f %16.3f %19.2e %15.3f   %11.3e\n',...
                 N,h,errornuevo,orden,telapsed)

     end
     errorviejo = errornuevo;
end

figure, surf(x,y,U), xlabel x, ylabel y, zlabel U(x,y),
title('ONDA  CALCULADA')

figure, surf(x,y,E), xlabel x, ylabel y, zlabel E(x,y),
title ERROR
```

B.3 The Mimetic method

For the mimetic method, the computation of the matrices **RD**, **RG** of 4th order and size N is performed in the *matriz_RD(N)* and *matriz_RG(N)* functions.

The structure of the code is very similar to the one presented in the previous section, the main difference lies in the way the updates are performed.

Initialization of common constants is performed in lines 7-13 of B.3.

- tfin: Simulation end time.
- epsilon: Precision required for convergence in the simultaneous tridiagonal solver.
- cflmax: Used with the grid size to determine the coarseness of the time step (heuristically tuned for the method).
- lamb: Wavelength of the solution.
- casos: List of grid sizes to test.

The for loop in Lines 14-118 performs the computation for each grid size given in *casos.*

1. 22-34 Computes the method matrices **G**,**RG**,**M1**,**D**,**RD** and **M**.
2. 36-49 Compute the grid which will be used along with the exact problem solution for error calculation.
3. 51-118 The for loop solves the acoustic problem for one time step:
   (a) 52-55 Update of the intermediate matrices $\mathbf{A}, \tilde{\mathbf{V}}, \tilde{\mathbf{W}}$.
   (b) 58-73 The for loop solves the first stage of the algorithm row by row.
   (c) 75-77 Update of the matrices $\mathbf{C}, \mathbf{V}, \mathbf{W}$ using the intermediate result from the previous step.
   (d) 80-97 The for loop solves the first stage of the algorithm column by column.
   (e) 99-117 The error is computed and various stats are printed.
4. 120-124 The result from the last computation is plotted along with the solution error.

```
1  % UNIVERSIDAD CENTRAL DE VENEZUELA
2  % POSTGRADO EN CIENCIAS DE LA COMPUTACION
3  % TUTOR: Dr. OTILIO ROJAS
4  % ESTUDIANTE: LUIS JOAQUIN CORDOVA DIAZ
5  %%% Compacto_Mimetico_2D.m
6
7  tfin = 5/sqrt(2);
8  epsilon = 1e-5;
9  dospi = 2*pi;
10 casos = [8 16 33 48 64 96 128 320 335];
11 ss = length(casos);
12 cflmax=0.815;
13 lamb=.25;
14 for caso = 1:ss
15      N = casos(caso);
16      h = 1/(N-1);
17      dt=cflmax*h;
18      niter = round(tfin/dt);
19      dts2 = .5*dt;
20      dts2_lamb2 = dts2*lamb*lamb;
```

```

    unoss=ones(N,1);
    G=spdiags([-unoss unoss],[0 1],N,N+1);
    G(1,1:3)=[-8/3 3 -1/3];
    G(N,N-1:N+1)=[1/3 -3 8/3];

    RG=matriz_RG(N-1);
    M1=RG*G/h;

    unos=ones(N-1,1);
    D = spdiags([-unos unos],[0 1],N-1,N);

    RD = matriz_RD(N-1);
    M = RD*D/h;

    xu=[0 linspace(h/2,1-h/2,N-1) 1];
    yu=xu';
    xv=linspace(0,1,N);
    yw=xv';

    U = sin(dospi/lamb*yu)*sin(dospi/lamb*xu);
    Utilde=U;
    % SOLUCION EXACTA DE LA U
    Uexacta = U*cos(sqrt(2)*dospi*tfin);
    V=zeros(N+1,N);
    W=zeros(N,N+1);
    s=-(1/lamb/sqrt(2))*sin(sqrt(2)*dospi*tfin)/sqrt(2);
    Vexacta=s*sin(dospi/lamb*yu)*cos(dospi/lamb*xv);
    Wexacta=s*cos(dospi/lamb*yw)*sin(dospi/lamb*xu);
    tstart = cputime;
    for n = 1:niter
        A = M*W(:,2:N);
        Vtilde = V;
        Wtilde = W;
        Wtilde(:,2:N) = W(:,2:N)-dts2*M1*U(:,2:N);
        % RESOLUCION DE LA PRIMERA ETAPA FILA POR FILA
        for i = 2:N
            ai = U(i,2:N)-dts2_lamb2*A(i-1,:);
            bi = V(i,:);
            uviejo = U(i,2:N);
            vviejo = V(i,:);
            test = 1; k = 0;
            while (test > epsilon) && (k < 12)
                unuevo = ai-dts2_lamb2*vviejo*M';
                vnuevo = bi-dts2*[0 unuevo 0]*M1';
                test = norm([unuevo-uviejo vnuevo-vviejo]);
                uviejo = unuevo;
                vviejo = vnuevo;
                k = k+1;
            end
            Utilde(i,2:N) = unuevo;
            Vtilde(i,:) = vnuevo;
        end       % FIN DE LA PRIMERA ETAPA

        C = Vtilde(2:N,:)*M';
        V = Vtilde;
        W = Wtilde;
        % RESOLUCION DE LA SEGUNDA ETAPA COLUMNA POR COLUMNA
        V(2:N,:) = Vtilde(2:N,:)-dts2*Utilde(2:N,:)*M1';
        for j = 2:N
            cj = Utilde(2:N,j)-dts2_lamb2*C(:,j-1);
            dj = Wtilde(:,j);
            uviejo = Utilde(2:N,j); wviejo = Wtilde(:,j); test = 1; k = 0;
            while (test > epsilon) && (k < 12)
                unuevo = cj-dts2_lamb2*M*wviejo;
                wnuevo = dj-dts2*M1*[0;unuevo;0];
                test = norm([unuevo-uviejo;wnuevo-wviejo]);
                uviejo = unuevo;
                wviejo = wnuevo;
                k = k+1;
            end
            U(2:N,j)=unuevo;
            W(:,j)=wviejo;
        end
        E = U-Uexacta; errornuevo = h*norm(E,'fro');
        % FIN DE LA SEGUNDA ETAPA
    end

    telapsed = cputime-tstart;
    if caso == 1
        disp(['------------------------------------------------',...
            '--------------------------'])
        disp(['Compact_mimetic␣ORDEN␣␣OBSERVADO␣␣CUANDO␣␣',...
            'cfl_max=0.815...␣errores␣para␣',...
            'U--->tfin=7/sqrt(2)--->epsilon=1e-5'])
        disp(['-------------------------------------------------------',...
            '------------------------'])
        disp(['␣␣␣␣␣␣␣␣␣␣␣␣N␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣h␣␣␣␣␣␣␣␣␣␣␣␣␣␣',...
            '␣␣␣error␣␣␣␣␣␣␣␣␣␣␣␣␣orden␣␣␣␣␣␣␣Tiempo'])
        fprintf('%16.1f␣%16.3f␣%19.2e␣␣␣␣␣␣␣␣␣----------%11.3e␣␣\n',...
                N,h,errornuevo,telapsed)
    else
```

```
            orden = log2(errorviejo/errornuevo);
            fprintf('%16.1f %16.3f %19.2e %15.3f   %11.3e\n', ....
                        N,h,errornuevo,orden,telapsed)
        end
        errorviejo = errornuevo;
end

figure, surf(xu,yu,U), xlabel x, ylabel y, zlabel U(xu,yu)
title('ONDA  CALCULADA U')

figure, surf(xu,yu,E), xlabel x, ylabel y, zlabel E(x,y)
title ('ERROR sobre U')
```

B.3.1 matriz_RG.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Joaquin Cordova (Nov - 2013); Otilio Rojas (Nov 16 - 2013)
%    Building the Mimetic RG and RD: 4-4 Order
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    N: Number of intervals (it should be at least 10 ) ...

function [RG] = matriz_RG(N)
    %%%%%%%%%%%%%% MATRIZ CONVERGENCIA
    %%%%%%%%%%%%%% RG:  G_4th_order = RG*G_2nd_order
    e=ones(N+1,1);
    RG = spdiags([-1/24*e 13/12*e -1/24*e],-1:1,N+1,N+1);
    RG(1,1:5) = [17958/14245 -8776/14245 154787/341880 ...
                  -3415/34188 25/9768];
    RG(2,1:4) = [-2/35 941/840 -29/420 1/168];
    RG(N+1,N-3:N+1) = fliplr(RG(1,1:5));
    RG(N,N-2:N+1) = fliplr(RG(2,1:4));
    full(RG);
```

B.3.2 matriz_RD.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Joaquin Cordova (Nov - 2013); Otilio Rojas (Nov 16 - 2013)
%    Building the Mimetic RG and RD: 4-4 Order
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    N: Number of intervals (it should be at least 10 ) ...

function [RD] = matriz_RD(N)
%%%%%%%%%%%%%%%% MATRIZ DIVERGENCIA RD:
%%%%%%%%%%%%%%%% D_4th_order = RD*D_2nd_order
    e=ones(N,1);
    RD = spdiags([-1/24*e 13/12*e -1/24*e],-1:1,N,N);
    RD(1,1) = 4751/5192;
    RD(1,2)  = RD(1,1)-909/1298;
    RD(1,3)  = RD(1,2)-6091/15576;
    RD(1,4)  = RD(1,3)+ 1165/5192;
    RD(1,5)  = RD(1,4)-129/2596;
    RD(1,6)  = RD(1,5)+25/15576;
    RD(1,6)  = 0.;
    %%% Component RD(1,6) is practically 0.
    RD(N,N-5:N) = fliplr(RD(1,1:6));
```

Appendix C

CUDA kernels

C.1 Basic matrix operations

```
/* Compute sin(x)
** ld: stride(x)
** offset: offset(x)
** n: rows(x)
** m: cols(x)
*/
template<class T>
__global__  void sin(T *x, int ld, int offset, int n, int m)
{
        int c =  blockDim.x*blockIdx.x + threadIdx.x;
        int r =  blockDim.y*blockIdx.y + threadIdx.y;

        if(r<n && c<m)
                x[offset+r+(c*ld)]=::sin(x[offset+r+(c*ld)]);
}

/* Compute cos(x)
** ld: stride(x)
** offset: offset(x)
** n: rows(x)
** m: cols(x)   */
template<class T>
__global__  void cos(T *x, int ld, int offset, int n, int m){
        int c =  blockDim.x*blockIdx.x + threadIdx.x;
        int r =  blockDim.y*blockIdx.y + threadIdx.y;

        if(r<n && c<m)
                x[offset+r+(c*ld)]=::cos(x[offset+r+(c*ld)]);
}

/* Fill matrix x with stride:ld, offset:offset and size :n x m
** with value:value*/
template<class T>
__global__ void fill(T *x, int ld, int offset, int n, int m, T value){
    int c =  blockDim.x*blockIdx.x + threadIdx.x;
    int r =  blockDim.y*blockIdx.y + threadIdx.y;

    if(r<n && c<m)
        x[offset+r+(c*ld)]=value;
}

/* Fill diag diagonal of a matrix x with stride:ld, offset:offset and
** size:n x m with value :value        */
template<class T>
__global__ void dfill(T *x, int ld, int offset, int n, int m,
                      T value, int diag){
    int c =  blockDim.x*blockIdx.x + threadIdx.x;
    int r =  blockDim.y*blockIdx.y + threadIdx.y;

    if(r<n && c<m && (diag==(c-r)))
        x[offset+r+(c*ld)]=value;
}

/* Copy matrix y(stride:ldy,offset:offset_y,rows:ny,cols:my) over
**          matrix x(stride:ldx,offset:offset_x,rows:nx,cols:mx)
** without transposition */
template<class T>
__global__ void copy_nn(T *x, int ldx, int offset_x, int nx, int mx,
                        const T *y, int ldy, int offset_y){
        int cx = blockDim.x*blockIdx.x + threadIdx.x;
        int rx = blockDim.y*blockIdx.y + threadIdx.y;

    if(rx<nx && cx<mx)
        x[offset_x+rx+(cx*ldx)]=y[offset_y+rx+(cx*ldy)];
}

/* Copy matrix y(stride:ldy,offset:offset_y,rows:ny,cols:my) over
**        matrix x(stride:ldx,offset:offset_x,rows:nx,cols:mx)
** with transposition */
template<class T>
__global__ void copy_tn(T *x, int ldx, int offset_x, int nx, int mx,
                        const T *y, int ldy, int offset_y ){
        int cx = blockDim.x*blockIdx.x + threadIdx.x;
        int rx = blockDim.y*blockIdx.y + threadIdx.y;

    if(rx<nx && cx<mx)
        x[offset_x+rx+(cx*ldx)]=y[offset_y+cx+(rx*ldy)];
}

// x = alpha * x (stride:ld, offset:offset,rows:n, cols:m)
template<class T>
```

```
__global__ void scale(T *x, int ld, int offset,
                      int n, int m, T alpha){
     int c =  blockDim.x*blockIdx.x + threadIdx.x;
     int r =  blockDim.y*blockIdx.y + threadIdx.y;

     if(r<n && c<m)
          x[offset+r+(c*ld)]*=alpha;
}

/* Construct a vector
** x(stride:ld,offset:offset,rows:n,cols:m)=start:inc:end */
template<class T>
__global__ void linspace(T *x, int ld, int offset,
                          int n, int m, T start,T inc){
     int c =  blockDim.x*blockIdx.x + threadIdx.x;
     int r =  blockDim.y*blockIdx.y + threadIdx.y;

     if(r<n && c<m){
          int pos = c + r*m;
          T val=start+inc*pos;
          x[offset+r+(c*ld)]=val;
     }
}
```

C.2 Reduction

```
// Compute g_odata[block.x]= sum(s_data[n]) n=0...sz-1
// blockDim.x must be power of two
template<class T> __forceinline__ __device__
void reduce(int i,T *sdata, T *g_odata, int sz, int dim){
          // each thread loads one element from global to shared mem
          //unsigned int tid =     threadIdx.x;

          // do reduction in shared mem
          for (unsigned int s=dim/2; s>0; s>>=1) {
                    if (i< s ) {
                              sdata[i] += sdata[i + s];
                    }
                    __syncthreads();
          }

          // write result for this block to global mem
          if(threadIdx.x == 0)
                    g_odata[blockIdx.x] = sdata[0];
}

// Compute g_odata[block.x]=||g_idata1-g_idata2||
// sz: vector size
template<class T>
__global__ void dist(T *g_idata1, T *g_idata2,T *g_odata,
                        int szx,int szy){
          extern __shared__ int   shared_data[];

          /* This conversion is needed to avoid redeclaration as
          ** different type of shared object */
          T* sdata=(T*)shared_data;
          T val1,val2=0;
          // each thread loads one element from global to shared mem
          unsigned int i = blockIdx.x*(szy) + threadIdx.x;

          // Blocks = Columns = szx
          // Threads = Rows = szy

          val1 = (g_idata1[i]-g_idata2[i]);
          int next=i+blockDim.x;
          if(threadIdx.x+blockDim.x<szy){
                    val2=(g_idata1[next]-g_idata2[next]);
          }
          sdata[threadIdx.x]=val1*val1+val2*val2;

          __syncthreads();
          reduce(threadIdx.x,sdata,g_odata,szx,blockDim.x);

          __syncthreads();
          // write result for this block to global mem
          if(threadIdx.x == 0)
                    g_odata[blockIdx.x] = sqrt(g_odata[blockIdx.x]);
}
```

C.3 Banded matrix multiplication

```
/* Perform transposed banded-matrix * dense-matrix multiply
** y := alpha*A*x + beta*y      trans:=0
** y := alpha*A*x + beta*y      trans!=0
** A = m x n     (kl,ku,lda)
** X = n x o     (ldx,incx)
** Y = m x o     (ldy,incy)
*/
template<typename T>
void __global__  algMM(int trans, int m, int n, int o, int kl, int ku,
                          T alpha,T* A, int lda, T* X, int ldx, int incx,
                          T beta, T* Y, int ldy, int incy){

        /* Get linear (i,j) indexes */
```

```
int i_y = blockIdx.y*blockDim.y + threadIdx.y;
int j_y = blockIdx.x*blockDim.x + threadIdx.x;

// Filter out of matrix threads
int idx_a,idx_x,d,step,bw;
if(!trans){
        if(i_y>=m || j_y>=o)
                return;
        int i_a,j_a_min,j_a_max;
        i_a=i_y;
        /* Limit columns by non-zero diagonals*/
        j_a_min=i_a-kl;
        j_a_max=i_a+ku;

        /* Limit columns by matrix size*/
        if(j_a_min<0)
                j_a_min=0;
        if(j_a_max>= n)
                j_a_max = n-1;

        bw=j_a_max-j_a_min;

        d = j_a_min-i_a;

        idx_a = (ku-d) + j_a_min*lda;
        step=lda-1;


        idx_x = j_a_min*incx + j_y*ldx;
}
else{
        if(i_y>=n || j_y>=o)
                return;

        int j_a,i_a_min,i_a_max;
        j_a = i_y;
        i_a_min=j_a-ku;
        i_a_max=j_a+kl;

        if(i_a_min<0)
                i_a_min=0;
        if(i_a_max>= m)
                i_a_max = m-1;

        bw=i_a_max-i_a_min;
```

```
                d = j_a-i_a_min;

                idx_a = (ku-d) + j_a*lda;
                step=1;

                idx_x = i_a_min*incx + j_y*ldx;
        }


        /* Perform element-wise multiplication*/
        T ret=0;

        T* a=&A[idx_a];
        T* x=&X[idx_x];

        // Tridiagonal matrix optimization
        #pragma unroll 3
        for(;bw>=0;bw--){
                ret += (*a)*(*x);
                a+=step;
                x+=incx;
        }

        int idx_y= i_y * incy + j_y*ldy;
        /* Perform multiply-add result update*/
        if(beta!=0)
                Y[idx_y]=alpha * ret + beta * Y[idx_y];
        else
                Y[idx_y]=alpha * ret;
}
```

C.4 Thomas algorithm

```
/* Parallel tridiagonal systems solver (AX=B) by Thomas Algorithm
**  N:   order of matrix A
**  NRHS: number of right hand sides (columns of B)
**      DU: A upper diagonal
**      D:  A main diagonal
**      DL: A lower diagonal
**  B: Dense matrix
**  X: Solution
** No partial-pivoting is performed so stability is only guaranteed
** when matrix is diagonally dominant or symmetric positive definite
*/
template<typename T>
void __global__ algTSM (int N, int NRHS, T* A, int tr_a,
                        T* B, int ldb, int incb, T* X,T* tmp){
```

```
/* Each thread requires N*sizeof(T)
shared memory for internal storage*/
/* One thread per right hand side*/
int rhs = blockIdx.x*blockDim.x + threadIdx.x;

if(rhs>=NRHS)
        return;

// c'[n-1], d'[n-1]
T* c_prime=&tmp[rhs*N];
T cp,dp;

//Friendlier names for input variables
T* x=&X[idx_dense(0,rhs,N,1)];
T* b=&A[1];
T* a=&A[-1];
T* c=&A[3];
if(tr_a){
        a=&A[0];
        c=&A[2];
}
T* d=&B[idx_dense(0,rhs,ldb,incb)];

/* Forward pass */
// Compute c'[i] and d'[i]
cp = c[0]/b[0];
dp = d[0]/b[0];
// c'[i-1] <- c'[i] / d'[i-1] <- d'[i]
c_prime[0]=cp;
x[0] = dp;
for(int i=1;i<N-1;i++){
        // Avanzar punteros
        a+=3; b+=3; c+=3; d+=incb;
        T denom = (*b)-((*a)*cp);
        // d'[i] = (d[i] - d'[i-1]*a[i])/(b[i] - c'[i-1]*a[i])
        dp = ((*d)-((*a)*dp))/denom;
        // c'[i] = c[i] / (b[i] - c'[i-1]*a[i])
        cp = (*c)/denom;
        // Store c'[i] and d'[i]
        c_prime[i]=cp;
        x[i] = dp;
        // c'[i-1] <- c'[i] / d'[i-1] <- d'[i]
        //cp_n_1=cp;
        //dp_n_1=dp;
}
```

```
            a+=3; b+=3; c+=3; d+=incb;
            T denom = (*b)-((*a)*cp);
            // d'[i] = (d[i] - d'[i-1]*a[i])/ (b[i] - c'[i-1]*a[i])
            dp = ((*d)-((*a)*dp))/denom;
/*          // c'[i] = c[i] / (b[i] - c'[i-1]*a[i])
            cp = (*c)/denom;
            // Store c'[i] and d'[i]
            c_prime[i]=cp;*/
            x[N-1] = dp;

            /* Back substitution*/
            //x[N-1] = d_p; // Implicit as d is temporally stored in x
            for(int i=N-2;i>=0;i--){
                    // x[i] = d'[i] - c'[i]*x[i+1]
                    x[i] -= c_prime[i]*x[i+1];
            }
}
```

References

[1] L. Córdova, O. Rojas, B. Otero, and J. Castillo, “Compact finite difference modeling of 2-d acoustic wave propagation,” *Journal of Computational and Applied Mathematics*, vol. 295, pp. 83 – 91, 2016. {VIII} Pan-American Workshop in Applied and Computational Mathematics.

[2] S. K. Lele, “Compact finite difference schemes with spectral-like resolution,” *Journal of Computational Physics*, vol. 103, no. 1, pp. 16 – 42, 1992.

[3] M. A. Dablain, “The application of high order differencing to the scalar wave equation,” *GEOPHYSICS*, vol. 51, no. 1, pp. 54–66, 1986.

[4] R. M. Alford, K. R. Kelly, and D. M. Boore, “Accuracy of finite-difference modeling of the acoustic wave equation,” *Geophysics*, vol. 39, no. 6, pp. 834–842, 1974.

[5] J. Virieux, “Sh-wave propagation in heterogeneous media; velocity-stress finite-difference method,” *Geophysics*, vol. 49, no. 11, pp. 1933–1942, 1984.

[6] J. T. Etgen and M. J. O’Brien, “Computational methods for large-scale 3d acoustic finite-difference modeling: A tutorial,” *Geophysics*, vol. 72, no. 5, pp. SM223–SM230, 2007.

[7] L. Di Bartolo, C. Dors, and W. J. Mansur, “A new family of finite-difference schemes to solve the heterogeneous acoustic wave equation,” *Geophysics*, vol. 77, no. 5, pp. T187–T199, 2012.

[8] S.-L. Chang and Y. Liu, “A truncated implicit high-order finite-difference scheme combined with boundary conditions,” *Applied Geophysics*, vol. 10, no. 1, pp. 53–62, 2013.

[9] C. A. Pérez Solano, D. Donno, and H. Chauris, “Finite-difference strategy for elastic wave modelling on curved staggered grids,” *Computational Geosciences*, vol. 20, no. 1, pp. 245–264, 2016.

[10] C. Gelis, D. Leparoux, J. Virieux, A. Bitri, S. Operto, and G. Grandjean, “Numerical modeling of surface waves over shallow cavities,” *Journal of Environmental and Engineering Geophysics*, vol. 10, no. 2, pp. 111–121, 2005.

[11] M. Abouali and J. E. Castillo, “High-order compact castillo-grone’ s mimetic operators,” techreport, Computational Sciences Research Center, 2012.

[12] J. E. Castillo and R. D. Grone, “A matrix analysis approach to higher-order approximations for divergence and gradients satisfying a global conservation law,” *SIAM Journal on Matrix Analysis and Applications*, vol. 25, no. 1, pp. 128–142, 2003.

[13] C. Grossmann, H.-G. Roos, and M. Stynes, *Numerical treatment of partial differential equations.* Universitext, Berlin, Heidelberg, New York: Springer, 2007.

[14] P. Roache, *Computational fluid dynamics.* Albuquerque, N.M: Hermosa Publishers, 1976.

[15] R. Sheppard, *Pricing Equity Derivatives under Stochastic Volatility: A Partial Differential Equation Approach.* 2007.

[16] J. Crank and P. Nicolson, "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type," *Advances in Computational Mathematics*, vol. 6, no. 1, pp. 207–226, 1996.

[17] D. W. Peaceman and J. H. H. Rachford, "The numerical solution of parabolic and elliptic differential equations," *Journal of the Society for Industrial and Applied Mathematics*, vol. 3, no. 1, pp. 28–41, 1955.

[18] H. D. Vries, "A comparative study of {ADI} splitting methods for parabolic equations in two space dimensions," *Journal of Computational and Applied Mathematics*, vol. 10, no. 2, pp. 179 – 193, 1984.

[19] J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations, Second Edition.* Society for Industrial and Applied Mathematics, 2004.

[20] J. Castillo, J. Hyman, M. Shashkov, and S. Steinberg, "Fourth- and sixth-order conservative finite difference approximations of the divergence and gradient," *Applied Numerical Mathematics*, vol. 37, no. 1–2, pp. 171 – 187, 2001.

[21] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, pp. 308–323, Sept. 1979.

[22] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[23] Z. Y. Zhang Xianyi, Wang Qian, "Model-driven level 3 blas performance optimization on loongson 3a processor," *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, dec 2012.

[24] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide.* Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[26] nVidia, *CUBLAS Library User Guide.* nVidia, v5.0 ed., Oct. 2012.

[27] J. W. Eaton *et al.*, "Gnu octave."

[28] J. O. Coplien, "Curiously recurring template patterns," *C++ Rep.*, vol. 7, pp. 24–27, Feb. 1995.

[29] NVIDIA, "Nvidia's next generation cuda compute architecture: Fermi."

[30] NVIDIA, "Nvidia geforce gtx 980: Featuring maxwell, the most advanced gpu ever made."

[31] M. Harris, "Optimizing parallel reduction in cuda," 2014.

[32] C. De Boor, *Elementary numerical analysis :*. New York :: McGraw-Hill,, 2nd ed. / ed., 1972. Title page imprint: London.

[33] S. Skiena, *The Algorithm Design Manual: Software*. The Algorithm Design Manual, TELOS–the Electronic Library of Science, 1998.

[34] D. Coppersmith and S. Winograd, "Computational algebraic complexity editorial matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251 – 280, 1990.

List of Figures