University of Minho
School of Engineering
Informatics Department

Filipe Pereira da Silva

Parallelization of the ADI Method Exploring Vector Computing in GPUs

November 2021

University of Minho
School of Engineering
Informatics Department

Filipe Pereira da Silva

Parallelization of the ADI Method Exploring Vector Computing in GPUs

Master dissertation
Master Degree in Integrated Master's in Informatics Engineering

Dissertation supervised by
Alberto José Proença
Diogo Alberto Rocha Lopes

November 2021

ACKNOWLEDGEMENTS

I want to thank everyone who helped me with this dissertation!

First, I want to thank Raquel Moreira for being my pillar on this journey, without you, I could not finish this, I might as well have given up, but you were always there to help me with my struggles, so I want to give you a HUGE "Thank You".

I want to thank my mom and dad, without them, I would have no way to study outside my little island, even with all the times I missed you, you were always there for me, and I want to thank you for that, for believing in me and never giving up.

I want to thank my brother, I knew I could always call you whenever I was bored or stressed and you would help me forget my problems.

I want to thank my friends, we spend quality time together in the past years, all the stories, the memories and everything we went through together. You guys were the best, and I could not ask for anything else from you.

I want to thank Diogo Lopes, my co-supervisor for being present whenever I needed and with all the help gave me. I also want to thank my professor Alberto Proença for all the help and ideas you gave to me, without your help I would most definitely not test as much and would not find as many errors as I did.

Furthermore, I want to thank my roommates for all the time I spent asking you for help.

Not only that, but I want to thank FCT for believing in me with a grant for this project, without it, I maybe would have not chosen it, and I had so much fun learning new things, learning more and more about GPU computing and learning new things about vector computing I did not even know were possible to do.

STATEMENT OF INTEGRITY

ABSTRACT

The 2D convection-diffusion is a well-known problem in scientific simulation that often uses a direct method to solve a system of N linear equations, which requires N^3 operations.

This problem can be solved using a more efficient computational method, known as the alternating direction implicit (ADI). It solves a system of N linear equations in 2N times with N operations each, implemented in two steps, one to solve row by row, the other column by column. Each N operation is fully independent in each step, which opens an opportunity to an embarrassingly parallel solution. This method also explores the way matrices are stored in computer memory, either in row-major or column-major, by splitting each iteration in two.

The major bottleneck of this method is solving the system of linear equations. These systems of linear equations can be described as tridiagonal matrices since the elements are always stored on the three main diagonals of the matrices. Algorithms tailored for tridiagonal matrices, can significantly improve the performance. These can be sequential (*i.e.* the Thomas algorithm) or parallel (*i.e.* the cyclic reduction CR, and the parallel cyclic reduction PCR).

Current vector extensions in conventional scalar processing units, such as x86-64 and ARM devices, require the vector elements to be in contiguous memory locations to avoid performance penalties. To overcome these limitations in dot products several approaches are proposed and evaluated in this work, both in general-purpose processing units and in specific accelerators, namely NVidia GPUs.

Profiling the code execution on a server based on x86-64 devices showed that the ADI method needs a combination of CPU computation power and memory transfer speed. This is best showed on a server based on the Intel manycore device, KNL, where the algorithm scales until the memory bandwidth is no longer enough to feed all 64 computing cores. A dual-socket server based on 16-core Xeon Skylakes, with AVX-512 vector support, proved to be a better choice: the algorithm executes in less time and scales better.

The introduction of GPU computing to further improve the execution performance (and also using other optimisation techniques, namely a different thread scheme and shared memory to speed up the process) showed better results for larger grid sizes (above 32Ki x 32Ki). The CUDA development environment also showed a better performance than using OpenCL, in most cases. The largest difference was using a hybrid CR-PCR, where the OpenCL code displayed a major performance improvement when compared to CUDA. But even with this speedup, the better average time for the ADI method on all tested configurations on a NVidia GPU was using CUDA on an available updated GPU (with a Pascal architecture) and the CR as the auxiliary method.

Keywords: Master thesis, GPU Computing, Physics, HPC, Mathematics

RESUMO

O problema da convecção-difusão é utilizado em simulações cientificas que regularmente utilizam métodos diretos para solucionar um sistema de N equações lineares e necessitam de N^3 operações.

O problema pode ser resolvido utilizando um método computacionalmente mais eficiente para resolver um sistema de N equações lineares com N operações cada, implementado em dois passos, um solucionando linha a linha e outro solucionando coluna a coluna. Cada par de N operações são independentes em cada passo, havendo assim uma oportunidade de utilizar uma solução embaraçosamente paralela. Este método também explora o modo de guardar as matrizes na memória do computados, sendo esta por linhas ou em colunas, dividindo cada iteração em duas, este método é conhecido como o método de direção alternada.

O maior *bottleneck* deste problema é a resolução dos sistemas de equações lineares criados pelo ADI. Estes sistemas podem ser descritos como matrizes tridiagonais, visto todos os seus elementos se encontrarem nas 3 diagonais interiores e a utilização de métodos estudados para este caso é necessário para conseguir atingir a melhor *performance* possível. Esses métodos podem ser sequenciais (como o algoritmo de Thomas) ou paralelos (como o CR e o PCR)

As extensões vectoriais utilizadas nas atuais unidades de processamento, como dispositivos x86-64 e ARM, necessitam que os elementos do vetor estejam em blocos de memória contíguos para não sofrer penalizações. Algumas abordagens foram estudadas neste trabalho para as ultrapassar, tanto em processadores convencionais como em aceleradores de computação. Os registos do tempo em servidores baseado em dispositivos x86-64 mostram que o ADI necessitam de uma combinação de poder de processamento assim como velocidade de transferência de dados. Isto é demonstrado especialmente no servidor baseado no dispositivo KNL da Intel, no qual o algoritmo escala até que a largura de banda deixe de ser suficiente para o problema. Um servidor com dois *sockets* em que cada é composto por um dispositivo com 16 *cores* baseado na arquitetura Xeon Skylake, com acesso ao AVX-512, mostrou ser a melhor escolha: o algoritmo faz as mesmas operações em menos tempo e escala melhor.

Com a introdução de computação com GPUs para melhorar a *performance* do programa mostrou melhores resultados para problemas de maiores dimensões (tamanho acima de 32Ki x 32Ki celulas). O desenvolvimento em CUDA também mostrou melhores resultados que em OpenCL na maioria dos casos. A maior divergência foi observada ao utilizar o método CR-PCR, onde o OpenCL mostrou melhor *performance* que em CUDA. Mas mesmo com este método sendo mais eficaz que o mesmo em CUDA, o melhor *performance* com o método ADI foi observado utilizando CUDA no GPU mais recente estudado com o método CR.

Palavras-chave: Dissertação de mestrado, Computação em GPU, Física, HPC, Matemática

CONTENTS

ACRONYMS

A

ADI Alternating Direction Implicit. 4–6, 8, 10, 19–21, 24, 29–32, 34, 35, 37–40, 45, 47, 50–52, 56, 58, 59, 61, 62

AMD Advanced Micro Devices. 25

AMX Advanced Matrix Extensions. 32

AOS Arrays of Structures. 24

API Aplication Programming Interface. 5, 21, 22, 26, 27, 34, 38, 44, 55, 58, 61

AVX Advanced Vector Extensions. 25, 37, 38, 47, 49, 51, 52, 57–59

C

CPU Central Processing Unit. 5, 9, 15, 21, 23, 24, 26–29, 31, 34, 38–40, 44, 45, 47–49, 52, 53, 57–62

CR Cyclic Reduction. 11–13, 17–20, 40–44, 53–61

CUDA Compute Unified Device Architecture. 5, 21, 26, 28, 40, 44, 47, 53–59, 61

G

GPGPU General-Purpose Computing in Graphics Processing Units. 27

GPU Graphics Processing Unit. 5, 9, 14, 15, 18, 21, 24, 26–31, 38–40, 44, 45, 47, 48, 52–62, 68

H

HBM High Bandwidth Memory. 50, 58

HPC High Performance Computing. 27

I

ISA Instruction Set Architecture. 28

K

KNL Knigths Landing. 5, 47, 50, 51, 58, 62

L

LU Lower Upper. 8, 9, 11

N

NUMA Non-Uniform Memory Access. 5, 24, 34, 51, 52

O	
OPENCL	Open Computing Language. 5, 21, 26, 28, 31, 40, 44, 47, 55–58, 61
P	
PCR	Parallel Cyclic Reduction. 11, 13, 14, 17, 18, 20, 40, 42–44, 53–58, 61
PTX	Parallel Thread Execution. 28
PU	Processing Unit. 23–26, 31
R	
RD	Recursive Doubling. 15, 18–20, 40
S	
SIMD	Single Instruction Multiple Data. 24–26, 28, 49, 58, 59, 61
SIMT	Single Instruction Multiple Threads. 28, 30, 59
SM	Streaming Multiprocessor. 26, 28, 40, 43, 45, 53, 56, 59
SMT	Simultaneous MultiThreading. 49, 51, 52, 59
SOA	Structures of Arrays. 24
SOC	System on Chip. 32
SSE	Streaming SIMD Extensions. 25, 51
U	
UM	Universidade do Minho. 47
V	
VPU	Vectorial Processing Unit. 25

Part I

INTRODUCTORY MATERIAL

1

INTRODUCTION

The main objective of this dissertation is to develop a method to efficiently solve a system of linear equations that describe the convection-diffusion equations on a given domain. These can be translated into heat flow or shallow water simulations. These systems of linear equations are commonly used to solve problems such as petroleum reservoir simulation, subsurface contaminant remediation, among others [Saif and Al-Saadawi (2011)].

The convection-diffusion equations are parabolic partial equations that combine the convection equation and the diffusion equation [Strang (2006)]. Since solving these equations directly can be computationally time-consuming, a better approach is to use an iterative method. This method will not produce exact results, but it will repeatedly work on improving an approximation to the solution.

Among several competitive well-known methods to computationally solve these equations, the *Alternating Direction Implicit (ADI)* method seemed the most adequate to lead to a faster solution. When compared to some other known methods it shows better accuracy and it is better suited for parallel environments. It can also be applied to all domains without requiring specific conditions [Islam (2010)].

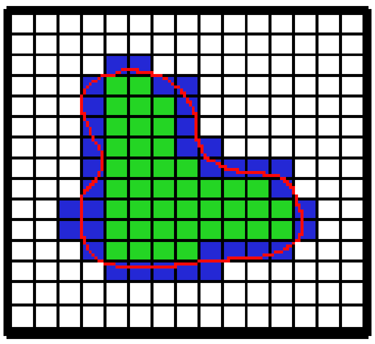
Figure 1: **Example of a 2D domain.**

The (ir)regular domain in this work is described through a structured n-dimensional mesh, always padded with at least 1 extra empty row/column next to the border, to simplify the algorithm. Figure 1 shows a simple 2D example, clearly identifying the border cells of the domain that will need special treatment and padded with two extra rows and columns in the mesh border.

The advantage of the ADI method to solve the convection-diffusion equations is that the equations that have to be solved in each step have a simpler structure and can be efficiently solved with a tridiagonal matrix solver.

This approach splits the 2D grid into simpler 1D lines to minimise the data dependency between iterations. First, it resolves the heat dispersion through rows and then columns, one at a time. An attempt to develop efficient parallel code was already pursuit [Lopes et al. (2019)], for shared and distributed memory parallel systems based on multicore *Central Processing Unit (CPU)* devices, but few attempts were yet tried on computing accelerators, such as *Graphics Processing Unit (GPU)* devices [Wei et al. (2013)].

Theoretically, these techniques can also be expanded to a 3D environment. It would need more iterations since instead of subdividing the problem into various 1D problems it would be necessary to first divide the domain into 2D subproblems and these into 1D subproblems.

1.1 CHALLENGES AND GOALS

Most experimental work on evaluation was on servers, nodes in a cluster system, based on Intel x86-64 CPU devices, while only a few of them took advantage of additional computing accelerators, namely GPU devices. A GPU algorithm was developed based on the sequential version of the code, with both *Compute Unified Device Architecture (CUDA)* and *Open Computing Language (OpenCL)* versions, stressing the difference between these two approaches. The code in itself does not differ much, so studying how to port CUDA code to OpenCL code was pertinent. The performance of the algorithms was measured and compared on different architectures, using the best practices tailored for each.

The original execution performance was compared to an efficient parallel version of the code on a server based on Intel manycore *Knigths Landing (KNL)* and a dual-socket server based on Intel multicore Skylake devices. The study took into consideration the *Non-Uniform Memory Access (NUMA)* architecture on the dual-socket servers and tests used different parallel configurations for a more efficient method.

1.2 DISSERTATION OUTLINE

The current chapter identified the key challenges and goals of the dissertation work. The next chapter explains why the ADI method was chosen for this specific problem, how the domain is interpreted and what tridiagonal methods were chosen. The third chapter is dedicated to the computational efficiency issues, stressing the vector extensions and how it works on both CPU and GPU devices, identifying potential bottlenecks. The fourth and fifth chapters are the core of this dissertation: the former describes the implementation of the ADI using C++ and using the CUDA and OpenCL *Aplication Programming Interface (API)* for the GPU devices; the latter discusses the comparative evaluation of their performance on different computing platforms, without and with computing accelerators. The sixth and last chapter presents some conclusions, giving suggestions for future work to enrich the obtained results.

2

THE ALTERNATING DIRECTION IMPLICIT METHOD

In this chapter the main problem will be studied and transformed into a numerical problem, being discretised as a system of linear equations. To solve this, various approaches to solve this problem are studied. Since the best approach for the problem is the ADI method [Islam (2010)], possible tridiagonal solvers will also be studied as they are used as auxiliary methods. At the end of the chapter, the various algorithms will be compared and some will be discarded due to constraints that some algorithms may have.

2.1 CONTEXT

The main problem of this dissertation is the simulation of the convection-diffusion equations in a computational domain. These equations can simulate real-world problems like heat transfer or shallow waters. These equations can be described as a system of linear equations.

To simulate these problems on a computer it is necessary to discretise the continuous domain into a subset of problems, these being measurable and quantified, which transforms the real-world problem into a computable problem.

When discretising the problem, two main approaches are possible. Dividing the problem into small triangles, which is one of the main approaches in computer graphics, filling the whole domain with a finite number of triangles, or using a grid, therefore dividing the problem into rows and columns.

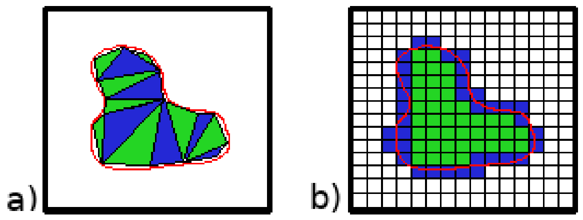

Figure 2: **Alternative ways to discretise a domain: a) With triangles; b) With squares.**

While using triangles the domain representation can be closer to reality, more calculations are needed per cell. This happens because the data is not structured and extra computations are required to verify the neighbouring triangles. While using a grid that is overlaid on top of the real problem, the simulated domain will become less likely to the real domain. This grid makes the memory management of the cells become trivial since they can be stored as a matrix, retaining the original order. Therefore finding the neighbouring cells is less expensive, since they are stored in the neighbouring indexes.

Using a grid to store the domain information has a major downside because since the domain is not regular, there might be a lot of empty cells. These empty cells still need to be processed and stored, since without them the grid stops having the neighbours indexed next to each other, therefore removing the best advantage of using a grid. This discretised grid, storing only the information of the epicentre of each cell.

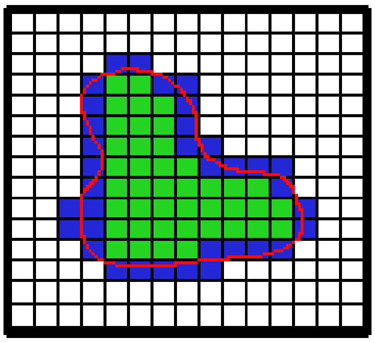

Figure 3: **Domain example: outer cells are white, boundary or ghost cells are blue and inner cells are green.**

Each discretisation method has some advantages and disadvantages. While using triangles have a more realistic approach, using a grid lets the usage of more efficient methods. It is also possible to make the grid more likely to the real counterpart by having more cells and more information of the whole domain at a time. Because of this, the discretisation method chosen is the grid. With this in mind, a study will be made to discover efficient methods based on this approach.

Due to the matrix being composed of rows and columns it is possible to describe the problem in each row/column with simple linear equations. These equations will simulate the convection-diffusion equations through all the cells from the frontier to the inner cells. These will be spread in rows/columns so it is possible to create a system with all the equations to be solved. Making a system of linear equations in this environment is easy since the matrix can be divided into linear equations, making the transformations between the matrix and the system of linear equations easy [Lopes et al. (2019)].

2.2 SOLVING SYSTEMS OF LINEAR EQUATIONS

Gaussian Elimination

A simple direct method to solve a system of linear equations is the Gaussian elimination [Gauss (1809)] [Robert (1990)] [Grcar (2011)].

This is one of the most used and most well-known method to find a direct solution to a system of equations in the form of $A\vec{x} = \vec{b}$ where A is a matrix and b is a known vector (from now on, this system of linear equations will be treated called as $Ax = Z$). First of all the algorithm starts by clearing the lower tridiagonal of the matrix A.

After eliminating the lower tridiagonal, the algorithm transforms the matrix into a diagonal matrix by also nullifying the upper diagonal of the matrix by using the lower lines to create zeros in the upper lines.

After having a diagonal matrix the algorithm simply divides both remaining elements of each line (the element from the matrix A's diagonal and the element in the vector $\vec{b}$).

This method is used as an auxiliary method to help solve problems like the computation of partial differential equations. It is also known to be a time-consuming method, so optimisations were studied [Zhou et al. (2018)].

Using the Gaussian elimination on the whole problem would be computationally intensive since the complexity of this algorithm with the Gaussian elimination method would be $O(N) = N^3$ [Strassen (1969)]. Other methods were studied [Islam (2010)], some of these being ADI method, Jacobi, and Gauss-Seidel method.

LU Algorithm

The *Lower Upper (LU)* decomposition is another algorithm used to solve systems of linear equations. This algorithm decomposes, as the name suggests, the system $Ax = z$ into $LUx = z$ by decomposition of the matrix A into the upper diagonal matrix U and the lower diagonal matrix L, and it is strictly sequential [Lindfield and Penny (2019)].

This algorithm has three steps: the decomposition of A into LU; solving the equation $Ly = z$, then solving the equation $LUx = Ly$ or $Ux = y$.

In the first step, the algorithm decomposes the matrix A into the matrices L and U using the following equations[1]:

$$A = \begin{vmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \dots & & \\ & & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & a_N & b_N \end{vmatrix} L = \begin{vmatrix} 1 & & & & \\ m_2 & 1 & & & \\ & m_3 & 1 & & \\ & & \dots & & \\ & & m_{N-1} & 1 & \\ & & & m_N & 1 \end{vmatrix} U = \begin{vmatrix} u_1 & c_1 & & & \\ & u_2 & c_2 & & \\ & & u_3 & c_3 & \\ & & \dots & & \\ & & & u_{N-1} & c_{N-1} \\ & & & & u_N \end{vmatrix}$$

$$\begin{cases} u_i = b_i & i = 1 \\ u_i = b_i - \frac{a_i * c_{i-1}}{u_{i-1}} & 2 \leq i \leq N \end{cases}$$
$$m_i = \frac{a_i}{u_{i-1}} 2 \leq i \leq N$$

After decomposing the matrix A, the algorithm solves $Ly = z$ using a simplified forward sweep. To solve this step the algorithm performs the following operations:

1 These example equations are used for a tridiagonal matrix [Ömer EGECIOGLU et al. (1990)], but the base algorithm can be used on regular matrices.

$$L = \begin{vmatrix} 1 & & & & & \\ m_2 & 1 & & & & \\ & m_3 & 1 & & & \\ & & ... & & & \\ & & m_{N-1} & 1 & & \\ & & & m_N & 1 & \end{vmatrix} y = [y_1, y_2, ..., y_{N-1}, y_N] z = [z_1, z_2, ..., z_{N-1}, z_N]$$

$$\begin{cases} y_i = z_i & i = 1 \\ y_i = z_i - m_i * y_{i-1} & 2 \leq i \leq N \end{cases}$$

After those two steps, the algorithm solves the vector $\vec{x}$ by doing the following equations in the system $Ux = y$:

$$U = \begin{vmatrix} u_1 & c_1 & & & & \\ & u_2 & c_2 & & & \\ & & u_3 & c_3 & & \\ & & ... & & & \\ & & & u_{N-1} & c_{N-1} & \\ & & & & u_N & \end{vmatrix} y = [y_1, y_2, ..., y_{N-1}, y_N] x = [x_1, x_2, ..., x_{N-1}, x_N]$$

$$\begin{cases} x_i = \frac{yi}{u_i} & i = N \\ x_i = \frac{y_i - x_{i+1} * c_i}{u_i} & i \neq N \end{cases}$$

The LU Decomposition can be used to solve regular matrices, but this algorithm is more difficult to parallelise, which would make it a bad choice to use on parallel environments like Manycore CPU and GPU architectures.

Jacobi and Gauss-Seidel methods

The Jacobi method is an iterative method used to solve systems of linear equations [Adsuara et al. (2016)] [Yang and Mittal (2014)].

First of all, an initial guess is given for the possible solution. The closer the guess is to the solution, the faster is the convergence. This method is considered finished when the difference between 2 iterations is smaller than a given threshold. The matrix A can be divided into 2 different matrices, one with the diagonal (D) and the other with the remaining values (R), such as $A = D + R$.

In every iteration the following equation is used to get a more precise solution, the variable x^k is the solution of iteration k, x_i^k is the solution of the element x_i on iteration k, variable z_i is the element i of the vector z, and variable a_{ij} is the element from the matrix A in the position (i, j):

$$x^{(k+1)} = D^{-1}(z - Rx^k)$$

$$x_i^{(k+1)} = \frac{1}{a_i}(z_i - \sum_{j \neq i} a_{ij} x_j^k), 1 \leq i \leq n$$

In every iteration, the termination criteria is evaluated to check that the method is converging. The method checks the difference between each iteration and if this difference is smaller than the given threshold, the solution is considered valid and the algorithm stops.

This algorithm does not update the values of $\vec{x}$ and needs to have at least 2 vectors saving the values of $\vec{x}$, one for the updated values and one for the values that will be updated this iteration.

The Jacobi algorithm also needs to make sure that the system is converging each iteration, and not all systems can be solved with this algorithm.

There is also another similar algorithm called Gauss-Seidel method [Yang (2018)]. This method differs from the Jacobi method because it uses updated values as soon as they are available. This little change makes it so that some systems of linear equations take fewer iterations to complete. This method also uses less memory space than the Jacobi method since it does not need to store the updated values and the non-updated values both at the same time.

The Gauss-Seidel, as the Jacobi method, can also not converge, so some systems can not be solved by it, and even if it takes fewer iterations to solve a system and takes less memory space, this method is strictly sequential [Islam (2010)].

2.2.1 *The ADI Method*

A potentially faster approach to solving the system of linear equations is using the ADI method [Islam (2010)]. This method partitions the grid into individual rows and columns, transforming a single large problem into various smaller independent problems.

Dividing the 2D grid into 1D lines, the algorithm becomes embarrassingly parallel because every line is independent from all others and that can be exploited by multi-threading environments [Wei et al. (2013)]. The ADI method splits each line/column and using its internal equations (explored below) creates a tridiagonal matrix and a solution vector. Each cell is deconstructed into 4 different variables (each fitting on each diagonal and one for the solution vector), these cells are deconstructed based on what type of cells they are. The three types of cells are:

- Inner cell - These cells are the unknown cells to be solved, they start with a pre-determined value and in every iteration, their value is updated.
- Boundary cell - These cells are the only truly known cells, their value never changes between iterations and will be the ones to determine the final solution. These are also known as ghost cells.
- Outer cell - These cells are also known as outside cells, these cells are used mostly for padding the grid and fitting the domain into it. Their values are nullified and do not contribute to the problem.

With this setup, an auxiliary method can be used to solve this system. The auxiliary methods used in these problems are tridiagonal solvers which are used to solve $Ax = z$.

This matrix and result vector are built using each cell in the line/column. The first row is composed of the information on the first cell, the second row on the second cell, and so on. The ADI method creates tridiagonal matrices from a single line/columns by following the next equations, once for every cell in the current line:

$$a_i = c_i = \begin{cases} \frac{ySize}{xSize}, & inner\ cell \\ 0, & if\ cell\ is\ not\ inner\ cell \end{cases}$$

$$b_i = \begin{cases} alpha + 2 * \frac{ySize}{xSize}, & inner\ cell \\ 1, & if\ cell\ is\ not\ inner\ cell \end{cases}$$

$$z_i = \begin{cases} x_{i-1} + x_{i+1} - 2 * x_i * Cell\ Area - \frac{5e^{Px+2Py} * xSize}{ySize} + alpha * x_i, & Inner\ Cell \\ 0, & Outter\ Cell \\ Cell\ Solution, & Ghost\ Cell \end{cases}$$

These formulas are used when computing the rows, The values of *Px* and *Py* are the position of the cell and the *xSize* and *ySize* are the width and height of the cell respectively. When computing the values for the columns, the *xSize* and *ySize* are switched due to the problem being transposed. The element *i* is the respective position in the current row/column, *x* is the solution of the last iteration and *alpha* is a predetermined value.

2.3 EFFICIENTLY SOLVING TRIDIAGONAL MATRICES

Using Gaussian Elimination is not efficient for solving tridiagonal matrices [Bottoni (1994)]. The Gaussian Elimination has a complexity of $O(N) = N^2$ and most of the computations done are unnecessary since most values of a tridiagonal matrix are zeros. Some better methods were developed for this specific matrices, namely the Thomas algorithm, *Cyclic Reduction (CR)*,*Parallel Cyclic Reduction (PCR)* between others [Zhang et al. (2010)].

2.3.1 *Thomas Algorithm*

The Thomas algorithm can replace the Gaussian elimination since it is optimised to only operate on a small part of the matrix. This is because it only needs to compute the values for the diagonals of the matrix. Thomas algorithm is based on LU Decomposition [Lee] and designed to be more efficient by having fewer steps.

$$\begin{vmatrix} a_1 & b_1 & 0 & 0 \\ c_2 & a_2 & b_2 & 0 \\ 0 & c_3 & a_3 & b_3 \\ 0 & 0 & c_4 & a_4 \end{vmatrix} * \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{vmatrix}$$

The Thomas algorithm is subdivided into two different phases, the forward sweep and the backwards substitution. The first phase eliminates the information on the lower diagonal of the matrix, changing the values of the main diagonal, the next equations are used on every row but the first one [2]:

$$c_i = c_i - \frac{c_i}{a_{i-1}} * a_{i-1} = 0$$
$$a_i = a_i - \frac{c_i}{a_{i-1}} * b_{b-1}$$
$$b_i = b_i - \frac{c_i}{a_{i-1}} * 0 = b_i$$
$$z_i = z_i - \frac{c_i}{a_{i-1}} * z_{i-1}$$

2 Based on https://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-_TDMA_(Thomas_algorithm)

$$\begin{vmatrix} a_1 & b_1 & 0 & 0 \\ 0 & a_2^* & b_2 & 0 \\ 0 & 0 & a_3^* & b_3 \\ 0 & 0 & 0 & a_4^* \end{vmatrix} * \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{vmatrix}$$

After the first step is finished the matrix only has 2 diagonals with information. In the second phase of the algorithm the upper diagonal of the algorithm is set to zero and the last computation per row is done. The following equation is used to compute the x values.

$$\begin{cases} x_i = \frac{z_i}{a_i} & i = N-1 \\ x_i = \frac{z_i - b_i * x_{i+1}}{a_i} & i \neq N-1 \end{cases}$$

After finishing this step, the solution is stored on the respective memory.

With this algorithm studied, the table 1 has the gathered information about the theoretical operation and iteration number.

		#Operations/Iteration	#Iterations	Total Operations
Per Step	Forward Sweep	5	N-1	5N-5
	Backward Substitution	3	N-1	3N-3
Whole Algorithm		4	2N-2	8N-8

Table 1: **Number of operations and iterations in the Thomas algorithm**

As seen in table 1, the complexity of the Thomas algorithm is $O(N) = N$ and is strictly sequential. Therefore, in a parallel environment, the complexity is the same.

2.3.2 *Cyclic Reduction*

A method to solve a tridiagonal matrix system in a parallel environment is CR [Hockney (1965a)] [Hockney (1965b)] [Zhang et al. (2010)] [Hwu (2011)] [Bini and Meini (2009)].

This method reduces the number of unknowns in a system of linear equations by half every iteration. When there are only 2 unknowns, the algorithm solves them independently. With these unknowns solved, the algorithm solves the remainder unknowns, doubling the number of solved at a time.

In the first step of the algorithm, it reduces the number of active equations. Because of this the stride (the size between each element) is multiplied by 2, and delta (the number of active equations) is divided by 2, performing the following operations:

$$\begin{aligned} b_i &= b_i - \frac{c_{i-\delta} * a_i}{b_{i-\delta}} - \frac{a_{i+\delta} * c_i}{b_{i+\delta}} \\ z_i &= z_i - \frac{z_{i-\delta} * a_i}{b_{i-\delta}} - \frac{z_{i+\delta} * c_i}{b_{i+\delta}} \\ a_i &= -\frac{a_{i-\delta} * a_i}{b_{i-\delta}} \\ c_i &= \frac{-c_{i+\delta} * c_i}{b_{i+\delta}} \end{aligned}$$

These operations only happen to active equations. These active equations are deactivated by the algorithm by selecting the odd active equations every iteration, halving the number of active equations. An example is shown in Figure 4

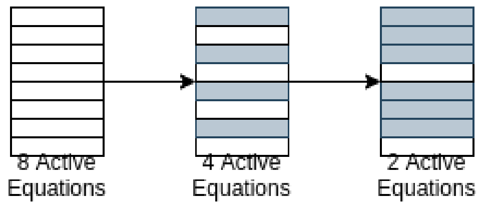

Figure 4: **How 1st step of Cyclic Reduction works: white cells are active, grey cells are inactive.**

After these reductions, the algorithm solves a couple of equations, the only active ones. After having the solution to these equations, the algorithm solves the other equations. Since the active equations are doubled every iteration, the stride is halved and the delta is doubled, shown in the Figure 5.

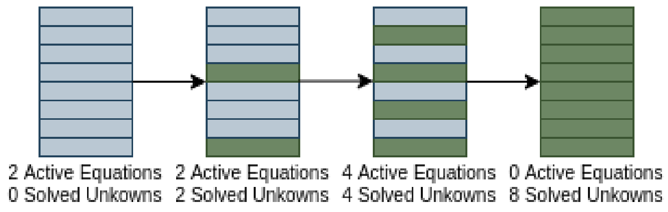
2 Active Equations 0 Solved Unkowns
2 Active Equations 2 Solved Unkowns
4 Active Equations 4 Solved Unkowns
0 Active Equations 8 Solved Unkowns

Figure 5: **Example on how the 2nd step of CR works: green cells are known, grey cells are unknown.**

To solve the remaining unknowns the algorithm performs the following to every active equations:

$$\begin{cases} x_i = \frac{z_i - c_i * x_{1+\delta}}{b_i} & i = \delta - 1 \\ x_i = \frac{z_i * x_{i-\delta} - c_i * x_{i+\delta}}{b_i} & i \neq \delta - 1 \end{cases}$$

By studying this algorithm, it is possible to check how many operations and iterations need to happen. This data was gathered and stored in table 2.

		#Operations/Iteration	#Iteration	Total Operations
Per Step	FS	$12\frac{N}{2^{\#it}}$	$log_2(\frac{N}{2})$	$12\frac{N}{2^{\#it}}log_2(\frac{N}{2})$
	BS	$4 * 2^{\#it}$	$log_2(\frac{N}{2})$	$4 * 2^{\#it}log_2(\frac{N}{2})$
Whole Algorithm		$8N$	$2log_2(\frac{N}{2})$	$16N * log_2(\frac{N}{2})$

Table 2: **Number of operations needed with Cyclic Reduction**

As noticeable in table 2, the complexity of the algorithm is $O(N) = N * log_2(N)$ but is highly parallel. On a parallel environment with N threads, each thread has a complexity of $O(N) = log_2(N)$.

2.3.3 *Parallel Cyclic Reduction*

Parallel Cyclic Reduction is a variant of CR algorithm [Hwu (2011)]. This algorithm uses most of the equations from CR's first phase, but its workload does not get reduced every iteration. On the other hand, PCR does not have a second phase, only making use of a simple equation to solve all the unknowns at the same time.

This algorithm reduces the number of dependable unknowns by creating more systems of equations, all independent as shown in Figure 6. The first step performs the reduction of unknowns in each system of equations by dividing the existing systems by 2, doubling the number of systems of equations but halving the number of unknowns per system. The second step simply solves every unknown in a single iteration. This happens because every 2 equations are bound to the same system of equations, and so these 2 are solved simultaneously [Wei et al. (2013)].

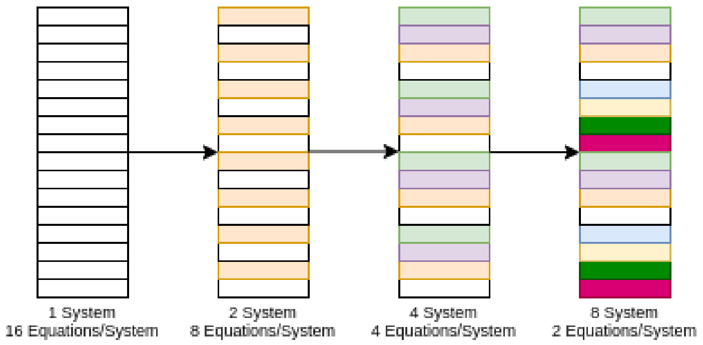

Figure 6: **How Parallel Cyclic Reduction works: each cell is colour coded to the independent system it belongs.**

After reducing the N sized system into $\frac{N}{2}$ systems the algorithm uses the following equation for both equations on each system, solving all of the initial unknowns in a single step:

$$x_i = \frac{b_{i+\delta} * z_i - c_i * z_{i+\delta}}{b_{i+\delta} * b_i - c_i * a_{i+\delta}}$$
$$x_{i+\delta} = \frac{z_{i+\delta} * b_i - z_i * a_{i+\delta}}{b_{i+\delta} * b_i - c_i * a_{i+\delta}}$$

Observing the executed operations and iterations on this algorithm, that information was recorded in table 3.

		#Operations/Iteration	#Iteration	Total Operations
Per Step	FS	$12N$	$log_2(\frac{N}{2})$	$12N * log_2(\frac{N}{2})$
	BS	12	1	12
Whole Algorithm		$12(N+1)$	$log_2(\frac{N}{2})$+1	$12N * log_2(\frac{N}{2})$+12

Table 3: **Number of operations needed with Cyclic Reduction**

The complexity of PCR according to the table 3 is $O(N) = N * log_2(N)$ and is highly parallel. In a parallel environment with N threads, each thread has a complexity of $O(N) = log_2(N)$

2.3.4 *SPIKE Algorithm*

SPIKE algorithm is a tridiagonal solver with high scalability but a considerable overhead. This algorithm is a highly parallel algorithm to solve diagonal dominant matrices [Kjelgaard Mikkelsen and Manguoglu (2008)].

This algorithm splits the tridiagonal matrix into smaller partitions. These partitions can then be solved independently (using Shared memory in 1 GPU or distributed memory with various GPU). This algorithm can be defined in various ways [Polizzi and Sameh (2007)], since the main focus of this algorithm is splitting the tridiagonal matrix into smaller ones, then using backward substitution to solve these smaller ones.

SPIKE's partitioning is different when working on different platforms. While working on GPU it uses data marshalling to rearrange the data into smaller partitions, while working on a distributed memory scheme it partitions the memory to all different machines. For each partition, a smaller matrix is created with the first and last elements of each partition. This algorithm focuses mostly on distributed memory, but it has some variations using shared memory [Chang et al. (2012)].

After partitioning, SPIKE uses a pivoted method to solve the new smaller matrices. Then the SPIKE algorithm uses other auxiliary methods to solve all the remaining unknowns. After solving all the unknowns, the results are gathered into a single solution, merging all the partitions into one.

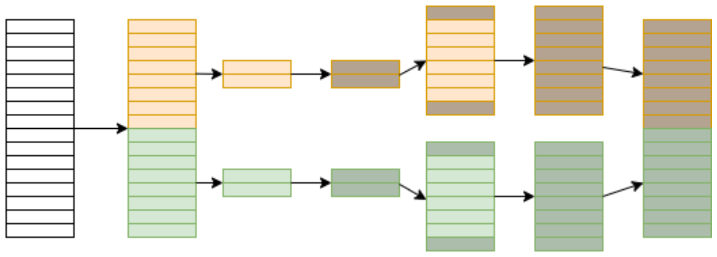

Figure 7: **How SPIKE algorithm works: each cell is colour coded to the independent system it belongs; darker cells are solved.**

This algorithm does not have a countable operation, since they directly depend on the number of partitions. But it is important to note that partitioning the matrix requires more memory allocation, leading to a larger overhead, which makes the algorithm perform significantly worse for smaller matrices when compared to other algorithms with less overhead.

Some studies were done to reduce the overhead and increase the scalability of this algorithm [Spring et al. (2018)]. This study aimed to better manage the load balance between threads, and also studies how the transpose could potentially reduce the computations needed for this algorithm. Just transposing the mid matrices of this problem had granted it with a speedup of over 1.1 over the non transposed one.

While using heterogeneous computing, some studies were made to use CPU, GPU and Field Programmable Gate Array devices [Macintosh (2019)], which proved to be more efficiently than using homogeneous GPU.

2.3.5 *Recursive Doubling*

The *Recursive Doubling (RD)* is a tridiagonal solver algorithm to solve non-dominant diagonal matrices [Stone (1973)]. This algorithm is more efficient when using a higher number of processors than the number of equations to be solved [Ömer EGECIOGLU et al. (1990)].

This algorithm works with some of the LU Decomposition equations, changing them to fit a parallel environment. To initialize, some assumptions are made to easily deduce some of the equations:

$$a_0 = c_{n-1} = 1 \text{ and } x_{-1} = x_n = 0$$

Assuming these, a formula can be deduced from the main tridiagonal equation $Ax = z$

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = z_i, \quad 0 \leqslant i \leqslant n-1$$

After these deductions it is possible to simplify some of the equations to create new matrices for the algorithm like the following:

$$\alpha_i = -\frac{b_i}{c_i} \quad \beta_i = -\frac{a_i}{c_i} \quad \gamma_i = \frac{z_i}{c_i}$$

$$B_i = \begin{vmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

After creating the new matrices it is possible to simplify the equations using the following formulas:

$$\begin{cases} C_i = B_i & i = 0 \\ C_i = B_i C_{i-1} & 1 \leqslant i \leqslant n-1 \end{cases}$$

For C_{n-1} (last one) it is possible to use the following form:

$$C_{n-1} = \begin{vmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & 0 & 1 \end{vmatrix}$$

Each g_{ij} variable is computed using the equations above, for every matrix C_n only the third line is static (having the values [0,0,1]). Then it is possible to solve for x using the following equations, these being independent of each other.

$$X_i = \begin{vmatrix} x_i \\ x_{i-1} \\ 1 \end{vmatrix} = C_{i-1} X_0$$

It is also possible to precompute C_0 and g_{ij} by using the following equations:

$$X_0 = \begin{vmatrix} x_0 \\ x_{-1} \\ 1 \end{vmatrix} = \begin{vmatrix} x_0 \\ 0 \\ 1 \end{vmatrix}$$

$$x_0 = -\frac{g_{02}}{g_{00}}$$

This algorithm has more steps than the other algorithms, the total number of operations and iterations was gathered and shown in table 4.

	#Operations/Iteration	#Iteration	Total Operations
Creating B	3	N	3N
Creating C	10	N-1	10N-10
Solving x_0	1	1	1
Solving for x	2	N-1	2N-2
Total Algorithm	5+1	3N-2+1	15N-11

Table 4: **Total number of operations and iterations using the recursive doubling algorithm**

The bottleneck of this algorithm is computing the second step [Ömer EGECIOGLU et al. (1990)]. This step can be parallelised using a thread to compute more than one element. This divides the problem into various chunks, one for each thread. For a parallel algorithm, a few extra operations must be performed. Each thread must first compute its coefficient to then treat a chunk of the overall problem. The overall complexity of each thread becomes $\frac{n}{p+log(p)}$, where p is the number of processors.

2.3.6 *Hybrid Algorithms*

While most tridiagonal algorithms are composed of various steps, these algorithms have pros and cons. Using hybrid algorithms can use their pros to nullify the cons of others. Therefore, to better accommodate these algorithms to various sizes, some algorithms can be merged to create improved versions.

CR-PCR

While CR is very scalable and efficient for large matrices, PCR is more efficient for smaller matrices. It is possible to merge these two algorithms to create a more efficient way to solve the tridiagonal matrix [Hwu (2011)].

The CR-PCR method uses the first step of CR algorithm to reduce the number of equations to a more manageable size. This will help with the usage of the PCR since it has more operations to do per iteration. The CR-PCR can solve the missing equations of CR by using a strided PCR.

After solving using PCR, the algorithm resumes the CR algorithm, solving the remaining unknowns. The number of operations for this algorithm was stored in table 5. In this table, the variable SS is the size of the PCR system and LS is solved using $M = SS + LS$.

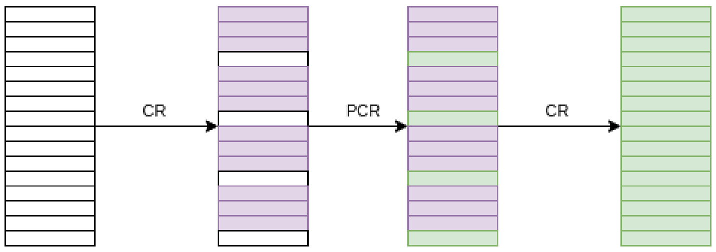

Figure 8: **How CR-PCR works: the CR reduces the number of active unknowns, then the remaining are solved using PCR followed by CR; green cells are solved.**

		#Operations/Iterations	Iteration	Total Operations
Per Step	CR FS	$12\frac{N}{2^{\#it}}$	$log_2(\frac{LS}{2})$	$12\frac{N}{2^{\#it}}log_2(\frac{LS}{2})$
	PCR FS	12SS	$log_2(\frac{SS}{2})$	$12SS * log_2(\frac{SS}{2})$
	PCR BS	12	1	12
	CR BS	$4 * 2^{\#it+log_2(\frac{SS}{2})}$	$log_2(\frac{LS}{2})$	$4 * 2^{\#it+log_2(\frac{SS}{2})}log_2(\frac{SS}{2})$
Whole Algorithm		—	—	$10N(2log_2(\frac{LS}{2}) + log_2(\frac{SS}{2}))$

Table 5: **Total number of operations and iterations using CR-PCR algorithm.**

PCR-Thomas

While PCR is considered an efficient method to solve smaller matrices, this method does not have good scalability like CR. However, its second step is solved in a single iteration being computationally heavy. Due to this, the algorithm changed and instead of it solving the matrix, the Thomas algorithm is used instead since the smaller systems are independent.

The Thomas algorithm is strictly sequential, but since it has to solve various systems of equations that are independent it is possible to execute various algorithms in parallel [Hwu (2011)].

The use of various Thomas algorithms inside each GPU block can also help hide the memory latency of the algorithm by having more than 1 warp (a group of threads, having more than one lets the device still do calculations while waiting for data) working at a time. This will be explored further on Chapter 3 and results shown in Chapter 5. The number of operations and iterations were stored in table 6, where the variable SS is the size of the Thomas algorithm systems and LS is solved using $M = SS + LS$.

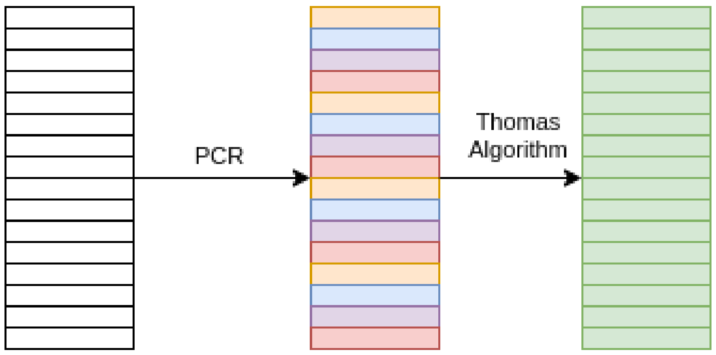

Figure 9: **How PCR-Thomas works: the PCR reduces the number of dependant unknowns by creating independent systems, where each is solved with Thomas algorithm; each independent system is colour coded and green cells are solved.**

		#Operations/Iterations	Iteration	Total Operations
Per Step	PCR FS	12N	$log_2(\frac{LS}{2})$	$12N * log_2(\frac{LS}{2})$
	Thomas algorithm	4	2SS-2	8SS-8
Whole Algorithm		—	—	$12N * log_2(\frac{LS}{2}) + 8SS - 8$

Table 6: **Total number of operations and iterations using PCR-Thomas algorithm.**

CR-RD

While CR is very scalable when compared to other algorithms, having sleeping threads amidst the steps makes so that the algorithm can become less efficient in the last iterations. This algorithm also reduces the number of equations active at a time.

As mentioned before, the RD is very efficient when the ratio between the number of processors and the number of equations is the highest, and with the reduction of the equations to be solved by the CR, the algorithm can achieve better performance.

The use of RD with conjunction with CR can help mitigate the disadvantages of both algorithms, namely by reducing the number of inactive threads of CR with the introduction of RD. The RD can then use all threads with a lower number of equations, increasing the ratio between them [Zhang et al. (2010)]. The number of operations and iterations were stored in table 7, where the variable SS is the size of RD algorithm systems and LS is solved using $M = SS + LS$.

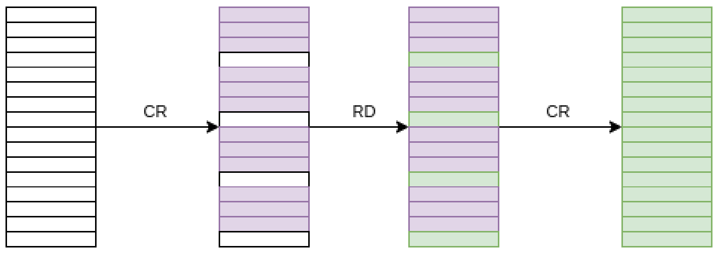

Figure 10: **How CR-RD works: the CR reduces the number of active unknowns, then the remaining are solved using RD followed by CR; green cells are solved.**

		#Operations/Iterations	Iteration	Total Operations
Per Step	CR FS	$12\frac{N}{2^{\#it}}$	$log_2(\frac{LS}{2})$	$12\frac{N}{2^{\#it}}log_2(\frac{LS}{2})$
	Thomas algorithm	5+1	3SS-2+1	15SS-11
	CR BS	$4*2^{\#it+log_2(\frac{SS}{2})}$	$log_2(\frac{LS}{2})$	$4*2^{\#it+log_2(\frac{SS}{2})}log_2(\frac{SS}{2})$
Whole Algorithm		—	—	$4*2^{\#it+log_2(\frac{SS}{2})}log_2(\frac{SS}{2})$ $+12\frac{N}{2^{\#it}}log_2(\frac{LS}{2})+15SS-11$

Table 7: **Total number of operations and iterations using the CR-RD algorithm.**

2.3.7 *Comparative Evaluation*

Even though most tridiagonal solvers are used to solve the same problem, they often perform very different operations and occur on different contexts. It is important to understand these contexts to fully comprehend what are the best suited algorithms for the studied problem.

	Total Operations	Works on Diagonal Dominant Matrices	Can be Parallelised
Thomas Algorithm	$8N-8$	✓	X
Cyclic Reduction	$16N*log_2(\frac{N}{2})$	✓	✓
Parallel Cyclic Reduction	$12N*log_2(\frac{N}{2})+12$	✓	✓
SPIKE Algorithm	Varies from partition size	✓	✓
Recursive Doubling	15N-11	X	✓

Table 8: **Comparison between tridiagonal solvers**

Looking at the table 8, it is possible to notice that the Thomas algorithm may not be the best suited for a parallel environment when used alone. However, when used with the ADI method, it is possible to have various Thomas algorithms performing at the same time.

When working in a multicore environment where the number of threads is below the number of systems to perform, having the number of systems equal to the size of the domain, the Thomas algorithm may be the best choice to have, since it has the lowest number of operations and in conjunction with ADI it can perform as a basic parallel algorithm.

While studying the other algorithms and their potential to explore parallelism, it may be possible to study improved ways to use threads to compute a single line/column. These threads could also be used to accelerate the computation of various lines, computing fewer lines at a time, but in less time.

The RD needs the upper and lower diagonal to not have zeroes due to the equations used. The ADI method creates these tridiagonal matrices based on a grid, and when the grid has outside cells, the values for the lower and upper diagonal are set as zero. Hence RD can not work on most of the matrices created by ADI.

It is also good to point out that the SPIKE algorithm is mostly used in distributed memory schemes, and even being possible to use in shared memory, this algorithm has a large overhead but scales very well, so being used in parallel for various tridiagonal matrices, this algorithm would benefit using only a single and larger tridiagonal matrix.

The use of hybrid algorithms will also be studied. The use of PCR in conjunction with the CR can help mitigate the problem of sleeping threads slowing down the code, while CR helps to reduce the total number of operations that need to be executed with the PCR. Thomas algorithm can also be used in conjunction with PCR. This algorithm can not be parallelised by itself, but the large tridiagonal matrix can be divided into smaller independent tridiagonal matrices by PCR. The use of Thomas algorithm also reduces the number of total operations of the PCR, by reducing the total workload of the algorithm.

The use of these hybrid tridiagonal solvers have the potential to be very efficient, since the advantages of each algorithm can, sometimes, nullify most of the disadvantages of others.

3 COMPUTATIONAL EFFICIENCY

In this chapter, a study about the GPU will be made, namely how the vector processing works on CPU and how differently it works on GPU. How manycore systems can achieve better performance is also studied, and how to grade scalable algorithms.

Various approaches for the ADI method will also be shown and the advantages and disadvantages of each approach.

The GPU architecture as well as the main differences between OpenCL and CUDA are analysed. A study of these API allows creating a comparable code which is essential to a fair evaluation of their overall performance.

3.1 SCALABILITY IN MANYCORE COMPUTING

Most modern CPU are designed and built with more than one physical core [Hennessy and Patterson (2011)]. These can work simultaneously on one or more algorithms so that they can achieve better performance; an example of a parallel algorithm is merge-sort, with a divide and conquer methodology [1].

To test the scalability of an algorithm it is possible to use Amdahl's Law [Amdahl (1967)], which describes the maximum theoretical possible speedup in an algorithm containing a part of sequential code. It can be described as following, where ρ is the percentage of parallel code and s is the number of threads working:

$$\frac{1}{1-\rho+\frac{\rho}{s}} \Rightarrow \lim_{s\to\infty} \frac{1}{1-\rho}$$

This law divides the code into two parts, a strictly sequential part and a possible parallel part. Since the sequential part can never be parallelised, this algorithm can be simplified to show the best speedup possible by removing the therm $\frac{s}{\rho}$, leaving the best possible speedup as:

$$\frac{1}{1-\rho}$$

To fully achieve the best performance possible while using various physical cores on a single algorithm, the program needs to be able to perform more than one instruction simultaneously. So

1 A reference paper from the School of Computer Science of Carnegie Mellon University: https://www.cs.cmu.edu/ guyb/papers/BM04.pdf

ideally the number of dependencies in the code should be nonexistent between different threads. Therefore, a code with a lot of data dependencies can not achieve a major performance speedup when using various threads, since some of the threads will be locked, waiting for others to finish their work [Tithi et al. (2013)]. An example of this can be seen in Figure 11, where the code takes double the time to solve a problem because of data dependencies between threads.

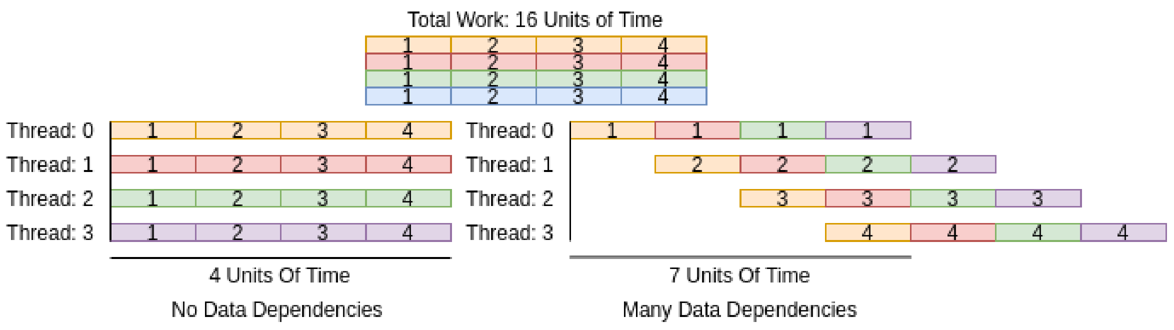

Figure 11: **Example on how having data dependencies between threads can slow down the code. Every colour means a dependency, orange cell #2 can not start until the end of orange cell #1.**

While having more physical cores working on a problem can lead to better performance, it is essential to make sure that every core does the most work it can. For that to happen, they should be working simultaneously and do the same amount of work, so the work should be divided as evenly as possible. This leads to work scheduling which, API such as OpenMP can do in different ways[2]:

- Static scheduling - Divides the problem into equal chunks, one for each thread. There is no extra overhead using this schedule type;
- Dynamic scheduling - Divides the problem into small chunks and assigns a single chunk for each thread, when a thread finishes the chunk assigned to it, the scheduler handles it another chunk. This scheduling type has an extra overhead when compared to static scheduling;
- Guided scheduling - Similar to Dynamic, but the chunk size varies, starting with larger chunks and reducing the size. This reduces the extra overhead from the dynamic scheduler.

An example is shown in Figure 12, each cell takes a unit of time, and because of bad balancing, it is possible to notice how the code takes almost double the execution time because of thread number 2, while the third thread only does little work.

For an algorithm to be scalable, it needs to be compute-bound. This is because having more threads can have an impact on the performance since the operations are ready to be executed but have no empty cores to execute them. This makes the memory layout to be of extra importance, since if the remaining operations are waiting for the data, having more threads only generates extra overhead, harming the performance [Hennessy and Patterson (2011)].

2 Information from Intel in https://software.intel.com/content/www/us/en/develop/articles/openmp-loop-scheduling.html

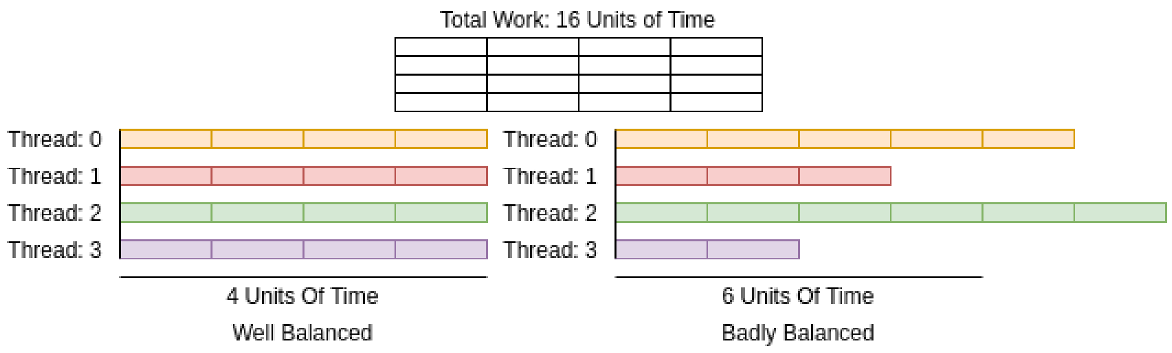

Figure 12: **Example on how having unbalanced work loads between threads can slow down the code. Colour coded to each assigned thread.**

3.2 DATA LAYOUT FOR EFFICIENT COMPUTING

Over the years the processing power of computers have grown exponentially, while the memory bandwidth grows steadily [Carvalho (2002)]. To have the best possible performance, the data usage should be planned beforehand. To obtain the best performance out of the code, a few points are studied in the context of the problem: memory bandwidth, memory latency and cache.

Whenever the program needs to retrieve data from memory, it suffers from latency for every transfer that needs to do. This is called memory latency, and this degrades the performance drastically. One of the ways to reduce the degradation caused by this latency is for the memory to retrieve more data than what is needed (usually the memory transfers 64 bytes of memory at a time) and to fill a line of cache. This line can then be used for subsequent operations without the need to go to the memory (hiding the memory performance degradation). This makes the code that uses data locality have less degradation due to lowering the total number of transfers.

The size of the memory bandwidth is the amount of data that can be transferred from the global memory to the *Processing Unit (PU)*. This size is theoretical, and as seen before, it has a few limitations (like the memory latency). The memory bandwidth is the amount of data to be transferred for every PU. Therefore, on parallel programs, it is of high importance to utilise this bandwidth as best as possible. Because of this, the usage of data locality and cache are of high importance, since the cache reduces the number of transfers that needs to happen (by storing the values and using them instead of always using the global memory) [Hennessy and Patterson (2011)].

To use the cache more efficiently the used data should be layed out depending on how the language stores the information (the C/C++ language stores the information of multi dimensional arrays in rows, so all the data should be layed out in rows). The cache itself can have more than one level, these levels are located on different parts of the processor. The lower levels of cache are smaller but have a lower penalty when accessed.

Accessing the registers (the smaller but faster memory of the hardware) has a smaller penalty when compared to other memory levels [Carvalho (2002)]. So when a program uses a variable more than once, the program stores it on hardware registers, which are scarce, but the few ones available are fast and have no penalty on their access. The smaller levels of cache can sometimes be inside CPU chips, making their size limited by the hardware itself, and are costlier to produce.

The memory can not be explicitly forced to be stored in the cache, so the programmer can use data locality to make the hardware store the data in the cache and use a low number of variables and use them the best way possible to store them all in the hardware registers.

In NUMA environments, the memory also needs to be stored in the right memory banks. If the memory is stored in a bank that is not directly connected to the working cores, the memory latency is even more severe, since the data needs to be passed from the various nodes until it reaches the correct core [Majo and Gross (2013)].

GPU ue a different memory scheme than it's CPU counterpart. This memory layout will be explored in the GPU section.

Having data locality in mind and studying how the ADI works, transposing the data is an option that may be explored. The C language has no matrix data type, so C programmers usually specify matrices as arrays of rows, as opposed to the way Fortran defines a matrix data type: as an array of columns. In typical C code, elements of a row are in consecutive memory locations, while elements of the same column are apart by the size of a row. Therefore, to guarantee that consecutive elements of a column are stored in consecutive locations (stride of 1), a transpose can be used when making operations column-wise.

Using both main matrix and transposed matrix the memory can be contiguously loaded and scatter stored, or contiguous stored and gather loaded. Since the store can happen while the CPU executes other operations, the chosen scheme is a contiguous load and scatter store.

For the other cell data (like the position, code, etc), the data structures need to be studied for memory efficiency. Since there are various cell data to be stored, it is possible to store them on either *Structures of Arrays (SoA)* or *Arrays of Structures (AoS)* [3]. While using AoS, each cell data is stored in a single structure, where a single array has all the pointers for every structure. Using SoA, each cell data is stored on various arrays, all stored on a single structure. For better use of memory locality, it is optimal to use SoA, making each data item contiguously located, with a better cache use[4].

3.3 VECTOR COMPUTING

Vector processing is the ability to use 1 instruction on various amounts of data (also known as *Single Instruction Multiple Data (SIMD)*) [Flynn (1972)].

Vector processing is possible when an operation is replicated to a vector of data, using the same instruction on a whole vector instead of 1 element at a time like scalar processors. Vector processing is used on generic PU that do not have native vector capabilities, by using vector extensions.

Using SIMD has some advantages when comparing it to regular scalar operations. One of the advantages is how the vector processing hides memory latency by using deeply pipelined loads and stores, having the latency happen only once per vector, instead of once per element [Hennessy and Patterson (2011)]. This made sure that the PU would not wait because of the memory latency, but mostly because of the limited bandwidth.

SIMD also reduces the number of operations that need to be read and, in some cases, potential loops from happening. Using the example in [Hennessy and Patterson (2011)], when computing the

3 https://software.intel.com/content/www/us/en/develop/articles/memory-layout-transformations.html
4 https://software.intel.com/content/www/us/en/develop/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html

Linpack benchmark's DAXPY (**D**ouble-Precision *a* times *X* **p**lus *Y*), using a vector instruction set (for example Intel's *Advanced Vector Extensions (AVX)*) it is possible to use a smaller number of instructions needed to compute this problem. This is possible because while using regular instruction, the program needs to process 2 loads, 1 store and 5 operations (not counting the jump if it is needed), the vector operations needed are only 2 loads, 1 store and 2 operations for the entirety of the vector (if the vector size is the same size as the capacity of the vector operation).

Using SIMD in some algorithms may be more difficult than others since the size of the data might be larger than the capacity of the vector operation. To treat all the data it is possible to use a technique called *strip-mining* [Weiss (1991)]. In *strip-mining* the program treats blocks of memory at a time (usually this block would be the size of the capacity of the vector operation), and then uses the last bits of data (that may not be enough to fill a vector) independently, not using SIMD on those. This approach can reduce the number of vector operations.

SIMD can also use chaining, which makes the vector processor use the vectors data almost instantly on the following operations, making sure that the vector is not stored and loaded into memory if the vector is used more than once at a time, only saving/loading into the vector registers [Hennessy and Patterson (2011)].

SIMD strives to use a single instruction in a whole vector of data, but when conditional appears some compilers have trouble dealing with them since in some pieces of data some code may happen and in others it does not. To hide divergence in code, the user can apply masks to it, making sure that a conditional instruction is masked as a regular instruction.

While using conditionals is very inefficient when it comes to SIMD and masks make the code easily vectorizable, using masks in regular scalar code is inefficient since it makes all operations happen in the whole data, instead of only happening in some bits of it [Hennessy and Patterson (2011)].

3.3.1 *Vector Extensions on Scalar Processors*

Vector extensions have been researched and developed for generic PU. Intel has started developing vector extensions since the 1990s, starting with MMX and *Streaming SIMD Extensions (SSE)*, in late 2000s, early 2010s developed AVX, with AVX2 and AVX-512 coming after. *Advanced Micro Devices (AMD)* also has its type of vector extension, known as 3DNow!.

Vector extensions have different instruction sets, which use the vectorial processors of the PU and these are called *Vectorial Processing Unit (VPU)*. These include vector registers, registers that can be filled with multiple data, instead of the single data that fills the regular registers and also have vector functional units. These units are pipelined and use the data in vectorial registers to use instructions on multiple data. Using SIMD on vector extensions is not the same as using it on vector machines.

Vector extensions on PU do not work well using data that are not contiguous in the memory. With the use of scatter-gather, vector machines can hide the memory latency pretty well when compared to vector extensions used in scalar PU. Gather-scatter was added to AVX512, but it is still heavily penalised when comparing to using contiguous memory. Using conditionals also causes performance degradation. Sometimes it can even disable the use of SIMD in these sets of data. Instead of using conditionals, when possible, the programmer could use masks, hiding possible divergences in the vectorized code.

Having data dependency between the data in the same vector also disables the use of SIMD. Because of this, when using operations with data dependencies, they should be studied to check if there is a way to avoid these.

3.4 WORKING WITH GPUS

For a program to work with the GPU it has to be written in a specific API, like CUDA[5] or OpenCL [6], or using specific libraries, like PyCUDA for python. CUDA is only used with Nvidia's Proprietary GPU while OpenCL can be used on all GPUs. These 2 API have some specific concepts and keywords that, even being the same, are named differently. Figure 13 and Figure 14 shows some concepts/keywords named differently on these 2 APIs. It is important to notice that sometimes the same name can lead to a different concept on the 2 APIs[7].

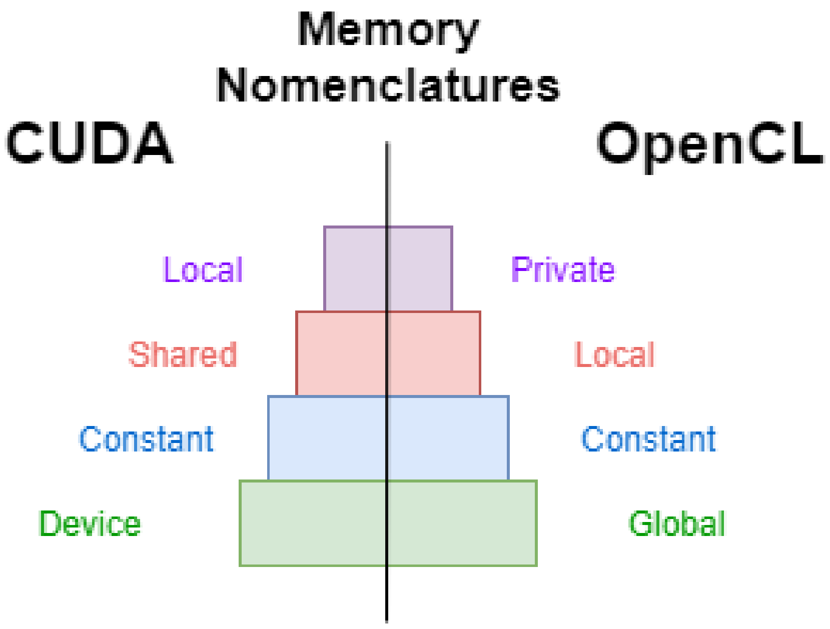

Figure 13: **CUDA and OpenCL memory designations.**

Memory hierarchy in GPU is similar to the one in conventional CPU, but with a few relevant differences: (i) each SIMD PU (a *Streaming Multiprocessor (SM)*, as labelled by NVidia) has a very large number of 32-bit registers (thousands); (ii) a physical memory on-chip can be configured as a local memory of SM at the same level as cache L1; (iii) cache L2 on-chip in a GPU is shared among all SM; and (iv) GPU have no cache L3[8].

5 CUDA Zone - https://developer.nvidia.com/cuda-zone
6 OpenCL Home Page - https://www.khronos.org/opencl/
7 Extra information can be found here:
https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf
https://on-demand.gputechconf.com/gtc/2012/presentations/S0642-GTC2012-Inside-Kepler.pdf
8 This memory hierarchy is based on the GPUs studied in this dissertation, other memory hierarchy can be found on other manufacturers, but those will not be studied.

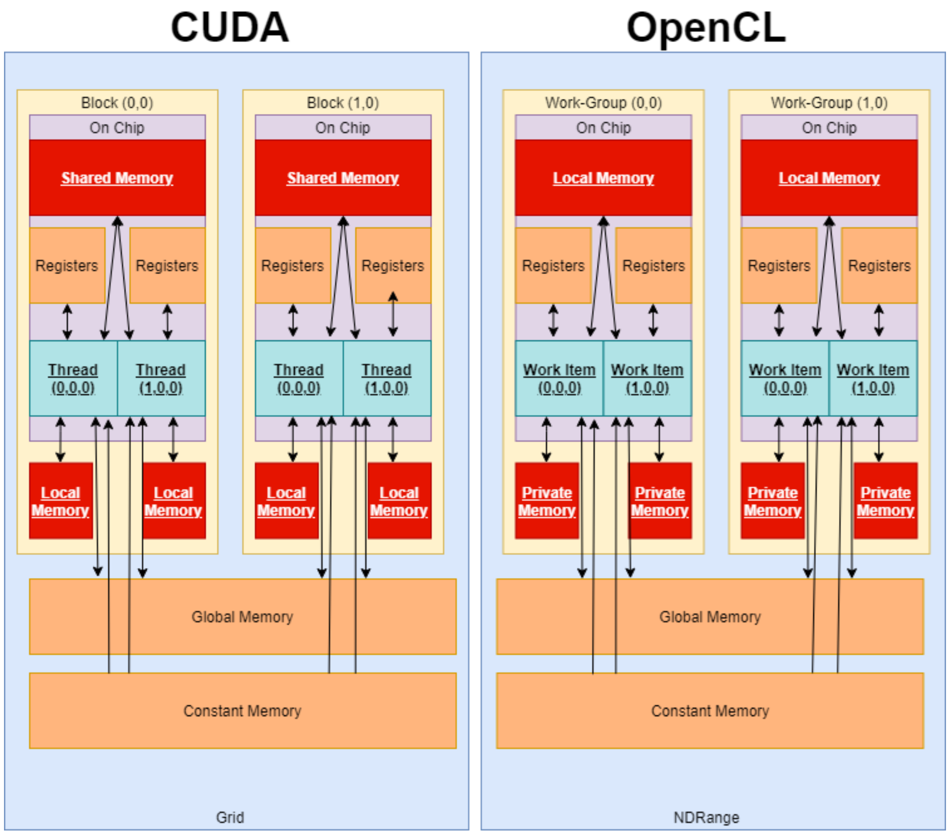

Figure 14: **CUDA and OpenCL different terminology.**

3.4.1 *Generic Computing on GPUs*

GPU is a specific type of a manycore architecture with vector capabilities and, as the name implies, were firstly designed to process graphics [Tomov (2016)]. GPU have high parallel processing capabilities with good power efficiency, therefore being widely used in *High Performance Computing (HPC)*.

GPU have a different architecture when compared to CPU. The GPU architecture is composed of kernel grids. These grids communicate with each other only using the main memory, not being able to synchronise in any other way. Therefore are almost independent of each other. In each grid, there are blocks of threads, where each block is organised by threads. Each thread can communicate with each other using shared memory, a memory type that is smaller, but also faster, that is shared between a block of threads. In every thread, there is also a private memory only accessible by it. In every block, it is also possible to synchronise using functions provided by the API developers.

In HPC, the key features of the GPU devices are used to perform vector computing operations, rather than generating graphic scenes on a monitor [Gadhikar and Rao (2018)]. This is known as *General-Purpose Computing in Graphics Processing Units (GPGPU)*. GPU were designed to have enormous computational power, allow high parallelism and very high bandwidth and low latency. Some of the memory bottlenecks are masked on GPU computing because of the higher memory bandwidth that they have when compared to CPU.

GPU can be used to compute matrix operations efficiently like matrix multiplication [Fatahalian et al. (2004)] using BLAS [Andrzej Chrzeszczyk (2017)] or parallel algorithms. This is due to their architecture, since the threads are automatically organised by groups, it is possible to use simpler instructions and think in sequential code that each thread will process concurrently.

Unlike CPU, the GPU architecture was built to work with its own type of SIMD, also known as *Single Instruction Multiple Threads (SIMT)*. This kind of vector computing uses more than one core grouped by warps (in CUDA or Wavefront in OpenCL). These warps are groups of threads, each running in its core, that all use the same set of instructions on their data (instead of 1 core using 1 instruction on vector-like vector extensions on CPU). There can be more than one warp assigned per SM, this is beneficial because while some of the warps are waiting for the memory to be gathered from the main memory, the cores are working on warps that already have the data ready to be used.

Vector computing in GPU can be more efficient when compared to CPU since GPU have hundreds to thousands of cores, when compared to the lower number of cores on CPU, having a higher number of operations performed per second, even with a lower clock rate and less synchronicity.

Exploring these vector capabilities can be really helpful performance-wise since it is noticeable that, outside of data transfers from and to the device, it is possible to get more performance when using the parallel and vector capabilities on GPU. The GPU can also benefit from SIMD and instruction-level parallelism by having its own set of intrinsic functions [Hennessy and Patterson (2011)].

Since GPU work in warps, it is better when all the threads that are working simultaneously have no divergence between each other, since only 1 instruction can happen at a time.

Some other important remarks about performance in GPU computing are how changing some values on the code to match the GPU specifications can improve the algorithm perform-wise. For example, it is best to use a number of threads that is a multiple of the warp number [Sethia et al. (2015)]. Warps issue the instructions to the threads in groups, so having no divergence (by masking these conditionals for example) in warps can make the code substantially more efficient, and threads being in a multiple number of the warp size means all threads will be working all the time, instead of some threads waiting for others to finish.

When working with CUDA, the instruction set architecture used is called *Parallel Thread Execution (PTX)*[9]. This *Instruction Set Architecture (ISA)* describes the operations to happen in a single CUDA thread. Using PTX is easier for the programmers since it hides the GPU hardware instructions, uses virtual registers that are distributed by the compiler itself, and can help to speed up some of the code by using the best instructions, chosen again by the compiler [Hennessy and Patterson (2011)].

PTX also hides conditional use of masks. These masks are used to eliminate possible divergence in code, making sure that every thread can perform a single instruction instead of having different instructions, making the code have a better performance overall. If the code has divergent conditional branching, the PTX predicts the best instruction flow by using a branch graph to perform the optimal instruction sequence.

Despite a lot of similarities between a vector computer and a GPU, there are some key differences between both. One of the main differences is how a vector computer gathers all the vector info in a vector register and performs the instructions on that block of data. A GPU would distribute all the data between various SIMD lanes. Besides the use of SIMD, the GPU can also benefit from multi-threading.

9 NVidia's Application guide to Parallel Thread Execution ISA: https://docs.nvidia.com/cuda/pdf/ptx_isa_7.1.pdf

Another difference between both is how each architecture tries to hide memory latency. While vector computers hide memory latency by getting all the elements of a vector at once (and only paying the latency once per load), GPU use multi-threading to hide it, using a thread scheduler, making sure that while some threads are waiting for the memory, other threads are active, making sure that the GPU is as active as possible [Hennessy and Patterson (2011)] [Cook (2013)]. There are also no scalar processing on GPU, making the scalar and sequential bits of the code inefficient on GPU while comparing them to vector machines.

3.4.2 *ADI on GPUs*

Since ADI is embarrassingly parallel, a GPU could be a very good option to implement a very efficient solution. Some previous work to port the ADI method to the GPU was already done [Wei et al. (2013)].

The ADI method itself only creates the vectors for the tridiagonal solvers to use for finding the solution. Since these vectors should fit on the GPU caches for better performance overall, some tweaks might need to be done. Another possibility is solving a limited number of rows/columns at a time. To achieve better performance some important remarks to remember are:

- Remembering how conditionals impact the performance of the code, and how the GPU get extra penalisation on branch divergences.
- The number of threads to utilise on each tridiagonal solver (or both if a hybrid solver is chosen), to make sure that the best performance is reached.
- Utilising the full vector and parallel capabilities of the GPU, using the available resources to maximise the performance.

When using ADI on GPU it is possible to diverge from the CPU using a different thread organisation. In the CPU code, for every thread, a single line/column is assigned at a time. This is used in CPU since the number of threads is very limited. It is possible to do this on the GPU, but since the number of threads is very superior when compared to CPU it is not good performance-wise, as the capacity of the GPU will be lower than 100%. Because of this, it is possible to assign a line/column to a block of threads. Using this thread scheme it is possible to use multi-threading inside each line/column, broadening the solvers to be able to be used. An example of the different schemes is showed in Figure 15.

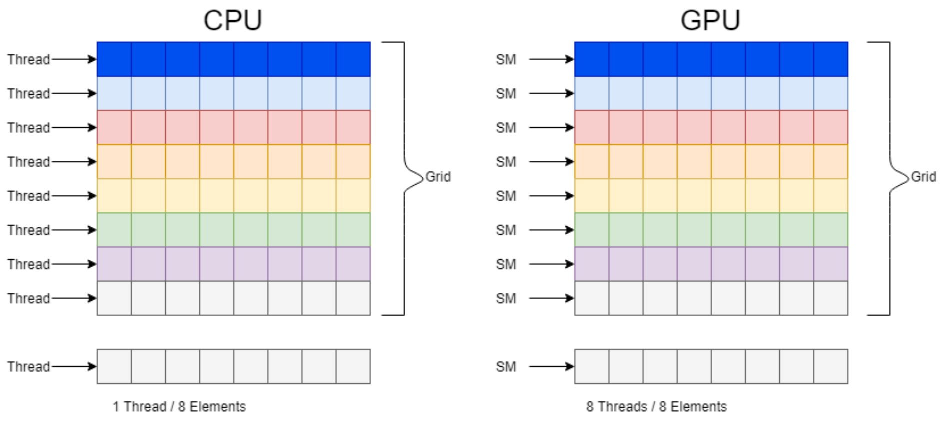

Figure 15: **Different thread Schemes for a grid for different devices**

Usually the main focus while studying the ADI method is the tridiagonal solver used since that algorithm is one of the main components of the ADI.

3.4.3 *Tridiagonal Solvers on GPUs*

Implemented ADI methods in GPU can use different tridiagonal solvers. This difference in tridiagonal solvers used is based on multi-threading in each line/column. Since the performance of the ADI is mostly based on the tridiagonal solver used, the ADI on itself only builds the tridiagonal solvers used.

The main focus on porting this algorithm to the GPU should be carefully planning all the resources. Reducing the number of operations that need to be performed sequentially by testing different algorithms can also be done. Divisions are computationally heavy when compared to other operations, so reducing the number of them, for example, by computing the inverses. These possibilities should be tested when using ADI since it helps to hide the sequential bits of the tridiagonal solvers used.

With the use of GPU, it is possible to use these architectures advantages like the higher bandwidth and the use of shared memory and multi-threading and SIMT to have better performance on these algorithms.

3.4.4 *Max Reduce Approach*

Since the GPU has a high number of threads available and has no scalar capabilities, the GPU can use a different algorithm for the maximum error search, *i.e.* a modified sum reduction[10].

This reduction was modified to, with a mask, chose which element has a higher error (instead of simply adding them). This new algorithm has a complexity of $O(N) = log_2(N)$.

Since the modulo function is computationally heavy and this implementation uses non-contiguous cells, the algorithm will be changed slightly to choose the first cells (reducing the number of operations to do due to every thread being assigned to cell $threadNum * 2$ and $threadNum * 2 + 1$).

10 Based on nVidia Optimising Parallel Reduction in CUDA on
http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Algorithm 1 Max Reduce based on Sum Reduce

```
1: function MAX REDUCE
2:     element = threadId
3:     nextElement = threadId+blockSize
4: loop elementsMissing > 1
5:     if Not Sleeping then
6:         if Array[element] < Array[nextElement] then
7:             Array[element] = Array[nextElement]
8:     elementsMissing = elementsMissing/2
```

3.5 HETEROGENEOUS COMPUTING

With the high parallelism in the ADI method, it may be possible to use the CPU as another device, instead of a simple host [Mittal and Vetter (2015)]. Using heterogeneous computing could potentially increase the performance of parallel algorithms due to having more raw processing power and the sheer number of PU.

CPU and GPU have different architectures and different capabilities. When using heterogeneous computing the division of the workload can make the best usage of each architecture, looking at the article [Mittal and Vetter (2015)] it is possible to make better use of the high-throughput GPU can offer and use the CPU for latency-critical zones of the programs.

The GPU also has problems dealing with divergence in code, so in areas of a program with a lot of divergences and not masked conditionals it is better to use the CPU. Another reason to use heterogeneous environments is the better management of power since the CPU stays idle waiting for the kernels to finish their assigned workload, some power is wasted in this wait. Therefore, using CPU for other work makes the program more power-efficient.

While working in OpenCL some studies [Grewe and O'Boyle (2011)] show that using CPU and GPU on different algorithms gives various speedups depending on the partition of the work for each device, hence the division of the work must be studied and tested when working with heterogeneous computing.

A problem that can be tailored to take advantage of heterogeneous computing is the SPIKE algorithm (studied on Section 2.3.4). This algorithm has the advantage of splitting the problem into sub-problems, and each sub-problem is solved independently. Each of these problems could be solved on various CPU, GPU and a combination of both, taking the best advantage of heterogeneous computing.

3.6 TENSOR COMPUTING

Vector computing allows the hardware to compute entire vectors at a time, allowing some programs to have speedups in terms of computational power (by computing more values at a time) but also in memory-bound programs (by hiding some of the memory latency). Tensor computing on the other hand allows the hardware to compute entire matrices (2D vectors) at a time, having a higher speedup on specific problems.

Tensor computing has been studied and some manufactures have started working on Tensor cores. These tensor processors are mostly used to work with Machine Learning, Intel is currently working on *Advanced Matrix Extensions (AMX)*, a new instruction set to work with tensor processing, this instruction set is scheduled to release with Saphire Falls in 2021, this instruction set uses tiles instead of vectors.

Intel has some experience already with tensor processing for neural networks in 2017 using Intel Neural Net Processors by acquiring Nervana Engine. Now Intel works with Habana, with a chip for training neural networks (using Gaudi chip) and an inference chip (using Goya chip). Habana Gaudi and Goya devices have tensor processing cores in their constitution, enabling them to use tensor processing. Nvidia has already added tensor core units to their hardware. In the Turing architecture, these tensor cores can have a throughput of at least 8 times larger [11]. Some vendors prefer to use *System on Chip (SoC)* devices with embedded applications that can use tensor computing, such as Google with Google Cloud TPU and Tesla with Tesla Full Self Drive.

When working with the ADI problem, it may be possible to use tensor computing to speed up the tridiagonal matrices creation, since most of the matrices are used by multiplying the state of the cell (if it is a boundary, inner or outside the cell, since they are coded with numbers) by the matrix values, hence using a custom matrix-multiplication.

While working with tridiagonal solvers it may also be possible to work with tensor computing, since the matrices are sparse, it may be possible to divide the usable part of the matrices in chunks and use a tensor computing unit per chunk.

11 Data from Nvidia: https://www.nvidia.com/en-us/data-center/tensor-cores/

Part II

CORE OF THE DISSERTATION

4 IMPLEMENTATION OF THE ALGORITHMS

ADI is a highly parallel algorithm since it splits the problem into many smaller and independent subproblems. In this chapter different implementations were studied and implemented, varying from different devices and APIs to different tridiagonal solvers.

All generated call-graphs for this work use a visibility threshold to have better readability: all functions that take less than 1 % of the overall execution time are not shown.

4.1 TARGETING CPU DEVICES

For the CPU code, a simple implementation was created and fine-tuned for the architecture used.

After getting an efficient sequential code, a parallel code was developed and implemented to run on three different device architectures. Using NUMA dual sockets and exploring the vector capacities of these architectures could potentially lead to an improved version of the code performance.

4.1.1 *Native Code*

The first implementation of the ADI method was straightforward and worked very precisely; however, it was slow since it was not built aiming performance.

For the initial tests, the sequential Thomas algorithm was chosen as the tridiagonal solver due to its simplicity and efficiency (as seen in Table 8 of Section 2.3.7).

The following algorithm shows a simple implementation of the ADI method using the Thomas algorithm as the auxiliary tridiagonal solver[1], followed by an image that shows a call-graph view of this version using a 1% threshold to remove those functions with a very small impact on the overall performance.

1 * Based on the equations shown in Section 2.2.1

Algorithm 2 ADI method with Thomas algorithm

```
1:  function ADI METHOD
2:  First Step:
3:  loop for each row
4:      $a_k = c_k = ComputeAC(k)^*$
5:      $b_k = ComputeB(k)^*$
6:      $z_k = ComputeZ(k)^*$
7:      $Thomas\_Algorithm(a, b, c, z, sol)$
8:  Second Step:
9:  loop for each column
10:     $a_k = c_k = ComputeAC(k)^*$
11:     $b_k = ComputeB(k)^*$
12:     $z_k = ComputeZ(k)^*$
13:     $Thomas\_Algorithm(a, b, c, z, sol)$
```

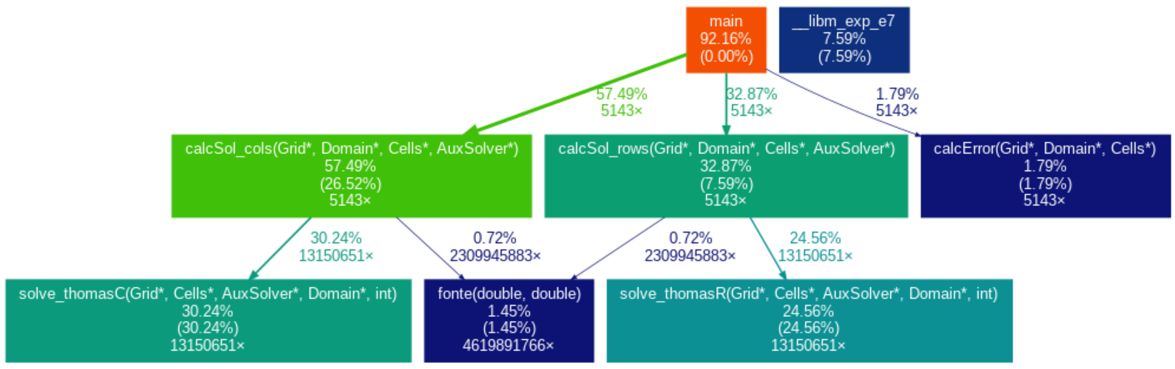

Figure 16: **A call-graph view of the ADI code with Thomas algorithm: *calcSol* are ADI functions with two steps, one for columns and another for rows; *fonte* is an auxiliary function.**

As Figure 16 shows, the Thomas algorithm is currently the slowest step of the algorithm (taking a total of around 55% of the overall execution time). It is also possible to notice how the ADI method is 3.5 times slower when comparing the columns to the rows, so studying a method to better use the memory layout is important. The *fonte* function and *exp* can also be moved to outside the loops, since they are static for each cell, computing it once per cell, instead of once per cell per iteration.

4.1.2 *Efficient Sequential Solutions*

After studying the problem and profiling the code, it was possible to notice how the column-wise functions take almost twice longer than the row-wise functions. One way to overcome this drawback is to store the result of the first step (the row-wise) in a transposed matrix such that the second step (column-wise) would access contiguous memory positions.

To test the memory layout and the performance memory-wise the size of the grid should be larger than all the combined caches. Since the data types used are doubles, every value has a size of 8 bytes. The grid is a square, so the following equation was used to reach a decision:

$$GridSize \gg \sqrt{\frac{CacheL3}{ValueSize}} * \sqrt{\frac{CacheL3}{ValueSize}} \approx \sqrt{\frac{32*10^6}{8}} * \sqrt{\frac{32*10^6}{8}} = 2000 * 2000$$
$$SelectedGridSize = 2560 * 2560$$

Every row with this size (2560 doubles, or 20480 bytes) is aligned with the cache line size (64 bytes) since its value is a multiple of it. The execution time and various cache information will be recorded to show how some methods will be studied and if they have an improvement in the execution time. A new implementation was created where a transpose of the grid is also stored, since reading in contiguous blocks and writing into the memory in non-contiguous blocks is more efficient and cache-friendly.

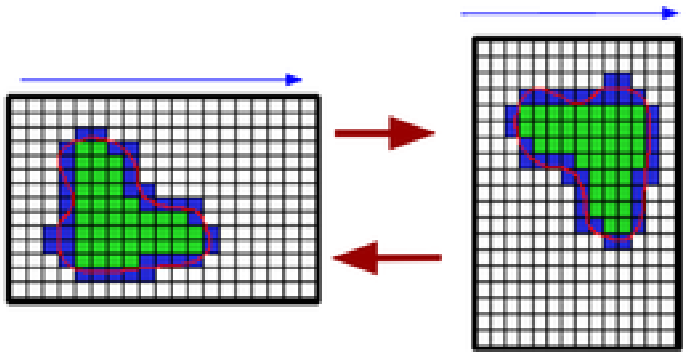

Figure 17: **The output of the row-wise step is transposed to run the column-wise step in row mode**

After changing the memory scheme a change to the exponential call frequency was also made to call it once per cell instead of one time per cell every iteration. In theory this workaround will decrease the time spent on computations, but will increase the memory usage, so cache misses might increase even if the execution time decreases.

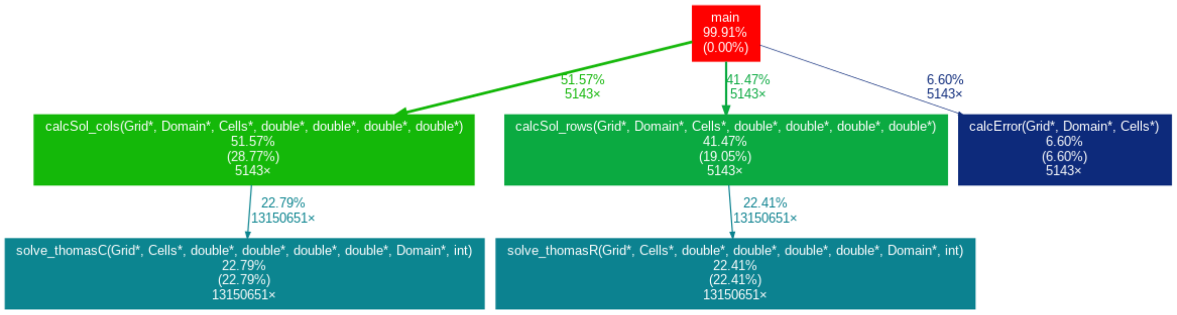
main
99.91%
(0.00%)
51.57%
5143×
41.47%
5143×
6.60%
5143×
calcSol_cols(Grid*, Domain*, Cells*, double*, double*, double*, double*)
51.57%
(28.77%)
5143×
calcSol_rows(Grid*, Domain*, Cells*, double*, double*, double*, double*)
41.47%
(19.05%)
5143×
calcError(Grid*, Domain*, Cells*)
6.60%
(6.60%)
5143×
22.79%
13150651×
22.41%
13150651×
solve_thomasC(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
22.79%
(22.79%)
13150651×
solve_thomasR(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
22.41%
(22.41%)
13150651×

Figure 18: **Call-graph of the cache friendly implementation with a 1% threshold**

This new implementation has low usage of auxiliary functions (like *exp* and *fonte*) as they do not appear now in the call-graph. The total difference between rows and columns is also lower, especially in the Thomas algorithm, since the difference is minimal between the 2 steps.

After optimising the program to use a better memory scheme, additional optimisations were made concerning memory accesses and reducing the overall usage of variables. The new operations to be made were recorded in Table 9.

Operations Done: Once/Outer Loop/Inner Loop	Cache-Friendly Implementation	Loop Optimisation
Add/Sub	0 / 0 / 16	1 / 0 / 11
Mult	1 / 1 / 9	3 / 1 / 4
Div	4 / 0 / 0	4 / 0 / 0

Table 9: **Almost half of the operations done inside the functions *calcSol_rows/cols* were now done outside the loop, decreasing the time spent inside the loop**

After these changes, the code should theoretically perform the loops without accessing the memory as much (since fewer operations were used and some array locations were stored on variables, being possible to store them on registers). Due to this reduction of the overall number of operations, the program performance improved.

Some modifications were also made on the Thomas algorithm. The first change relates to the division on the first step. After removing the division in the first step, a batch of 4 elements are solved at a time, removing the data dependencies between different batches. After this, the division is made for each cell. Making the division (the heaviest operation done in this loop) on another vectorizable loop, so these could be made in batches using the AVX present on this machine. The following code describes these changes:

Algorithm 3 Thomas Algorithm without data dependencies between divisions

```
1:  function THOMAS ALGORITHM
2:  First Step:
3:  loop for each line
4:      $Line_k = Line_{k-1} * A_k - Line_K * B_{k-1}$
5:  Second Step:
6:  loop for batch of 4 lines
7:      $P_k = B_k$
8:      $P_{k-1} = P_k * B_{k-1}$
9:      // Do for the other two lines ...
10:     $Q_k = Z_k - C_k * Sol_k^{iteration-1}$
11:     $Q_{k-1} = P_{k-1} * Z_{k-1} - C_k * Q_k$
12:     // Do for the other two lines ...
13: loop for each line
14:     $Sol_k^{iteration} = Q_k / P_k$
```

Using this implementation for the Thomas algorithm, a new call-graph was created and the results were the following:

This implementation has less time spent on the ADI method in itself because of the introduction of the optimisations that made the number of operations lower.

After the optimisations some small rearranges to the code were made. The arrays are forced to be aligned and anti-aliasing is forbidden. With these changes it was possible to use vectorization with better performance.

With these new changes to the code, some of the operations that were repeatedly performed inside the loops were now being done only once, therefore reducing the overall amount of operations to do.

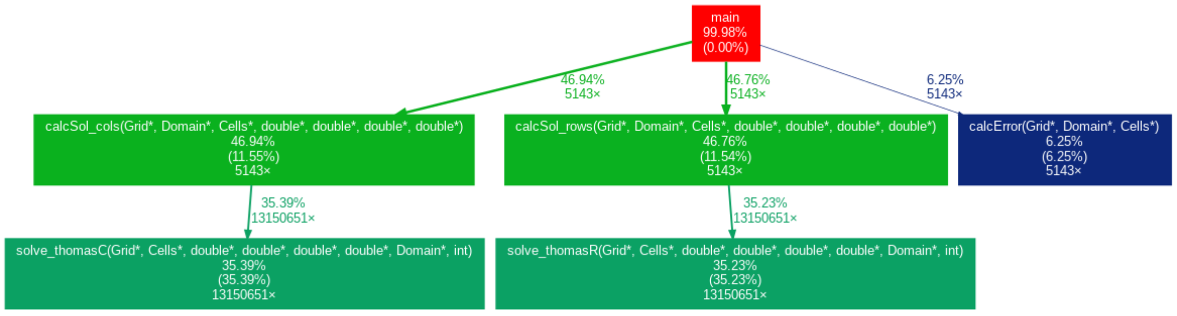
main
99.98%
(0.00%)
46.94%
5143×
46.76%
5143×
6.25%
5143×
calcSol_cols(Grid*, Domain*, Cells*, double*, double*, double*, double*)
46.94%
(11.55%)
5143×
calcSol_rows(Grid*, Domain*, Cells*, double*, double*, double*, double*)
46.76%
(11.54%)
5143×
calcError(Grid*, Domain*, Cells*)
6.25%
(6.25%)
5143×
35.39%
13150651×
35.23%
13150651×
solve_thomasC(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
35.39%
(35.39%)
13150651×
solve_thomasR(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
35.23%
(35.23%)
13150651×

Figure 19: **Call-graph of the loop optimisations implementation with a 1% threshold**

The use of vectorisation on the ADI and Thomas algorithm (mostly vectorising the divisions) made the code perform better, even with the disadvantages of the usage of the AVX (like the slower clock frequency for the whole program).

4.1.3 *Parallel approach*

Multicore Approach

One important aspect is to study the number of threads to run with the program and to start testing a parallel approach for GPU computing.

The first approach for a parallel algorithm was using the ADI way of splitting the problem into sub-problems and having each thread solve each sub-problem at a time since they are independent of each other. This approach also solves the tridiagonal matrices after computing the line.

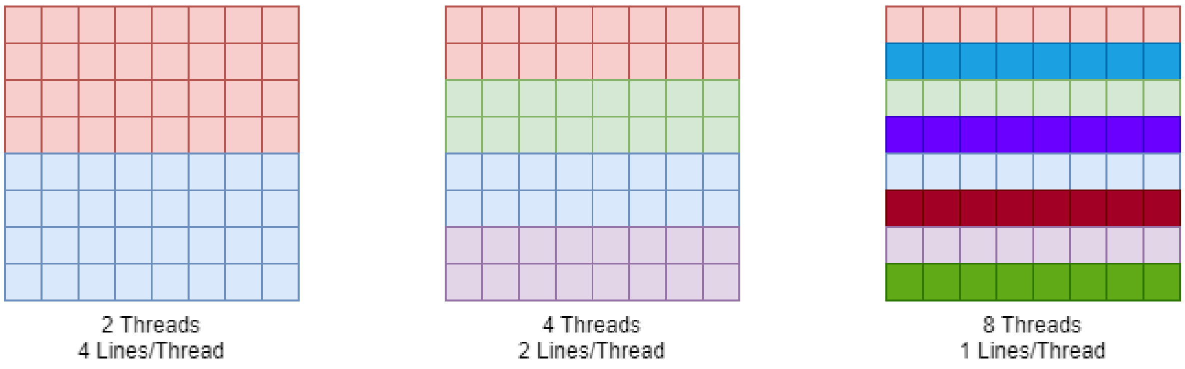

Figure 20: **Thread scheme for a CPU device. Each line is colour coded by what thread will solve it. This scheme only accounts for one step, as the column wise step work division is similar.**

This implementation used the OpenMP API to make use of the shared memory paradigm. Using this API also allows the use of various scheduling schemes (explained on Section 3.1). The results of each scheduling type with various thread numbers will be showcased in Chapter 5.

This approach does not have a parallel error calculation, therefore a new method was used to calculate the error in parallel. This parallel error calculator divides the grid into N sized chunks, where $N = GridSize/\#Threads$. After dividing the grid, each thread will look for the highest error in each chunk. After getting all the highest errors in each chunk in parallel, the master thread sequentially gets the highest error in the selected errors.

Algorithm 4 Parallel Error Checker

```
1: function PARALLEL ERROR CHECKER
2: First Step:
3: max = 0
4: loop for each element assigned to this thread
5:     $if sol_k > max : max = sol_k$
6: $maxvalues_{thread} = max$
7: Second Step (only for master thread):
8: max = 0
9: loop for each thread
10:     $if maxvalues_{selectedthread} > max : max = maxvalues_{selectedthread}$
```

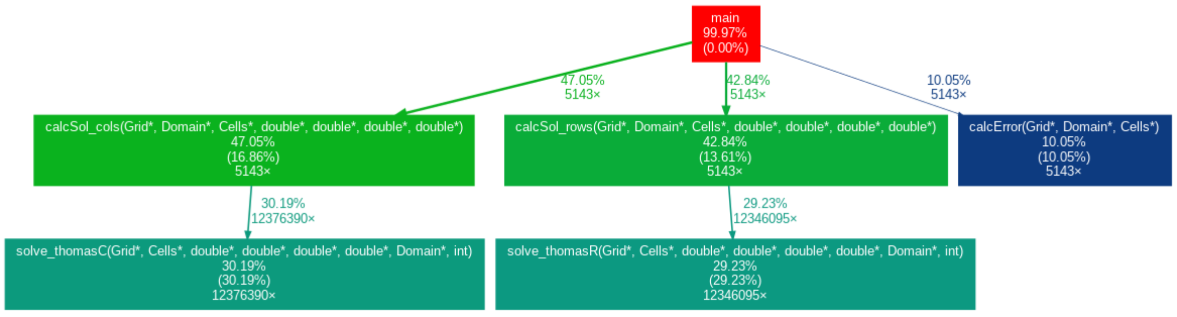
main
99.97%
(0.00%)
47.05%
5143×
42.84%
5143×
10.05%
5143×
calcSol_cols(Grid*, Domain*, Cells*, double*, double*, double*, double*)
47.05%
(16.86%)
5143×
calcSol_rows(Grid*, Domain*, Cells*, double*, double*, double*, double*)
42.84%
(13.61%)
5143×
calcError(Grid*, Domain*, Cells*)
10.05%
(10.05%)
5143×
30.19%
12376390×
29.23%
12346095×
solve_thomasC(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
30.19%
(30.19%)
12376390×
solve_thomasR(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
29.23%
(29.23%)
12346095×

Figure 21: **Call-graph of the latest implementation with a 1% threshold**

After implementing and tuning the code using best practices, it is possible to see how the difference between both steps of the ADI method (columns *vs.* rows) is minimal (less than 3%). The function that takes the most time is now Thomas algorithm (almost 60% of the whole program). The new function to calculate the error is also slower when comparing to their sequential version (observing only relative time between sequential and parallel) since a piece of the algorithm, even with vectorization, is still strictly sequential.

4.2 TARGETING GPU DEVICES

With an efficient implementation of the ADI using Thomas algorithm as the auxiliary method for the CPU device, the code for the GPU was developed. This code was based on the efficient methods used for the CPU, but the different tridiagonal solvers used could give the algorithm some extra performance. Using a parallel reduction for the error checker could also be an improvement.

4.2.1 *Tailoring ADI*

The ADI method has various steps, and the way device and host should communicate is a relevant factor to obtain the best performance with this algorithm. The device can use synchronisation only inside an SM and without extra recurrent locks. It is not possible to synchronise the whole device. Therefore, the synchronisation is done with the host. Figure 22 shows a possible scheme between these.

While having the host synchronising the device, the error is computed in the device and memory transfers from device to host are not cheap performance-wise. Besides this, it is possible to make memory transfers asynchronous. To make them asynchronous, instead of computing the error at the end of the iteration and transfer it to the host, it is possible to compute the error between both ADI steps and transfer the error value while the device solves the last step. This hides the latency of the host-device transfers.

The ADI algorithm for the GPU device was tailored so every thread was solving a single line/column at a time. With the use of the GPU high number of threads (having thousands available), the thread schemes will be different. Instead of assigning a single thread per line, a full SM was used.

By dividing the lines into each SM it is also possible to improve the use of shared memory (a faster memory than the global memory, which is private to each SM) and the GPU capacity since the number of threads deployed at a time can be far greater than the number of lines/columns to be processed.

Since the number of elements in a line/column in the ADI method might not be the same as the number of threads in a SM, more than 1 element is scheduled for each thread.

The ADI is computationally lighter when compared to the tridiagonal solvers to be used. So while the other tridiagonal solvers are computationally heavy when compared to the Thomas algorithm, they can be parallelised and end up performing better than the sequential one. It is also possible to create and store the data that fits into the shared memory, reducing the number of total accesses to the global memory from the start.

4.2.2 *Tridiagonal Solvers*

With the sheer number of threads to be used in a GPU, a different approach will be tested using parallel tridiagonal solvers instead of the sequential Thomas algorithm.

These tridiagonal solvers will be implemented using CUDA and OpenCL.

CUDA Implementation

While the tridiagonal solver used as an auxiliary to the ADI method in the CPU was the Thomas algorithm due to it being the least computational intensive, working with GPU it is possible to use multi-threading inside each line more effectively. Therefore the parallel tridiagonal solvers were tested.

For this analysis, the base algorithms were studied and tested, namely, the CR, the PCR and the SPIKE algorithm. The RD algorithm, as seen before, can not be used in the ADI context due to its limitations (will not solve diagonal dominant tridiagonal matrices [Ömer EGECIOGLU et al. (1990)]).

The base implementations of CR and PCR were based on the book [Hwu (2011)], the following code describes these algorithms:

This algorithm also takes some consideration about bit manipulation, since the variable stride and activeThread are always a power of 2, it is possible to shift the bit to the right to divide by 2 and shift the bit to the left to multiply by 2. It also reduces the number of divisions by storing the inverse of the division (tmp1 and tmp2) and multiplying the values by it instead of dividing.

Algorithm 5 CR Algorithm

```
function CR ALGORITHM
First Step:
activeThreads = N/2
stride = 1
loop iterations<numberOfSteps
    stride = stride*2
    delta = stride / 2
    i = stride * element + stride - 1
    iLeft = i - delta
    iRight = i + delta
    tmp1 = a[i] / b[iLeft]
    tmp2 = c[i] / b[iRight]
    b[i] = b[i] - c[iLeft] * tmp1 - a[iRight] * tmp2
    z[i] = z[i] - z[iLeft] * tmp1 - z[iRight] * tmp2
    a[i] = -a[iLeft] * tmp1
    c[i] = -c[iRight] * tmp2
    activeThreads = activeThreads/2
Second Step:
loop iterations<numberOfSteps
    delta = stride / 2;
    i = stride * idx + stride / 2 - 1
    if (i == delta - 1) x[i] = (z[i] - c[i] * x[i + delta]) / b[i]
    else x[i] = (z[i] - a[i] * x[i - delta] - c[i] * x[i + delta]) / b[i]
    stride = stride/2
    activeThreads = activeThreads*2
```

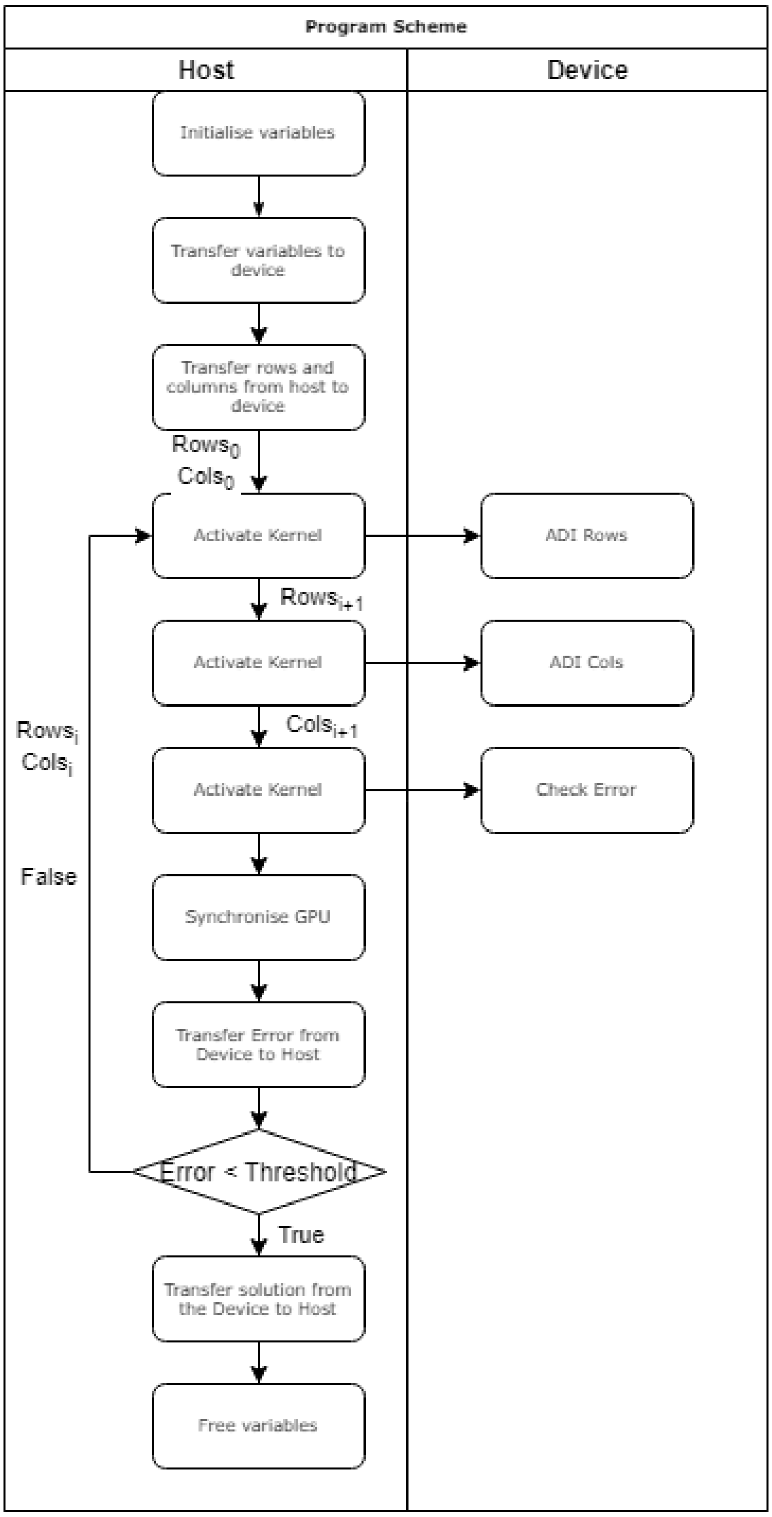

Figure 22: **Workload distribution between the host and the GPU.**

The PCR algorithm performs mostly the same calculations as CR. The major difference is that PCR does not disable threads and only does a set of computations in the second step, instead of having a loop.

Algorithm 6 PCR Algorithm

```
function PCR ALGORITHM
First Step:
delta = 1
i = element
loop iterations<numberOfSteps
    iLeft = i - delta
    iRight = i + delta
    tmp1 = a[i] / b[iLeft]
    tmp2 = c[i] / b[iRight]
    //Only update the values at the end of iteration
    b[i] = b[i] - c[iLeft] * tmp1 - a[iRight] * tmp2
    z[i] = z[i] - z[iLeft] * tmp1 - z[iRight] * tmp2
    a[i] = -a[iLeft] * tmp1
    c[i] = -c[iRight] * tmp2
    delta = delta * 2
Second Step:
addr1 = element;
addr2 = element + delta;
tmp3 = 1/(b[addr2] * b[addr1] - c[addr1] * a[addr2])
x[addr1] = (b[addr2] * z[addr1] - c[addr1] * z[addr2]) * tmp3
x[addr2] = (z[addr2] * b[addr1] - z[addr1] * a[addr2]) * tmp3
```

The PCR needs to assign a thread to each line of the tridiagonal solver, therefore the vanilla algorithm can not work on the 2048x2048 tridiagonal matrix (since there are only 1024 threads per SM). For testing purposes, the algorithm will be expanded to a hardcoded version with 2 lines per thread. For larger matrices, the CR and PCR implementations require additional tweaking from the originals. For the CR, stridden lines will be assigned per each thread. For the PCR, a temporary array will be created to store all the temporary variables during run time, which will have a great performance penalty, making it not viable for using it with grid sizes larger than 2048x2048.

The SPIKE algorithm is tailored for a parallel distributed memory scheme, but it has a larger overhead compared to other algorithms: it first partitions the problem (needs extra memory allocations and extra computations) and then merges them at the end. The algorithm is described below.

With the usage of CR, PCR and the Thomas algorithm it is possible to test the two new hybrid algorithms, these being the CR-PCR and the PCR-Thomas. The CR algorithm strives to reduce the number of active equations, and PCR strives to reduce the number of steps. With the conjunction of these two, it is possible to first reduce the number of equations, and then use PCR to solve these more efficiently. It is also possible to use Thomas algorithm with PCR since the PCR reduces the number of equations on a system by creating more independent systems. Then it is possible to use more than one Thomas algorithm at a time, one for each system.

With the introduction of the CR-PCR and the PCR-Thomas algorithm, a new implementation was made. This implementation reduces the tridiagonal matrix into a smaller matrix with the CR by reducing the number of elements to test, then the algorithm PCR-Thomas will solve this matrix.

Algorithm 7 SPIKE Algorithm

```
1:  function SPIKE ALGORITHM
2:  First Step:
3:  loop for each needed partition
4:      newPartition = new Array
5:      copyValues(initialArray,newPartition)
6:  Second Step:
7:  loop for each partition
8:      Solve outer Elements of partition
9:  Third Step:
10: loop for each element of partition
11:     Solve element using backward substitution with solved elements
12: Fourth Step:
13: loop for each partition
14:     copyValues(newPartition,initialArray)
```

Finally, the remaining elements are solved with the second step of the CR. An example with a system of 16 linear equations is shown in Figure 23. This algorithm will be called CR-PCR-Thomas.

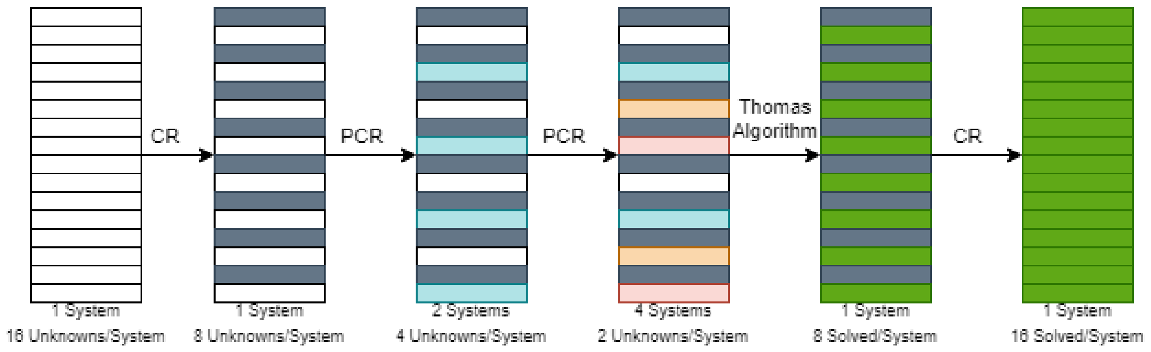

Figure 23: **Example of how CR-PCR-Thomas algorithm solves a system of 16 linear equations.**

OpenCL Implementation

While CUDA only works with GPU from NVidia, the OpenCL is more open to the number of devices it can work on [Fang et al. (2011)]. Therefore, for more widespread availability and testing purposes, a version of the code was ported for OpenCL. This new version is a direct translation from the CUDA code. The initial work on this version were the tridiagonal solvers.

The translation from CUDA to OpenCL was translating most of the kernels from the CUDA code to OpenCL. With the usage of C++ on both API, the sequential code on the CPU device was not changed, only changing how the setup was done.

When using OpenCL the compilation of the kernels was done at the startup of the program, providing an extra overhead to this algorithm version. The OpenCL also need to set up an environment and load the specific platform variables, so that setup was also done beforehand.

4.2.3 *Error Checking*

In the CPU implementation of the ADI algorithm, to look for the largest difference between expected results and computed results (error checking), the grid is divided into chunks and every available thread will get the maximum element in each chunk. In the end, a single thread will get the maximum between all maximums. While this works on a CPU with a low number of threads and vectorizing this search using their vector extensions, the different number of threads available in a GPU architecture leads to a different algorithm. Therefore, the GPU used a modified sum reduction[2].

This reduction was modified to, with a mask, chose which element has a higher error (instead of simply adding them). This new algorithm has a complexity of $O(N) = log_2(N)$. A modification to the tridiagonal solver was also made. Instead of computing the error in the error checker, it is computed while the solver computes the value of the cell. With this difference, the error checker can be performed more than once without voiding the previous results.

This algorithm loads the cells assigned to the SM to the shared memory before starting. After loading the elements, the threads assigned halve the number of errors to search for every iteration. Each thread checks if it is sleeping, and if not, chooses the element to work with using a modulo operation.

Since the modulo operation is computational heavy, threads will store the elements into the current thread number. This makes the memory contiguous, and the thread does not need to compute the value to look for. This reduces the number of operations per iteration and the thread only need to check if it is active or not.

To reduce the number of kernel launches, instead of storing a single element per thread, the thread can check 2 elements when loading the values to the shared memory and only store the larger error. This modification halves the number of kernel launches since each SM will process twice the number of threads available instead of only the number of threads.

The following code describes the latest version of the algorithm, using shared memory, loading the largest element of 2, masking the conditional and not using the modulo operator:

Algorithm 8 Max reduction algorithm

```
1: function MAXREDUCTION ALGORITHM
2: Allocate and Load memory:
3: SHARED sdata
4: i = threadNumber
5: i1 = elementToAnalyse
6: i2 = otherElementToAnalyse
7: sdata[i] = i1<i2?i2:i1 //Masked conditional to max
8: loop sqrt(number of elements)
9:     sdata[i] = sdata[i]<sdata[next]?sdata[next]:sdata[i]
10: if masterThread
11:     return sdata[0]
```

2 Based on nVidia's Optimizing Parallel Reduction in CUDA on http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

This version of the code has more operations overall since some cells are checked more than once (every iteration). But this code is parallel from beginning to end.

5

EXPERIMENTAL VALIDATION AND EVALUATION

In this chapter, results were recorded. Some of the ways to improve its performance are noted for the final GPU implementation. This algorithm was also ported to CUDA and OpenCL code to be tested on GPU with different micro-architectures from NVidia.

All the tests and results were gathered using compilers and programs that are listed on a tooling appendix of this dissertation.

5.1 TESTBED

To test the sequential code a testing environment was selected. The system that was chosen is a compute node in the SeARCH cluster of *Universidade do Minho (UM)*. The nodes in this cluster have a mix of different multicore and manycore devices, with or without computing accelerators.

For the sequential runs, the SeARCH-662 node was selected: it has a dual multicore Intel Xeon device and a NVidia GPU accelerator device. This node's specifications are in Table 10.

Three systems were selected to test the parallel code: the SeARCH-662 node (to compare with the sequential code), the SeARCH-881 node (based on more updated multicore Xeon devices with better vector capabilities, namely AVX-512) and a node with the Intel KNL device (a manycore CPU device). The specifications of these nodes are in Table 10.

To test the algorithms for the GPU, other hardware was also utilised, namely, two different GPU cards from NVidia, with two different microarchitectures: 1 with a Kepler microarchitecture and the other with a Pascal microarchitecture.

The results gathered in this dissertation used the K-Best algorithm to determine the time, K-Best specifications used was from 10 runs and the 3 best times had to have a difference of less than 5% to be usable. Then the average of those 3 values was stored.

5.2 EXPERIMENTAL OUTCOMES ON CPU DEVICES

Once the CPU ADI method was implemented and validated on homogeneous nodes with Intel-based devices. The ADI method used Thomas algorithm as its tridiagonal solver in the CPU devices as explained in Chapter 4 execution times were measured, starting with the Ivy Bridge CPU devices. The measured execution times are in Table 12.

Checking the table above it is possible to see how every new implementation performed better than the previous ones, comparing the first version and the single exp one, there was a speedup of around 1.7 but the cache accesses and misses did not decrease as much since the exp function stopped being

SeARCH Node	SeARCH-662	SeARCH KNL	SeARCH-881
Architecture	Ivy Bridge	Knights Landing	Skylake
CPU Chip	Xeon Processor E5-2695v2	Xeon Phi Processor 7210	Xeon Gold 6130
# Sockets	2	1	2
#Cores/Device	12	64	16
SMT/HT	Yes	Yes	Yes
#Threads/Device	24	256	32
Clock Frequency	2.4 GHz	1.3 GHz	2.1 GHz
Turbo Frequency	3.2 GHz	1.5 GHz	3.7 GHz
L1I Cache Size	12 x 32 KiB	64 x 16 KiB	16 x 32 KiB
L1D Cache Size	12 x 32 KiB	64 x 16 KiB	16 x 32 KiB
L2 Cache Size	12 x 256 KiB	38 x 1 MiB	16 x 1 MiB
L3 Cache Size (Shared)	12 x 2.5 MiB	—	16 x 1.375 MiB
Vector Extensions	AVX	AVX512	AVX512
Memory Channels	4	6	6
RAM Clock Rate	1600 MHz	2400 MHz	2666 MHz
Memory Bandwidth	50 GB/s	115 GB/s	128 GB/s
OS	CentOS 6.3	CentOS 6.3	CentOS 7.5
Others	—	MCDRAM (400 GB/s) Cluster Mode: Quadrant Memory: Flat Mode	—

Table 10: **Specifications of the cluster CPU nodes**

Microarchitecture	Kepler (K20m)	Pascal (GeForce GTX 1070)
CUDA Cores	2496	1920
Streaming Multiprocessors	13	15
Threads per SM	1024	1024
Clock Frequency	705 MHz	1746 MHz
Single to Double Precision Rate	1/3	1/32
Memory Bandwidth	208 GB/s	256 GB/s
Shared L2 Cache	1280 KiB	2048 KiB
Local Memory per SM	48 KiB	48 KiB

Table 11: **Specifications of the GPU devices**

Implementation	Time(s)	Cache L3 Accesses	Cache L3 Misses	Cache L3 Miss Rate
Regular	2618	102 299 285 882	32 592 111 835	31.86%
Cache-Friendly	1909	96 716 170 945	21 108 224 257	21.83%
Single Exp	1856	96 718 215 186	21 236 405 821	21.96%
Loop Optimisations	1542	37 764 164 967	9 192 664 442	24.34%
Vectorised Code	**1336.0**	45 847 516 068	8 878 739 151	19.37%

Table 12: **Comparison of all CPU sequential code using doubles and the Ivy Bridge CPU device**

called as much but each cell had its value stored on the memory, forcing more values to be loaded from the main memory.

Comparing that version to the vectorised one it is possible to notice another speedup of 1.4 and the number of loads reduced drastically, this was because some values were not being accessed so much because of the reduced number of operations inside the loops. The speedup of the vectorised code was not a big improvement as could be expected, but it is important to remark the usage of only the base AVX and the usage of doubles, which reduces the vectorisation performance.

5.2.1 *Multicore CPU Devices*

The validation and evaluation of the parallel implementation were performed with a various number of threads, from 2 to 48 threads. This was doable due to having 2 sockets with 1 CPU device on each and each having 12 physical cores and using 2 threads per core with *Simultaneous MultiThreading (SMT)*. The execution times were placed in Table 13.

Number of Threads	1	2	4	6	12	24	48
Time (s)	1336.0	737.1	505.6	343.2	252.9	**242.0**	355.2

Table 13: **All tested number of threads using Ivy Bridge CPU**

As Table 13 shows, it is possible to see that the best execution time is achieved when using 1 thread per core and both sockets are used. When using SMT the measured times got worse, due to the larger overhead of the SMT usage.

This latest version of the code has access to SIMD with AVX but does not benefit from the best available version of the AVX. The Skylake CPU used had access to a newer version of AVX, the AVX-512. To test the Skylake, a few changes to the executable were made, namely, change the compiler flags to use AVX-512 instead of AVX.

To test the scalability of the code using the Skylake, different scheduling options were used and these were recorded in Table 14 and a call-graph was created (Figure 24).

Number of Threads	2	4	8	16	32	64
Static (s)	525	332.5	179.3	94	59.8	63.4
Dynamic (s)	588.5	309	178.9	96.8	58.6	65.2
Guided (s)	570.5	295	171.3	89.5	**57.5**	70.2

Table 14: **Times measured using Skylake with three different scheduling types**

With the Table 14 it is easily seen that the best performance is achieved using the guided schedule, showing as the better scheduler between the threads among the others. This scheduling type has a larger overhead than static but scales better with the increasing number of threads since the work is better balanced into chunks. The best number of threads is when using both sockets and every physical core is busy with a single thread. When using SMT it is possible to notice a decline in performance, due to the overhead in having more threads without the speedup they could provide.

As expected, the functions of the code that took most advantage of the vectorization (*calcError*) had performance gains, while the parts that did not use vectorization (especially the Thomas algorithm

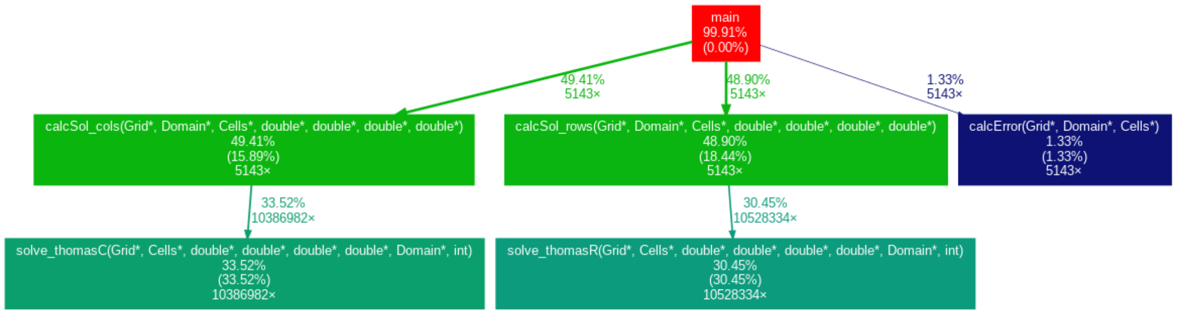
main
99.91%
(0.00%)
49.41%
5143×
48.90%
5143×
1.33%
5143×
calcSol_cols(Grid*, Domain*, Cells*, double*, double*, double*, double*)
49.41%
(15.89%)
5143×
calcSol_rows(Grid*, Domain*, Cells*, double*, double*, double*, double*)
48.90%
(18.44%)
5143×
calcError(Grid*, Domain*, Cells*)
1.33%
(1.33%)
5143×
33.52%
10386982×
30.45%
10528334×
solve_thomasC(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
33.52%
(33.52%)
10386982×
solve_thomasR(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
30.45%
(30.45%)
10528334×

Figure 24: **Call-graph using 32 threads and guided scheduling in Skylake with a 1% threshold**

inner calculations) had some performance loss due to the slower clock frequency, making the difference in times more noticeable.

5.2.2 *Manycore CPU Devices*

The Intel KNL device was used to evaluate the ADI scalability in manycore architectures, namely with reduced cache features (the KNL chip has no L3 cache), using the Skylake code to test this chip. Once again, different thread scheduling was tested (some with at least 12 threads) and these new times are in Table 15.

Number of Threads	2	4	6	8	12	16	24	32	48	64
Static Time (s)	2023	1257	954	768	603	588	530	552	515	540
Dynamic Time (s)	—	—	—	—	630	568	518	542	508	566
Guided Time (s)	—	—	—	—	587	535	**490**	537	494	582

Table 15: **Times measured using KNL with three different scheduling types**

Looking at the table it is possible to notice that after a certain threshold the ADI stops scaling, even with different scheduling types. When using KNL there is a new type of memory available, a *High Bandwidth Memory (HBM)* called MCDRAM. This RAM is only available in KNL and the bandwidth is almost 4 times larger when compared to the regular RAM (information in Table 10). Due to the machine memory being in flat mode no changes to the code should be necessary, but two other approaches were tested: one forcing the memory to allocate in the MCDRAM, and another that allocates the memory in the MCDRAM and then aligns it. The results can be seen in Table 16. All measured times were performed with at least twelve threads, due to the time plateauing around these number of threads.

Number of Threads	12	16	24	32	48	64
Regular Time (s)	586.7	535.3	**490.3**	537.3	493.9	581.8
Forced Alloc Time (s)	599.3	552.75	509.6	506.4	505.9	558.0
Alloc & Align Time (s)	742.9	663.9	574.4	569.8	501.4	537.8

Table 16: **Times measured while allocating with different allocation methods.**

As expected, since the memory mode was set to flat, the memory was already aligned and set in the MCDRAM, while forcing has an overhead, so using them is unnecessary and makes the code less performant. A call-graph was made for the best performance on the KNL (Figure 25).

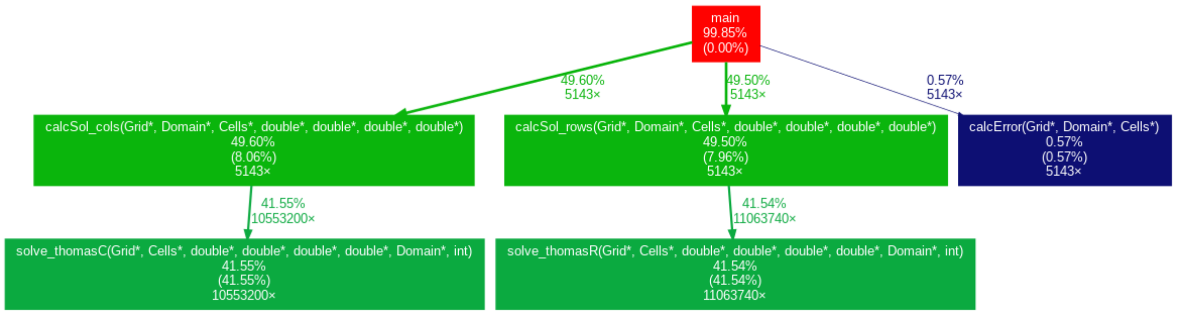
main
99.85%
(0.00%)
49.60%
5143×
49.50%
5143×
0.57%
5143×
calcSol_cols(Grid*, Domain*, Cells*, double*, double*, double*, double*)
49.60%
(8.06%)
5143×
calcSol_rows(Grid*, Domain*, Cells*, double*, double*, double*, double*)
49.50%
(7.96%)
5143×
calcError(Grid*, Domain*, Cells*)
0.57%
(0.57%)
5143×
41.55%
10553200×
41.54%
11063740×
solve_thomasC(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
41.55%
(41.55%)
10553200×
solve_thomasR(Grid*, Cells*, double*, double*, double*, double*, Domain*, int)
41.54%
(41.54%)
11063740×

Figure 25: **Call-graph using the best KNL algorithm with a 1% threshold**

This call graph shows that the KNL has a serious slowdown on the Thomas algorithm (from taking almost 60% of the relative time to more than 80%). This function uses a lot of memory since it will read from different arrays simultaneously, change these values and use them later on, making the memory latency more noticeable.

5.2.3 *CPU Results Discussion*

The multicore implementation of the ADI code is scalable on the architectures studied. When using a dual-socket the implementation treats the memory allocation on the thread level instead of a single memory allocation. This makes the use of a dual-socket more efficient than the use of a single socket, even when hiding the latency of the NUMA architectures by using the local RAM for each thread more precisely than a simple memory allocation. When using SMT on these architectures there is a penalty, most likely due to the physical cores being already busy when using a single thread per core and the overhead of the extra threads makes it not worthwhile.

When using the manycore implementation the algorithm performs differently than expected. The ideal number of threads was 24 threads, which is only using a thread per 3 physical cores, and when using 48 threads, which is using 2 threads per 3 physical cores. This implementation does not scale above that number of threads. Looking at the call-graph for this implementation (Figure 25) it is possible to notice how the Thomas algorithm takes over 80% of the time of the whole algorithm.

One of the reasons for the slowdown of the code can be explained by the lower clock frequency of the KNL. The KNL clock frequencies were recorded in Table 17, when using different threads and different instruction sets (like AVX or SSE4.2). These clock frequencies were recorded using the perf tool while running the DGEMM algorithm[1].

While studying the possible slowdowns on the scalability for the KNL, the memory bandwidth needed for full usage of the available cores was calculated. Using the clock frequency of Table 17, the clock frequency used will be 500 MHz. Using the formula $\#Threads * \#VPUPerThread * VectorSize * Clock\_Frequency$ the theoretical memory bandwidth for continuous data processing is around 4000

1 Test DGEMM program from https://github.com/jdmccalpin/simple-MKL-DGEMM-test

Clock Frequency	SSE4.2	AVX	AVX2	AVX512
1 Thread	1440 MHz	1450 MHz	1440 MHz	1410 MHz
24 Threads	998 MHz	980 MHz	580 MHz	580 MHz
48 Threads	912 MHz	880 MHz	518 MHz	507 MHz
64 Threads	1020 MHz	910 MHz	497 MHz	490 MHz

Table 17: **Clock frequencies recorded using perf while running a DGEMM algorithm**

GB/s. This is 10x larger than the theoretical peak bandwidth of the MCDRAM[2]. Looking at Table 15 it is possible to notice how the algorithm stops scaling as well around the 8 to twelve threads mark. This is indeed the same number of threads that when using the formula above should stop having enough bandwidth to supply all the data needed, increasing the time spent waiting for data.

When analysing the multicore architectures, it is possible to see that the ADI scales well using more threads until they start using SMT. This happens because of the usage of good NUMA practices, letting each thread store the data in the RAM available in each socket. The performance gain from the Ivy Bridge to the Skylake CPU is also possible with the upgraded clock frequency (the best skylake time has a speedup of 2 when compared to the best Ivy bridge time). This happens because it has a higher core number (running half extra threads at a time) and an upgraded version of AVX (even with a clock frequency penalty, the bandwidth is larger (almost 3 times larger), which makes the skylake have fewer data penalties. The cache information was recorded and stored in Table 18.

	Cache Accesses	Cache Misses	Current Miss Rate	Total Miss Rate
Cache L1	1 497 170M	230 170M	15.4%	15.4%
Cache L2	230 170M	24 616M	10.7%	1.6%
Cache L3	24 616M	8 350M	33.9%	0.6%

Table 18: **Number of cache accesses and cache misses on the dual Skylake server.**

As seen in Table 18, the better use of the cache makes the algorithm keep most of the needed data on-chip (almost 99% of the requested data is stored on the first two cache levels). With the high memory usage of the ADI and the whole problem does not fit in the L3 cache, one third of all L3 accesses miss the cache. This problem can not be avoided since each iteration needs to process all the data while using a smaller problem (that fits itself on the L3 cache) could reduce most of the memory accesses to the L3 cache (besides the first iteration).

5.3 GPU RESULTS

5.3.1 *CUDA Implementations*

After implementing all code for GPU devices, the tridiagonal solvers times were recorded. The first results gathered were to discover the best point to size to switch between algorithms in the hybrid algorithms. To do this a test was run for all the sizes (this information is stored in Appendix A) and the best switching size was stored in Table 19.

2 Peak bandwidth calculated by Intel in https://software.intel.com/content/www/us/en/develop/articles/multi-channel-dram-mcdram-and-high-bandwidth-memory-hbm.html

Matrix Size	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
Break point in CR-PCR	16	32	64	128	256	512	1024
Break point in PCR-Thomas	2	2	2	4	4	4	8

Table 19: **Best point to switch from the first algorithm to the other using CUDA in the Kepler GPU.**

With the best switching sizes known, the times were gathered and stored for different sizes (from 32x32 up to 2048x2048) and tridiagonal solvers (CR, PCR and hybrids) in Table 20.

Size	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
CR (ms)	0.79	0.99	1.11	1.34	1.67	2.00	2.53
PCR (ms)	0.65	0.72	0.80	0.84	0.97	1.42	2.48
CR-PCR (ms)	0.93	1.00	1.18	1.22	1.36	1.74	2.83
PCR-Thomas (ms)	**0.60**	**0.72**	**0.77**	**0.84**	**0.91**	**1.31**	**2.10**

Table 20: **Execution times for CR, PCR and Thomas algorithms, including hybrids for GPU, using doubles and CUDA on the Kepler GPU.**

In the algorithm CR-PCR, 2 cells were assigned to each thread, and while working on the PCR some of the threads simply become inactive. Because of this, the PCR is less performant the fewer threads are used in this algorithm. Therefore the higher use of the PCR makes the work more balanced, having a better performance overall. In the algorithm PCR-Thomas the usage of Thomas makes the algorithm more sequential, but unlike CR-PCR, the threads finish their assigned work, so they do not become inactive. While having a smaller Thomas algorithm size makes the algorithm more parallel, the usage of a larger size on the Thomas algorithm makes the total operations to perform lower, so encountering a middle size (using the Thomas algorithm size as 8) is beneficial for the algorithm. The number of threads working on the Thomas algorithm using the size as 8 is 256 threads, that is 8 groups of warps. Using more than 1 warp at a time also has benefits, since some threads will be sleeping while waiting for their data, other groups of threads can work, keeping the SM busy.

Analysing the Table 20 it is possible to notice how the best algorithm for this configuration is the PCR-Thomas. This algorithm has less inactive time (by keeping the number of active threads high) and reduces the total number of operations enough to make it worth using more than the other methods.

Studying the difference between the algorithms CR and PCR-Thomas, it is possible to notice the switching size at matrix sizes of 512 (up to this point, the CR scaled worse, after this point, the CR-Thomas scales worse). Because of that, the change from the CR algorithm will reduce the 2048x2048 system into a 512x512 system. After this change, the switching size between PCR and Thomas algorithm needs to be studied. Table 21 shows how the algorithm performs with different configurations.

It is possible to see that the best performance is recorded when the Thomas size is 8. With this configuration, the CPU uses 64 threads (there are 64 systems of 8 elements each) to solve the Thomas algorithms. This leaves 2 working warps on the GPU, while one is performing operations, the other can be waiting for data to be retrieved, hiding the memory latency and reducing the number of operations to be performed.

Thomas Algorithm Size	2	4	8	16	32	64	128	256
Time (ms)	2.28	2.27	**2.25**	2.27	2.70	3.12	3.50	6.23

Table 21: **Execution times of the CR-PCR-Thomas algorithm, on a 2048x2048 system, with 2 CR steps, reducing the total size to 512x512, and using doubles and CUDA on the Kepler GPU.**

Next, a scalability test was made for the SPIKE algorithm. This algorithm should have high scalability but a large overhead [Kjelgaard Mikkelsen and Manguoglu (2008)]. The algorithm will be compared to the CR algorithm since as Table 23 shows when comparing to other algorithms it has better scalability. The times were recorded on Table 22.

Grid Size	32 x 32	2Ki x 2Ki	32Ki x 32Ki	4Me x 4Me
CR (ms)	0.79	2.53	51.76	6896.60
SPIKE (ms)	4.43	5.12	11.40	418.62

Table 22: **Execution times for the CR and the SPIKE algorithms, using doubles and CUDA on the Kepler GPU**

As noticed in Table 22 the SPIKE algorithm scales well for very large matrices (32Kix32Ki) when compared to CR. However, the larger overhead hinders its performance for matrix sizes up to 2Ki, which are usually the size of matrices used to solve problems with convection-diffusion equations (like heat-transfer in 2D environmnets). This tridiagonal solver had the potential to solve systems of high dimensions, including 3D versions of the environment. Therefore the SPIKE algorithm will be discarded for following tests as it does remain competitive for the size chosen as it is too small to cover the overhead.

The execution times of all algorithms were measured on the Kepler GPU and also on the Pascal GPU. The times were recorded on Table 23.

Size	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
CR (ms)	0.31	0.37	0.43	0.50	0.57	0.67	0.80
PCR (ms)	0.19	0.26	0.26	0.32	0.48	0.80	1.49
CR-PCR (ms)	0.31	0.36	0.39	0.43	0.52	0.72	0.90
PCR-Thomas (ms)	0.20	0.23	0.26	0.31	0.43	0.67	1.10

Table 23: **Execution times for the CR, PCR and Thomas algorithms, including hybrids with GPU, using doubles and CUDA on the Pascal GPU.**

Differently from the Kepler GPU, the Pascal GPU CR algorithm scales better than all others. It has a slower start (on par with CR-PCR), but as the size grows, the CR time grows slower than all the others. This is achieved by the scalability achieved using the CR. The usage of the PCR can achieve better performance for smaller matrices because of the reduction of work in the second step, only having an equation for each cell, instead of a loop to solve every cell. The Thomas algorithm also helps to achieve better scalability with the PCR, as it reduces the number of total iterations and reduces the number of operations, having a downside of using fewer threads at a time.

The custom algorithm CR-PCR-Thomas implementation was also tested on the Pascal. Just as before, the switching size between CR and the PCR-Thomas is the size 512x512. Table 24 shows the performance of the algorithm using different Thomas algorithm sizes.

Thomas algorithm size	2	4	8	16	32	64	128
CR-PCR-Thomas (ms)	0.796	0.751	0.747	1.079	1.081	1.68	2.879

Table 24: **Execution times for the CR-PCR-Thomas algorithm, on a 2048x2048 system, with 2 steps of CR, reducing the total size to 512x512, using doubles with CUDA on a Pascal GPU**

As expected, the algorithm shows a performance likely to the PCR-Thomas, having the best performance on Thomas size set to 8. This is possible because of the number of threads to work at a time (64 threads at a time), this means that 2 warps work at a time, reducing the number of total operations (by having less PCR steps) but having enough steps to have more than 1 warp working at a time (to hide some of the memory latency).

When comparing the best CR-PCR-Thomas algorithm to the others, it is possible to notice how it performs better than all the other algorithms, this implementation takes all the advantages of all the others. First, it reduces the number of equations, reducing the whole size of the problem considerably, then the PCR divides the system of equations into various independent systems to be solved by the Thomas algorithm, using various warps to reduce the memory latency.

5.3.2 *OpenCL Implementations*

After recording the performance on all GPU devices of the CUDA code, the same was done to the OpenCL port (excluding the SPIKE algorithm).

When testing the OpenCL port of the tridiagonal solvers on the GPU, a few changes were made, and due to some differences in both API configurations, the tests using the OpenCL only performed the tridiagonal solver on 1 system at a time. Therefore the direct comparison of both implementations is misleading, but the scalability between both is not since the datatypes and matrix sizes are the same.

The following table shows the tridiagonal solvers performance using the Kepler GPU:

Size	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
CR (μs)	134.5	137.7	152.0	164.3	180.8	204.3	243.5
PCR (μs)	**100.2**	**117.7**	123.0	**126.1**	**124.1**	**151.0**	235.0
CR-PCR (μs)	126.5	132.4	131.0	133.6	143.2	157.0	**211.4**
PCR-Thomas (μs)	120.6	118.3	**122.1**	129.2	135.8	158.0	215.8

Table 25: **Execution times for the CR, PCR and Thomas algorithm, using doubles with OpenCL on a Kepler GPU**

As the CUDA implementation, the PCR and PCR-Thomas have a slightly better performance than the CR, with PCR-Thomas having the better overall results. Despite these similarities, in this version of the code, the best algorithm is now CR-PCR. This algorithm scales better than the others, starting as having the second worst performance, to having the best performance overall. This is most likely due to the different compilers and how inactive threads impact performance on both API since the inactive threads do not impact the performance as much as the CUDA counterpart.

With the code implemented, tests on the Pascal GPU were performed to check the scalability and how each algorithm performed. Table 26 has all the data recorded using a single system and the datatype used are doubles.

Size	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
CR	64.5	67.5	70.6	77.1	82.5	90.5	104.5
PCR	**43.9**	**45.1**	**46.7**	**49.3**	**55.6**	**69.9**	127.8
CR-PCR	55.9	56.3	56.9	59.8	67.0	76.0	**83.4**
PCR-Thomas	54.0	45.4	50.8	52.4	60.9	76.7	114.7

Table 26: **Execution times for the CR, PCR and Thomas algorithm, using doubles with OpenCL on a Pascal GPU**

As the CUDA code, the CR performs better than PCR and PCR-Thomas, but the CR-PCR algorithm perform better with this configuration. On this configuration, the PCR performs better than the CR for smaller matrices, but stills end up scaling worse and for the size of 2048x2048: it shows the worst performance.

5.3.3 *GPU Results Discussion*

After implementing and testing the necessary functions for the ADI method (tridiagonal solvers and the error calculation) the next step is testing everything together.

Since both GPU have a shared memory of 48 KiB, the algorithm will take advantage of it by having some of the arrays generated from the ADI be stored directly on it, removing the need to use the global memory as much as possible. To choose the arrays to store in the shared memory, all the accesses will be counted, and the arrays more used will be used. These computations are in Table 27.

Tridiagonal Solver Used	A	B	C	Z
CR	4	5	5	4
PCR	2	2	2	3
Thomas	1	3	2	3

Table 27: **Ratio between memory accesses per algorithm**

Studying the Table 27 and knowing that each array has 2048 elements (and these elements are doubles) each array has a size of 16 KiB. Therefore, each SM has a shared memory capable of storing 3 arrays. Since in every algorithm the array A is the less used the other 3 will be the ones chosen to be stored in it.

Since the ADI is an iterative method and the number of iterations may change, the times recorded will be the average per iteration, the domain to be used is a circle described in a grid and the tridiagonal solvers used will be variable for testing purpose. These times are recorded in Table 28.

When comparing the results of Table 28 with to the times of the algorithms on previous tables, the ADI using the tridiagonal solvers with the best performance show better performance overall. The disparity between times is higher this time around, mostly because of the sheer number of tridiagonal solvers used simultaneously (each step has 2048 systems running at a time). Nonetheless,

Tridiagonal solver used	CR	PCR	CR-PCR	PCR-Thomas	CR-PCR-Thomas
Kepler (ms)	38.4	27.6	65.9	26.6	31.3
Pascal (ms)	12.1	33.2	37.7	26.5	14.8

Table 28: **Execution times for various tridiagonal solvers, on a 2048x2048 grid, using doubles and CUDA. Execution time is the average per iteration.**

the introduction of the shared memory reduces drastically the number of memory accesses that need to happen, therefore, reducing the total time needed for each tridiagonal solver drastically.

The introduction of shared memory on the CUDA implementation happened flawlessly. With the OpenCL implementation there was an error using the entirety of the 48 KiB of local memory (shared memory in OpenCL nomenclature). The OpenCL implementation caused an overflow on local memory usage. This indicates that OpenCL can not provide the whole local memory to the program, which may indicate there are some internal OpenCL variables being stored in it. Because of this, only two of the tridiagonal solvers' arrays can be stored on the local memory. Looking back at Table 27, the chosen memory arrays to be stored in the local memory are B and Z.

The times recorded were also recorded as average time per iteration and stored on Table 29 using the doubles datatype and only 2 arrays stored in shared memory.

Tridiagonal solver used	CR	PCR	CR-PCR	PCR-Thomas
Kepler (ms)	45.4	37.5	48.7	40.8
Pascal (ms)	13.45	41.2	13.4	29.1

Table 29: **Execution times for various tridiagonal solvers, on a 2048x2048 grid, using doubles and OpenCL. Execution time is the average per iteration.**

While working with the kepler GPU, it is possible to notice how on the CUDA implementation (Table 28) performs better than using OpenCL on most algorithms but the CR-PCR. Using the Pascal GPU it is possible to notice that the algorithm that shows better performance is the CR-PCR (having CR perform almost the same).

5.4 RESULTS DISCUSSION

With all the implementations done, a comparison between the results was made using the defined size of 2560x2560 using doubles. Starting, all the execution times and memory information on sequential version on CPU devices was gathered into one table (Table 30) shown below.

Looking at Table 30 it is possible to notice an overall speedup of 2, a large decrease in total memory accesses (almost 27%). This was possible with the usage of the transpose, reduction of the functions used mid algorithm, the usage of operations fewer times than needed and the use of AVX.

A parallel code was implemented using 3 different CPU with different architectures, this time using the grid size of 2048x2048 to compare with the GPU. The best times of each architecture were gathered and each implementation-specific information into Table 31.

Implementation	Time(s)	Cache L3 Accesses	Cache L3 Misses	Cache L3 Miss Rate
Regular	2618.0	102 299 285 882	32 592 111 835	31.86%
Cache-Friendly	1909.6	96 716 170 945	21 108 224 257	21.83%
Single Exp	1856.7	96 718 215 186	21 236 405 821	21.96%
Loop Optimisations	1542.2	37 764 164 967	9 192 664 442	24.34%
Vectorized Code	1336.0	45 847 516 068	8 878 739 151	19.37%

Table 30: **Comparison among all sequential implementations**

Architecture	Ivy Bridge	Skylake	KNL
Average time per iteration (ms)	9.76	5.33	12.35
Number of Threads	2x12	2x16	24
Scheduling	Static	Guided	Guided
Socket Information	Dual Socket	Dual Socket	Single Socket
SIMD	AVX	AVX-512	dual AVX-512
Other Information	—	—	Automatic MCDRAM allocation

Table 31: **All best ececution times using the three CPU with additional information about each implementation.**

Studying the Table 31 it is possible to notice how the best performance is from the Skylake CPU using 32 threads (assigning 1 thread per physical core) and the guided scheduling (good work balance but does not have as much overhead as dynamic). The usage of KNL was not as good as expected, showing the worst performance of all tested CPU (even with the usage of a HBM).

To check the ADI potential efficiency on GPU, new versions were implemented, using the CUDA and OpenCL APIs. These implementations studied new tridiagonal solvers since the first implementation (simulating the CPU ADI directly) was very inefficient. These new tridiagonal solvers were studied and their times were recorded in Tables 20, 23, 25 and 26. While using these tridiagonal solvers on the ADI method, there might be a few differences in performance, since the ADI implementation introduced the shared memory and the most used arrays used it exclusively. The best algorithms for each implementation were stored on Table 32.

GPU Architecture	Kepler		Pascal	
	CUDA	OpenCL	CUDA	OpenCL
Time per iteration (ms)	26.6	37.5	12.1	13.4
Algorithm used	PCR-Thomas	PCR	CR	CR-PCR
Size of secondary matrix	8	—	—	32

Table 32: **Best average time per iteration between all GPU code. Using a grid size of 2048x2048 doubles.**

Checking Table 32 it is possible to notice that the best performance is recorded using CR algorithm on the Pascal GPU with CUDA. These times are also possible because of the change of the error checker. The new version makes good use of the shared memory and uses a high number of threads (against the simpler version with some scalar operations of the CPU). The time spent on memory transfers from the host-device also got reduced with the use of asynchronous copy.

When comparing these times between the CPU (Table 31) and the GPU (Table 32) it is possible to notice how the best performance on CPU is almost 3 times faster than the best performance on GPU considering a grid with a size of 2048x2048. The usage of SIMD on the CPU can be compared to the usage of SIMT on the GPU. While the AVX512 can spread an operation for up to 8 doubles at a time, the usage of warps in GPU can lead to the spread of an operation for up to 32 doubles (4 times more operations), and can these warps can hide memory latency with other warps on stand-by, reducing the total stall time due to the memory. The Skylake can have up to 32 threads (without using SMT), each processing 8 elements with a maximum clock frequency of 1900 MHz, while the Pascal GPU can only have fifteen SM processing 32 elements at a time with a clock frequency of 1746 MHz. In theory, more elements are processed at a time in the Skylake CPU, but since only the division of the Thomas algorithm is vectorised, all the remaining operations in this algorithm are done without the usage of SIMD, resulting in fewer operations happening in simultaneous.

Due to the parallel nature of ADI, it was expected, that versions of this algorithm on GPU would perform better than any version on CPU. That was not the case when comparing the results gathered in this dissertation, especially for grid sizes up to 2048x2048. So additional tests were made to evaluate if this would happen for larger grid sizes, or if that was a common feature of this method. ADI was tested for multiple sizes using CR algorithm in CUDA using the Pascal architecture (since it had the best time from all GPU times) and using the Skylake with the Thomas algorithm. These times were recorded using the average time per iteration and stored on Table 33.

Size	512*512	1024*1024	2048*2048	4096*4096	8192*8192
Skylake (ms)	0.66	1.67	4.97	74.33	900.1
Pascal (ms)	0.95	4.3	12.1	127.8	677.2

Table 33: **Execution times recorded for various grid sizes using the same initial grid**

As seen in Table 33, the larger grids can get more performance from the GPU algorithms as these algorithms show better performance. The algorithm used, CR, reduces each matrix to half every iteration, so the exponential growth of the grid lead to linear growth of iterations to execute while using Thomas algorithm by itself also increases the number of iterations exponentially. This leads to different performances with different sizes, as the CPU algorithms have less overhead, the GPU algorithms scale better.

Comparing now the SPIKE's algorithm to CR (data in Table 22), is possible to notice how using the SPIKE algorithm tends to be more efficient for grid sizes larger than 32Kix32Ki, in this case, when studies for larger matrices are performed, SPIKE algorithm may be one of the algorithms to be tested and tuned to that problem.

The GPU ADI also has a disadvantage on their memory usage on Pascal, with the limited shared memory, there is not enough data to use it on even 1 array in the larger size. Only able to use 1 array on the size 4096*4096.

Using different tridiagonal solvers can also lead to disadvantages using GPU. While the Thomas algorithm can not be used in parallel by itself, the usage of the ADI method to create independent systems, it becomes possible to use more than 1 Thomas algorithm at a time. The usage of the other methods is heavily penalised with the increase of total operations needed per iteration, increasing the total workload of the algorithm.

With these findings, it is possible to confirm, that for these implementations, the implementation best suited for sizes smaller than 4Kix4Ki is the CPU implementation using the Skylake CPU, for sizes between 4Kix4Ki and 32Kix32Ki the implementation best suited is the GPU implementation using the CR algorithm in the Pascal GPU, and, in theory, for sizes larger than 32Kix32Ki the best implementation could be using SPIKE algorithm if it proves to be as scalable as the earlier tests show.

6 CONCLUSION

The ADI method is an algorithm to solve the convection-diffusion equations iteratively. This method splits the main problem into smaller independent sub-problems. These smaller sub-problems are independent, so they can be solved in parallel.

A first implementation was done for the CPU. This implementation was then explored and made more efficient with the usage of better memory layout, reduction of the total operations to perform and the use of SIMD.

Using parallelism with shared memory on a CPU added some significant performance speedups, reducing the execution times to less than a fifth of the time on the Ivy Bridge CPU. The scheduling types were then tested using the Skylake CPU, showing an improvement while using the guided schedule type when compared to the static scheduling used before.

Using the GPU and the many parallel tridiagonal solvers showed that each API and GPU architecture can have different outputs for the tridiagonal solvers used. The best results were using the Pascal GPU (these were better than the Kepler GPU in all times recorded) with CUDA and CR. The usage of OpenCL when comparing to CUDA makes most of the tridiagonal solvers perform with less performance (excluding CR-PCR using the Pascal GPU.

The usage of a regular grid size of 2048x2048 (which are usually the sizes used for this convection-diffusion problem) leads to a better performance using the CPU. Due to this, the better use for these smaller grids is the CPU implementation.

While the basic grid can be more efficiently solved using the studied CPU, when the grid size goes up the problem stops being as efficient on the CPU as the GPU implementation. This happens because the GPU use various parallel algorithms, that have better scalability than the regular Thomas algorithm, with larger matrices the GPU performs better, even with the disadvantage of not being able to use their shared memory on the studied GPU. These times become more efficient on grid sizes larger than 8192x8192 for the Pascal GPU using CUDA, when compared to a multicore CPU.

Testing the problem with the SPIKE algorithm can be beneficial if a new strategy for distributed memory is created, using more GPU and CPU. If more shared memory is available, it can also be possible to switch the memory allocation to the shared memory, reducing some of the overhead caused by this. While these would need different configurations, with the tests done, the SPIKE algorithm started being more efficient for grid sizes larger than 32Ki x 32Ki.

Using OpenCL showed some problems using the local memory when comparing it to CUDA, since some of it is already used by the language itself. Due to this, the arrays allocated in this faster memory is not as used on OpenCL and it is used in CUDA, leading to more waits on the memory.

While working with the KNL did not prove to be efficient with the ADI implemented, it is possible to notice how it stops speeding up after reaching the threshold of the memory bandwidth. The KNL does not have an L3 cache, so it ends up having to use the global memory often. This paired with the its lower clock frequency (around 4 times slower than the Skylake clock frequency) makes the code run with lower performance than when using a multicore CPU.

With the current usage of the Pascal GPU, it is not possible to test tensor computing. This could theoretically speed up the creation of the tridiagonal matrices by creating chunks of matrices at a time. The usage of tensor computing could also be used instead of vector computing for tridiagonal solvers.

It is also possible to use heterogeneous computing to speed up the current GPU implementation, this could increase the number of simultaneous lines/columns being processed due the higher number of processors and more computation power. This implementation would need more transfers from each device. Therefore, it could be inefficient, depending on the implementation, so it must be studied carefully.

7

BIBLIOGRAPHY

J.E. Adsuara, I. Cordero-Carrión, P. Cerdá-Durán, and M.A. Aloy. Scheduled relaxation jacobi method: Improvements and applications. *Journal of Computational Physics*, 321:369–413, Sep 2016. ISSN 0021-9991. doi: 10.1016/j.jcp.2016.05.053. URL http://dx.doi.org/10.1016/j.jcp.2016.05.053.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450378956. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

Jacob Anders Andrzej Chrzeszczyk. Matrix computationson the gpucublas, cusolver and magma by example. 2017.

Dario Bini and Beatrice Meini. The cyclic reduction algorithm. *Numerical Algorithms*, 51:23–60, 05 2009. doi: 10.1007/s11075-008-9253-0.

M. Bottoni. A variant of the adi method for two-phase flow calculations. *Computers & Fluids*, 23 (2):305–321, 1994. ISSN 0045-7930. doi: https://doi.org/10.1016/0045-7930(94)90043-4. URL https://www.sciencedirect.com/science/article/pii/0045793094900434.

Carlos Carvalho. The gap between processor and memory speeds. 2002.

Li-Wen Chang, John Stratton, Hee-Seok Kim, and Wen-mei Hwu. A scalable, numerically stable, high-performance tridiagonal solver using gpus. pages 1–11, 11 2012. ISBN 978-1-4673-0805-2. doi: 10.1109/SC.2012.12.

Shane Cook. Chapter 9 - optimizing your application. In Shane Cook, editor, *CUDA Programming*, Applications of GPU Computing Series, pages 305 – 440. Morgan Kaufmann, Boston, 2013. ISBN 978-0-12-415933-4. doi: https://doi.org/10.1016/B978-0-12-415933-4.00009-0. URL http://www.sciencedirect.com/science/article/pii/B9780124159334000090.

J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225, 2011. doi: 10.1109/ICPP.2011.45.

Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. pages 133–137, 01 2004. doi: 10.1145/1058129.1058148.

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972. ISSN 2326-3814. doi: 10.1109/TC.1972.5009071.

L. M. Gadhikar and Y. S. Rao. Analysis of programs for gpgpu architectures. In *2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 1–4, May 2018. doi: 10.1109/ICOEI.2018.8553918.

C.F. Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. Carl Friedrich Gauss Werke. sumtibus F. Perthes et I. H. Besser, 1809. URL https://books.google.pt/books?id=ORUOAAAAQAAJ.

Joseph Grcar. Mathematicians of gaussian elimination. *Notices of the American Mathematical Society*, 58, 06 2011.

Dominik Grewe and Michael O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. pages 286–305, 03 2011. ISBN 978-3-642-19860-1. doi: 10.1007/978-3-642-19861-8_16.

John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X.

R. W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *J. ACM*, 12(1): 95–113, January 1965a. ISSN 0004-5411. doi: 10.1145/321250.321259. URL https://doi.org/10.1145/321250.321259.

Roger Hockney. A fast direct solution of poisson's equation using fourier analysis. *Journal of the ACM (JACM)*, 12:95–113, 01 1965b. doi: 10.1145/321250.321259.

W.W. Hwu. *GPU Computing Gems Jade Edition*. Applications of GPU Computing Series. Elsevier Science, 2011. ISBN 9780123859648. URL https://books.google.pt/books?id=LsNVFUnzcVMC.

Rajibul Islam. An efficient one dimensional parabolic equation solver using parallel computing. 2010.

Carl Christian Kjelgaard Mikkelsen and Murat Manguoglu. Analysis of the truncated spike algorithm. *SIAM J. Matrix Analysis Applications*, 30:1500–1519, 01 2008. doi: 10.1137/080719571.

William T. Lee. Tridiagonal matrices: Thomas algorithm. URL http://www.industrial-maths.com/ms6021_thomas.pdf.

George Lindfield and John Penny. Chapter 2 - linear equations and eigensystems. In George Lindfield and John Penny, editors, *Numerical Methods (Fourth Edition)*, pages 73 – 156. Academic Press, fourth edition edition, 2019. ISBN 978-0-12-812256-3. doi: https://doi.org/10.1016/B978-0-12-812256-3.00011-7. URL http://www.sciencedirect.com/science/article/pii/B9780128122563000117.

Diogo Lopes, Rui Pereira, and Stéphane Clain. High performance computation with adi on cartesian grid to solve the steady state 2d convection diffusion equation. Associação Portuguesa de Mecânica Teórica, Aplicada e Computacional, 2019. URL http://hdl.handle.net/1822/60272.

Hamish J. Macintosh. *Solving diagonally dominant tridiagonal linear systems with FPGAs in an heterogeneous computing environment*. PhD thesis, Queensland University of Technology, 2019. URL https://eprints.qut.edu.au/130762/.

Zoltan Majo and Thomas Gross. (mis)understanding the numa memory system performance of multithreaded workloads. pages 11–22, 09 2013. ISBN 978-1-4799-0553-9. doi: 10.1109/IISWC.2013.6704666.

Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), July 2015. ISSN 0360-0300. doi: 10.1145/2788396. URL https://doi.org/10.1145/2788396.

Eric Polizzi and Ahmed Sameh. Spike: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113 – 120, 2007. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2005.07.005. URL http://www.sciencedirect.com/science/article/pii/S0045793005001325. Challenges and Advances in Flow Simulation and Modeling.

Yves Robert. *The impact of vector and parallel architectures on the Gaussian elimination algorithm*. Manchester University Press, 1990.

A.S.J. Saif and Firas Al-Saadawi. Bernstein differential quadrature method for solving the unsteady state convection-diffusion equation. *Indian Journal of Applied Research*, 3:20–26, 10 2011. doi: 10.15373/2249555X/SEPT2013/60.

Ankit Sethia, Delaram Jamshidi, and Scott Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015*, pages 174–185, 03 2015. doi: 10.1109/HPCA.2015.7056031.

Braegan S. Spring, Eric Polizzi, and Ahmed H. Sameh. A feature complete spike banded algorithm and solver, 2018.

Harold S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM*, 20(1):27–38, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321741. URL https://doi.org/10.1145/321738.321741.

William Gilbert Strang. The heat equation and convection-diffusion. In *Mathematical Methods for Engineers II*, 2006. URL https://archive.org/details/flooved1699.

V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

Jesmin Jahan Tithi, Dhruv Matani, Gaurav Menghani, and Rezaul Chowdhury. Avoiding locks and atomic instructions in shared-memory parallel bfs using optimistic parallelization. 05 2013. doi: 10.1109/IPDPSW.2013.241.

Stanimire Tomov. Hpc with multicore and gpus. 02 2016.

Zhangping Wei, Byunghyun Jang, Yaoxin Zhang, and Yafei Jia. Parallelizing alternating direction implicit solver on gpus. *Procedia Computer Science*, 18:389–398, 12 2013. doi: 10.1016/j.procs.2013.05.202.

Michael Weiss. Strip mining on simd architectures. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, page 234–243, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914341. doi: 10.1145/109025.109083. URL https://doi.org/10.1145/109025.109083.

King H. Yang. Chapter 7 - stepping through finite element analysis. In King-Hay Yang, editor, *Basic Finite Element Method as Applied to Injury Biomechanics*, pages 281 – 308. Academic Press, 2018. ISBN 978-0-12-809831-8. doi: https://doi.org/10.1016/B978-0-12-809831-8.00007-6. URL http://www.sciencedirect.com/science/article/pii/B9780128098318000076.

Xiyang I.A. Yang and Rajat Mittal. Acceleration of the jacobi iterative method by factors exceeding 100 using scheduled relaxation. *Journal of Computational Physics*, 274:695 – 708, 2014. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2014.06.010. URL http://www.sciencedirect.com/science/article/pii/S0021999114004173.

Yao Zhang, Jonathan Cohen, and John Owens. Fast tridiagonal solvers on the gpu. volume 45, 05 2010. doi: 10.1145/1837853.1693472.

D. Zhou, F. Gao, E. Breaz, A. Ravey, and A. Miraoui. Tridiagonal matrix algorithm for real-time simulation of a two-dimensional pem fuel cell model. *IEEE Transactions on Industrial Electronics*, 65 (9):7106–7118, Sep. 2018. ISSN 1557-9948. doi: 10.1109/TIE.2017.2787598.

Ömer EGECIOGLU, Cetin K. KOC, and Alan J. LAUB. A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors* *technical report no. trcs88–1, department of computer science, university of california, santa barbara, january 1988. this article was presented at the third siam conference on parallel processing for scientific computing, los angeles, california, december 1–4, 1987, and the third conference on hypercube concurrent computers and applications, california institute of technology, jpl, pasadena, california, january 19–20, 1988. In Henk A. van der Vorst and Paul van Dooren, editors, *Parallel Algorithms for Numerical Linear Algebra*, volume 1 of *Advances in Parallel Computing*, pages 95 – 108. North-Holland, 1990. doi: https://doi.org/10.1016/B978-0-444-88621-7.50010-4. URL http://www.sciencedirect.com/science/article/pii/B9780444886217500104.

Part III

APPENDICES

A

EXECUTION TIMES

In this chapter, all the tridiagonal solvers execution times will be shown. Both GPU implementations will be shown and for every implementation, both GPU times will be shown. For every GPU, a table for each tested matrix size will be also shown. The Smaller Sizes in the tables are the size of the secondary algorithm (PCR in CR-PCR and Thomas in PCR-Thomas) unless specified otherwise.

A.1 CUDA EXECUTION TIMES

A.1.1 *Kepler GPU*

Smaller Sizes	2	4	8	16
CR	0.31			
PCR	0.19			
CRPCR	0.50	0.45	0.40	0.31
PCRThomas	0.20	0.20	0.23	0.32

Table 34: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 32x32**

Smaller Sizes	2	4	8	16	32
CR	0.37				
PCR	0.26				
CRPCR	0.61	0.55	0.49	0.42	0.36
PCRThomas	0.23	0.23	0.26	0.35	0.57

Table 35: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 64x64**

Smaller Sizes	2	4	8	16	32	64
CR	0.43					
PCR	0.32					
CRPCR	0.85	0.79	0.72	0.59	0.51	0.43
PCRThomas	0.32	0.31	0.34	0.43	1.36	2.17

Table 36: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 128x128**

Smaller Sizes	2	4	8	16	32	64	128
CR	1.34						
PCR	0.84						
CRPCR	2.11	2.05	2.00	1.98	1.84	1.44	1.08
PCRThomas	0.84	0.88	0.95	1.01	1.40	2.2	3.88

Table 37: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 256x256**

Smaller Sizes	2	4	8	16	32	64	128	256
CR	1.67							
PCR	0.97							
CRPCR	2.73	2.60	2.53	2.58	2.79	2.17	1.53	1.19
PCRThomas	0.91	0.91	0.98	1.20	1.48	2.29	3.96	7.44

Table 38: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 512x512**

Smaller Sizes	2	4	8	16	32	64	128	256	512
CR	2.00								
PCR	1.42								
CRPCR	3.31	3.23	3.20	3.39	3.17	3.19	2.36	1.92	1.67
PCRThomas	1.33	1.31	1.34	1.45	1.90	2.50	4.19	7.55	14.50

Table 39: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 1024x1024**

Smaller Sizes	2	4	8	16	32	64	128	256	512	1024
CR	2.53									
PCR	2.48									
CRPCR	4.32	4.20	4.10	4.12	4.25	4.08	3.70	2.92	3.00	2.83
PCRThomas	2.26	2.16	2.10	2.17	2.53	3.50	4.60	8.00	14.92	29.30
CRPCRThomas*	2.28	2.27	2.25	2.27	2.70	3.12	3.50	6.23	—	

Table 40: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 2048x2048**
*** The CRPCRThomas algorithm was coded in a way that the CR always had 2 steps and the PCRThomas steps were changeable. In this case, the Smaller Size is the size of the Thomas algorithm.**

A.1.2 *Pascal GPU*

Smaller Sizes	2	4	8	16
CR	0.31			
PCR	0.19			
CRPCR	0.50	0.45	0.40	0.31
PCRThomas	0.20	0.20	0.23	0.32

Table 41: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 32x32**

Smaller Sizes	2	4	8	16	32
CR	0.37				
PCR	0.26				
CRPCR	0.61	0.55	0.49	0.42	0.36
PCRThomas	0.23	0.23	0.26	0.35	0.57

Table 42: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 64x64**

Smaller Sizes	2	4	8	16	32	64
CR	0.43					
PCR	0.27					
CRPCR	0.73	0.66	0.60	0.53	0.47	0.39
PCRThomas	0.26	0.26	0.28	0.38	0.60	1.06

Table 43: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 128x128**

Smaller Sizes	2	4	8	16	32	64	128
CR	0.49						
PCR	0.32						
CRPCR	0.85	0.78	0.71	0.65	0.59	0.51	0.43
PCRThomas	0.32	0.31	0.34	0.42	0.64	1.10	2.07

Table 44: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 256x256**

Smaller Sizes	2	4	8	16	32	64	128	256
CR	0.57							
PCR	0.48							
CRPCR	0.97	0.90	0.84	0.77	0.71	0.64	0.58	0.52
PCRThomas	0.45	0.43	0.44	0.53	0.72	1.17	2.12	4.06

Table 45: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 512x512**

Smaller Sizes	2	4	8	16	32	64	128	256	512
CR	0.67								
PCR	0.80								
CRPCR	1.09	1.04	0.97	0.89	0.83	0.77	0.72	0.72	0.74
PCRThomas	0.74	0.69	0.67	0.72	0.90	1.32	2.24	4.16	8.07

Table 46: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 1024x1024**

Smaller Sizes	2	4	8	16	32	64	128	256	512	1024
CR	0.80									
PCR	1.49									
CRPCR	4.28	1.21	1.14	1.07	1.01	0.94	0.90	0.90	1.01	1.17
PCRThomas	1.33	1.19	1.10	1.10	1.24	1.62	2.47	4.30	8.19	16.03
CRPCRThomas*	0.80	0.75	0.75	1.08	1.08	1.68	2.88	5.27	—	

Table 47: **Times (in ms) for 16 systems using various tridiagonal solvers for matrices with size of 2048x2048**
The CRPCRThomas algorithm was coded in a way that the CR always had 2 steps and the PCRThomas steps were changeable. In this case, the Smaller Size is the size of the Thomas algorithm.

A.2 OPENCL EXECUTION TIMES

A.2.1 *Kepler GPU*

Smaller Sizes	2	4	8	16
CR	135			
PCR	100			
CRPCR	132	137	128	127
PCRThomas	122	121	132	134

Table 48: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 32x32**

Smaller Sizes	2	4	8	16	32
CR	138				
PCR	118				
CRPCR	137	137	133	132	144
PCRThomas	121	118	125	134	158

Table 49: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 64x64**

Smaller Sizes	2	4	8	16	32	64
CR	152					
PCR	123					
CRPCR	152	148	144	143	139	131
PCRThomas	122	123	127	103	167	178

Table 50: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 128x128**

Smaller Sizes	2	4	8	16	32	64	128
CR	164						
PCR	126						
CRPCR	162	172	159	167	159	146	134
PCRThomas	129	134	135	145	168	219	325

Table 51: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 256x256**

Smaller Sizes	2	4	8	16	32	64	128	256
CR	181							
PCR	124							
CRPCR	181	180	177	149	199	171	153	143
PCRThomas	138	136	141	172	173	222	331	544

Table 52: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 512x512**

Smaller Sizes	2	4	8	16	32	64	128	256	512
CR	204								
PCR	151								
CRPCR	202	202	157	205	223	218	180	189	168
PCRThomas	161	158	163	176	203	241	34	561	997

Table 53: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 1024x1024**

Smaller Sizes	2	4	8	16	32	64	128	256	512	1024
CR	244									
PCR	235									
CRPCR	233	251	262	245	265	257	242	211	240	233
PCRThomas	226	219	216	230	249	327	372	588	1025	1908

Table 54: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 2048x2048**

A.2.2 *Pascal GPU*

Smaller Sizes	2	4	8	16
CR	65			
PCR	44			
CRPCR	61	61	58	49
PCRThomas	54	54	57	56

Table 55: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 32x32**

Smaller Sizes	2	4	8	16	32
CR	68				
PCR	45				
CRPCR	55	70	52	58	50
PCRThomas	45	47	49	55	70

Table 56: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 64x64**

Smaller Sizes	2	4	8	16	32	64
CR	71					
PCR	47					
CRPCR	65	60	60	54	60	54
PCRThomas	57	57	51	57	70	99

Table 57: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 128x128**

Smaller Sizes	2	4	8	16	32	64	128
CR	77						
PCR	49						
CRPCR	64	61	65	59	56	56	53
PCRThomas	54	52	55	60	72	101	162

Table 58: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 256x256**

Smaller Sizes	2	4	8	16	32	64	128	256
CR	83							
PCR	57							
CRPCR	68	67	72	67	65	68	67	67
PCRThomas	70	70	61	67	78	107	167	287

Table 59: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 512x512**

Smaller Sizes	2	4	8	16	32	64	128	256	512
CR	91								
PCR	70								
CRPCR	80	80	77	77	73	74	69	71	71
PCRThomas	77	78	101	93	116	160	246	423	772

Table 60: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 1024x1024**

Smaller Sizes	2	4	8	16	32	64	128	256	512	1024
CR	105									
PCR	128									
CRPCR	82	82	80	82	81	85	93	82	91	106
PCRThomas	123	123	116	116	115	142	193	310	553	1045

Table 61: **Times (in us) for 1 system using various tridiagonal solvers for matrices with size of 2048x2048.**

B

TOOLING

Used In	Tool	Version	Switches	Other Information
CPU Implementation	ICC	19.0.5.281	-O3 -std=c++11 -fno-inline -qopenmp -no-multibyte-chars -xHost	Used with Intel Libraries
CUDA Implementation	GCC	4.9.0	—	Used with CUDA
	nvcc	10.1.105	-Xptxas -O3 -ccbin=gcc	—
OpenCL Implementation	g++	7.5.0	-O3	Used with OpenCL Library
Other Tools	gprof	2.27-41. base.el7_7.1	—	Used to profile the CPU implementation
	gprof2dot	2019.11.30	—	—
	nvprof	10.1.105	—	Used to profile the GPU implementation

The author acknowledge the financial support by FEDER – Fundo Europeu de Desenvolvimento Regional, through COMPETE 2020 – Programa Operacional Fatores de Competitividade, and the National Funds through FCT – Fundação para a Ciência e a Tecnologia, project no. POCI-01-0145-FEDER-028118 e PTDC/MAT-APL/28118/2017. The project was funded with a research grant with the reference of CFUM-46-SEA