Master in *Photonics for Security Reliability and Safety* (PSRS)

DESIGN OF THE BACK-END PROCESSING SYSTEM FOR THE ACCELERATION OF THE MICROWAVE IMAGING RECONSTRUCTION ALGORITHMS

Master Thesis Report

Presented by

Nathalia Della Giustina Ballmann

and defended at

Université Jean Monnet

31 August 2022

Academic Supervisors:

PhD, Francesca Vipiana

PhD, Mario Casu

PhD, Jorge Tobon

Jury Committee:

PhD, Nathalia Destouches

PhD, Carlo Ricciardi

PhD, Baptiste Moine

MSc, Arnaud Meyer

Abstract

Microwave imaging (MWI) is a diagnostic tool whose working principle relies on the dielectric contrast between lesions and healthy tissues, and could be used, for instance, to detect breast cancer or brain strokes. This work aims to improve the speed of the processing of MWI data acquired when using a finite element contrast source inversion method. The processing consists of solving a large sparse and complex linear system with 24 right-hand sides.

Multiple open-source solvers (UMFPACK, KLU, Eigen and MUMPS) were tested to solve this linear system, including direct and indirect methods, and the precision obtained from each trial was compared. The only direct solver tested that could solve the all the MWI linear systems of interest was MUMPS. The indirect methods tested did not achieve precise results.

The original proposal of this thesis was to perform the factorization of the matrix and then use a GPU to accelerate the solution of the triangular linear systems, MUMPS does not support exporting the factorization results, though. Therefore, UMFPACK was used to factorize matrices from the SuiteSparse Collection in order to test OpenCL implementations in a GPU so to employ parallelism.

Two different types of kernels to solve triangular linear systems were implemented: a) column block algorithms, which only worked for very small matrices and presented synchronization issues for bigger matrices; b) solving multiple right-hand sides in parallel. When solving multiple right-hand sides in parallel, the row-compressed format could be executed faster than the column-compressed format. The latter format had worst performance because it required more accesses to the GPU's global memory. Two modified versions of the kernel to deal with column-compressed format using local memory were also implemented, which resulted in smaller run times, but still slower than executing the same task in a CPU. In general, the observed overhead of moving data to and from the GPU was greater than the time to execute the same task sequentially in a CPU for the tested matrices.

Acknowledgments

My first and earnest acknowledge to the PSRS Consortium for awarding me the EMJMD scholarship. Enrolling in an Erasmus Mundus Joint Master Degree had been a dream of mine for years, and you enabled this dream to come true.

I would like to extend the acknowledgements to my advisors at Politecnico di Torino, for providing guidance for this work.

I wish to express my gratitude to the Professors and instructors I had in Université Jean Monnet, University of Eastern Finland and Politecnico di Torino.

I would like to show my appreciation to my family and friends for the support I received. I can only hope to make up someday for the time I'm away. I swear it is more than I originally planned.

To the friends I made along the way and the people I got closer to during this two-year period, for making things easier when the end of the world seemed to be happening every couple of weeks. Merci. Kiitos. Grazie. Obrigada.

Finally, to my partner in life, Yuri, thank you for always supporting me in all aspects of life and for making the best of every situation.

Table of Contents

Abstract I
Acknowledgments II
1 Introduction 1
2 Microwave Imaging 3
2.1 System Prototype 3
2.2 Contrast Source Inversion 4
3 Backend processing practical aspects 8
3.1 Computers used 8
3.2 Available data 9
3.2.1 Matrix MWI 1 9
3.2.2 Matrix MWI 2 9
3.2.3 Suite Sparse Matrix Collection 10
3.3 Data format 11
3.4 Data compression 11
3.4.1 List of non-zero elements 11
3.4.2 Column-compressed Storage 13
3.4.3 Row-Compressed Storage 14
4 Factorizing Sparse Matrices 14
4.1 Methodology 15
4.2 UMFPACK 16
4.3 MUMPS 19
4.4 Other attempts 20
4.5 Summary of results 20
5 Iterative Solvers 21
5.1 Matrix MWI 1 22
5.2 Matrix MWI 2 25

6 Solving Triangular Linear Systems 25
6.1 Algorithms and parallelization 27
6.2 OpenCL 32
6.3 Prototype 1 – Solve multiple RHSs in parallel 35
6.5 Prototype 3 – Solve multiple RHSs in parallel and store accumulation variable in local memory 39
6.6 Prototype 4 – Column Block Algorithm 41
6.7 Comparison between CPU and GPU 43
7 Conclusion 45
8 Bibliography 47

1 Introduction

Microwave imaging (MWI) is a diagnostic whose working principle relies on dielectric contrast between lesions and healthy tissues. This technology could be used as a complement to other imaging techniques like mammography and magnetic resonance. It presents advantages, such as not offering risk to the patient and the cost is potentially low [1]. It is also particularly well suited for prehospital use since the devices can be built in compact and portable formats. [2]

One application of MWI would be brain imaging for stroke detection, since permittivity and conductivity for blood are higher than for grey and white matter. It is estimated that stroke is ranked as the second most common cause of death and the third cause of years of disability-affected life [2]. For instance, [3] reports a prototype for brain stroke 3D imaging.

Another example of application would be breast cancer detection, as presented in [1] along with a proposal of COTS (Commercial off-the-shelf) equipment that would make the technology more affordable. It has also been proposed to use Microwave techniques in applications like "bone imaging and bone density measurements, thermal monitoring in hyperthermia, cardiac imaging, imaging of soft tissue in extremities, detection of compartment syndrome, and detection of thoracic and abdominal injuries." [2]

This thesis focuses on the back-end processing of data arising from the Finite-Element Contrast Source Inversion Method applied to MWI. This algorithm allows for accurate quantitative reconstructions of images but needs higher computational resources [4]. The processing of the data consists primarily of a large sparse linear system to be solved. There are two possible ways to solve the resulting linear system: with direct or indirect methods (or iterative ones). The direct methods start with the factorization of the matrix and, to perform this task, it's possible find open-source software solutions. As a result of the direct methods' factorization, the initial matrix is transformed into a multiplication of two sparse triangular matrices, which, in practice represents two simpler to solve linear systems, also called the *SpTRSV* kernel. This kernel is also present when solving many numerical methods and is one of the building blocks of sparse Numerical Linear Algebra [5].

SpTRSV can be parallelized to accelerate the algorithm execution. Different kind of devices could be used: GPUs (Graphic Processing Unit), FPGAs (Field Programmable Gate Array) or ASICs (Application Specific Integrated Circuit). As a rule of thumb, there is always a compromise between cost, time necessary to develop a solution and achieved acceleration (which is related to how specific can be the computer architecture to tackle a problem). An ASIC would be the most expensive solution and take more time to develop, but potentially the best in terms of performance, as it enables building hardware extremely specialized for a certain application. Hardware acceleration with GPUs is constrained by the GPU architecture but could be implemented more quickly. FPGAs are usually seen as a compromise between the other two solutions.

There are cases in which the use of FPGA may not be beneficial, mainly when the data transfer time to an FPGA is too big. For instance, a Monte-Carlo computation could benefit of being run in a FPGA because it has few inputs and many computations are performed; a vector addition operation, however, would likely not benefit, since there are only twice as many inputs are there are computations. [6] GPUs can be programmed using parallel computing platforms like CUDA and OpenCL. The latter has the advantage of being portable for more platforms, including GPUs and FPGAs, and was chosen to be used in this work in prototypes that attempted to accelerate the execution of the *SpTRSV* kernel.

In literature that are many examples of works that cover the implementation of the *SpTRSV* kernel: [7] implements block algorithms in a GPU, [8] solves multiple right-hand sides using GPUs, [5] measures the performance of the kernel implemented in FPGAs using the OpenCL platform.

This work is presented as follows. Section 2 presents a short description of the hardware used in a prototype developed at Politecnico di Torino for 3D Brain stroke imaging, and a brief introduction on how the Finite Element Contrast Source Inversion Method, which produces the data used in this thesis. Section 3 contains practical aspects of this work, such as hardware specifications of the computers used, as well as a description of the data used to evaluate the

prototypes, data formats and different compression formats. Section 4 presents the different direct solvers tested to factorize MWI matrices and a quantitative analysis of the results, whereas Section 5 presents the indirect solvers tested and how they compare to the direct solvers. Section 6 contains a bibliography review covering the main strategies used to accelerate the solution of triangular linear systems, a summary of OpenCL's memory architecture, the description of the prototypes developed for this task using GPUs and interpretation of the obtained results. Conclusions sum up the findings of this work and provides suggestions on how to proceed to speed MWI reconstructions algorithms.

2 Microwave Imaging

2.1 System Prototype

The MWI system prototype developed at Politecnico di Torino [3] has the goal of generating images of the brain in order to detect brain strokes. The different human tissues exhibit different electrical properties at microwave frequencies. The goal of detecting brain strokes can be achieved by creating a 3D plot of the dielectric permittivity and conductivity of the human head. A CAD model of the developed device is shown in Figure 1 along with a photography of the prototype, presented in Figure 2.

The prototype's working principle is based on 24 printed monopole antennas placed around the head mimicking a wearable helmet. The antennae are connected to a two-port vector network analyser (VNA) through a 24×24 switching matrix. The working frequency is around 1 GHz. The process of acquiring data to generate the images consists of turning on alternately each of the 24 antennae and then measuring the received signal in the remaining 23.

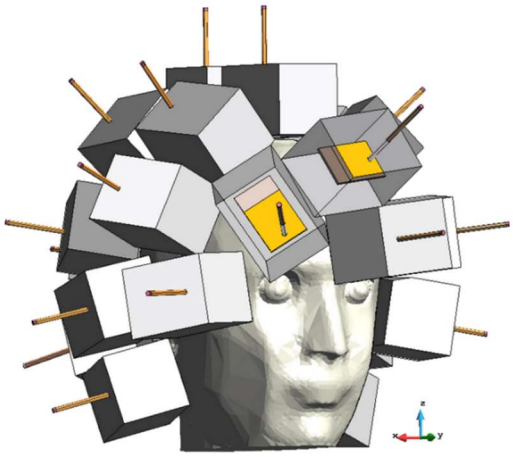

Figure 1 – CAD model of the MWI system prototype. Image extracted from [3].

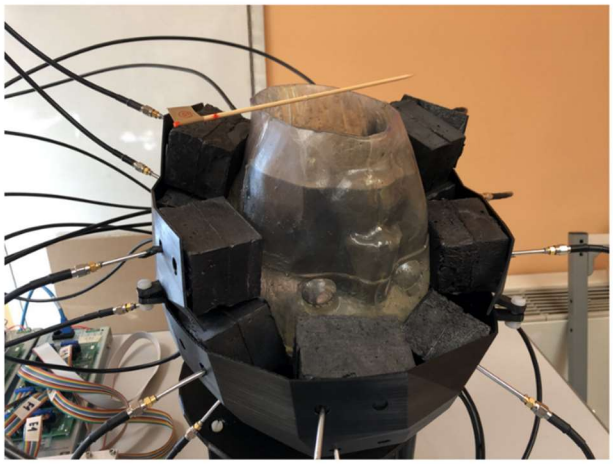

Figure 2 – Photography of the MWI system prototype. Image extracted from [3].

2.2 Contrast Source Inversion

The contrast source inversion (CSI) is a non-linear iterative algorithm that is widely used to numerically solve microwave inversion problems. It allows for accurate quantitative reconstructions, even though the use of the algorithm yields in a high computational cost [4].

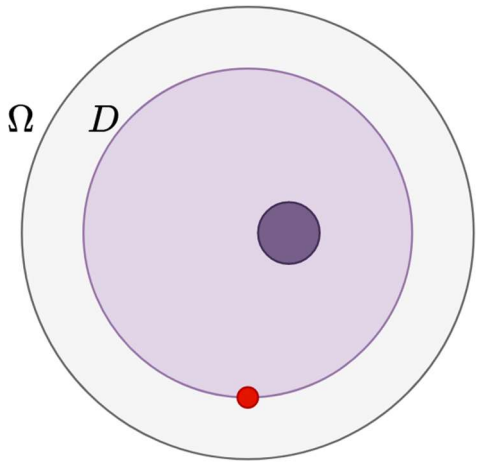

Figure 3 – Illustration of the regions of the 3D scattering problem. In red is an example of placement of an antenna.

The way the CSI algorithm works is briefly described in this section, taking [4] as reference. The 3-D scattering problem is depicted in Figure 3. The whole 3-D domain is denoted by Ω, which is filled with a medium of known complex relative permittivity ϵ_b. Within Ω, we have a region of interest D with unknown complex relative permittivity ϵ_r. The antennae are placed in the boundary of D.

Each antenna t illuminates Ω without the target and this measure corresponds to the incident field $\underline{E}_t^{inc}(\underline{r})$, whereas $\underline{E}_t^{tot}(\underline{r})$ correspond to the illuminated Ω when the target is present. The scattered field $\underline{E}_t^{sct}(\underline{r})$ is defined as:

$$\underline{E}_t^{sct}(\underline{r}) = \underline{E}_t^{tot}(\underline{r}) - \underline{E}_t^{inc}(\underline{r}) \tag{1}$$

The dielectric contrast $\chi(\underline{r})$ between the background medium and the target is defined in Equation 2 and a parameter called contrast source is defined in equation 3, which links the total field radiated by the t-th antenna and the dielectric contrast.

$$\chi(\underline{r}) \triangleq \frac{\epsilon_r(\underline{r}) - \epsilon_b(\underline{r})}{\epsilon_b(\underline{r})} \tag{2}$$

$$\underline{\omega}_t(\underline{r}) \triangleq \chi(\underline{r})\underline{E}_t^{tot}(\underline{r}) \tag{3}$$

$\underline{E}_t^{sct}(\underline{r})$ and $\underline{\omega}_t(\underline{r})$ are linked by the wave equation, as presented In Equation 4, with the wave number $k_b^2(\underline{r}) = \omega^2\mu_0\epsilon_0\epsilon_b(\underline{r})$, ω is the angular frequency, μ_0 and ϵ_0 are, respectively, the free space permeability and permittivity [9].

$$\nabla \times \nabla \times \underline{E}_t^{sct}(\underline{r}) - k_b^2(\underline{r})\underline{E}_t^{sct}(\underline{r}) = k_b^2(\underline{r})\underline{\omega}_t(\underline{r}) \tag{4}$$

The solution for the CSI problem is achieved by minimizing the cost function in Equation 5, where F^S and F^D represent, respectively, the mismatch at the antennae locations and the mismatch in the region of interest D. The minimization process is done through iterative optimization.

$$F^{CSI}(\chi_n, \underline{\omega}_{t,n}) = F^S(\underline{\omega}_{t,n}) + F^D(\chi_n, \underline{\omega}_{t,n}) \tag{5}$$

The implementation of the CSI algorithm requires the discretization of the 3-D domain Ω. Illustrations of this discretization are shown in Figure 4 and Figure 5, in which are depicted, respectively the antennae and the area to be detected. The work in [4] proposes a novel way of discretization which involves only scalar coefficients and simplifies the CSI implementation.

This Finite Element Method (FEM) produces a linear system $Ax = b$, which must be solved so to reconstruct the 3-D image of the brain based on its electric properties and be able to detect, for instance, a blood clot.

A is a square sparse complex and symmetric matrix and b are the multiple right-hand side vectors. The number of right-hand sides is equal to the number of antennae. x is of the same

dimension as b. Since most of the elements of Matrix A are zero, the matrix is said to be sparse. The matrix has dimensions of 2050637×2050637 and has 28722845 non-zero elements.

The placement of the non-zeroes depends on how the FEM problem was formulated. The size of the matrix depends roughly on the ratio between the domain under analysis and the emitted wavelength by the antennae cubed, that is, on the granularity of the discretization.

The resolution of this FEM problem was previously done using a solver called *Pardiso* [10] [11] [12] through a *MATLAB* interface. This solver, however, is not available for academic use anymore.

Considering data obtained from a previous run of the solution with *Pardiso*, of all the time it took to solve the FEM problem (97 min), 85.5% the time was spent in factorizing matrix A, and an additional 7.5% in solving the triangular linear systems after obtaining the factorization results.

The total time to solve the system was also considered excessive and not suitable for the goal of enabling a fast diagnosis. The scope of this work aims to find alternatives to accelerate the solution of this linear system, and at first, focus on the acceleration the resolution triangular linear systems through parallelism.

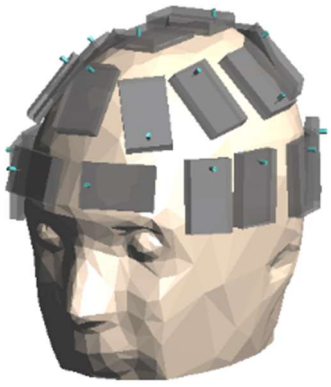

Figure 4 – Illustration of 3D discretization used in the CSI algorithm with the antennae.

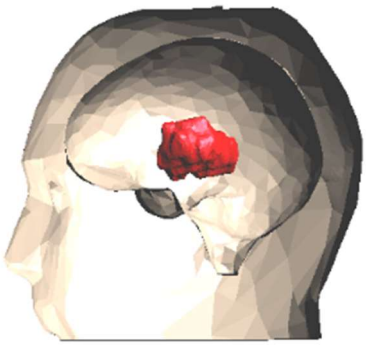

Figure 5 - Illustration of 3D discretization used in the CSI algorithm with the region to be detected.

3 Backend processing practical aspects

3.1 Computers used

Most of the computation were performed in a Windows PC within Ubuntu WSL. The PC specifications are shown in Table 1. WSL stands for Windows Subsystem for Linux, and it allows to run a Linux environment on Windows [13]. The Linux-like environment simplifies the process of installing open-source programs, as it can be done directly from the command line.

Processor	AMD Ryzen 5 5500U with Radeon Graphics 2.10 GHz
Installed RAM	16.0 GB (15.4 GB usable)
Operational System	Windows 10 Home
WSL	Ubuntu 20.04.4

Table 1 - PC specifications

A server that belongs to the VLSI Research Group from Politecnico di Torino was used for the Hardware Acceleration section of this work due to its GPU. The server specifications are presented in Table 2.

Operational System	CentOS 7
GPU	NVIDIA Tesla P100

Table 2 - Server specifications

Some tests were also run in a Google Colab environment, which is a free *Jupyter* notebook environment that runs in the cloud and does not require any installations.

3.2 Available data

3.2.1 Matrix MWI 1

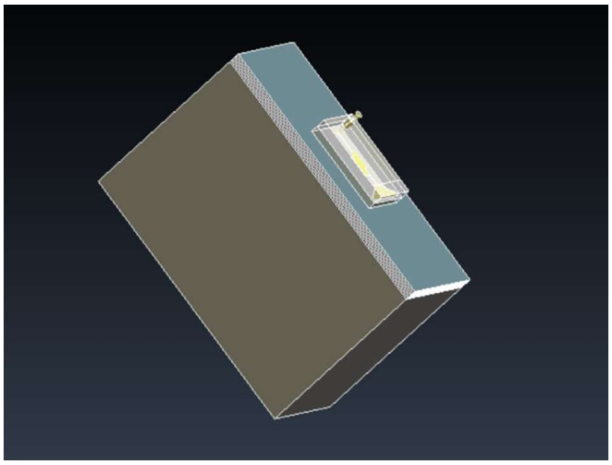

Figure 6 – 3D problem: Box and single radiating antenna

Matrix MWI 1 refers to a small 3D problem which consists of a box and a single radiating antenna. This matrix is smaller compared to Matrix MWI 2 and allowed to validate some of the algorithms developed. Its size is 257361×257361 and it has 3995111 non-zero elements. Thus, also a sparse matrix.

3.2.2 Matrix MWI 2

Matrix MWI 2 refers to data acquired with the prototype and the CSI algorithm described in Section 2.2.

3.2.3 Suite Sparse Matrix Collection

The SuiteSparse Matrix Collection [14] is a set of sparse matrices resulting from various applications to be used to test algorithms for sparse matrices. From this collection, 11 matrices were selected to assist in developing the work presented in this document. The list of matrices used is displayed in Table 3.

Name	Size	NNZ	Pattern symmetry	Origin
rajat11 †	135	812	89.10%	Circuit Simulation Problem
cavity05 †	1182	32747	90.50%	Computational Fluid Dynamics Problem Sequence
dw1024 †	2048	10114	98.50%	Electromagnetics Problem
cavity17 †	4562	131735	95.30%	Subsequent Computational Fluid Dynamics Problem
LeGresley_4908 †	4908	30482	97.70%	Power Network Problem
cell2 †	7055	30082	99.70%	Directed Weighted Graph
cryg10000 †	10000	49699	99.70%	Materials Problem
rajat27 †	20640	97353	96.50%	Circuit Simulation Problem
hvdc1 †	24842	158426	98.20%	Power Network Problem
t2em †	921632	4590832	99.90%	Electromagnetics Problem
young1c *	841	4,089	100.00%	Acoustics Problem
MWI 1 *	257361	3995111	100%	MWI - Box
MWI 2 *	2050637	28722845	100%	MWI – Head

Table 3 - List of matrices used to analyse algorithms

† real matrix, * complex matrix

The criterium to select which matrices to use was based on pattern symmetry, opting for the ones with pattern symmetry close to 100%, which is also the case for MWI 1 and MWI 2.

3.3 Data format

MATLAB was used to perform the finite element modelling and processing of measured data of the MWI prototype described in Section 2. The data that was exported from *MATLAB* is in *mat* format, which is a binary format.

Since it was not possible to find a straight-forward tool to import *mat* files using C++ (the primarily used programming language), it was opted for using a text format to store matrix data. Examples of file formats to store matrices are *mtx* (Matrix Market) or *rb* (Rutherford Boeing). The SuiteSparse Matrix Collection provides all matrix data in *mat*, *mtx* and *rb* formats.

The time to convert a matrix or vector from *mat* to *mtx* or *rb* depends on the number of non-zero elements of the matrices. It can, thus, become significant for large matrices. In this work, this data conversion process was done through a *MATLAB* script.

3.4 Data compression

One important consideration when dealing with the back-end processing of MWI data is that the matrices are sparse. For a non-sparse matrix, that is, a dense matrix of dimensions $n \times n$, the expected software complexity is at least $O(n^2)$. When the sparsity is considered, most of the arithmetical computations are not necessary.

When exploiting the sparsity of matrices, the goal is to make the software complexity to be approximately proportional to the number of non-zeros, that is $O(nnz)$. This is also valid for the format sparce matrices are stored.

The three most common storage formats are: list of non-zero elements, column-compressed storage and row-compressed storage. The format in which *mtx* and *rb* file extensions store matrix data is a list of the non-zeros elements, whereas the result of a factorization performed by a solver can be provided in either column or row-compressed Storage formats.

3.4.1 List of non-zero elements

The row index, column index and data of non-zero elements are stored in three separate vectors, starting from the first column, and then going through the successive columns.

An example of how a sparse matrix A would be stored is shown in Figure 7: A_i and A_j are vector of integers; vector A_x is of the same format as the matrix (integer, float, double, etc). In case the matrix is complex, there is a fourth vector to store the imaginary data information.

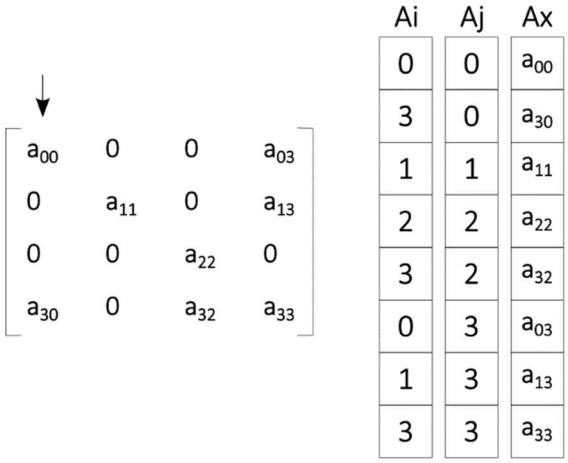

Figure 7 – List of non-zero elements example

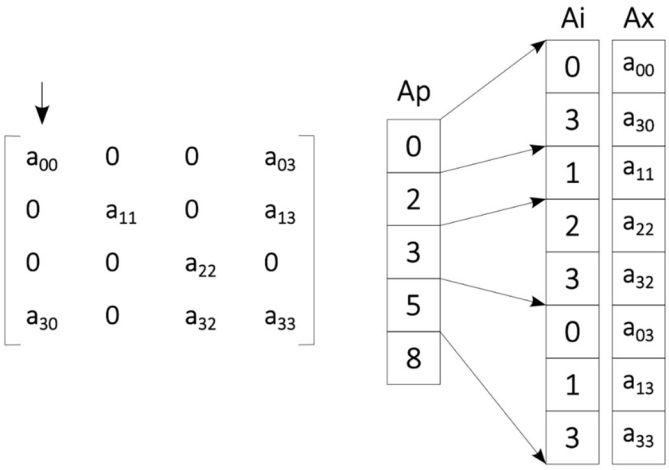

Figure 8 – Column-compressed storage example

3.4.2 Column-compressed Storage

In the column-compressed format, storage starts by the non-zero elements of first column. Instead of storing both row and column information, only row index is stored in vector A_i while vector A_p, also called "pointer vector", stores an index that indicates at which index the information of the first element of a certain column is stored.

Figure 8 helps illustrate how this compression works. For instance, information regarding the second column is equal to $A_p[1]$, whose value is 2. $A_i[2]$ is equal to 1, which means that the first non-zero element of the second column is in the second row. To know how many non-zeros there are in a hypothetical column c, the value of $A_p[c]$ should be subtracted of $A_p[c+1]$.

The length of A_i is equal to the number of non-zeros. The length of vector A_p is equal to the number of columns added by one. The last element of vector A_p is equal to the number of non-zeros for the whole matrix. In the occasion a column does not have any non-zero element, an add-in zero is included for that column.

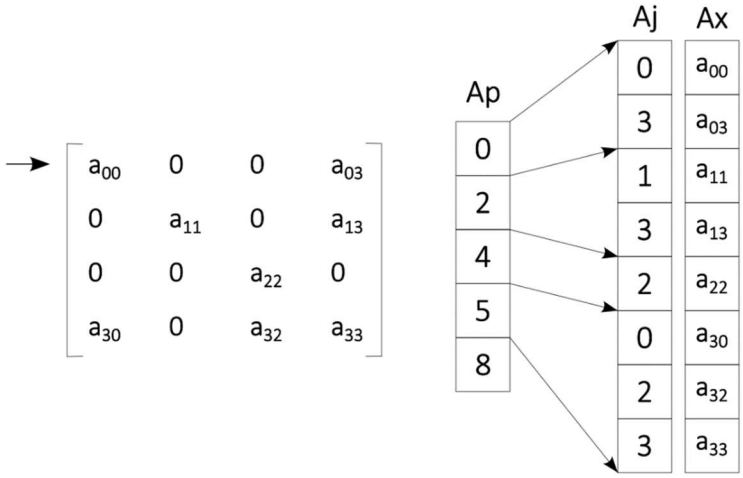

Figure 9 – Row-compressed storage example

3.4.3 Row-Compressed Storage

Like column-compressed, another possibility is to use the row-compressed Storage format. It follows the same reasoning as the column-compressed Storage format, however, instead of starting from the first column, it starts form the first row and then proceeds to store information that refers to the subsequent rows. Column information is stored in vector A_j and A_p stores the column index at which the first element of a row is stored, as depicted in Figure 9.

4 Factorizing Sparse Matrices

The idea behind the factorization of matrices is to split the solution of a linear systems is to split the solution of the linear system into the solution of two simpler linear systems. The most common factorization is called LU, which consists of transforming matrix A into a multiplication of matrices L and U, respectively, a lower and an upper triangular linear system.

After the factorization, the way to solve the system is presented in Equations 6-9.

$$Ax = b \tag{6}$$

$$L(Ux) = b \tag{7}$$

$$Ux = y \tag{8}$$

$$Ly = b \tag{9}$$

In case the matrix is symmetric positive defined (SPD), it possible to perform the Cholesky factorization, which turns matrix A into a multiplication between a lower triangular matrix L and its transposed (L^T). That is, $A = LL^T$.

If the matrix is only symmetric, but not positive defined, another possibility is a LDL^T factorization, which is a multiplication between a lower triangular matrix L, a diagonal matrix D and L^T, so that $A = LDL^T$. LU, Cholesky and LDL^T are said to be direct methods of factorization, that is, the result of the factorization should correspond to exactly the value of

Another concern is the number of zeros the resulting. That's it may be interesting to perform a permutation of the matrix, which can employ methods like AMD, COLAMD or Natural ordering. The goal of the reordering is usually to reduce the number of fill-ins (non-zero elements)

of the matrices L and U keeping the sparsity as low as possible, but also to maintain accuracy (when the numerical value of the different non-zero elements are considerable different). [15]

For instance, AMD, or the Approximate Minimum Degree ordering algorithm, "uses techniques based on the quotient graph for matrix factorization that allow us to obtain computationally cheap bounds for the minimum degree." [16] As a result of the ordering algorithms, matrix A elements permutated according to row permutation matrix P and column permutation matrix Q . The matrices are multiplied in the following order: PAQ.

Most of these algorithms are made available via open-source software and are updated by the open-source community. Examples of sparse solvers are: UMFPACK, KLU, Eigen, MUMPS, PasTix.

4.1 Methodology

The goal for the factorization step was to successfully factorize matrix MWI 2 and use the factorization results (lower and upper matrix) as inputs to the software that performs the solution of triangular linear systems (which would be accelerated using GPUs).

To check whether the factorization result was successful in an easy way, an artificial vector B was created, where B is the result of the multiplication of the sparse matrices (A) and a unit vector of appropriate size. This way, at the end of the solution of the linear system, it would be possible to easily assess whether the algorithm was working as expected.

Furthermore, since the expected value of vector X is a unity vector of size n, it was possible to define an error parameter to compare different methods of factorization, which is shown in the Equation 5.

$$Err = \frac{\sum_i^n |X[i] - 1|}{n} \tag{10}$$

4.2 UMFPACK

UMFPACK uses Unsymmetric MultiFrontal method and direct sparse LU factorization to solve unsymmetric sparse linear systems. It is written in ANSI/ISO C and it's possible to use the MATLAB interface. It's also what is in the backend of *MATLAB*'s *LU* function and $x = A\backslash \mathrm{b}$. [15]

UMFPACK has two versions: standard and SuiteSparse_long. The standard version limits the memory usage to 2GB, whereas the SuiteSparse_long version can use as much as there is available in the computer. In both cases, the matrix can be either real or complex.

The factorization step is divided in two steps: symbolic and numeric factorization. The output is an object that stores the factorization result. Following the numeric factorization, it is possible to either use the mentioned object to solve a linear system by providing a dense vector B. The other possibility is to export the factorization data.

It was decided to use the UMFPACK solver as a C library, since the solution of triangular linear systems would later be done using OpenCL's C++ interface. Starting from the smaller matrices and increasingly incrementing the size of the matrix the factorization process using UMFPACK was successfully validated. Most of the issues solved during this scaling process were related to allocation of memory.

Matrix MWI 1 was successfully factorized using UMFPACK in around 3 min and resulted in lower and upper triangular matrices of over 230 million non-zero elements, despite the original matrix presenting only around 4 million non-zeroes. It was not possible to factorize Matrix MWI 2 in the regular PC due to lack of memory.

It was then attempted to factorize MWI 2 in the server, due to its much superior processing power and memory. The factorization, however, was not possible either even after 2h. Figure 10 shows the amount of memory the process was using during the factorization attempt: 55.9 GB and 70.5 GB of virtual memory, which is considerably more memory than a regular PC would have.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
[illegible]											
[illegible]											
18019	nathali+	20	0	70.5g	55.9g	2568	R	100.0	56.9	50:13.73	main
[illegible]											

Figure 10 – Resources used while processing matrix MWI 2 in server

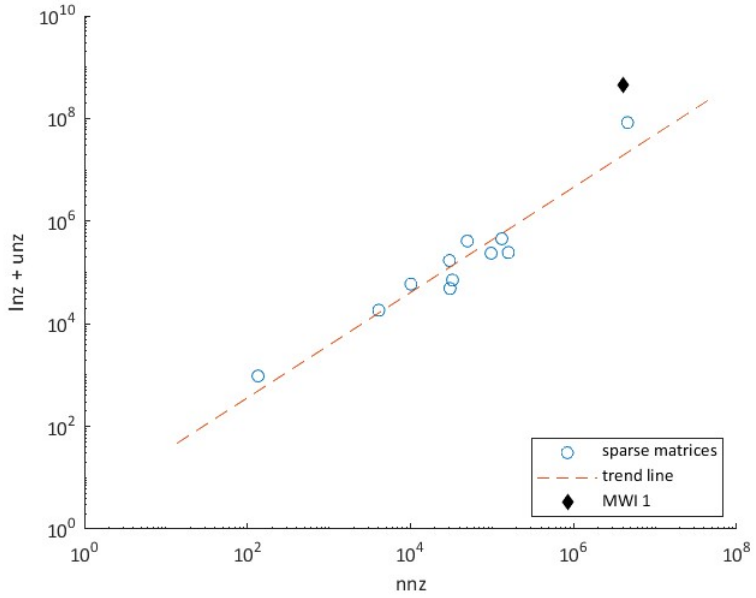

Figure 11 – Number of non-zeros before and after factorization

The data obtained from the factorization of other matrices was analysed and the number of non-zeros (nnz) before the factorization and the sum of the non-zeros of the lower (lnz) and upper (unz) was plotted in a graph, which is displayed in Figure 5. It can be observed that the matrices from the SuiteSparse collection followed approximately the trend line in red, while matrix MWI 1 has considerably more non-zero elements in the lower and upper matrices.

If MWI 2 followed the same proportion as MWI 1, it would have 3 billion non-zeros in matrix L and matrix U. The complexity that processing this number of non-zeros can be the explanation why it required so much memory to perform factorization and it took so long without completing it.

The placing of the non-zero elements plays a role in the resulting lower and upper triangular matrices. In the case of the MWI matrices, this placement depends on how the finite element problem is formulated.

An example of how this process works can be observed in the two linear systems presented in Equations 11 and 12. In both Equations, the four matrices represent, in order, the matrices P (permutation matrix), A (original matrix), L (lower triangular matrix) and U (upper triangular matrix). Both A matrices have 11 non-zeros elements, however, in Equation 11 $lnz + unz$ is 12, whereas $lnz + unz$ is 22 for Equation 12.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 9 & 0 & 9 & 0 & 9 \\ 0 & 7 & 0 & 0 & 0 \\ 9 & 0 & 9 & 0 & 9 \\ 0 & 0 & 0 & 4 & 0 \\ 9 & 0 & 9 & 0 & 9 \end{pmatrix} \tag{11}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 9 & 0 & 9 & 0 & 9 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 9 & 8 & 5 & 3 \\ 0 & 8 & 8 & 0 & 0 \\ 0 & 5 & 0 & 8 & 0 \\ 0 & 3 & 0 & 0 & 6 \end{pmatrix} \tag{12}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 5/9 & 1 & 0 & 0 \\ 0 & 1/3 & 3/5 & 1 & 0 \\ 0 & 8/9 & -2/10 & 17/24 & 1 \end{pmatrix} \begin{pmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 9 & 8 & 5 & 3 \\ 0 & 0 & -40/9 & 47/9 & -5/3 \\ 0 & 0 & 0 & -24/5 & 6 \\ 0 & 0 & 0 & 0 & -29/4 \end{pmatrix}$$

4.3 MUMPS

MUMPS was discarded at first as an option in this project as it does not allow to export the result of a factorization to allow working on accelerating the solution of triangular linear systems. However, due to the limitations found in UMFPACK, MUMPS was tested to factorize matrices MWI 1 and MWI 2.

MUMPS supports both single and double precision computations for real and complex systems. When this solver is installed, it comprises an example code in Fortran to solve non-symmetric matrices. The input for this solver is a text file in a certain format that could be easily generated using *MATLAB* based on the *mat* files.

The Fortran example program was also updated and compiled to allow using symmetric matrices as input, which is both the case for MWI 1 and MWI 2. This way, only half of the matrix needs to be included in the input file.

When the program is run, after importing the data, an analysis step takes place before the factorization. During the analysis step the amount of memory needed is estimated and in case it is not possible to allocate enough memory, the program execution fails.

By observing the execution log, it was possible to check that when the symmetric matrix solver is run, a LDL^T factorization is performed. When the asymmetric solver is run, it performs LU factorization. This information was also found in the solver's user guide [17].

The solution of matrix MWI 1's linear system was successfully run considering both single and double precision for both the symmetric (LDL^T factorization) and asymmetric solvers (LU factorization). The solution of matrix MWI 2's linear system, however, was only possible considering single precision in a LDL^T factorization. Factorization of MWI 2 failed due to lack of memory when considering double precision for symmetric and asymmetric solvers and single precision for the asymmetric solver.

4.4 Other attempts

Other direct solvers were also tested to find a tool that would allow to both factorize MWI 2 and export the factorization results. The results obtained were unsuccessful and are briefly cited next:

- KLU. It was developed to solve matrices arising in SPICE-like circuit simulation applications. It was used to successfully factorize some of the other matrices listed in Table 3 with results close to UMFPACK's, however it was not capable of factorizing neither MWI 1 nor MWI 2 after 10s of minutes. [18]

- Scipy in Google Colab. UMFPACK is in the back end of factorization functions of this Python library. Although, it is possible to use up to 25 GB of RAM memory, the machine ran out of memory during factorization of MWI 2. [19]

- SparseLU, SparseQR and SimplicialLDLT from Eigen. In all cases the factorization was not completed after several minutes neither for MWI 1 nor MWI 2. All Ordering options were tested (COLAMD, AMD, Natural). [20]

4.5 Summary of results

Tables 4 and 5 compare the obtained results for MWI 1 and MWI 2.

Solver	**MUMPS LDL^T**		**MUMPS LU**		**UMFPACK**
Precision	Single	Double	Single	Double	Double
Error	1.38E-05	2.12E-14	1.40E-05	1.99E-14	1.72E-20
Time	9m30s	9m24s	5m34s	5m26s	2m50
Memory Use Estimation	1207 Mb	2368 Mb	2292 Mb	4536 Mb	NA

Table 4 - Factorization results for MWI 1

From the obtained results, LDL^T factorization is the most memory-efficient factorization method to deal with the data from MWI problems. Considering the double precision and only

MWI 1, UMFPACK offers the smallest error, although it was not capable of solving the linear system with MWI 2.

Solver	**MUMPS LDL^T**		**MUMPS LU**		**UMFPACK**
Precision	Single	Double	Single	Double	Double
Error	8.03E-04	NA	NA	NA	NA
Time	62m32s	NA	NA	NA	NA
Memory Use Estimation	13623 Mb	26886 Mb	25549 Mb	50713 Mb	NA

Table 5 - Factorization results for MWI 2

Pardiso, which was the previous solution used for solving the linear system, also employed a LDL^T factorization. According to the User guide:

> "The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm or the nested dissection algorithm from the METIS package, followed by the parallel left-right looking numerical Cholesky factorization $PAP^T = LL^T$ or $PAP^T = LDL^T$ for symmetric, indefinite matrices. The solver uses diagonal pivoting or 1×1 and 2×2 Bunch-Kaufman pivoting for symmetric indefinite matrices and an approximation of X is found by forward and backward substitution and iterative refinement." [21]

5 Iterative Solvers

Given the observed limitations when using direct solvers to factorize and solve the linear systems related to MWI, some attempts on using iterative solvers were done. As the name suggests, an iterative solver computes the solution of a linear system through a sequence of approximations until either a desired tolerance or the set maximum number of iterations is

reached. The used library for this type of solver is called *Eigen* and two of the built-in iterative solvers were tested: *ConjugateGradient* and *BiCGSTAB*. *BiCGSTAB* stands for Biconjugate gradient stabilized method.

For iterative solvers, it is possible to use preconditioners, which may accelerate convergence to the solution. There are three preconditioners available: Diagonal, Identity and *IncompleteLUT*.

The *IncompleteLUT* preconditioner is based on a direct method and for both tested MWI matrices, it did not complete the first iteration and thus, results for this preconditioner are not presented.

5.1 Matrix MWI 1

When running an iterative solver, it is necessary to set a tolerance and maximum number of iterations. When either of those parameters are met, the solver solution is presented. For this problem, the tolerance (Equation 13) was set to be very small ($1e-10$) and the solver was run considering number of iterations from 100 up to 3000.

$$Tolerance = AX - B \tag{13}$$

A summary of the results is presented in Figure 12. *BiCGSTAB* solver presents an overall tendency to converge to a solution as the number of iterations increases. Tolerance, as well as the defined Error metric tend to decrease (Equation 10).

The *ConjugateGradient* solver does not converge to a solution. In *Eigen*'s documentation [20], it is said that the *ConjugateGradient* method is most suited for SPD matrices (symmetric and positive defined), which is not the case for MWI 1. According to [22] "For the CG method the matrix A should ideally be positive-definite. The application of CG to indefinite matrices may lead to failure, or to lack of convergence."

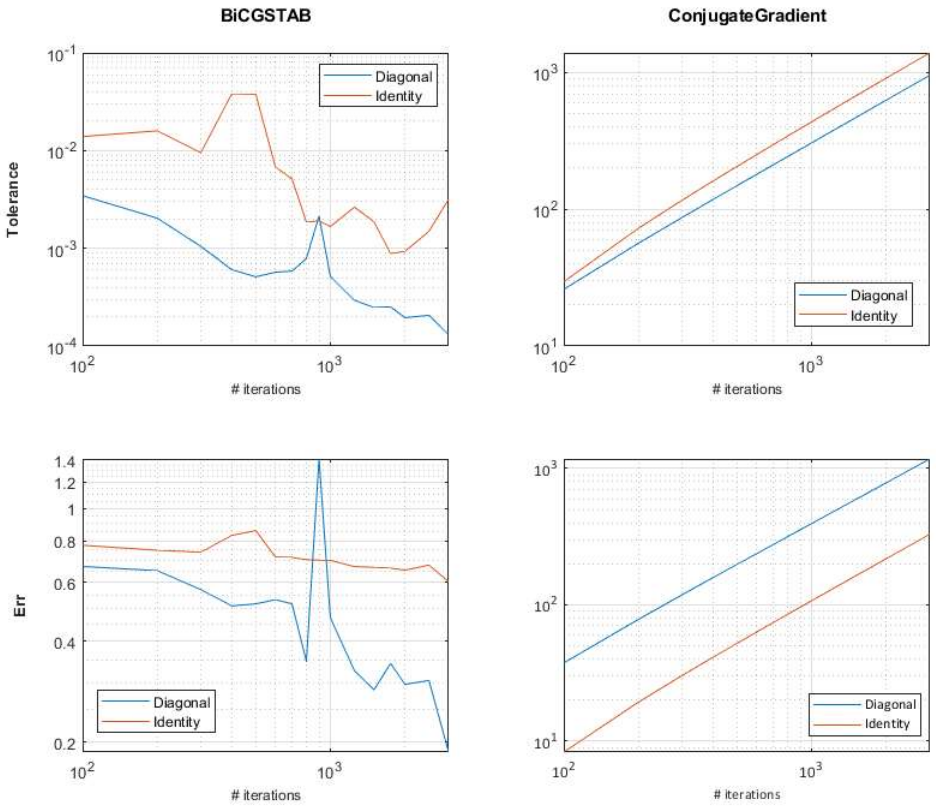

Figure 12 - Err and Tolerance evolution with number of iterations for BiCGSTAB and ConjugateGradient iterative solvers

Vector Index	Found solution for X		Expected X		B vector	
	Real	Imaginary	Real	Imaginary	Real	Imaginary
0	1.19109	-0.168486	1	0	-2.31484e-3	6.845032e-9
1	0.41729	-0.121286	1	0	1.01977e-3	0
6	1	-2.69182e-13	1	0	1	0
7	1	-2.69182e-13	1	0	1	0

Table 6 – Samples of the iterative solver solution

When analysing the results obtained that presented the smallest tolerance (*BiCGSTAB*, Diagonal Preconditioner and 3000 iterations), it was possible to notice that the solution to the linear system varies considerably, although the expected value of X is a unit vector.

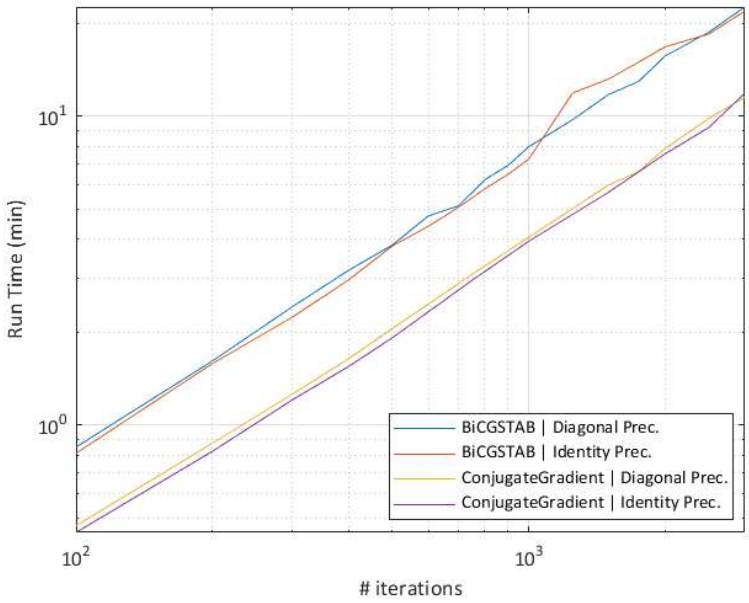

Figure 13 – Run time of iterative solvers

Samples of the obtained soluiton are presented in Table 6, alongside the expect value and the corresponding right-hand side value. It can be observed that the solver does not seem to converge for some of the elements of the vector. This issue may be due the algortihm that aims to minimize the tolerance, which is proportional to $AX - B$. Thus, smaller values of vector B affect less the overall tolerance.

When comparing run time of direct and iterative solvers to solve matrix MWI 1, although they are in the same order of magnitude of magnitude (between 1 and 10 minutes), the results obtained from iterative solver were not as precise. A plot of the time taken for each solver to run is shown in Figure 13.

5.2 Matrix MWI 2

Some efforts were made to try to factorize matrix MWI 2 with *BiCGSTAB* solver, however, the results were also not satisfactory. The solver's error metrics did not decrease as the number of iterations reached hundreds of iterations, and when observing the resulting X vector, the results were like those reported for matrix MWI 1 and shown in Table 6.

6 Solving Triangular Linear Systems

The initial proposal of this work was to focus on the acceleration of solution of the triangular linear systems resulting from the factorization of MWI 2. However, among the open-source solvers tested, it was not possible to find a direct solver that would allow to both factorize and export the results for this matrix. To cover the initial intent of this thesis, the developed prototypes of triangular linear system solvers using a GPU were therefore tested using matrices from the SuiteSparse Collection.

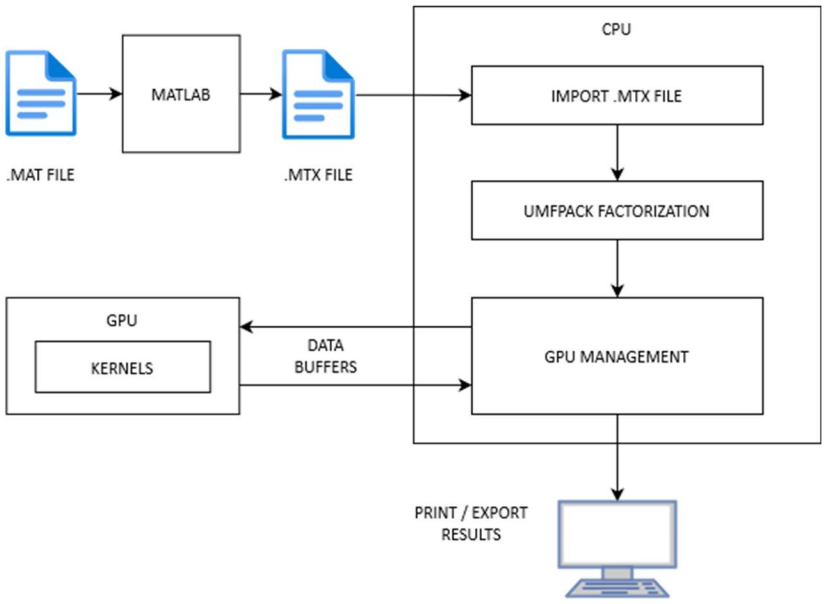

Figure 14 – Block diagram of the programs that use GPU

For the prototypes described next, it was necessary to import the factorization done by UMFPACK. A block diagram that illustrates all the process the data is subject to is shown in Figure 14. The conversion from *mat* to *mtx* file was done through a *MATLAB* script. The process of importing *mtx* files for both input matrix *A* and right-hand sides B was done through a C++ script based on the *fstream* library.

The output of the UMFPACK factorization consists of 2 matrices and 3 vectors, as follows:

a) L: lower triangular matrix in row-compressed format.
b) U: upper triangular matrix in column-compressed format.
c) P: vector with indexes corresponding to the row permutation matrix of A.
d) Q: vector with indexes corresponding to the column permutation matrix A.
e) R_S: scaling factor for each row of matrix A.

Given an input matrix A, the result of the factorization would correspond to:

$$P(R_s \cdot A)Q = LU \tag{14}$$

Based on UMFPACK's factorization outputs, it is possible to split the solution of linear system $AX = B$ into four steps, which are:

- Step 1: $B' = P(B \cdot R_s)$
- Step 2: $Y = L\backslash B'$
- Step 3: $X' = U\backslash Y$
- Step 4: $X = QX'$

Steps 1 and 4 represent the reordering operations of a vector. Steps 2 and 3 represent the resolution of a lower and upper triangular linear system, respectively. Steps 1-4 were developed in C++ using sequential algorithms and successfully validated to serve as a starting point to work on the hardware acceleration algorithms.

6.1 Algorithms and parallelization

Assuming that the factorization process was successfully performed resulting in a lower triangular matrix L and an upper triangular matrix U, a linear system $Ly = b$ (Equation 14) could be then solved as shown in Equations 15-18. This process is sequential and requires that the previous step was solved beforehand, which illustrates the challenges in performing this task with parallelism.

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \tag{15}$$

$$y_1 = b_1 / l_{11} \tag{16}$$

$$y_2 = (b_2 - l_{21} y_1) / l_{22} \tag{17}$$

$$y_3 = (b_3 - l_{32} y_2 - l_{31} y_1) / l_{33} \tag{18}$$

$$y_4 = (b_4 - l_{43} y_3 - l_{42} y_2 - l_{41} y_1) / l_{44} \tag{19}$$

Alternatively, however, the linear system $Ly = b$ could be solved as shown in Equations 10-16. Still, all steps must be solved sequentially, although it is possible to solve independently each vector line in Equations 11 and 13.

$$y_1 = b_1 / l_{11} \tag{20}$$

$$\begin{pmatrix} b'_2 \\ b'_3 \\ b'_4 \end{pmatrix} = \begin{pmatrix} b_2 \\ b_3 \\ b_4 \end{pmatrix} - \begin{pmatrix} l_{21} \\ l_{31} \\ l_{41} \end{pmatrix} x_1 \tag{21}$$

$$y_2 = b'_2 / l_{22} \tag{22}$$

$$\begin{pmatrix} b_3'' \\ b_4'' \end{pmatrix} = \begin{pmatrix} b_3' \\ b_4' \end{pmatrix} - \begin{pmatrix} l_{32} \\ l_{42} \end{pmatrix} x_2 \tag{23}$$

$$y_3 = b_3''/l_{33} \tag{24}$$

$$(b_4''') = (b_4'') - (l_{43})x_3 \tag{25}$$

$$y_4 = b_4'''/l_{44} \tag{26}$$

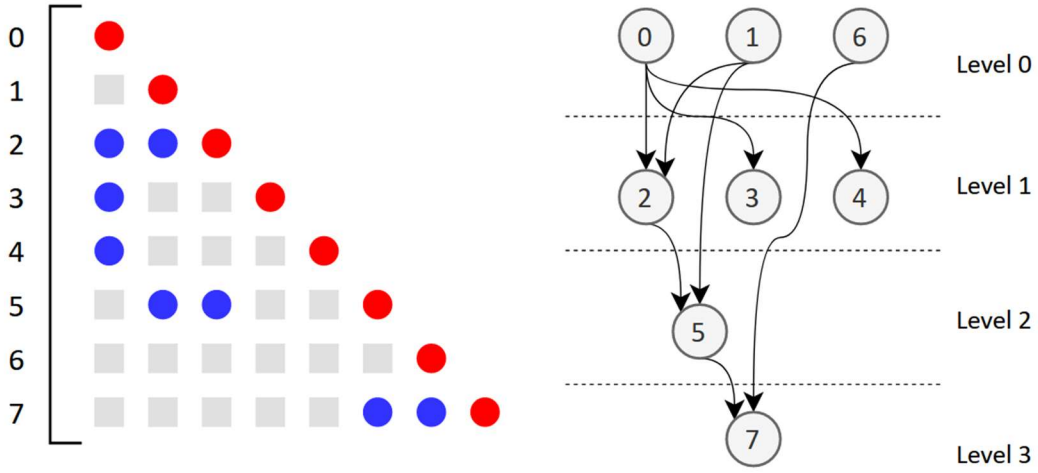

Figure 15 – Parallelization of a sparse matrix based on graphs. Non-zeros are represented by blue and red circles. Red circles for the diagonal elements and blue for non-diagonal elements.

When matrices are sparse there may be more room for parallelism since most of the arithmetical operations are not necessary. An example on how to parallelize a sparse matrix is presented in Figure 15. It consists of mapping the dependencies between rows to find which rows could be solved in parallel and generating a graph based on that. In the example, rows 0, 1 and 6 only have one element (in the matrix diagonal), therefore, they are at Level 0 and can be solved with no dependencies. Row 5 depends on rows 1 and 2. Although row 1 is ready when Level 0 rows are solved, it depends also on Row 2, which also depends on Row 1.

This approach is often referred to as graph colouring (each execution path is represented by a colour) [23] and has a mandatory pre-processing phase to map the parallelism. The pre-processing increases in complexity as the number of non-zeros/sizes of the matrix increases as well. This approach is not advised for matrices that have strong links between rows, that is, when the matrix's rows depend on the result from the previous one.

One way to orchestrate the solution is to maintain a counter for each node that is decremented whenever a dependency is solved. Once the counter reaches zero, the unknown value of the row can be solved. This is called a self-scheduling algorithm. [23] With this algorithm, it is necessary to synchronize rows. Algorithms that do not required synchronization are called sync-free.

Another alternative is to explore the parallelism of the matrix-vector multiplication (also referred to as *SpMV* kernel). In general, *SpMV* has better parallelism than solving a triangular system [7]. The basic premise would be similar to solving in parallel each row of Equations 20 and 22.

The first algorithm presented by [7] is called Column Block. As illustrated in Figure 16, the idea consists in solving the linear system $A_1X_1 = B_1$ sequentially to find the values of vector X_1. Then, the value column matrix C_1 is multiplied by X_1 and the resulting column vector is used to subtract the values of B_2, B_3 and B_4. Next, the linear system $A_2X_2 = B_2$ is solved and, similarly, C_2 is multiplied by X_2 and the values is subtracted of B_3 and B_4, and so on. *SpMV* parallelism is explored in the multiplications C_1X_1, C_2X_2 and C_3X_3.

The reference paper also presents two other algorithms that would explore parallelism in a similar way: they are called Row Block and Recursive Block algorithms. The sectioning of matrix A is shown in Figure 17. The order to solve the linear system follows the same reasoning that was explained for the Column Block algorithm. In [7] the three block algorithms (column, row and recursive) are implemented on modern GPUs, and an adaptive approach that can automatically select the best kernels according to input sparsity structures is proposed.

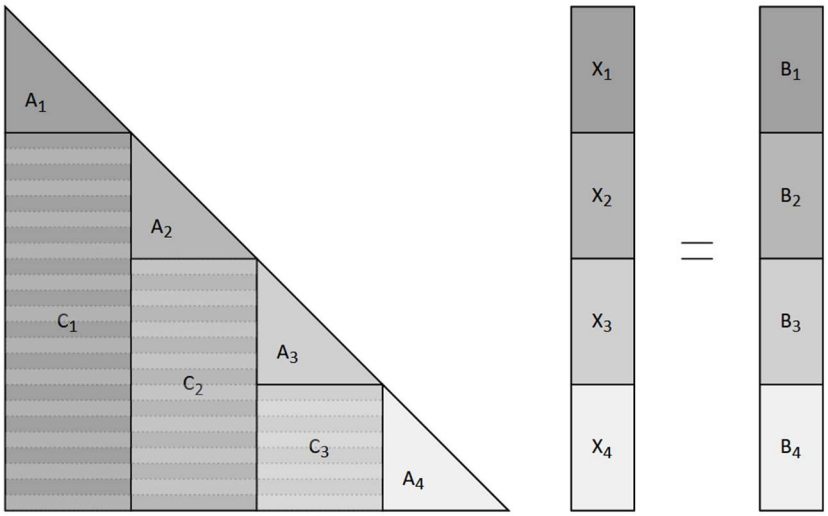

Figure 16 – Column Block algorithms' graphical representation

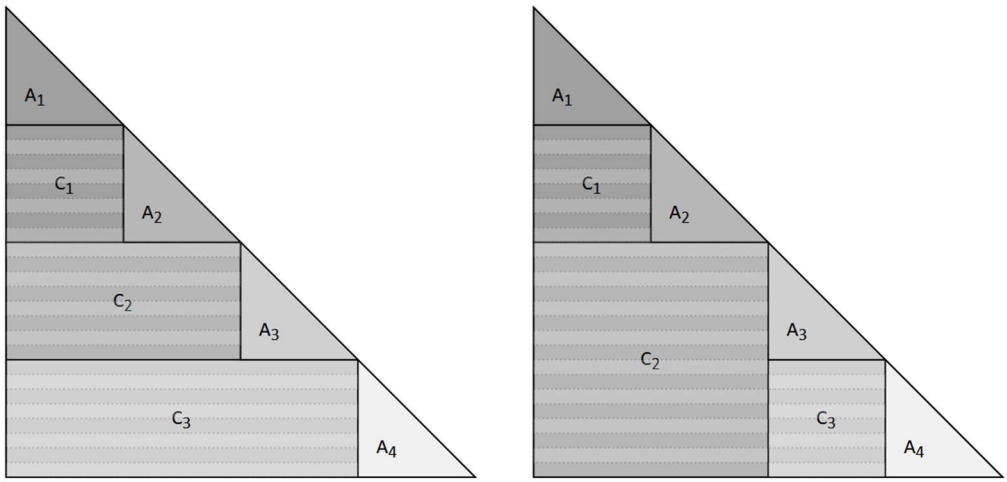

Figure 17 – Row Block and Recursive Block algorithms' graphical representation

In case the linear system has multiple right-hand sides, which can be solved parallelly. This is called a *SpTRSM* kernel. According to [8], when solving multiple right-hands sides, the cost of the

pre-processing and synchronization is reduced, which may lead to improvements in the overall performance.

Another detail to account for is how the matrix data is provided. For instance, taking the data format which is the output of the UMFPACK solver. The lower triangular matrix (L) is provided in row-compressed format and the upper triangular matrix (U), in column-compressed format.

The algorithm to solve the linear system with L is presented in Algorithm 1 and the algorithm to solve the linear system with U is presented in Algorithm 2.

```
1   for j = 0 to n-1 do
2         row_ptr = Lp[j]
3         next_row_ptr = Lp[j+1]
4         nnz_in_row = next_row_ptr-row_ptr
5
6         acc = 0
7         for k = 0 to nnz_in_row-2 do
8               col = Lj[row_ptr+k]
9               acc = acc + Lx[row_ptr+k]*Y[col]
10        end for
11
12        B'[j] = B'[j] - acc
13        Y[j] = Bl[j]/Lx[next_row_ptr-1]
14  end for
```

Algorithm 1 – Solving $LY = B'$ with L is in row-compressed format

In Algorithm 1, since data is in Row-Compressed Storage format, the algorithm starts from the first row and propagates onwards (line 1). Starting from the second row, in case the element in the first column is not null, it is multiplied by the value of Y computed in the previous step (line 7-10) and subtracted of vector B (line 12). Then, the value of the second element is computed (line 13). This process is repeated for every row of the triangular matrix.

```
1   for j = n-1 to 0 do
2         col_ptr = Up[j]
3         next_col_ptr = Up[j+1]
4         nnz_in_col = next_col_ptr-col_ptr
5
6         X'[j] = Y[j]/Ux[next_col_ptr-1]
7
8         for k = 0 to nnz_in_col-1 do
9               row = Ui[col_ptr+k]
10              Y[row] = Y[row] - Ux[col_ptr+k]*X'[j]
11        end for
12  end for
```

Algorithm 2 - Solving $UX' = Y$ with U is in column-compressed format

For the resolution of the upper triangular matrix (Algorithm 2), the process starts from the last row of the matrix. The data is in column-compressed storage format, so to make the best of the way data is stored, after each value of X' is found (line 6), for all the non-zeros of that column, the relative element value of Y is updated (lines 8-11).

6.2 OpenCL

OpenCL stands for Open Computing Language is an open. It is an open royalty-free standard for general purpose parallel programming across different platforms, which includes CPUs, GPUs, and other processors. It gives software developers more portability and efficiency, as the same code can be compiled for different processing platforms. It also supports a wide range of applications and provides a low-level, high-performance, and portable abstraction.

The programs that run in parallel in a platform are called kernels and they use a subset of ISO C99 with extensions for parallelism.

A schematic with the abstractions used by OpenCL is shown in Figure 18. The kernels are executed parallelly in the work-items and to each work-item, an item index is attributed. All work-

items execute the same code. A group of work-items is called a computer unit, or work-group. Each work-group has also an associated group index.

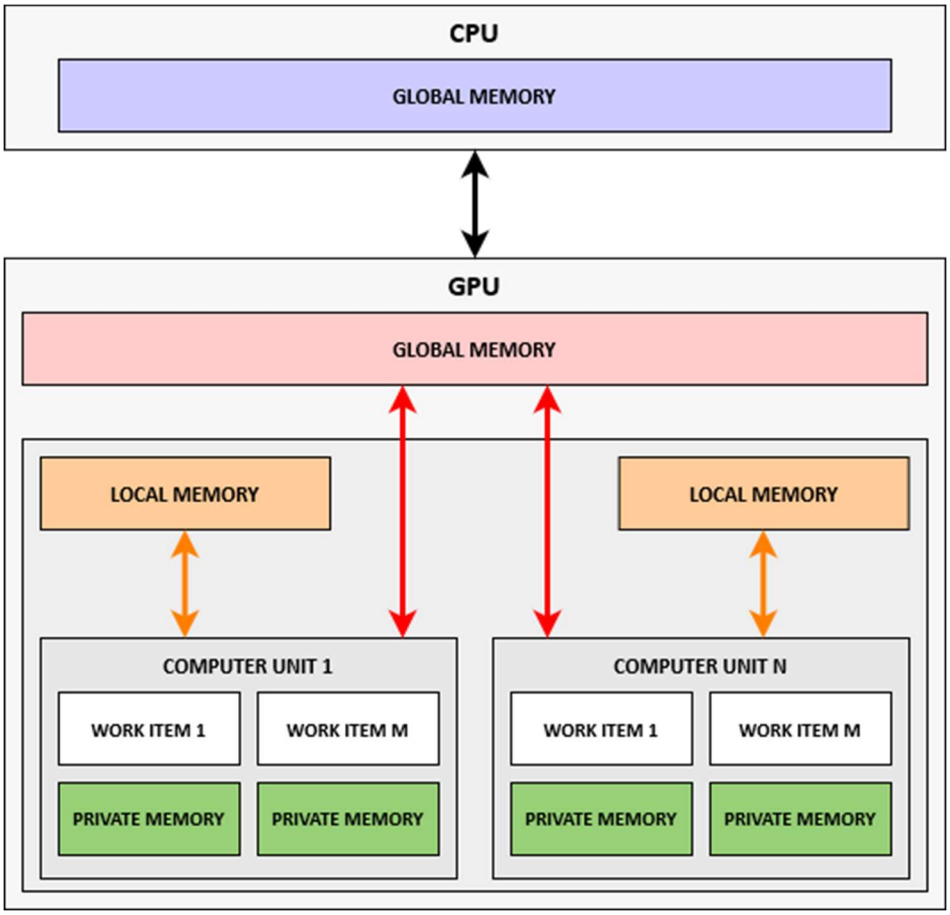

Figure 18 – OpenCL memory architecture

In Figure 18 it is also possible to see the 3 different levels of memory in a GPU: global, local, and private. Global memory is common for all work-items and work-groups in a GPU. Global memory it may be cached depending on the capabilities of the device.

Local memory is common for all work-items in the same workgroup. It may be implemented as a dedicate region of memory on the OpenCL device. Alternatively, the local

memory region may be mapped onto sections of the global memory. Lastly, private memory, which refers only to a specific work-item.

The GPU used in this work, the global memory size is 15.9 Gb and local memory size is 48 kb (according to OpenCL's *clGetDeviceInfo* function). The private memory size is only a few bytes. [24] Access time to global memory is the slowest, followed by local memory and, lastly, private memory. [24] Ideally the access to global memory should be minimized whenever possible, as it hinders the kernel performance.

The communication between CPU and GPU (represented by the arrow in black) is done by data buffers, which must be declared and serve as inputs for the kernels. The biggest challenge when dealing with GPUs is the memory management, which must be done explicitly.

To successfully run a program in a GPU using OpenCL, it is necessary to follow the following steps:

a) Set up OpenCL
    I. Identify devices and pick device with which to work
    II. Create command queue for the picked device
b) Compile kernels – The process consists of importing the kernel programs written in C and compiling then in run-time.
c) Buffer input data - Each data input must be added into an OpenCL data buffer, which takes parameters like data type, size, whether it's a read only, write only or read-write data. The process of buffering data may significantly affect the performance of a program, as will be presented later in this section.
d) Run kernels - After they are compiled, it's possible to create function-like structures and then run them with the data buffers as inputs, possibly also some constants.
e) Copy output data from buffer – After running all kernels, the results are exported from the GPU to the CPU.

6.3 Prototype 1 – Solve multiple RHSs in parallel

The matrix MWI 2 originates from a MWI problem has 24 right-hand side vectors, one for each antenna. Thus, the next prototype aims to do is to solve the triangular linear systems of the 24 right-hand side problems parallelly.

After exporting data from UMFPACK, the implementation consisted of solving sequentially step 1, which consisted of a reorder and multiplication operation between vectors. The output vectors B_{0-2} , along with L_i, L_p, L_x, U_j, U_p and U_x were added to read-only buffers of appropriate size and type, a read-write buffer for Y_{0-23} and a read-only buffer for X'_{0-23} were also created. The OpenCL kernels were written following Algorithm 1 and Algorithm 2.

The comparison between the execution of this algorithm sequentially in the CPU repeat 24 times and the parallel execution of 24 work-groups is shown in Table 6. The prototype was evaluated for the 3 largest real matrices because from the SuiteSparse collection. For smaller matrices, the run time was not consistent and varied widely. The smaller matrices took a shorter amount of time to be run and were more influenced by other processes running on the same machine

Kernel	Platform	rajat27	hvdc1	t2em
Step 2	CPU	15.120 ms	20.808 ms	3520.008 ms
	GPU	1.716 ms	1.877 ms	211.905 ms
Step 3	CPU	20.232 ms	20.952 ms	4784.856 ms
	GPU	121.699 ms	155.228 ms	6704.505 ms

Table 7 – Performance of prototype 1 compared to CPU

From the obtained run times, it can be observed that step 2 performed consistently better when run parallelly in a GPU. Step 3, however performed generally worse. This run time analysis does not include the overhead of compiling kernels and moving memory to and from GPU, which will be done in a later section.

The explanation as to why step 2 was accelerated lies in the data from it being represented in a row-compressed format, which minimizes the need to update data in global memory. The accumulation variable acc is stored in the work-item's private memory (lines 6 and 9 of Algorithm 1), minimizing the accesses to global memory by only updating vector Y_{0-23} once per row.

In case step 3 (Algorithm 2) were to be altered so that it is execute more similarly to Algorithm 1, thus updating X' only once per row, an exhaustive algorithm to check whether a column has one element of the row would also result in a lot of accesses to global memory. The likely better solution would be to convert data to compress-row storage before buffering data into the GPU. The conversion would have to be performed only once per matrix, as all right-hand sides have the same data as input.

This is an algorithm that has complexity close to $O(n\ nnz)$: for all rows, almost all non-zero elements should have their row index checked. When we consider, for instance, the factorization results for matrix MWI 1, it has nnz of around 300 million elements and n equals to around 250 thousand.

6.4 Prototype 2 – Solve multiple RHSs in parallel and store Y in local memory

Based on the results of Prototype 1, in order to improve the speed of step 3, the proposal was to use local memory to minimize accesses to global memory.

To solve sequentially the problem $UX' = Y$, element j of vector Y is first fetched to find the respective value of $X'[j]$. Vector Y is subtracted of the multiplication of $X'[j]$ and column j of matrix U. Taking the assumption that matrix U elements are predominantly placed around the diagonal (which is true for matrix MWI 1 and MWI 2), the proposal was to store parts of Y that are more likely to be needed in the local vector.

Figure 19 illustrates how the adapted algorithm works. As it was already shown in Algorithm 2, it starts by the last row. First, local Y is updated with the k values of Y (in the image, $k = 4$). Next, the value of $X'[j]$ is found by dividing $Y[j]$ by the element in the diagonal of matrix

U's j-th row. Finally, either Y or local Y are updated according to the non-zero elements in matrix U's j-th column accordingly.

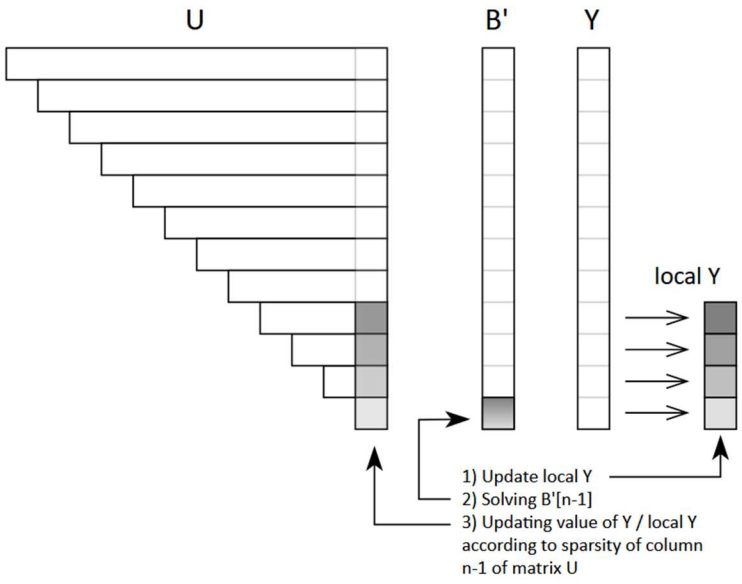

Figure 19 – Visual representation of Prototype 2's algorithm

The way local Y is updated is shown in Figure 20 in an example where $k = 4$ and size of Y (n) is 12 during 5 iterations. Each colour represents one different index of the local Y vector. As it can be inferred, local Y acts like a ring buffer. A pointer variable is used to track at which index of local Y the corresponding Y vector element is stored.

The amount of memory that can be used for local Y is, however, limited. The GPU used in this implementation has 48 Mb of local memory available. This would allow to store 2 Mb for each right-hand side. All information in this computation is stored with double precision, that is, 8 bytes per matrix/vector element. It is possible to store 256 doubles with 2 Mb of memory. Since this prototype was to then be altered to support MWI data, which is in complex format, the code was written to store 128 elements in local Y.

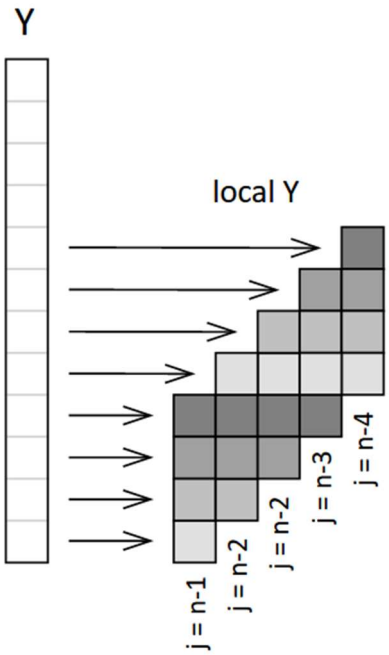

Figure 20 – Ring buffer behaviour for local Y in Prototype 2.

Since all right-hand side calculation are independent, there is no need to any synchronization between work-items. The proposed prototype was successfully implemented, run, and compared to what was obtained for Prototype 1 and the CPU. The obtained results for this implementation are presented in Table 8, where it can be observed that some improvements compared to prototype 1 for all the 3 analysed matrices by 13-18%. The performance, however, is still far from the performance obtained with the CPU.

Table 8 also presents how many times local Y and global Y were updated for each of the analysed matrices. For *hvdc1*, local Y updates represented 81.1% (the highest percentage) and a reduction in 16.6% in execution time of the kernel when compared to Prototype 1. *t2em* has both the smallest acceleration rate (13.1%) and smallest percentage of access to local Y (26.9%).

Kernel	Platform	rajat27	hvdc1	t2em
Step 3	CPU	20.232 ms	20.952 ms	4784.856 ms
	GPU – Prototype 1	121.699 ms	155.228 ms	6704.505 ms
	GPU – Prototype 2	99.586 ms	129.501 ms	5832.055 ms
	Prototype 2 / Prototype 1	81.8% -18.2%	83.4% -16.6%	86.9% -13.1%
	Update local Y	60025	78221	10968137
	Update global Y	43931	18199	29849594
	Update local Y / total	57.7%	81.1%	26.9%

Table 8 - Performance of prototype 2 compared to CPU and prototype 1

6.5 Prototype 3 – Solve multiple RHSs in parallel and store accumulation variable in local memory

Another proposal to use local memory was by creating an accumulation variable to store the multiplication between matrix U elements and the computed values of X'. This accumulation variable vector also acts as a ring buffer, like in Prototype 2.

In Figure 21, an illustration of the ring buffer is shown, where each colour represents a different index of the accumulation vector. In each block, the index to which the vector element refers to is written. In this case, the size of the ring size is $k = 4$ and it illustrates also that, to solve the upper triangular linear system, we start from the last (n-th) row.

The changes to Algorithm 2 consisted of, after the value $X'[j]$ was found, testing whether the row index of matrix U's column j had a respective accumulation variable in the local memory at that iteration of the algorithm. Before computing the value of a X' element, the value of $Y[j]$ is subtracted of the accumulation vector.

Considering the same 48 Mb of available data, the size of the ring buffer is still 128 for each of the 24 right-hand sides. The obtained results for this prototype are presented in Table 9,

which are very close to the results obtained for Prototype 2. Prototype 3 performed better than Prototype 1 for the analysed kernel, however, it is still slower than running the same task sequentially in a CPU.

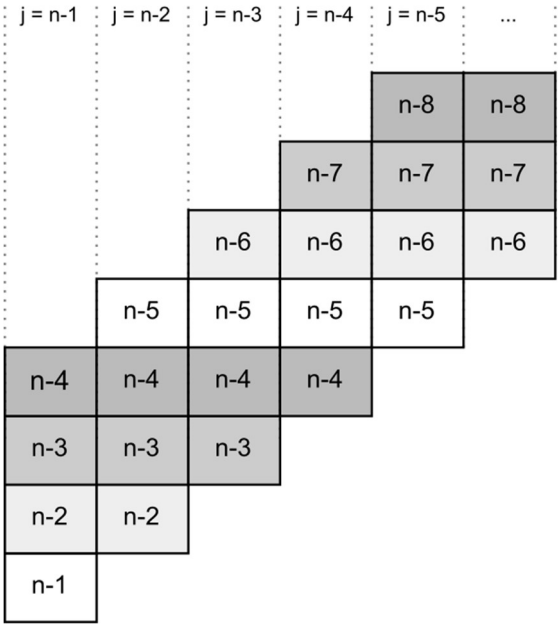

Figure 21 – Ring buffer behaviour for accumulator in Prototype 3

The number of updates to local acc for each matrix presented in Table 9 is the same as those reported in Table 8 for Prototype 2. This indicates that both prototypes attempt to optimize the kernel in an extremely similar way, which also explains the almost identical run times.

Kernel	**Platform**	**rajat27**	**hvdc1**	**t2em**
Step 3	CPU	20.232 ms	20.952 ms	4784.856 ms
	GPU – Prototype 1	121.699 ms	155.228 ms	6704.505 ms
	GPU – Prototype 3	99.678 ms	129.357 ms	5829.387 ms
	Prototype 3 / Prototype 1	81.9% -18.1%	83.3% -16.7%	87% -13%

	Update local acc	60025	78221	10968137
	Update global Y	43931	18199	29849594
	Update local acc / total	57.7%	81.1%	26.9%

Table 9 - Performance of prototype 3 compared to CPU

6.6 Prototype 4 – Column Block Algorithm

The next prototype presented attempts to use another parallelization technique that was presented and described previously in this section: the Column Block Algorithm.

The algorithm was implemented considering that the data input is in Row-Compressed format. Even though this is not also the format the output format that UMFPACK exports the upper matrix, it was deemed more appropriate to implement the parallelization of the *SpMV* kernel. Additionally, the upper matrix in Column-Compressed converted to Row-Compressed format to solve the very similar problem.

When this kernel was implemented, the first thing to be noticed is that the complexity of the algorithm increased considerably. Algorithm 1 has around 10 lines of code, whereas the Column Block algorithm has around 60 lines.

The most relevant implementation decisions taken while developing this kernel's algorithm are the list below:

a) The kernel was developed so that it would be possible to run the algorithm considering any size for the small triangular matrices. This parameter was named granularity.
b) From the granularity, it was possible to determine a parameter called chunks, which is the number of times the sequence of solving the small linear system, matrix-vector multiplication and updating vector B.
c) The number of work-items for this kernel was said to be equal to the number of lines of the matrix, but at each step of the algorithm, there would be a test to assess whether the work-item would be executed.

d) The triangular linear system is solved parallelly in all work-items. This represents a redundancy, but the motivation for that is that there would be no need to synchronize the obtained values of X vector. Also, in case only work-items were to solve the linear system, the others would be in an idle state waiting for the resolution of this step.

e) To assure that all work-items had updated the value of vector B before solving the following triangular linear system, local and global memory barriers were added. According to OpenCL's user manual:

> "The OpenCL C programming language provides a built-in work-group barrier function. This barrier built-in function can be used by a kernel executing on a device to perform synchronization between work-items in a work-group executing the kernel. All the work-items of a work-group must execute the barrier construct before any are allowed to continue execution beyond the barrier". [25]

The prototype worked appropriately well for the smallest matrix only (*rajat11*), which served to validate that the algorithm was correctly implemented and for different values of granularity. No acceleration could be observed (instead, the prototype took 400 times more to be executed than a CPU), which is likely do to the relatively small size of the matrix that would take a short time to solve in a CPU (only a few milliseconds). The overhead previously mentioned when dealing with GPUs and the much more complicated algorithm also must be considered.

The implementation, however, did not work for any other of the listed matrices, starting from *cavity05*. When solving the *cavity05*'s linear system, the results obtained were close in value to the expected in first few values of X (unit vector), but increasingly became more different, indicating a propagation of the error. The values of X were also different at each run. This last characteristic and having had the algorithm validated for matrix *rajat11* signals that the failure was due to the synchronization between work-items.

The problem observed may be due to the relaxed memory model, which means that "the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times" [25]. This is a characteristic of the memory type:

> "Local memory is consistent across work-items in a single workgroup at a work-group barrier. Global memory is consistent across work-items in a single workgroup at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel." [25]

After encountering this and not finding a solution for them in OpenCL's, it was decided not to continue implementing solution. The referenced paper [7] does not hint on how to deal with such issues.

6.7 Comparison between CPU and GPU

From the previous session, it could be observed that the solution of the lower triangular linear system using the developed kernel executed 24 times in parallel was faster than executing the same task sequentially in the CPU. The comparison between the execution time in CPU and GPU should also consider the overhead of transferring data, compiling kernels, and searching for the target to be used.

The best result among the prototypes (Prototype 3) is presented in the Figure 22 alongside the necessary time to run the same task in a CPU. The graph also represents the fraction of time for each different task. As it was seen from previous results, step 2 was accelerated and represents a very small fraction of the total time to run the whole task in a GPU. Step 3, however, was not accelerated.

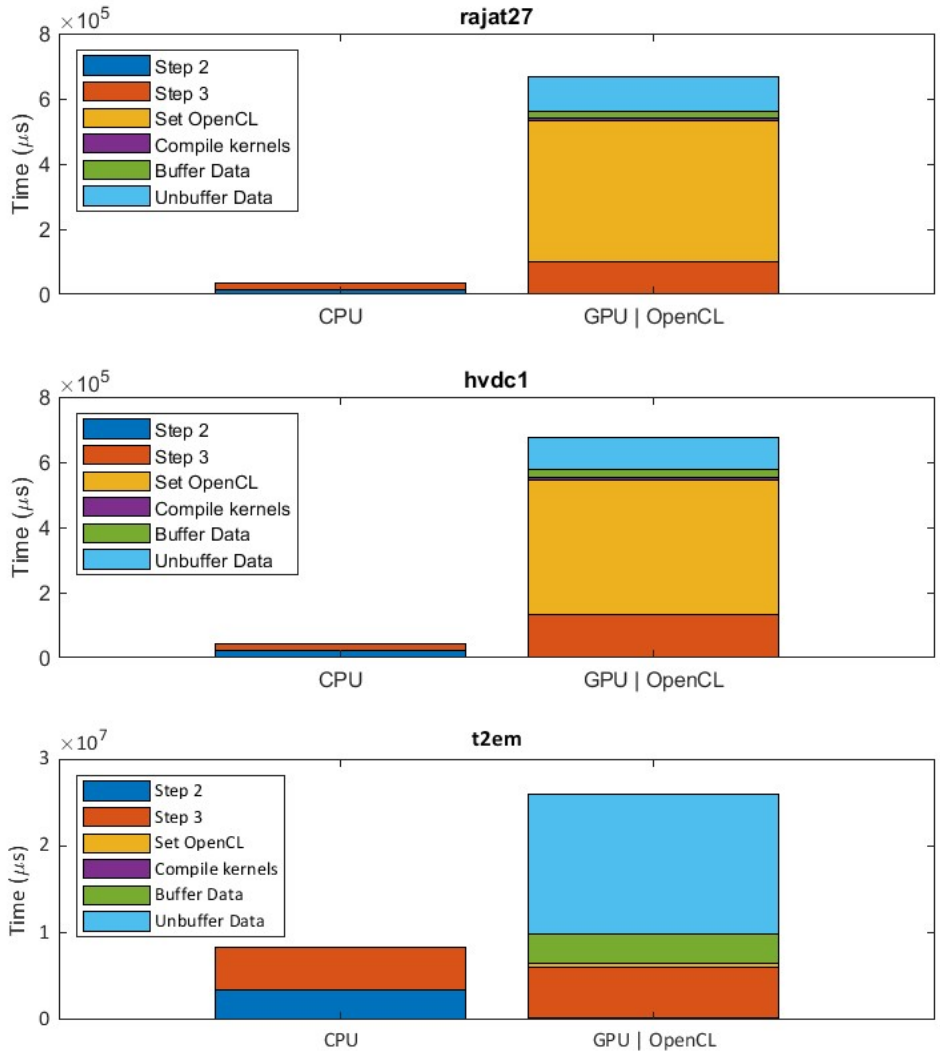

Figure 22

Most of the time to run the task on the GPU was spent in buffering and unbuffering data (in the case of the largest analysed matrix – *t2em*) and in setting OpenCL for the other two smaller

matrices (*rajat27* and *hvdc1*). Therefore, for this specific application, it can be said that the use of GPU did not accelerate the software execution.

It can be observed though, that the time necessary to compile kernel is very small and the time needed in the "Set OpenCL" step becomes less significant as the matrix increases in size.

Another issue in using GPU to solve large linear system, such as those arising from MWI, would be the available memory. Global memory for this GPU is around 16 Gb. The amount of data resulting from the factorization of MWI 1 would be around 6 Gb (over 300 million non-zero double precision and complex elements for each triangular matrix). For bigger matrices, it could be necessary to think of strategies to split the solution of the linear system, which would also increase the time portion that represents the transfer of data to the GPU.

7 Conclusion

An extensive list of open-source solvers were tested to solve sparse linear systems arising from MWI problems. Matrix MWI 1 was successfully solved using various methods, however, this was not the focus of this thesis, as it can be easily solved using *MATLAB* in a few minutes. To be solved using LU factorization and MUMPS, Matrix MWI 2 required more RAM memory than regular computers have. With UMFPACK, however, it was not possible to factorize it in the server, which has a lot of computational resources.

The type of factorization that is the most appropriate to solve MWI Matrix 2 is likely LDL^T, which takes advantage of the symmetry of the matrix, and was employed by MUMPS and *Pardiso* (solver previously used by the Research Group). The main indication for the research group to continue solving this problem is, therefore, to use MUMPS, which has an interface with MATLAB. Another possibility would be to use the *PasTiX* solver, that was not tested but also performs LDL^T factorization. According to the solver's documentation, however, it also does not support exporting the results of a factorization [26], which would prevent other students to work on the parallelization of the solution of sparse triangular linear systems with MWI data.

The iterative methods tested presented a considerably worse precision in the solution for matrix MWI 1 considering the same run time (limited by the number of iterations). The obtained results were not satisfactory, as shown in Table 6.

OpenCL was successfully used to create kernels that executed the *SpTRSV* kernel using parallelism, including the explicit management of the GPU memory, although with limitations. The use of local memory was explored in two of the prototypes and the results indicated improvements.

It could also be observed that row-compressed format was more efficient to solve the *SpTRSV* kernel than the column-compressed format for a matrix roughly of the same size.

It was also observed that the overhead of moving data to and from the GPU is considerably large. The time to set *OpenCL*, compile kernels, buffer and unbuffer data may be greater than time to run the same process sequentially in a CPU. This aspect was not analysed by the bibliography covered for this work and is relevant when applying a GPU algorithm to a real-life scenario. Additionally, MWI 1 factorization data is already almost as large as the global memory of the GPU used, which means that it would be necessary to find strategies to buffer data arising from the factorization of bigger matrices in this GPU.

Other strategies that could be tried to accelerate the solution of the studied linear systems: using tools to explore parallelism in CPUs (such as OpenMP), installing versions of solvers that explore parallelism according to the hardware available.

Another point to consider is that, since the placement of non-zeros elements affects greatly how fast the factorization of a matrix can be executed, and not necessarily the number of non-zeros, other formulation of the FEM problem could be tested. Whenever possible, it is preferable to reduce the size of the problems to allow a faster solution of the linear system and enable the goal of a fast diagnosis using MWI.

8 Bibliography

[1] M. R. Casu, M. Vacca, J. A. Tobon, A. Pulimeno, I. Sarwar, R. Solimene and F. Vipiana, "A COTS-Based Microwave Imaging System for Breast-Cancer Detection," *IEEE Transactions on Biomedical Circuits and Systems,* vol. 11, no. 4, pp. 804-814, 2017.

[2] A. Fhager, S. Candefjord, M. Elam and M. Persson, "Microwave Diagnostics Ahead: Saving Time and the Lives of Trauma and Stroke Patients," *IEEE Microwave Magazine,* vol. 19, no. 3, pp. 78-90, 2018.

[3] J. A. Vasquez, R. Scapaticci, G. Turvani, G. Bellizzi, D. O. Rodriguez-Duarte, N. Joachimowicz, B. Duchêne, E. Tedeschi, M. R. Casu, L. Crocco and F. Vipiana, "A prototype microwave system for 3d brain stroke imaging," *Sensors,* vol. 20, no. 9, 5 2020.

[4] V. Mariano, J. A. Tobon Vasquez and F. Vipiana, "Discretization Error Analysis in the Contrast Source Inversion Algorithm".

[5] F. Favaro, E. Dufrechou, P. Ezzatti and J. P. Oliver, "Exploring FPGA optimizations to compute sparse Numerical Linear Algebra kernels".

[6] Xilinx, *Fundamentals of FPGA-based Acceleration,* Frankfurt: Xilinx Developer Forum, 2018.

[7] Z. Lu, Y. Niu and W. Liu, "Efficient Block Algorithms for Parallel Sparse Triangular Solve," *ACM International Conference Proceeding Series,* 8 2020.

[8] W. Liu, A. Li, J. Hogg, I. Duff and B. Vinter, "Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides," *Concurrency and Computation: Practice and Experience,* vol. 29, 2017.

[9] A. Zakaria, "The Finite-Element Contrast Source Inversion Method for Microwave Imaging Applications".

[10] M. Bollhöfer, A. Eftekhari, S. Scheidegger and O. Schenk, "Large-scale Sparse Inverse Covariance Matrix Estimation," *SIAM Journal on Scientific Computing,* vol. 41, no. 1, pp. A380-A401, 2019.

[11] M. Bollhöfer, O. Schenk, R. Janalik, S. Hamm and K. Gullapalli, State-of-the-Art Sparse Direct

Solvers, 2020.

[12] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies and G. Wellein, “A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication,” *ACM Trans. Parallel Comput.,* vol. 7, no. 3, 2020.

[13] Microsoft, “What is the Windows Subsystem for Linux?,” 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows/wsl/about. [Accessed 17 08 2022].

[14] T. A. Davis and Y. Hu, “The University of Florida Sparse Matrix Collection,” *ACM Trans. Math. Softw.,* vol. 38, 2011.

[15] T. A. Davis, “Algorithm 832: UMFPACK V4.3---an Unsymmetric-Pattern Multifrontal Method,” *ACM Trans. Math. Softw.,* vol. 30, no. 2, p. 196–199, 2004.

[16] P. R. Amestoy, T. A. Davis and I. S. Duff, “An Approximate Minimum Degree Ordering Algorithm,” *SIAM Journal on Matrix Analysis and Applications,* vol. 17, no. 4, pp. 886-905, 1996.

[17] Mumps Technologies, *MUltifrontal Massively Parallel Solver (MUMPS 5.4.1) Users’ guide,* 2021.

[18] T. A. Davis and E. Palamadai Natarajan, “Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems,” *ACM Trans. Math. Softw.,* vol. 37, no. 3, 2010.

[19] Scipy, “Sparse linear algebra (scipy.sparse.linalg),” [Online]. Available: https://docs.scipy.org/doc/scipy/reference/sparse.linalg.html. [Accessed 17 08 2022].

[20] Eigen, “Solving Sparse Linear Systems,” [Online]. Available: https://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html. [Accessed 17 08 2022].

[21] O. Schenk and K. Gärtner, “PRADISO - User Guide Version 7.2,” 28 12 2020. [Online]. Available: https://pardiso-project.org/manual/manual.pdf. [Accessed 18 08 2022].

[22] The Numerical Algorithms Group Ltd. 2022, “Large Scale Linear Systems,” [Online]. Available: https://www.nag.com/numeric/nl/nagdoc_28.3/flhtml/f11/f11intro.html. [Accessed 17 08 2022].

[23] R. Li, “On Parallel Solution of Sparse Triangular Linear Systems in CUDA,” *arXiv preprint arXiv:1710.04985,* 10 2017.

[24] National Energy Research Scientific Computing Center, “OpenCL: A Hands-on Introduction,” [Online]. Available: https://www.nersc.gov/assets/pubs_presos/MattsonTutorialSC14.pdf. [Accessed 17 08 2022].

[25] Khronos, “The OpenCL Specification,” 14 11 2012. [Online]. Available: https://registry.khronos.org/OpenCL/specs/opencl-1.2.pdf. [Accessed 17 08 2022].

[26] PaStiX Handbook, “PaStiX: A sparse direct solver,” [Online]. Available: https://solverstack.gitlabpages.inria.fr/pastix/. [Accessed 17 08 2022].