

## GPU Acceleration of a Domain Decomposition Solver

**Auteur :** Geleleens, Emil

**Promoteur(s) :** Geuzaine, Christophe

**Faculté :** Faculté des Sciences appliquées

**Diplôme :** Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"

**Année académique :** 2023-2024

**URI/URL :** <https://gitlab.uliege.be/Emil.Geleleens/master-thesis>; <http://hdl.handle.net/2268.2/20960>

---

### Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

---



# GPU Acceleration of a Domain Decomposition Solver

A Thesis Submitted for the Degree of  
Master of Science in Computer Science and Engineering

*in the Faculty of Applied Sciences  
at the University of Liège*

*Author:*

Emil GELELEENS

*Supervisor:*

Christophe GEUZAINÉ

Academic Year 2023-2024

# Contents

<b>List of Figures</b>	<b>iii</b>
<b>List of Tables</b>	<b>vi</b>
<b>List of Algorithms</b>	<b>vii</b>
<b>Listings</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 General Context and Objectives . . . . .	1
1.2 Overview of Linear Solvers . . . . .	2
1.2.1 LU Decomposition . . . . .	3
1.2.2 Krylov Subspace Solvers . . . . .	4
1.2.3 Domain Decomposition Methods . . . . .	8
<b>2 GPU-Accelerated Sparse Triangular Solver</b>	<b>11</b>
2.1 Preprocessing Step . . . . .	12
2.1.1 Level-Set . . . . .	13
2.1.2 Compressed Level-Set . . . . .	16
2.1.3 Balanced Level-Set . . . . .	18
2.2 Solution Step . . . . .	19
2.2.1 CUDA Kernels . . . . .	20
2.2.2 Optimized Reduction . . . . .	21
2.2.3 Matrix Storage Format . . . . .	24
2.2.4 Task Distribution . . . . .	24
2.2.5 Smart Parameters . . . . .	26
2.2.6 Sequential Methods . . . . .	26
2.2.7 Scheduled Methods . . . . .	31
2.3 Solution Step with Multiple Right-Hand Sides . . . . .	37
2.3.1 Naive Method . . . . .	37
2.3.2 Block Methods . . . . .	38
2.3.3 Flipped Methods . . . . .	45
<b>3 Numerical Experiments</b>	<b>49</b>

3.1	Hardware Setup . . . . .	49
3.2	Direct Method . . . . .	50
3.2.1	Toy Problem . . . . .	50
3.2.2	Comparison of Preprocessing Strategies . . . . .	51
3.2.3	Comparison of Solution Strategies . . . . .	55
3.2.4	Comparison of Multi-RHS Solution Strategies . . . . .	61
3.3	ORAS . . . . .	66
<b>4</b>	<b>Conclusion</b>	<b>70</b>
	<b>Bibliography</b>	<b>71</b>



# List of Figures

1.1	Scattering of waves in an aircraft engine . . . . .	2
1.2	Example of a mesh generated by Gmsh [13] and partitioned into non-overlapping subdomains using METIS [19]. . . . .	9
2.1	DAG representing the dependencies of the unknowns of the system defined by the matrix $L$ in Equation 2.10. . . . .	15
2.2	Scheduler associated with the matrix $L$ defined in Equation 2.10 built with the <b>level-set</b> method. . . . .	15
2.3	Scheduler associated with the matrix $L$ defined in Equation 2.10. . . . .	15
2.4	Scheduler associated with the matrix $L$ defined in Equation 2.10 built with the <b>compressed-level-set</b> method. An arrow between two unknowns means that these two unknowns share the same node. . . . .	18
3.1	Preprocessing time for the $L$ and $U$ matrices for each strategy as a function of the number of DOFs in the system to solve. . . . .	51
3.2	Solve time as a function of the number of DOFs for each scheduling strategy. The solution step is performed using the <b>default</b> method. The scheduler-free method uses the <b>sequential-singleblock</b> method for solving the problem. . . . .	52
3.3	Forward substitution and backward substitution time as a function of the number of DOFs for each scheduling strategy. The solution step is performed using the <b>default</b> method. The scheduler-free method uses the <b>sequential-singleblock</b> method for solving the problem. . . . .	53
3.4	Bar plot representing the number of rows having a given number of nonzero elements for the $L$ and $U$ matrices corresponding to the problem with the most coarse mesh ( <b>-clscale</b> = 1, i.e., the smallest system $1156 \times 1156$ ). . . . .	54
3.5	GPU solve performance as a function of the number of DOFs with several strategies, using the <b>balanced-level-set</b> method for constructing the scheduler when one is needed. The cuSPARSE strategy uses the <b>cusparseSpSV_solve</b> routine of the cuSPARSE library to perform the forward and backward substitution. . . . .	55
3.6	GPU solve time as a function of the number of DOFs with several strategies, using the <b>balanced-level-set</b> method for constructing the scheduler when one is needed. . . . .	56

3.7	GPU forward substitution time as a function of the number of DOFs with several strategies, using the <b>balanced-level-set</b> method for constructing the scheduler when one is needed. . . . .	56
3.8	GPU backward substitution time as a function of the number of DOFs with several strategies, using the <b>balanced-level-set</b> method for constructing the scheduler when one is needed. . . . .	57
3.9	CPU solve performance as a function of the number of DOFs. We compare several computation methods. The plots labeled “1 CPU” and “MKL SPARSE” correspond to single-core methods. The “1 CPU” method employs a simple implementation of the forward and backward substitution algorithm, while the “MKL SPARSE” method solves the system using the <code>mk1_sparse_z_trsv</code> routine from the Intel oneAPI Math Kernel Library [17]. For the multi-core methods, computations are parallelized using OpenMP similarly to the <b>default</b> method on GPU, utilizing a scheduler built with the <b>balanced-level-set</b> strategy. . . . .	58
3.10	CPU solve time as a function of the number of DOFs. We compare several computation methods. The plots labeled “1 CPU” and “MKL SPARSE” correspond to single-core methods. The “1 CPU” method employs a simple implementation of the forward and backward substitution algorithm, while the “MKL SPARSE” method solves the system using the <code>mk1_sparse_z_trsv</code> routine from the Intel oneAPI Math Kernel Library [17]. For the multi-core methods, computations are parallelized using OpenMP similarly to the <b>default</b> method on GPU, utilizing a scheduler built with the <b>balanced-level-set</b> strategy. . . . .	59
3.11	Speedup of the GPU solve using the <b>adaptative</b> strategy compared to the fastest CPU solve. . . . .	59
3.12	. . . . .	60
3.13	Solve performance as a function of the number of right-hand sides stored in column-major order for different system sizes and solution strategies. The cuSPARSE strategy uses the <code>cusparseSpSM_solve</code> routine to solve the two triangular systems with multiple right-hand sides. . . . .	62
3.14	Solve performance as a function of the number of right-hand sides stored in row-major order for different system sizes and solution strategies. The cuSPARSE strategy uses the <code>cusparseSpSM_solve</code> routine to solve the two triangular systems with multiple right-hand sides. . . . .	63
3.15	Colormap illustrating the optimal solution strategy for several system sizes and number of right-hand sides. Note that when there is only one right-hand side, this figure only shows the best multi-right-hand side strategy. However, in this case the best strategy would be the <b>adaptative</b> strategy. . . . .	64
3.16	Heatmap illustrating the performance of the GPU solve when the best multi-right-hand-side strategy is used. . . . .	64
3.17	Heatmap representing the speedup of the best GPU multi-RHS solution strategy compared to the best CPU multi-RHS solution strategy. . . . .	65

3.18 ORAS tests with varying characteristic length (CL) of the mesh elements.	67
3.19 ORAS tests with varying finite element order. . . . .	67

# List of Tables

2.1	List of intrinsic functions and variables used in the CUDA kernels. . . . .	21
2.2	Example of task distribution using the distribution function defined in Equation 2.14 when the number of tasks is $N = 23$ and the number of threads is $T = 7$ . . . . .	25
2.3	Example of task distribution using the distribution function defined in Equation 2.17 when the number of tasks is $N = 23$ and the number of threads is $T = 7$ . . . . .	26
3.1	Node details of the GPU partition of the Lucia cluster. [16] . . . . .	49
3.2	Node details of the CPU ( <code>batch</code> ) partition of the Lucia cluster. [16] . . . .	50
3.3	ORAS test configuration . . . . .	66
3.4	Results of the ORAS experiment on GPU. The first column contains the order of the finite elements. The second is the characteristic length of one mesh element. The last column contains the average time of each GMRES iteration. The third column contains the average number of DOFs per subdomain. If one multiplies this number by the number of subdomains (16), then the result is larger than the total number of DOFs in the system, because the subdomains are overlapping. . . . .	68
3.5	Results of the ORAS experiment on CPU. The first column contains the order of the finite elements. The second is the characteristic length of one mesh element. The last column contains the average time of each GMRES iteration. The third column contains the average number of DOFs per subdomain. If one multiplies this number by the number of subdomains (16), then the result is larger than the total number of DOFs in the system, because the subdomains are overlapping. . . . .	69

# List of Algorithms

1	Basic <i>LU</i> Decomposition Algorithm . . . . .	4
2	GCR Method . . . . .	7
3	Preprocessing Step: <b>level-set</b> . . . . .	14
4	Preprocessing Step: <b>compressed-level-set</b> . . . . .	17
5	Preprocessing Step: <b>balanced-level-set</b> . . . . .	19
6	Solution Step: <b>sequential-singleblock</b> . . . . .	29
7	Solution Step: <b>sequential-multiblock</b> . . . . .	31
8	Solution Step: <b>default</b> . . . . .	33
9	Solution Step CPU: <b>adaptative</b> . . . . .	35
10	Solution Step Kernel: <b>adaptative</b> . . . . .	36
11	Solution Step: <b>naive</b> . . . . .	38
12	Solution Step: <b>block-outer</b> . . . . .	41
13	Solution Step: <b>block-middle</b> . . . . .	42
14	Solution Step CPU: <b>block-inner</b> . . . . .	43
15	Solution Step Kernel: <b>block-inner</b> . . . . .	44
16	Solution Step: <b>flipped</b> . . . . .	46
17	Solution Step CPU: <b>flipped-enhanced</b> . . . . .	47
18	Solution Step Kernel: <b>flipped-enhanced</b> . . . . .	48

# Listings

2.1	Optimized reduction . . . . .	21
3.1	Gmsh script for generating a mesh of a unit cubic domain with specified mesh size and physical entities. . . . .	50

# Acknowledgments

I want to sincerely thank Professor Geuzaine and Boris Martin for their guidance and support throughout this year. I am very grateful to have had the opportunity to work on such an engaging topic.

The present research benefited from computational resources made available on Lucia, the Tier-1 supercomputer of the Walloon Region, infrastructure funded by the Walloon Region under the grant agreement n°1910247.

# Chapter 1

## Introduction

### 1.1 General Context and Objectives

Time-harmonic wave problems arise in many fields of physics and engineering such as acoustics, electromagnetics or mechanics. Whether it is predicting the noise of an aircraft engine (see Figure 1.1), the propagation of seismic waves, or the electromagnetic/optical behavior of future space telescopes, these problems are ubiquitous. Sadly, these problems are known to be computationally demanding to solve, especially in the high-frequency regime [27]. One approach that can be used consists in translating the problem into a linear system using the Finite Element Method (FEM) with absorbing boundary conditions and then solving the system using a linear solver. This method usually leads to large complex-valued linear systems which can be indefinite [21]. Direct solvers do not scale well for such systems, and Krylov subspace iterative solvers (such as GMRES [24]) converge too slowly or not at all. Fortunately, the preconditioners come to our rescue. Preconditioners allow to accelerate the convergence of Krylov solvers by modifying the system into a form that is more favorable. A preconditioner is a matrix  $M^{-1}$  which is applied to both sides of the system such that the newly generated system has the same solution as the original one:

$$M^{-1}Ax = M^{-1}b. \tag{1.1}$$

Constructing an effective preconditioner is a challenging task. However, domain decomposition methods (DDM) enable the efficient construction of robust preconditioners for problems of interest in massively parallel environments.

The aim of this thesis is to accelerate the solution of time-harmonic wave problems using Krylov iterative solvers with domain decomposition preconditioners on GPUs. Specifically, we are interested in accelerating the Generalized Minimal Residual (GMRES) Krylov method, which is preconditioned with the Optimized Restricted Additive Schwarz (ORAS) preconditioner. To achieve this, we will use the GmshFEM library [23], which is a FEM library capable of solving systems directly or iteratively. In any case, the computations are currently performed by the CPU. GmshFEM relies on



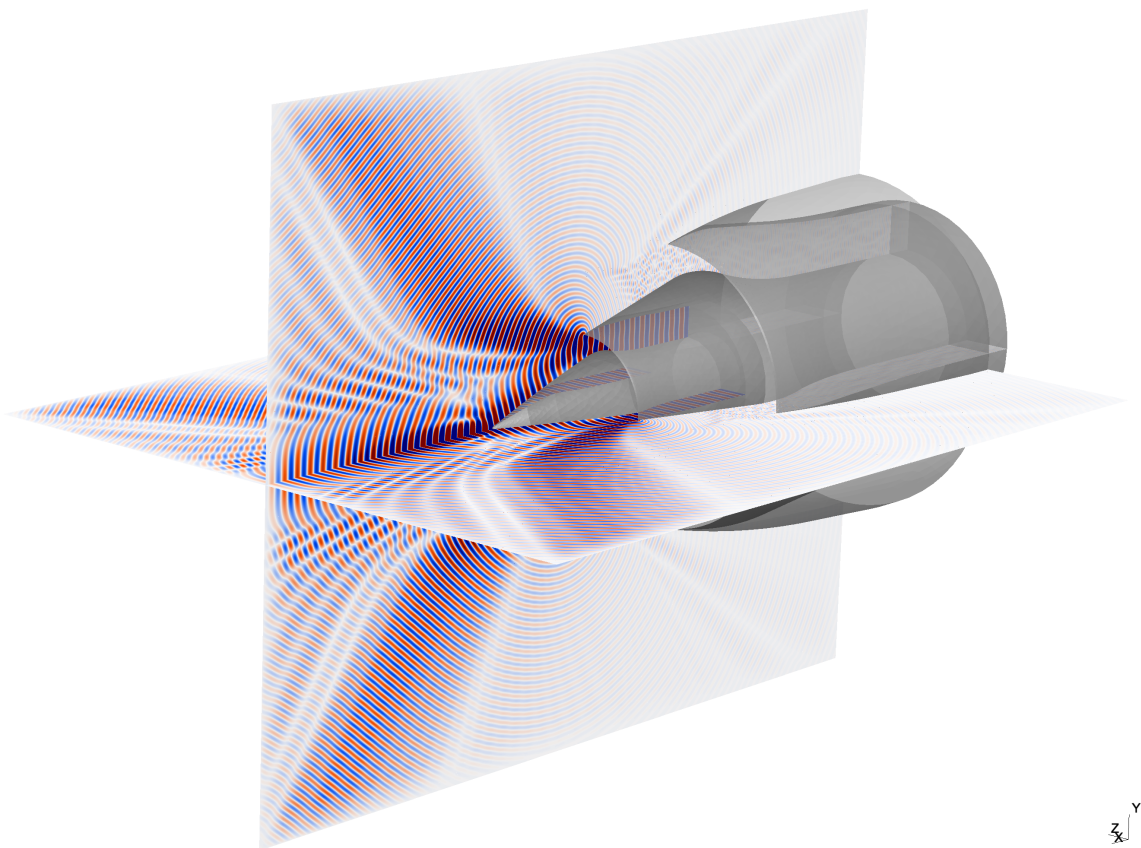


Figure 1.1: Scattering of waves in an aircraft engine

PETSc [5, 4, 6] for solving the assembled systems with Krylov subspace solvers. Luckily, some of PETSc's code has been ported to the GPU [20]. For example, iterative solvers are available on the GPU. However, there are still missing features. For instance, each iteration of the GMRES algorithm requires applying the preconditioner  $M^{-1}$ , which, as we will discuss, in the case of Schwarz preconditioners necessitates solving multiple smaller systems. These smaller systems are well-suited for direct solvers such as MUMPS [2] or UMFPACK [9], which are entirely CPU-based. Therefore, there is potential for improvement by porting this part of the computation to the GPU as well.

## 1.2 Overview of Linear Solvers

In this section, we review multiple methods of interest for solving linear systems.

### 1.2.1 LU Decomposition

The idea behind  $LU$  decomposition is to factorize the matrix  $A$  into two triangular matrices  $L$  and  $U$ .  $L$  is lower triangular and  $U$  is upper triangular. One can then easily compute  $x$  by solving two triangular systems:

$$\begin{cases} Ly = b, \\ Ux = y. \end{cases} \quad (1.2)$$

The first system can be solved with the forward substitution algorithm:

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij}y_j}{L_{ii}}, \quad (1.3)$$

and the second one with the backward substitution algorithm:

$$x_i = \frac{y_i - \sum_{j=i+1}^n U_{ij}x_j}{U_{ii}}. \quad (1.4)$$

Having an  $LU$  decomposition of a matrix  $A$  is advantageous when solving systems multiple times with various right-hand sides, as the  $L$  and  $U$  factors can be reused. An alternative approach is to first compute and store  $A^{-1}$ . One would then have to compute the matrix-vector multiplication  $x = A^{-1}b$ . This requires  $n^2 - n$  additions and  $n^2$  multiplications. In contrast, forward and backward substitution combined require  $n^2 - 3n$  additions,  $2n$  subtractions,  $n^2 - n$  multiplications and  $2n$  divisions. Although matrix-vector multiplication might initially seem more efficient, this comparison does not consider that  $A$  is sparse. In general, the inverse of a sparse matrix is not sparse, but it is often possible to obtain sparse  $L$  and  $U$  factors. With sparse factors, the number of floating-point operations decreases, as multiplications involving zero elements can be ignored. Additionally, computing the inverse is generally more computationally expensive and less numerically stable. Therefore, it is generally preferable to solve a system using  $LU$  decomposition rather than using the inverse  $A^{-1}$ .

The  $LU$  decomposition itself can be performed using Gaussian elimination. Algorithm 1 shows how the  $L$  and  $U$  factors can be computed. The problem with this algorithm is that it would perform a division by zero if some diagonal elements of  $A$  are zero. This does however not mean that matrices with zero diagonal elements are singular. The way to resolve this issue is to permute certain rows such that all the diagonal elements are nonzero, and then factorize this new matrix:

$$PA = LU, \quad (1.5)$$

where  $P$  is a permutation matrix. If this is not possible, it indicates that a column contains only zeros, which implies that the matrix is singular. If the matrix is invertible, there are usually multiple permutation matrices that could work. It is therefore important to choose the best one. While there are many ways to reorder the rows in

---

**Algorithm 1:** Basic  $LU$  Decomposition Algorithm

---

**Input** :  $A \in \mathbb{C}^{n \times n}$   
**Output:**  $L \in \mathbb{C}^{n \times n}$   
 $U \in \mathbb{C}^{n \times n}$

```

1  $L \leftarrow \mathbb{I}_n$ 
2  $U \leftarrow A$ 
3 for  $k \leftarrow 1$  to  $n - 1$  do
4   for  $i \leftarrow k + 1$  to  $n$  do
5      $L_{ik} \leftarrow \frac{U_{ik}}{U_{kk}}$ 
6     for  $j \leftarrow k$  to  $n$  do
7        $U_{ij} \leftarrow U_{ij} - L_{ik}U_{kj}$ 
8     end
9   end
10 end

```

---

$A$ , the objective is usually to reduce fill-in, that is, minimize the number of nonzero elements in the factors. For example, METIS [19] can be used to produce fill reducing orderings for sparse matrices.

Linear direct sparse solvers, such as MUMPS [2], use more advanced multifrontal methods to perform the factorization. In the multifrontal method, all elimination operations take place within dense submatrices, called frontal matrices [1]. The concept of a frontal matrix was first introduced by [18]. The multifrontal method allows for more than one front to occur at the same time [11], enabling the parallelization of the factorization process.

Currently, all major sparse  $LU$  factorization libraries are CPU-based, and porting them to the GPU would be a challenging task that is beyond the scope of this thesis.

Unfortunately, direct solvers do not scale well for large-scale systems, which is why alternative methods are preferred for solving such problems. Although direct solvers may not be suitable for solving large-scale problems, they are often used as subsolvers within a larger iterative method.

### 1.2.2 Krylov Subspace Solvers

In this section, we review the concepts underlying Krylov subspace methods.

Consider the well-posed linear system  $Ax = b$ . For simplicity, assume that the system is real-valued; however, the methods can be extended to the complex case. A Krylov iterative solver starts from an initial guess  $x^0$  and improves the guess at each iteration until the approximation  $x^k$  is close enough to the actual solution. To determine whether, at iteration  $k$ , the estimate  $x^k$  is good enough, one computes the norm of the residual  $r^k = b - Ax^k$  and if this norm is small enough, the algorithm stops.

Let  $M$  be an easily invertible matrix of the same size as  $A$ . By easily invertible, we

mean that computing  $M^{-1}r$ , with  $r = b - Ax$ , is cheap [10]. The fixed point method for solving this system is defined as follows:

$$x^{k+1} = x^k + M^{-1}r^k, \quad (1.6)$$

where  $x^k$  is the approximation of the solution at iteration  $k$  and  $r^k = b - Ax^k$  is the residual. When this algorithm converges,  $x^k$  converges to the solution of the system

$$M^{-1}Ax = M^{-1}b. \quad (1.7)$$

Note that the solution of this system is the same as the solution of the original system. In this case,  $M^{-1}$  is called the preconditioner. For more details on the convergence of the fixed point method see [14], but in general it is hard to find a preconditioner which makes the method convergent. This drawback motivates the use of Krylov methods which are much more robust.

For a given matrix  $B$  and vector  $y$ , the Krylov subspace of dimension  $k$  associated with  $B$  and  $y$  is defined as

$$\mathcal{K}^k(B, y) := \text{span}\{y, By, B^2y, \dots, B^{k-1}y\}. \quad (1.8)$$

It can be shown that the solution of a fixed point method belongs to a Krylov subspace [10]. Therefore, a way to iteratively compute a solution would be to look for an optimal element in the above mentioned space. This is the idea of Krylov subspace solvers.

Krylov subspace solvers approximate the solution of a linear system  $Ax = b$  by iteratively refining an initial guess  $x^0$ . At iteration  $k$ , the approximated solution  $x^k$  is such that  $x^k \in x^0 + \mathcal{K}^k(A, r^0)$ , where  $r^0 = b - Ax^0$  is the initial residual. At iteration  $k$ , the approximation is

$$x^k = x^0 + \sum_{i=1}^k \alpha_i p^i, \quad (1.9)$$

where  $p^1, p^2, \dots, p^k$  is a basis of the Krylov subspace  $\mathcal{K}^k(A, r^0)$ .

### Generalized Conjugate Residual Method

For example, in the Generalized Conjugate Residual (GCR) method [12], the basis is chosen such that its vectors are  $A^T A$ -orthogonal [25], i.e., such that

$$(Ap^i)^T Ap^j = 0, \quad \text{for } i \neq j. \quad (1.10)$$

The best approximate solution  $x^k \in x^0 + \mathcal{K}^k(A, r^0)$  is then given by

$$x^k = x^0 + \sum_{i=1}^k \alpha_i p^i, \quad (1.11)$$

where  $\alpha_i$  is a coefficient computed so that the 2-norm of the residual is minimized. We can also define  $x^k$  with the following recurrence relation:

$$x^k = x^{k-1} + \alpha_k p^k. \quad (1.12)$$

The residual can be expressed as follows:

$$\begin{aligned} r^k &= b - A \left( x^0 + \sum_{i=1}^k \alpha_i p^i \right) \\ &= r^0 - \sum_{i=1}^k \alpha_i A p^i. \end{aligned} \quad (1.13)$$

We can minimize the 2-norm of the residual by solving the following least square problem:

$$\min_{\alpha_j} \left\| r^0 - \sum_{i=1}^k \alpha_i A p^i \right\|_2^2. \quad (1.14)$$

This yields the following equations for  $j = 1, \dots, k$ :

$$\frac{\partial}{\partial \alpha_j} \left\| r^0 - \sum_{i=1}^k \alpha_i A p^i \right\|_2^2 = -2(r^0)^T A p^j + 2 \underbrace{\sum_{i \neq j} \alpha_i (A p^j)^T A p^i}_{=0 \text{ because of 1.10}} + 2\alpha_j \|A p^j\|_2^2 = 0. \quad (1.15)$$

By solving these equations, we get

$$\alpha_j = \frac{(r^0)^T A p^j}{(A p^j)^T A p^j}. \quad (1.16)$$

From 1.13 and 1.10, we find the following:

$$(r^{j-1})^T A p^j = (r^0)^T A p^j - \sum_{i=1}^{j-1} \alpha_i (A p^i)^T A p^j = (r^0)^T A p^j. \quad (1.17)$$

Equation 1.16 can now be rewritten as follows:

$$\alpha_j = \frac{(r^{j-1})^T A p^j}{(A p^j)^T A p^j}. \quad (1.18)$$

This modified expression for  $\alpha_j$  is usually preferred because it only requires the previous residual  $r^{j-1}$ , while in 1.16, one needs to store the initial residual  $r^0$  as well, which increases memory usage. Additionally, this second representation of  $\alpha_j$  is more numerically stable.

It is now necessary to determine how to choose the basis of the Krylov subspace. The simplest option computes the next basis vector  $p^{k+1} \in \mathcal{K}^{k+1}(A, r^0)$  as a linear combination of the current residual  $r^k$  and all previous basis vectors. Note that this can work because  $r^k \in \mathcal{K}^{k+1}(A, r^0)$ . We obtain an expression of the form

$$p^{k+1} = r^k + \sum_{i=1}^k \beta_{ik} p^i, \quad (1.19)$$

where  $\beta_{ik}$  are coefficients that need to be determined. Since the basis vectors need to be  $A^T A$ -orthogonal, we have, for  $j \leq k$ ,

$$\begin{aligned} (Ap^j)^T Ap^{k+1} &= 0, \\ \iff (Ap^j)^T \left( Ar^k + \sum_{i=1}^k \beta_{ik} Ap^i \right) &= 0, \\ \iff \beta_{jk} &= -\frac{(Ap^j)^T Ar^k}{(Ap^j)^T Ap^j}. \end{aligned} \tag{1.20}$$

Algorithm 2 illustrates how to implement the GCR algorithm. One drawback of this method is that, at iteration  $k$ , all the  $p^i$  vectors with  $i \leq k$  need to be stored, resulting in increased memory usage at each iteration. One way to circumvent this issue is to implement a restarted version of the algorithm. Instead of orthogonalizing against all previous basis vectors, this version orthogonalizes only against the last  $m$  vectors, where  $m$  is a fixed parameter. This approach reduces both the cost of each iteration and memory usage, but it can slow down the convergence of the algorithm. There is a trade-off between using a small value for  $m$ , which leads to faster iterations but slower convergence, and a large  $m$ , which results in slower iterations but improved convergence. In libraries such as PETSc [5, 4, 6], the default value is  $m = 30$ , though it can be adjusted as needed.

---

**Algorithm 2:** GCR Method

---

**Input** :  $A \in \mathbb{R}^{n \times n}$   
 $b \in \mathbb{R}^n$   
 $x^0 \in \mathbb{R}^n$

- 1  $r^0 \leftarrow b - Ax^0$
- 2  $p^1 \leftarrow r^0$
- 3 **for**  $k \leftarrow 1, 2, 3, \dots$ , *until convergence* **do**
- 4      $\alpha_k \leftarrow \frac{(r^{k-1})^T Ap^k}{(Ap^k)^T Ap^k}$
- 5      $x^k = x^{k-1} + \alpha_k p^k$
- 6      $r^k \leftarrow r^{k-1} - \alpha_k Ap^k$
- 7     **for**  $i \leftarrow 1$  **to**  $k$  **do**
- 8          $\beta_{ik} \leftarrow -\frac{(Ap^i)^T Ar^k}{(Ap^i)^T Ap^i}$
- 9     **end**
- 10     $p^{k+1} \leftarrow r^k + \sum_{i=1}^k \beta_{ik} p^i$
- 11 **end**

---

**Generalized Minimal Residual Method**

The Generalized Minimal Residual (GMRES) method is another Krylov subspace iterative method for solving linear systems which was first introduced by [24]. In this

method, instead of building a basis  $p^1, \dots, p^k$  of the Krylov subspace  $\mathcal{K}^k(A, r^0)$  which is  $A^T A$ -orthogonal, we build a basis which is orthonormal [10]. This is done by using a Gram-Schmidt algorithm. That is,

$$p^1 = \frac{r^0}{\|r^0\|_2} \text{ and } p^{k+1} = \frac{Ap^k - \sum_{i=1}^k ((Ap^k)^T p^i) p^i}{\left\| Ap^k - \sum_{i=1}^k ((Ap^k)^T p^i) p^i \right\|_2}. \quad (1.21)$$

The GMRES algorithm also suffers from increasing memory usage as the number of iterations grows. To manage this issue, as for the GCR method, it can be implemented with a restart strategy, where the algorithm is restarted after  $m$  iterations. Compared to GCR, GMRES requires less memory and fewer arithmetic operations, making it the better choice in most cases.

There exists many variants of the GMRES algorithm, one of which being Block-GMRES which is particularly useful when solving a system  $AX = B$ , where  $X$  and  $B$  are matrices. This kind of system arises when one wants to solve a problem with multiple sources.

### 1.2.3 Domain Decomposition Methods

In this section, we describe the Restricted Additive Schwarz method (RAS) which is a fixed point iteration for solving linear systems.

We consider a domain  $\Omega$  which is partitioned into overlapping subdomains  $\Omega_i$ , with  $i = 1, 2, \dots, n_d$  [10]. Let  $\mathcal{N}$  be the set of indices of degrees of freedom and  $n = |\mathcal{N}|$ . We consider the decomposition of  $\mathcal{N}$  in  $n_d$  overlapping subsets  $\mathcal{N}_i$ , with  $i = 1, 2, \dots, n_d$  ( $\mathcal{N} = \bigcup_i \mathcal{N}_i$ ) such that  $\mathcal{N}_i$  contains the indices of the degrees of freedom in the subdomain  $\Omega_i$ . This set of indices can be partitioned using a graph partitioner such as METIS [19] (see Figure 1.2) or SCOTCH [7]. In practice, the domain is often represented by a mesh, in which case the mesh itself is partitioned rather than the set of indices.

Let  $R_i$  be the restriction of a vector  $x \in \mathbb{C}^n$  to a subdomain  $\Omega_i$ , with  $i = 1, 2, \dots, n_d$ . This operator is represented by a rectangular  $|\mathcal{N}_i| \times n$  boolean matrix such that

$$\begin{cases} (R_i)_{kj} = 1, & \text{if } j \in \mathcal{N}_i \wedge (R_i)_{pj} = 0 \ \forall p \in \{1, \dots, |\mathcal{N}_i|\} \text{ with } p \neq k, \\ (R_i)_{kj} = 0, & \text{otherwise.} \end{cases} \quad (1.22)$$

The extension operator, which projects a local vector into the full space, is the transpose matrix  $R_i^T$ . Let  $D_i$  be the partition of unity operator represented by a square diagonal matrix of degree  $|\mathcal{N}_i|$ . This operator is constructed to ensure that

$$\mathbb{I}_n = \sum_{i=1}^{n_d} R_i^T D_i R_i, \quad (1.23)$$

where  $\mathbb{I}_n \in \{0, 1\}^{n \times n}$  is the identity matrix. Note that there exists another Schwarz method called additive Schwarz method (ASM) which is the same except that it does not use the partition of unity operator, which is equivalent to considering that  $D_i = \mathbb{I}_{|\mathcal{N}_i|}$ .

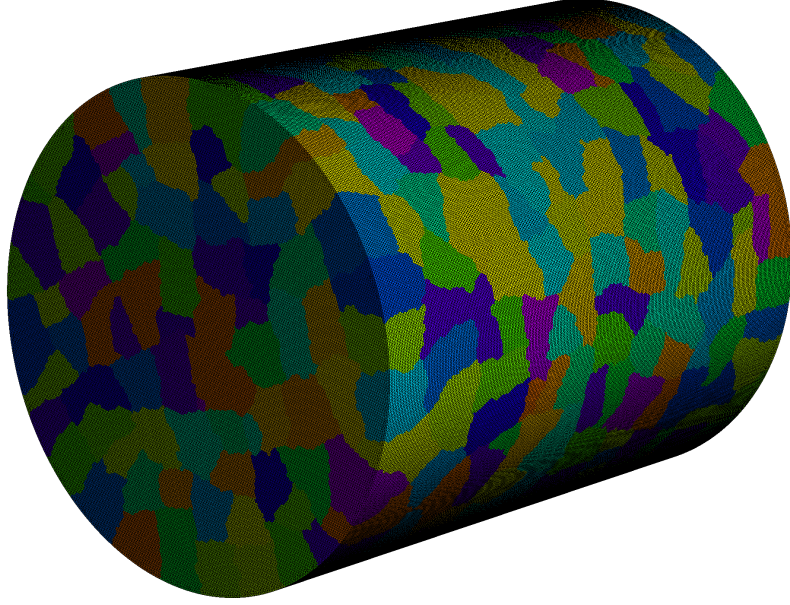


Figure 1.2: Example of a mesh generated by Gmsh [13] and partitioned into non-overlapping subdomains using METIS [19].

Suppose we want to solve  $Ax = b$ . We define the local matrices  $A_{\text{loc},i} = R_i A R_i^T$ , with  $i = 1, 2, \dots, n_d$ . These local matrices are  $|\mathcal{N}_i| \times |\mathcal{N}_i|$  matrices which contain the elements  $A_{kj}$  where  $k \in \mathcal{N}_i$  and  $j \in \mathcal{N}_i$ . Each local  $A_i$  matrix is owned by one process. We define the preconditioner matrix

$$M_{\text{RAS}}^{-1} = \sum_{i=1}^{n_d} R_i^T D_i A_{\text{loc},i}^{-1} R_i, \quad (1.24)$$

which is used as an approximation of  $A^{-1}$  which has the advantage of being computed in parallel by as many processes as there are subdomains. The Schwarz algorithm is the preconditioned fixed point iteration defined by

$$x^{k+1} = x^k + M_{\text{RAS}}^{-1} r^k, \quad (1.25)$$

where  $x^k$  is the vector approximating  $x$  at iteration  $k$  and  $r^k = b - Ax^k$  is the residual.

This method does however not converge with Helmholtz problems. These fixed point iterations are not often used in practice. Instead, the RAS preconditioner is used as a preconditioner for a Krylov subspace method such as GMRES.



To further improve the convergence of GMRES, one can use the Optimized Restricted Additive Schwarz (ORAS) preconditioner  $M_{\text{ORAS}}^{-1}$  instead. The only difference with the RAS method consists in replacing the local Dirichlet matrices  $A_{\text{loc},i} = R_i A R_i^T$  by matrices  $A_{\text{Robin},i}$  which correspond to local Robin subproblems [10]. At iteration  $k$  of GMRES, the preconditioner is applied to the current residual:

$$M_{\text{ORAS}}^{-1} r^k = \sum_{i=1}^{n_d} R_i^T D_i A_{\text{Robin},i}^{-1} r^{k,i}, \quad (1.26)$$

where  $r^{k,i}$  is the restriction of the residual to subdomain  $i$ . This means that each process  $i$  must solve the system

$$A_{\text{Robin},i} x = r^{k,i}. \quad (1.27)$$

As this local system is quite small compared to the global one, a direct solver can be used to factorize the matrix  $A_{\text{Robin},i}$  before starting the GMRES iteration. Then, at each GMRES iteration, one can solve the system by performing one forward and backward substitution. Our goal now is to perform this forward and backward substitution on the GPU by developing a GPU-accelerated sparse triangular solver.

## Chapter 2

# GPU-Accelerated Sparse Triangular Solver

In this chapter, we will see how we can solve a complex sparse triangular system on the GPU efficiently. We will only consider lower triangular systems, but what we will see is also applicable to upper triangular systems where the order of computations is reversed. We consider the lower triangular matrix  $L \in \mathbb{C}^{n \times n}$  such that

$$L = \begin{pmatrix} L_{11} & 0 & 0 & \dots \\ L_{21} & L_{22} & 0 & \dots \\ L_{31} & L_{32} & L_{33} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \quad (2.1)$$

We want to solve the system

$$Lx = b, \quad (2.2)$$

where  $x \in \mathbb{C}^n$  and  $b \in \mathbb{C}^n$ .

The computation that has to be performed to solve the present problem is the following:

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij}x_j}{L_{ii}}, \quad (2.3)$$

where  $i = 1, 2, \dots, n$ . We notice that computing each unknown  $x_i$  must be done in a sequence starting from  $i = 1$  to  $n$ . We can however parallelize the computation of each  $x_i$ . We can do this by assigning to each available thread a portion of the sum  $s_i = \sum_{j=1}^{i-1} -L_{ij}x_j$  that we will call a segment and then, with a reduction operation, one can compute the total sum and the solution is then given by  $x_i = \frac{b_i + s_i}{L_{ii}}$ . This method allows to have some level of parallelism, but it is quite limited as the system we want to solve is sparse, meaning that the number of elements to sum might be quite low and performing a reduction is not free either as it requires  $O(\log N)$  steps, where  $N$  is the number of threads. This strategy is what we will later refer to as the **sequential** strategy as the  $x_i$  are computed sequentially.

We slightly modify the problem to reduce it to a unit diagonal triangular system. First, let us define the variables  $y_i \in \mathbb{C}^n$  so that

$$y_i = x_i L_{ii} = b_i - \sum_{j \in D_i} L_{ij} x_j = b_i - \sum_{j \in D_i} \frac{L_{ij}}{L_{jj}} y_j. \quad (2.4)$$

This means that the solution of the system  $L'y = b$ , where  $L' = \left( \frac{L_{ij}}{L_{jj}} \right)_{i=1, \dots, n, j=1, \dots, n}$  can be used to solve  $Lx = b$  since  $x_i = y_i / L_{ii}$ . Calculating the  $x_i$  given the  $y_i$  can be done very efficiently on the GPU as it is a massively parallelizable operation and if we store  $1/L_{ii}$  instead of  $L_{ii}$  in memory we can compute the  $x_i$  using a complex multiplication instead of a complex division which is much faster. Note that computing the inverse of each  $L_{ii}$  and computing the  $L_{ij}/L_{jj}$  has only to be done once and this computation's cost is amortized by the fact that the triangular solve is only a small part of a larger iterative method where each triangular system is solved several times with different right-hand sides. Moreover, reducing the problem to a unit diagonal system also allows to get rid of the division by  $L_{ii}$  which requires to synchronize all the threads which are involved in the computation of a certain  $x_i$ . For all these reasons, from now on, we will consider that the problem we want to solve is simply a unit-diagonal triangular system.

## 2.1 Preprocessing Step

We want to solve  $Lx = b$ , where  $L$  is a complex sparse unit-diagonal lower triangular matrix. The system can thus be solved as follows:

$$x_i = b_i - \sum_{j=1}^{i-1} L_{ij} x_j, \quad (2.5)$$

where  $i = 1, 2, \dots, n$ . We observe that to compute  $x_i$ , we need to have computed all the  $x_j$  such that  $j < i$ . This means that we cannot compute in parallel several  $x_i$ . However, if  $L$  is sparse which is the case here, we can reduce the amount of computations in Equation 2.3, since if  $L_{ij} = 0$ , then  $L_{ij} x_j = 0$  as well. This means that

$$x_i = b_i - \sum_{j \in D_i} L_{ij} x_j, \quad (2.6)$$

with  $D_i = \{j \mid L_{ij} \neq 0 \wedge i \neq j\}$ . Since  $L$  is sparse, we know that  $|D_i| \ll i - 1$  for most  $i$ . Consequently, the computation of  $x_i$  does not depend on all the  $x_j$  with  $j < i$ . We can exploit this fact to compute several unknowns in parallel.

The goal of the preprocessing step is to analyze the sparsity pattern of  $L$  to build a structure called a scheduler which will tell the GPU how to parallelize the computation of the solution in the solution step. It is important to note that the preprocessing step is entirely done by the CPU and the scheduler is then copied to the GPU. Only the solution step will be carried out by the GPU.

The idea of the scheduler denoted as  $\mathcal{S}$  is to divide the solution step into a sequence of stages  $(S_1, S_2, \dots)$ , where each stage  $S_i$  consists of a set of what we call nodes. A node  $N$  is an ordered list of increasing row indices  $N = (i_1, i_2, \dots)$  (in the case of an upper triangular system, the list must be decreasing). The row indices maintained by each node, must be independent of the row indices maintained by the other nodes in the same stage. When a node  $N$  is computed, it means that we sequentially compute the unknowns  $x_{i_1}, x_{i_2}, \dots$ . While each node computes unknowns sequentially, different nodes in the same stage are executed in parallel. Once all nodes in the current stage have been computed, we proceed to the next stage, and continue until we reach the final stage.

For this computation to be correct, we must impose several constraints. First, we want each  $x_i$  to be computed only once, which means that row  $i$  can only appear in one node once. Furthermore, since nodes within the same stage are computed in parallel, they must be independent. This means that no two nodes should contain rows that depend on rows in the other node. We say that two rows  $i$  and  $j$  are independent if and only if  $L_{ij} = L_{ji} = 0$ . This allows us to formally define the necessary and sufficient condition for two nodes to be independent. Let  $N_1 = (i_1, i_2, \dots)$  and  $N_2 = (j_1, j_2, \dots)$  be two nodes. Then,  $N_1$  and  $N_2$  are independent if and only if

$$(i_p \neq j_q) \wedge (L_{i_p j_q} = L_{j_q i_p} = 0), \forall p = 1, 2, \dots, \forall q = 1, 2, \dots \quad (2.7)$$

Next, we must make sure that any row  $i$  in a given stage only depends on rows from earlier stages or on rows which are computed earlier within the same node since computations inside a node are sequential. Let us define some useful functions:

- $s(i)$  maps a row index to its stage index.
- $n_l(i)$  maps a row index to its local node index inside  $s(i)$ .

The constraint can be expressed as follows:

$$\forall i, j \text{ s.t. } L_{ij} \neq 0 : s(j) < s(i) \vee (s(j) = s(i) \wedge n_l(i) = n_l(j)). \quad (2.8)$$

In practice, the scheduler is stored in memory using three vectors:  $\mathcal{S} = (S, N, R)$ .  $S$  is a vector where each pair of consecutive elements defines a range of indices in  $N$ . The global indices of the nodes in stage  $i$  are given by  $S_i, S_i + 1, \dots, S_{i+1} - 1$ . The  $N$  vector is such that each pair of consecutive elements defines a range of indices in  $R$ . In node  $j$  (where  $j$  is the global index of the node), the row indices can be accessed in  $R$  at the following indices:  $N_j, N_j + 1, \dots, N_{j+1} - 1$ . For example, to access the  $k$ -th row of the  $j$ -th node of the  $i$ -th stage, one accesses  $R_{N_{S_i+j}+k}$ .

### 2.1.1 Level-Set

A simple way to build a scheduler  $\mathcal{S}$  would be to assign to each node one single row. This would mean that each stage computes as many unknowns as there are nodes. One

way of assigning unknowns to stages is to loop from  $i = 1$  to  $n$ . At each iteration, if  $D_i$  is empty, we set  $s(i)$  to zero. If  $D_i$  is not empty, we know that  $s(j)$  is already set for each  $j \in D_i$  since the system is lower triangular. We can thus simply update  $s(i)$  as follows:

$$s(i) \leftarrow \max_{j \in D_i} \{s(j)\} + 1. \quad (2.9)$$

This method is well-known and often called **level-set** [3] [26]. Algorithm 3 shows how the **level-set** method can be implemented. Note that for each unknown which is dependency-free, its stage will be zero at the end of the algorithm. These unknowns do not have to be added to the scheduler, as no computations are required to find them ( $x_i = b_i$ ,  $\forall i$  such that  $D_i = \emptyset$ ).

---

**Algorithm 3:** Preprocessing Step: **level-set**

---

```

1 Function PreprocessLS():
2    $p(j) := 0 \ \forall j \in \mathbb{N}$ 
3   for  $i \leftarrow 1$  to  $n$  do
4      $s(i) \leftarrow 0$ 
5      $n_l(i) \leftarrow 0$ 
6     foreach  $j \in D_i$  do
7       UpdateLS( $i, j$ )
8     end
9     if  $n_l(i) > p(s(i))$  then
10       $p(s(i)) \leftarrow n_l(i)$ 
11    end
12  end
13 end
14 Function UpdateLS( $i, j$ ):
15   if  $s(i) \leq s(j)$  then
16      $s(i) \leftarrow s(j) + 1$ 
17      $n_l(i) \leftarrow p(s(i)) + 1$ 
18   end
19 end

```

---

Let us see how it would work on a simple example. We assume we want to solve the system  $Lx = b$ , where  $L$  is defined as follows:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix}. \quad (2.10)$$

Figure 2.1 shows the directed acyclic graph (DAG) representing the dependencies between the unknowns of the system. If we build the scheduler using the `level-set` method, we would get the scheduler depicted in Figure 2.2.

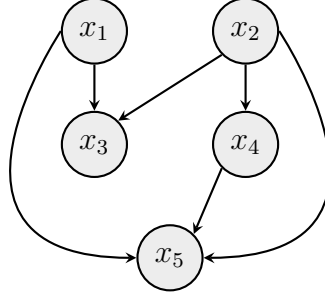


Figure 2.1: DAG representing the dependencies of the unknowns of the system defined by the matrix  $L$  in Equation 2.10.

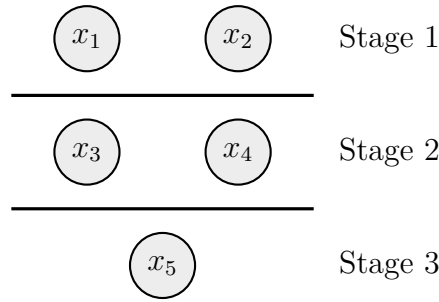


Figure 2.2: Scheduler associated with the matrix  $L$  defined in Equation 2.10 built with the `level-set` method.

Note that the scheduler obtained is not the only possible single unknown per node scheduler we can build. Another possibility is shown in Figure 2.3.

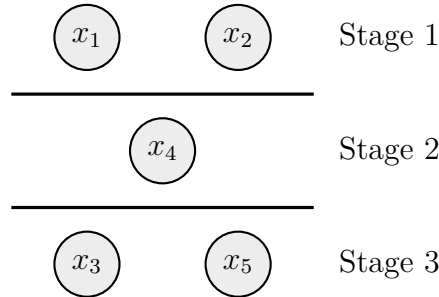


Figure 2.3: Scheduler associated with the matrix  $L$  defined in Equation 2.10.

In order to be able to create better scheduler building algorithms, it would be useful to have a way of evaluating a given scheduler. We could do this experimentally by measuring the time necessary to solve the system with it. Another approach is to

define a computable cost measure which represents how time-consuming it is to solve the system with the given scheduler. We can do this by computing the critical path length or time span of the computation, which represents the minimum time that the computation would take in an idealized setting where we have an infinite number of parallel threads. We know from Equation 2.6 that computing  $x_i$  has a time complexity  $O(|D_i|)$  if we compute the sum sequentially. Since the unknowns in a node are computed sequentially, we can conclude that the time span of a node  $N$  is  $O(\sum_{i \in N} |D_i|)$ . The span of a stage  $S$  is  $O(\max_{N \in S} \{\sum_{i \in N} |D_i|\})$  since all the nodes are executed in parallel. This means that the total time span for solving a system given a scheduler  $\mathcal{S}$  is  $O(C(\mathcal{S}))$  where

$$C(\mathcal{S}) = \sum_{S \in \mathcal{S}} \max_{N \in S} \left\{ \sum_{i \in N} |D_i| \right\}. \quad (2.11)$$

This measure  $C(\mathcal{S})$  can be used to determine the quality of a scheduler  $\mathcal{S}$ .

The cost measure for the scheduler  $\mathcal{S}_1$  shown in Figure 2.2 is  $C(\mathcal{S}_1) = 5$ . For the scheduler  $\mathcal{S}_2$  shown in Figure 2.3 on the other hand we find  $C(\mathcal{S}_2) = 4$  which indicates that this scheduler is better than  $\mathcal{S}_1$ .  $\mathcal{S}_2$  is better because it delays the computation of  $x_3$  by moving it from stage 2 to stage 3, and by doing so, it decreases the cost of stage 2 without increasing the cost of stage 3.

### 2.1.2 Compressed Level-Set

Until now, we did not take advantage of the fact that a scheduler can have several unknown computations per node. In the **level-set** method, we used the formula

$$s(i) \leftarrow \max_{j \in D_i} \{s(j)\} + 1 \quad (2.12)$$

to assign a stage to each unknown. This makes sure that each unknown is always one stage after its latest dependency. This is not always necessary to guarantee that the scheduler is correct. If all the dependencies which are in stage  $\max_{j \in D_i} \{s(j)\}$  are in the same node, we can put the  $i$ -th unknown in this same node as well. Doing this can greatly reduce the number of stages, which is why this method was named **compressed-level-set**. Reducing the number of stages is generally a good thing because, as we will see later, each stage will correspond to a CUDA kernel launch which is not free. Algorithm 4 shows how it can be implemented. Notice that, as for the **level-set** algorithm, the unknown indices  $i$  such that  $s(i) = 0$  are the dependency-free unknowns and these are excluded from the scheduler.

We can once again build a scheduler using this new algorithm based on the system defined in Equation 2.10. The resulting scheduler  $\mathcal{S}_3$  is shown in Figure 2.4. We observe that  $C(\mathcal{S}_3) = 4$ , which is better than what we had with the simple **level-set** algorithm and it uses less stages which is also better. This does not mean that the **compressed-level-set** algorithm is better than **level-set** in all cases. Let us consider a case in which there is only one single dependency-free unknown (it would necessarily

---

**Algorithm 4:** Preprocessing Step: compressed-level-set

---

```

1 Function PreprocessCLS():
2    $p(j) := 0 \forall j \in \mathbb{N}$  // keeps track of the number of nodes in each
   stage
3    $f(i) := \perp \forall i \in \{1, \dots, n\}$  // stores whether the  $i$ -th unknown
   depends on some other unknown which is in the same node
4   for  $i \leftarrow 1$  to  $n$  do
5      $s(i) \leftarrow 0$ 
6      $n_l(i) \leftarrow 0$ 
7     foreach  $j \in D_i$  do
8       | UpdateCLS( $i, j$ )
9     end
10    if  $n_l(i) > p(s(i))$  then
11      |  $p(s(i)) \leftarrow n_l(i)$ 
12    end
13  end
14 end
15 Function UpdateCLS( $i, j$ ):
16   if  $s(i) < s(j) \vee (\neg f(i) \wedge s(i) = s(j) \wedge s(i) > 0)$  then
17     |  $s(i) \leftarrow s(j)$ 
18     |  $f(i) \leftarrow \top$ 
19     |  $n_l(i) \leftarrow n_l(j)$ 
20   else if  $s(i) = s(j) \wedge (n_l(i) \neq n_l(j) \vee s(i) = 0)$  then
21     |  $s(i) \leftarrow s(j) + 1$ 
22     |  $f(i) \leftarrow \perp$ 
23     |  $n_l(i) \leftarrow p(s(i)) + 1$ 
24   end
25 end

```

---

be  $x_1$ ). In this scenario, the algorithm will build a scheduler with only one stage and one node which will contain all the unknowns  $x_i$  with  $i > 1$ . There would be no parallelism at all.



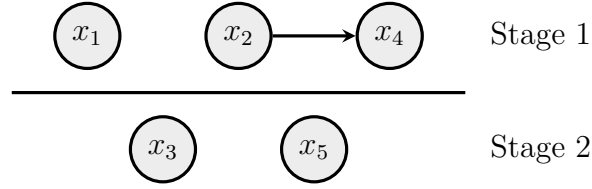


Figure 2.4: Scheduler associated with the matrix  $L$  defined in Equation 2.10 built with the **compressed-level-set** method. An arrow between two unknowns means that these two unknowns share the same node.

### 2.1.3 Balanced Level-Set

Until now, we have seen two algorithms to build a scheduler. We can combine these two algorithms into one by using a heuristic which will select which of the two previously seen strategies is best to schedule each individual unknown. Algorithm 5 shows how it can be implemented. The only difference with Algorithm 4 is that, if some criterion is not met, we revert to the **level-set** strategy. By choosing a good criterion, we can improve significantly the obtained scheduler. One heuristic that worked well was to combine two criteria by setting a maximum node cost  $C_{\max}$  and a maximum number of unknowns per node  $U_{\max}$ . We say that the heuristic meets the criterion when for a given unknown  $x_i$ , adding  $x_i$  to the current stage  $s(i)$  and node  $n_l(i)$  does not make the total node cost exceed  $C_{\max}$  and the total number of rows exceed  $U_{\max}$ . The disadvantage of this heuristic is that it requires to adjust  $C_{\max}$  and  $U_{\max}$  which can be challenging.

---

**Algorithm 5:** Preprocessing Step: balanced-level-set

---

```

1 Function PreprocessBLS():
2    $p(j) := 0 \forall j \in \mathbb{N}$     // keeps track of the number of nodes in each
   stage
3    $f(i) := \perp \forall i \in \{1, \dots, n\}$     // stores whether the  $i$ -th unknown
   depends on some other unknown which is in the same node
4   for  $i \leftarrow 1$  to  $n$  do
5      $s(i) \leftarrow 0$ 
6      $n_l(i) \leftarrow 0$ 
7     foreach  $j \in D_i$  do
8       | UpdateCLS( $i, j$ )
9     end
10    if  $n_l(i) > p(s(i))$  then
11      |  $p(s(i)) \leftarrow n_l(i)$ 
12    end
13  end
14 end
15 Function UpdateBLS( $i, j$ ):
16   if  $s(i) < s(j) \vee (\neg f(i) \wedge s(i) = s(j) \wedge s(i) > 0)$  then
17     |  $s(i) \leftarrow s(j)$ 
18     |  $f(i) \leftarrow \top$ 
19     |  $n_l(i) \leftarrow n_l(j)$ 
20     | if  $\neg \text{MeetsHeuristicCriterion}(i)$  then
21       | |  $s(i) \leftarrow s(j) + 1$ 
22       | |  $f(i) \leftarrow \perp$ 
23       | |  $n_l(i) \leftarrow p(s(i)) + 1$ 
24     | end
25   else if  $s(i) = s(j) \wedge (n_l(i) \neq n_l(j) \vee s(i) = 0)$  then
26     |  $s(i) \leftarrow s(j) + 1$ 
27     |  $f(i) \leftarrow \perp$ 
28     |  $n_l(i) \leftarrow p(s(i)) + 1$ 
29   end
30 end

```

---

## 2.2 Solution Step

We can now look into how the solution is actually computed. We will implement several kernels ranging from simplest to most sophisticated and their performances will be compared in the next chapter.

### 2.2.1 CUDA Kernels

To understand the various algorithms we will discuss, it is necessary to review how to run code on the GPU. The approach taken involves combining the CUDA runtime API with CUDA kernels. The runtime API provides C and C++ functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc [8]. CUDA kernels are functions which are executed on the GPU. These are written in an extension of C++ which provides several intrinsics and variables. When a kernel is launched, the function is then executed on several GPU threads concurrently.

Launching a kernel is similar to calling a function, but it requires additional configuration parameters. Threads are organized into thread blocks which can contain at most 1024 threads. The size of a thread block is specified by a three-dimensional number. Inside the kernel we can access the current thread index within the current block with the variable `threadIdx` which is a three-dimensional number and the dimensions of the current block using `blockDim`. Thread blocks are themselves organized into a three-dimensional grid of blocks. The maximum number of blocks per kernel launch is  $2^{31} - 1$ . The current thread block index can be accessed with the variable `blockIdx` and the kernel grid dimensions with `gridDim`. The dimensions of the grid as well as the dimensions of the thread blocks is specified when the kernel is launched. The fact that threads and thread blocks are organized into a 3-dimensional grid, is mainly for the convenience of the developer, as it simplifies the code writing process for many algorithms. Finally, thread blocks are also divided into warps. A warp is a group of 32 consecutive threads which execute in lockstep, meaning that these threads are always synchronized.

Another important element to understand is memory. A GPU has several memory regions. Each thread has its own local memory and registers. Each thread block has some amount of shared memory which is shared between all threads within the same thread block. Shared memory can either be statically allocated or dynamically allocated at kernel launch. It can be seen as a self managed cache. Finally, there is global memory which is shared across all GPU kernels on the same device. This is where most of the data structures will be stored.

The three configuration parameters necessary to launch a kernel are: the grid dimensions, the block dimensions and the amount of shared memory required per thread block. In the algorithms that we will see, we will use the notation: `Kernel < gridDim, blockDim, sharedMem >` to specify those parameters. The `sharedMem` variable will have a value in bytes.

Finally, Table 2.1 lists the different variables and intrinsics that we will use which are accessible within a CUDA kernel.

Variable	Description
<code>threadIdx</code>	3D index of the current thread
<code>blockDim</code>	3D dimensions of the current thread block
<code>blockIdx</code>	3D index of the current block
<code>gridDim</code>	3D dimensions of the thread block grid

Intrinsic Function	Description
<code>__syncthreads</code>	Synchronizes all threads in a thread block
<code>__shfl_down_sync</code>	Exchanges a variable between threads within a warp

Table 2.1: List of intrinsic functions and variables used in the CUDA kernels.

### 2.2.2 Optimized Reduction

The kernels we will implement will often need to perform reductions to sum elements owned by several threads in a thread block. For this, we implement a block-level reduction function which is as optimized as possible. The implementation of this function is heavily based on [15]. We implement a `Reduce` function which takes three arguments: the array to reduce, the current block-wide thread index and the number of threads participating in the reduction.

Instead of writing the algorithm in pseudo-code, we decided to directly present the actual C++ code (see Listing 2.1), as the presence of templates and constant expressions makes it challenging to translate into pseudo-code.

Note that, the `.x` and `.y` accessors represent, respectively, the real and the imaginary parts of the scalar value being accessed. The main difference with [15] is that we use a warp-level primitive `__shfl_down_sync` which allows for fast reduction operations within a warp. This primitive is used to efficiently get the value of a variable stored in another thread part of the same warp. One can notice that the reduction function can only accept block sizes which are a power of two. This will be important to keep in mind when launching kernels.

```

1 template<unsigned int block_size>
2 __device__ static void warp_reduce(volatile cuDoubleComplex values[],
   int64_t thread) {
3     if constexpr (block_size >= 64) {
4         values[thread].x += values[thread + 32].x;
5         values[thread].y += values[thread + 32].y;
6     }
7
8     volatile cuDoubleComplex value;
9     std::memcpy((void *)&value, (const void *)&values[thread], sizeof(
   cuDoubleComplex));
10
11     if constexpr (block_size >= 32) {

```

```

12     value.x += __shfl_down_sync(0xffffffff, value.x, 16);
13     value.y += __shfl_down_sync(0xffffffff, value.y, 16);
14 }
15 if constexpr (block_size >= 16) {
16     value.x += __shfl_down_sync(0xffffffff, value.x, 8);
17     value.y += __shfl_down_sync(0xffffffff, value.y, 8);
18 }
19 if constexpr (block_size >= 8) {
20     value.x += __shfl_down_sync(0xffffffff, value.x, 4);
21     value.y += __shfl_down_sync(0xffffffff, value.y, 4);
22 }
23 if constexpr (block_size >= 4) {
24     value.x += __shfl_down_sync(0xffffffff, value.x, 2);
25     value.y += __shfl_down_sync(0xffffffff, value.y, 2);
26 }
27 if constexpr (block_size >= 2) {
28     value.x += __shfl_down_sync(0xffffffff, value.x, 1);
29     value.y += __shfl_down_sync(0xffffffff, value.y, 1);
30 }
31
32     std::memcpy((void *)&values[thread], (const void *)&value, sizeof(
cuDoubleComplex));
33 }
34
35 template<unsigned int block_size>
36 __device__ static void block_reduce(cuDoubleComplex values[], int64_t
thread) {
37     if constexpr (block_size >= 1024) {
38         if (thread < 512) {
39             values[thread] = scalar_add(values[thread], values[thread +
512]);
40         }
41         __syncthreads();
42     }
43     if constexpr (block_size >= 512) {
44         if (thread < 256) {
45             values[thread] = scalar_add(values[thread], values[thread +
256]);
46         }
47         __syncthreads();
48     }
49     if constexpr (block_size >= 256) {
50         if (thread < 128) {
51             values[thread] = scalar_add(values[thread], values[thread +
128]);
52         }
53         __syncthreads();
54     }
55     if constexpr (block_size >= 128) {
56         if (thread < 64) {
57             values[thread] = scalar_add(values[thread], values[thread +
64]);

```

```

58     }
59     __syncthreads();
60 }
61 if (thread < 32) {
62     warp_reduce<block_size>(values, thread);
63 }
64 }
65
66 __device__ __forceinline__ static void reduce(cuDoubleComplex *values,
67 int64_t thread, int64_t n_threads) {
68     switch(n_threads) {
69         case 1024:
70             block_reduce<1024>(values, thread);
71             break;
72         case 512:
73             block_reduce<512>(values, thread);
74             break;
75         case 256:
76             block_reduce<256>(values, thread);
77             break;
78         case 128:
79             block_reduce<128>(values, thread);
80             break;
81         case 64:
82             block_reduce<64>(values, thread);
83             break;
84         case 32:
85             block_reduce<32>(values, thread);
86             break;
87         case 16:
88             block_reduce<16>(values, thread);
89             break;
90         case 8:
91             block_reduce<8>(values, thread);
92             break;
93         case 4:
94             block_reduce<4>(values, thread);
95             break;
96         case 2:
97             block_reduce<2>(values, thread);
98             break;
99         case 1:
100             block_reduce<1>(values, thread);
101             break;
102     }

```

Listing 2.1: Optimized reduction

### 2.2.3 Matrix Storage Format

Our goal is to solve the lower triangular sparse system  $Lx = b$ , where  $L \in \mathbb{C}^{n \times n}$ . It is important to discuss how the matrix  $L$  should be stored in the memory of the program. During the solution step, no transformations will be performed on  $L$  which means that the only crucial aspects to consider are the access times and the space complexity.

The two simplest storage formats, setting aside the use of a two-dimensional array, use an array with  $n \times n$  entries. There is the row-major format in which each element  $L_{ij}$  is stored at the index  $(i - 1)n + j - 1$ , where the array is indexed with zero-based numbering while the matrix element  $L_{ij}$  uses as usual one-based numbering. The column-major format on the other hand stores the element  $L_{ij}$  at the index  $(j - 1)n + i - 1$ . These two formats have a space complexity  $O(n^2)$ . The choice between the two should primarily be based on the memory access patterns of the program. The access times are constant. These two formats will be used when discussing the multiple right-hand side case.

The matrix  $L$  is sparse, which means that most of its entries are zero. This fact suggests that it should be possible to store  $L$  more efficiently by avoiding to store zeros. The COOrdinate format (COO) is a list of triples  $(i, j, z)$ , where  $z = L_{ij}$ . If there is no triple corresponding to some row  $i$  and column  $j$  in the list, it means that  $L_{ij} = 0$ . This format has a space complexity  $O(\text{nnz}(L))$ , where  $\text{nnz}(L)$  is the number of nonzero elements in  $L$ . The main drawback of this format is that it has linear random, row and column access times ( $O(\text{nnz}(L))$ ).

The format that we will use is the Compressed Sparse Row (CSR) format. It represents  $L$  using three vectors:  $L^{(\text{row})} \in \mathbb{N}^{n+1}$ ,  $L^{(\text{col})} \in \mathbb{N}^{\text{nnz}(L)}$  and  $L^{(\text{val})} \in \mathbb{C}^{\text{nnz}(L)}$ .  $L^{(\text{val})}$  contains all the nonzero values in the  $L$  matrix and  $L^{(\text{col})}$  stores the corresponding columns.  $L^{(\text{row})}$  contains the start and end indices of each row in  $L^{(\text{val})}$  and  $L^{(\text{col})}$ . We assume that each row in  $L^{(\text{val})}$  and  $L^{(\text{col})}$  is stored in increasing column index order (the CSR format does not necessarily impose this). We can see that

$$D_i = \left\{ L_k^{(\text{col})} \mid k \in \left\{ L_i^{(\text{row})}, L_i^{(\text{row})} + 1, \dots, L_{i+1}^{(\text{row})} - 2 \right\} \right\}. \quad (2.13)$$

This makes it simple to iterate over the set  $D_i$  when the matrix  $L$  is in CSR format.

### 2.2.4 Task Distribution

Let us introduce a function  $d : \mathbb{N}^3 \rightarrow \mathbb{N}^2$ . This function is used to assign a certain amount of tasks to a certain amount of slots. Each slot identified by its slot index will be given a range of tasks which are identified by a task index. For the task and slot indices, exclusively zero-based indexing will be used. The function  $d$  takes three arguments: the slot index  $t$ , the number of tasks  $N$  and the total number of slots  $T$ . We will later see that by slot we will usually mean a thread block. The idea is that each slot should get a range of tasks that doesn't overlap with any other slot and the union of all the ranges should contain all tasks from 0 to  $T - 1$ . Moreover, the convention that we use is that if the function returns  $(a, b)$ , then the range of tasks is  $a, a + 1, \dots, b - 1$ .

When  $a = b$ , we have a zero-size range which means that this particular slot does not receive any task.

One natural definition could be the following:

$$d(t, N, T) = \left( \left\lfloor \frac{tN}{T} \right\rfloor, \left\lfloor \frac{(t+1)N}{T} \right\rfloor \right). \quad (2.14)$$

This approach works both when the number of tasks is greater than the number of slots and when the number of slots is greater, in which case some slots will receive zero-size ranges. If we look at the range sizes of adjacent slots, we notice that they can differ on several occasions (e.g., see Table 2.2). This is not optimal because we want nearby slots to have equal task range sizes. If the slots correspond to threads, assuming that each thread loops over its task range, it is better to make sure that consecutive threads have the same task range size. This is the case because threads which are part of the same warp execute in lockstep, which means that if the task range sizes differ between those threads, the ones that terminate earlier will have to wait for the others.

Thread	0	1	2	3	4	5	6
$d(t, N, T)$	(0, 3)	(3, 6)	(6, 9)	(9, 13)	(13, 16)	(16, 19)	(19, 23)
Range Size	3	3	3	4	3	3	4

Table 2.2: Example of task distribution using the distribution function defined in Equation 2.14 when the number of tasks is  $N = 23$  and the number of threads is  $T = 7$ .

It can be shown that it is always possible to distribute  $N > 0 \in \mathbb{N}$  tasks among  $T > 0 \in \mathbb{N}$  threads such that each thread receives either  $r$  or  $r+1$  tasks for some  $r \in \mathbb{N}$ . Let us assume that there are  $M$  threads with  $r+1$  tasks (and thus  $T-M$  with  $r$  tasks), we need to prove that we can find  $M \in \{0, 1, \dots, T\}$  and  $r \in \mathbb{N}$  such that

$$M(r+1) + (T-M)r = N. \quad (2.15)$$

Let

$$M = N - T \left\lfloor \frac{N}{T} \right\rfloor = N \bmod T \quad \text{and} \quad r = \left\lfloor \frac{N}{T} \right\rfloor. \quad (2.16)$$

This satisfies Equation 2.15, both values are obviously integers and  $M \in \{0, 1, \dots, T-1\}$ . This suggests that we can always define an assignment with at most one pair of adjacent threads which have a different number of tasks ( $r$  and  $r+1$ ) by simply assigning  $r+1$  tasks to the  $M$  first threads and then  $r$  tasks to the  $T-M$  last ones (or vice versa). We can achieve this with the following function:

$$d(t, N, T) = \left( \begin{array}{c} t \left\lfloor \frac{N}{T} \right\rfloor + \min(t, N \bmod T) \\ (t+1) \left\lfloor \frac{N}{T} \right\rfloor + \min(t, N \bmod T) + 1 - H(t - N \bmod T) \end{array} \right), \quad (2.17)$$

where  $H$  is the unit step function. Table 2.3 shows how this new function would distribute tasks with an example.



Thread	0	1	2	3	4	5	6
$d(t, N, T)$	(0, 4)	(4, 8)	(8, 11)	(11, 14)	(14, 17)	(17, 20)	(20, 23)
Range Size	4	4	3	3	3	3	3

Table 2.3: Example of task distribution using the distribution function defined in Equation 2.17 when the number of tasks is  $N = 23$  and the number of threads is  $T = 7$ .

### 2.2.5 Smart Parameters

It is often challenging to select the optimal dimensions for the thread blocks and the grid of a given kernel to achieve the best performance. CUDA provides the `cudaOccupancy` API to address this issue, which contains routines to heuristically determine the best launch configuration for a given kernel to maximize occupancy. In the different kernels we will discuss, we will sometimes use, in the pseudo-code, the function `GetIdealKernelConfig`, which will return an estimation of the ideal grid and block size. In practice, this is implemented with the `cudaOccupancy` API.

Unfortunately, configurations are chosen heuristically, which may not correspond to the actual optimal configuration. Luckily, since our triangular solver is meant to be used within an iterative method, we can hope to execute the same kernel multiple times for the same system. We could thus at each new kernel launch change slightly the configuration and assess whether it improves the overall performance of the kernel. This is the idea underlying smart parameters. Some kernels will have as input a **params** variable which will be a vector of values used to control the configuration of the kernel. After each iteration, the **params** vector is updated. The approach chosen is similar to a simplified simulated annealing algorithm, where the value we want to minimize is the execution time of the kernel. When the execution time of the kernel is worse than the previous kernel execution time, the **params** vector is reset to its previous value and one of its elements is chosen at random and randomly modified (either increased or decreased). On the other hand, if the execution time is better than in the previous kernel execution, then we update the same element as the one updated at the previous iteration and adjust it in the same direction as earlier. The key is that with each new iteration, the magnitude of change in the chosen element is scaled down. This approach allows a broad exploration of the search space initially, followed by a convergence towards either a local minimum or, ideally, a global one.

### 2.2.6 Sequential Methods

We implement the **sequential** strategy discussed earlier. This strategy does not require any scheduler and comes in two flavours: **sequential-singleblock** which only uses one CUDA kernel launch and one single thread block and **sequential-multiblock** which uses several blocks and  $n$  kernel launches, where  $n$  is the size of the system we want to solve. Contrary to what the name suggests, these strategies are not entirely sequential, but the unknowns are computed in sequence ( $x_i$  is computed before  $x_{i+1}$ ).

if the system is lower triangular). We introduce parallelism in the computation of each individual unknown by distributing partial sums that we will call segments to the available threads. As a reminder, each unknown  $x_i$  is computed using Equation 2.6.

Let us define the set of thread indices  $\mathcal{T} = \{0, 1, \dots, T-1\}$  as well as the set of unknown indices  $\mathcal{U} = \{1, 2, \dots, n\}$ . In the **sequential-singleblock** method, at iteration  $i$ , each thread  $t$  will compute a value

$$y_{it} = - \sum_{j \in D_{it}} L_{ij} x_j, \quad (2.18)$$

where  $\forall i \in \mathcal{U}$ , it is necessary that  $\bigcup_{t \in \mathcal{T}} D_{it} = D_i$  and  $D_{it_1} \cap D_{it_2} = \emptyset \forall t_1 \neq t_2$  such that  $(t_1, t_2) \in \mathcal{T}^2$ . It is clear that these two conditions do not uniquely determine  $D_{it}$ . The chosen definition was the following:

$$D_{it} = \left\{ L_p^{(\text{col})} \mid p = t + L_i^{(\text{row})} + kT, \text{ where } k \in \mathbb{N} \text{ and } p \leq L_{i+1}^{(\text{row})} - 2 \right\}. \quad (2.19)$$

This definition was chosen to maximize bandwidth. To understand why, it is necessary to review the memory layout of a GPU. The GPU has two main memory regions: global memory and shared memory. The matrix  $L$  and the unknowns  $x_i$  are stored in global memory, while the segments  $y_{it}$  are stored in shared memory.

Global memory is generally much slower than shared memory, but its access can be made relatively efficient through coalesced memory accesses. Memory coalescing involves arranging memory accesses so that successive threads in a warp (a group of 32 consecutive threads that execute in SIMD fashion) access consecutive memory locations. Doing this allows the GPU to combine several memory accesses into a single memory transaction, which significantly improves performance.

In our case, when some thread  $t$  computes  $y_{it}$ , at iteration  $k$ , it accesses  $L_i^{(\text{row})}$ ,  $L_p^{(\text{col})}$ ,  $L_p^{(\text{val})}$  and  $x_{L_p^{(\text{col})}}$ , with  $p = t + L_i^{(\text{row})} + kT$  in global memory. The next thread  $t+1$  will request  $L_i^{(\text{row})}$ ,  $L_{p+1}^{(\text{col})}$  and  $L_{p+1}^{(\text{val})}$ . As a result, memory accesses to  $L^{(\text{col})}$  and  $L^{(\text{val})}$  are coalesced as these vectors are stored contiguously in global memory. Since all the threads request  $L_i^{(\text{row})}$ , this memory access will only require a single transaction, broadcasting the value to the requesting threads. On the other hand, successive threads will not necessarily access consecutive memory locations in the vector  $x$ . The accesses to  $x$  will only be coalesced to some degree if successive nonzero columns in  $L$  have consecutive indices.

Note that we could have defined  $D_{it}$  using the task distribution function  $d$ :

$$\begin{aligned} D_{it} = \left\{ L_p^{(\text{col})} \mid p = L_i^{(\text{row})} + k, \text{ where} \right. \\ k \in \{s, s+1, \dots, e-1\}, \\ (s, e) = d(t, N, T) \text{ and} \\ \left. N = L_{i+1}^{(\text{row})} - L_i^{(\text{row})} - 1 \right\}. \end{aligned} \quad (2.20)$$

This works, however we do not have coalesced memory accesses anymore. This is so, because successive  $p$  values will be assigned to the same thread.

Next, the shared memory region is shared among threads within a thread block and is structured with several banks arranged so that successive 32-bit words are stored in consecutive banks and each bank has a memory bandwidth of 32 bits per clock cycle. This arrangement is crucial because if two threads within the same warp attempt to access different addresses in the same bank, the accesses will be serialized, resulting in a bank conflict. The simplest way to avoid bank conflicts is to have consecutive threads in a thread block access successive words in shared memory, ensuring that threads in the same warp access different banks simultaneously. It is important to note that if multiple threads access the same address in the same bank, there is no bank conflict either as the accessed value will be multicast to the requesting threads. In our scenario, the segments  $y_{it}$  are double precision complex values stored contiguously in shared memory, occupying 128 bits or equivalently 16 bytes. If  $y_{it}$  is stored at address  $A$ , then  $y_{i,t+1}$  is stored at address  $A + 16$  bytes. Each  $y_{it}$  is spread across 4 consecutive banks. When each successive thread in a warp attempts to access a successive 16-byte value, the total warp-wide memory request size is  $32 \times 16 = 512$  bytes. However, each memory request is limited to 128 bytes (one word per bank). Consequently, the hardware will divide the memory request into 4 conflict-free requests (for instance, the first request loads values requested by the first 8 threads, the second request loads the values requested by the next 8 threads, and so on). Even though the memory request had to be split into 4 requests, maximum throughput is still achieved (128 bytes per clock cycle) so there are no bank conflicts.

Recall that Table 2.1 lists the variables and intrinsic functions that will be used in the kernels that will be seen. These variables and intrinsics are accessible only within a CUDA kernel. Algorithm 6 illustrates how the `sequential-singleblock` method can be implemented. The `KernelSeqSingleblock` function is the function which is actually executed by the GPU, while the remaining code runs on the CPU. The kernel is launched with one single thread block with  $16 \times \text{blockDim.x}$  bytes of shared memory which is needed to allow each thread  $t$  to store a segment  $y_{it}$ . The `cache` variable is an array of scalars representing the shared memory. To index the shared memory array, the notation `cache[cacheldx]`, where `cacheldx = t` was used instead of a subscript (`cachecacheldx`), as used for other vectors in the algorithm. This distinction was done to differentiate zero-based indexed arrays from one-based indexed arrays like  $x$  or  $L^{(\text{row})}$ . The algorithm launches a single GPU kernel which will do all the computations by iterating over the rows of  $L$  (from  $i = 1$  to  $n$ , as the system of interest is lower triangular). At iteration  $i$ , the value computed and stored in `cache[cacheldx]` is  $y_{it}$ . A reduction operation sums all the values in `cache` and stores the result in `cache[0]`. Finally, `cache[0]` is added to  $x_i$ , which was initially set to  $b_i$ . The `__syncthreads` intrinsic synchronizes all threads within a thread block. It is called before the reduction to ensure that the computation of  $y_{it}$  is complete for each thread  $t$ . It is also called after updating  $x_i$  to prevent any thread  $t$  from computing  $y_{jt}$  for some  $j > i$  for which  $y_{jt}$  depends on  $x_i$ .

---

**Algorithm 6:** Solution Step: **sequential-singleblock**

---

**Input:**  $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})}) \in \mathbb{N}^{n+1} \times \mathbb{N}^{\text{nnz}(L)} \times \mathbb{C}^{\text{nnz}(L)}$   
 $b \in \mathbb{C}^n$ **Output:**  $x \in \mathbb{C}^n$ 

```

1  $x \leftarrow b$ 
2  $\text{gridDim} \leftarrow \{x = 1, y = 1, z = 1\}$ 
3  $\text{blockDim} \leftarrow \{x = 1024, y = 1, z = 1\}$ 
4  $\text{KernelSeqSingleblock} \langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle (L, x)$ 
5 Function  $\text{KernelSeqSingleblock}(L, x)$ :
6    $t \leftarrow \text{threadIdx}.x$ 
7    $T \leftarrow \text{blockDim}.x$ 
8    $\text{cacheIdx} \leftarrow \text{threadIdx}.x$ 
9    $\text{cacheSize} \leftarrow \text{blockDim}.x$ 
10  for  $i \leftarrow 1$  to  $n$  do
11     $\text{cache}[\text{cacheIdx}] \leftarrow 0$ 
12    for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
13       $j \leftarrow L_p^{(\text{col})}$ 
14       $\text{cache}[\text{cacheIdx}] \leftarrow \text{cache}[\text{cacheIdx}] - L_p^{(\text{val})} \times x_j$ 
15    end
16     $\_\_\text{syncthreads}()$ 
17     $\text{Reduce}(\text{cache}, \text{cacheIdx}, \text{cacheSize})$ 
18    if  $\text{cacheIdx} = 0$  then
19       $x_i \leftarrow x_i + \text{cache}[0]$ 
20    end
21     $\_\_\text{syncthreads}()$ 
22  end
23 end

```

---

One of the limitations of the **sequential-singleblock** method is that we can only launch a kernel with a single thread block, which means that the maximum number of parallel threads is 1024. If one could launch a kernel with several blocks it could drastically increase the maximum parallelism. This is the goal of the **sequential-multiblock** method. The problem when several blocks are created at launch is that there is no global synchronization primitive that allows synchronization between threads of different blocks. Global synchronization is achieved by terminating the current kernel and then starting a new one. Note that the CPU initiates kernel launches asynchronously, which means that when it initiates a kernel launch, it does not wait for the kernel to complete before continuing its execution. If one wants two kernels to be executed serially, one has to make sure that they are part of the same CUDA stream. If no stream is specified at launch, a default one is chosen. We will only ever use this default stream, as we will not need to execute several CUDA kernels concurrently.

Algorithm 7 shows how the **sequential-multiblock** method can be implemented. In this algorithm, the outermost loop, iterating over the rows in  $L$ , is not part of the kernel, and a new kernel is launched for each row  $i$ . The specified amount of shared memory per thread block is  $16 \times \text{blockDim}.x$  bytes, which means that each single thread in the kernel can have its own 16 byte value. This will be the case for all the kernels that will be seen. Each thread owns the value stored at the index  $\text{cacheldx} = \text{threadldx}.x$  in the shared memory array **cache** specific to its thread block. The segment index  $t = \text{threadldx}.x + \text{blockldx}.x \times \text{blockDim}.x$  is equal to the global thread index of the current thread. By global, we mean kernel-wide.  $T$  is the total number of segments to be computed and it is equivalent to the total number of threads in the kernel. Since there are several thread blocks, after the block-level reduction, it is necessary to add to  $x_i$  the value stored at **cache**[0] in each thread block. One approach to achieve this, is to store, for each thread block, **cache**[0] into a global memory temporary array of length **gridDim}.x** at index **blockldx}.x** and then terminate the kernel. A new kernel would then be launched to sum the values stored in this temporary array using a reduction. This kernel launch is necessary to make sure that each block in the previous kernel has computed the temporary value required before reducing the array. Assuming there would never be more than 1024 thread blocks, the reduction could be carried out with a single additional kernel launch consisting of one block and as many threads as there were blocks in the previous kernel. The block-level reduction function **Reduce** could then be invoked after copying the temporary array to shared memory. However, an alternative approach was selected to avoid the necessity of launching an additional kernel. CUDA offers atomic operation functions as **AtomicAdd** which can be invoked to add **cache**[0] to  $x_i$ . In fact, it was possible to completely bypass the use of a reduction operation by instead directly updating the vector  $x_i$  with **AtomicAdd**( $x_i, -L_p^{(\text{val})} \times x_j$ ) at each iteration in the innermost loop. This approach is however not great. Concurrent atomic operations on the same data would be serialized, resulting in a linear time complexity relative to the number of threads, which is less efficient than the logarithmic complexity achieved with a reduction. Therefore, atomic operations should only be used sparingly. However, using the **AtomicAdd** function solely for summing the partial sums computed by each thread block leads to better performance compared to a reduction with an additional kernel launch.

**Algorithm 7:** Solution Step: `sequential-multiblock`


---

**Input:**  $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $b \in \mathbb{C}^n$   
**Output:**  $x \in \mathbb{C}^n$

```

1  $x \leftarrow b$ 
2  $(\text{idealGridSize}, \text{idealBlockSize}) \leftarrow \text{GetIdealKernelConfig}(\text{KernelSeqMultiblock})$ 
3  $\text{gridDim} \leftarrow \{x = \text{idealGridSize}, y = 1, z = 1\}$ 
4  $\text{blockDim} \leftarrow \{x = 2^{\lfloor \log_2 \text{idealBlockSize} \rfloor}, y = 1, z = 1\}$ 
5 for  $i \leftarrow 1$  to  $n$  do
6   |  $\text{KernelSeqMultiblock} \langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle (L, x, i)$ 
7 end
8 Function  $\text{KernelSeqMultiblock}(L, x, i)$ :
9   |  $\text{cacheldx} \leftarrow \text{threadldx}.x$ 
10  |  $\text{cacheSize} \leftarrow \text{blockDim}.x$ 
11  |  $t \leftarrow \text{threadldx}.x + \text{blockldx}.x \times \text{blockDim}.x$ 
12  |  $T \leftarrow \text{blockDim}.x \times \text{gridDim}.x$ 
13  |  $\text{cache}[\text{cacheldx}] \leftarrow 0$ 
14  | for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
15    |  $j \leftarrow L_p^{(\text{col})}$ 
16    |  $\text{cache}[\text{cacheldx}] \leftarrow \text{cache}[\text{cacheldx}] - L_p^{(\text{val})} \times x_j$ 
17  | end
18  |  $\_\_\text{syncthreads}()$ 
19  |  $\text{Reduce}(\text{cache}, \text{cacheldx}, \text{cacheSize})$ 
20  | if  $\text{cacheldx} = 0$  then
21    |  $\text{AtomicAdd}(x_i, \text{cache}[0])$ 
22  | end
23 end

```

---

## 2.2.7 Scheduled Methods

### Default Method

In this section, we first describe the `default` method, which corresponds to the most natural way of solving the system using a scheduler as the ones seen in Section 2.1. Algorithm 8 demonstrates how to implement this method.

One can notice that each stage corresponds to one kernel launch. Distributing the computation across several kernel launches allows to synchronize between stages. The number of blocks in the grid is precisely chosen so that there are as many blocks as there are nodes in the scheduler. Once again, there is no need for a multidimensional grid or multidimensional blocks, so the  $y$  and  $z$  coordinates are set to one.

To understand how the number of threads per thread block is chosen, it is necessary to remember that the computation of a single unknown  $x_i$  requires as many multipli-

cations and additions as there are nonzero elements in the  $i$ -th row of  $L$ . With the sequential methods, we saw that we could distribute these additions and multiplications among the threads, with each thread computing what we called a segment (i.e., a partial sum). Ideally, there would be as many segments as there are nonzero elements in row  $i$  of  $L$ . In such a scenario, every thread would simply perform one addition and one multiplication before calling the **Reduce** function. Since every kernel computes several unknowns, we decided to choose, when possible, a block size that is greater than or equal to the maximum number of nonzero elements in a row  $i$  of  $L$  for which  $x_i$  is computed in the kernel which is about to be launched. These values are computed during the scheduler's construction and stored in a vector  $M \in \mathbb{N}^s$ , where  $s$  is the number of stages in the scheduler (i.e.,  $s = \dim(S) - 1$ ). One can see that this vector is then used in the computation of the chosen block size, given by  $\max(1, \min(1024, 2^{\lceil \log_2 M_{\text{stage}} \rceil - \lfloor \text{params}_1 \rfloor}))$ , which gives a power of two between 1 and 1024. Initially,  $\text{params}_1$  is set to 0 and is used to reduce the chosen block size in subsequent iterations if it is found to be too large. The chosen block size aims to maximize the number of segments  $y_{it}$  (as defined in Equation 2.18, with  $D_{it}$  defined as in Equation 2.19) computed in parallel while minimizing their sizes. Note that as long as the thread block size is a power of two between 1 and 1024, the algorithm remains correct. A power of two is necessary for the **Reduce** function to work as expected (see Section 2.2.2).

Each thread block is associated with a node in the current stage. This means that each thread block will compute in sequence the unknowns in its associated node. Each thread  $t$ , in a thread block will as for the **sequential-singleblock** method compute the segment  $y_{it}$ , with  $t = \text{threadIdx.x}$  being the local thread index within the current thread block. This method is similar to the **sequential-singleblock** method, the only major difference being that, thanks to the scheduler, we can launch multiple thread blocks computing different independent unknowns in parallel and we use multiple kernel launches to synchronize between stages.

### Adaptative Method

The **default** method works well, but it does not fully exploit the parallel processing capabilities of the GPU, as the number of threads allocated to the computation of one unknown  $x_i$  is limited to 1024. If the number of nonzero elements in row  $i$  of  $L$  is greater than some threshold greater than 1024, then it might be better to spread the computation of a single unknown across multiple thread blocks. This idea led to the creation of an **adaptative** method which would, at each stage, choose between the **default** single block per node kernel and a new kernel which would assign multiple blocks to each node. The first difficulty arises from the fact that if a single node is computed across several blocks, it will not be possible to synchronize the blocks once an unknown has been computed. This is problematic because unknowns inside a node must be computed in sequence, as they might depend on each other. There are two ways to address this issue: either restrict the scheduler to unit-size nodes (e.g., by using the **level-set** strategy) or launch multiple kernels to synchronize between the

---

**Algorithm 8:** Solution Step: default

---

**Input:**  $\mathcal{S} := (S, N, R, M)$   
 $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $b \in \mathbb{C}^n$   
 $\text{params} \in \mathbb{R}^1$

**Output:**  $x \in \mathbb{C}^n$

```

1  $x \leftarrow b$ 
2 for stage  $\leftarrow 1$  to  $\dim(S) - 1$  do
3   gridDim  $\leftarrow \{x = S_{\text{stage}+1} - S_{\text{stage}}, y = 1, z = 1\}$ 
4   blockSize  $\leftarrow \max(1, \min(1024, 2^{\lceil \log_2 M_{\text{stage}} \rceil - \lfloor \text{params}_1 \rfloor}))$ 
5   blockDim  $\leftarrow \{x = \text{blockSize}, y = 1, z = 1\}$ 
6   KernelDefault  $\langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle(L, x, \mathcal{S}, \text{stage})$ 
7 end
8 Function KernelDefault( $L, x, \mathcal{S}, \text{stage}$ ):
9    $t \leftarrow \text{threadIdx}.x$ 
10   $T \leftarrow \text{blockDim}.x$ 
11  node  $\leftarrow S_{\text{stage}} + \text{blockIdx}.x$ 
12  for row  $\leftarrow N_{\text{node}}$  to  $N_{\text{node}+1} - 1$  do
13    cache[ $t$ ]  $\leftarrow 0$ 
14     $i \leftarrow R_{\text{row}}$ 
15    for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
16       $j \leftarrow L_p^{(\text{col})}$ 
17      cache[ $t$ ]  $\leftarrow \text{cache}[t] - L_p^{(\text{val})} \times x_j$ 
18    end
19    __syncthreads()
20    Reduce(cache,  $t, T$ )
21    if  $t = 0$  then
22       $x_i \leftarrow x_i + \text{cache}[0]$ 
23    end
24    __syncthreads()
25  end
26 end

```

---

computation of different unknowns. The latter option was chosen because it offers more flexibility for the scheduler. Algorithms 9 and 10 illustrate how the method was implemented.

At each stage, there are two possibilities, either the **default** strategy is used or the **KernelMultiblockNode** kernel is launched a certain amount of times. This new kernel is launched with a two-dimensional grid where the  $x$ -dimension represents the node, and multiple blocks assigned to the same node are distributed along the  $y$ -dimension. For the multiblock-per-node strategy to be chosen, two conditions have to be satis-



fied. First, we check that  $M_{\text{stage}}$  is greater than some threshold  $1024 \times \text{params}_2$ , where  $\text{params}_2$  is a smart parameter greater than or equal to one. Second, the chosen number of blocks per node needs to be strictly greater than one, otherwise the **default** method would be better suited. The chosen number of blocks per node is stored in the **numNodeBlocks** variable. It is defined as the minimum of the largest value for which the total number of thread blocks is less than or equal to **maxGridSize** and the smallest value for which the total number of threads assigned to a node is more than  $M_{\text{stage}}$ . The **idealGridSize** obtained with the function **GetIdealKernelConfig** is the minimum grid size which achieves maximum occupancy, as discussed in Section 2.2.1. In practice, it has proven beneficial to launch kernels with larger grid sizes. Therefore, a variable  $\text{maxGridSize} = \text{idealGridSize} \times \text{params}_1$  was defined. The  $\text{params}_1$  parameter is a smart parameter greater than or equal to one.

If the multiblock-per-node strategy is chosen, the kernel needs to be launched a certain number of times, given by  $P_{\text{stage}}$ . The vector  $P \in \mathbb{N}^s$ , where  $s$  is the number of stages, is a new addition to the scheduler. It contains the maximum number of unknowns among the nodes in each stage. Each kernel launch will compute one unknown in each node, and to inform the kernel which unknown it needs to compute, a **step** variable is passed as an argument to indicate the index of the unknown to compute in the current node. Since the number of unknowns per node is not constant, some nodes will terminate before others. It is thus necessary to check in the kernel whether the current node has not already completed its computations. This is what line 9 does. The rest of the kernel is similar to the one described in the **sequential-multiblock** method.

### Cooperative Groups

In CUDA 9, new synchronization mechanisms were introduced with the Cooperative Groups API. This API allows to synchronize multiple thread blocks without having to start a new kernel. However, this results in more limited kernel configurations. Either way, an equivalent of the **default** method was implemented using only one kernel launch with cooperative groups. This method will be referred to as **cooperative-groups**.

---

**Algorithm 9:** Solution Step CPU: adaptative

---

**Input:**  $\mathcal{S} := (S, N, R, M, P)$   
 $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $b \in \mathbb{C}^n$   
 $\text{params} \in \mathbb{R}^3$

**Output:**  $x \in \mathbb{C}^n$

```

1  $x \leftarrow b$ 
2  $(\text{idealGridSize}, \text{idealBlockSize}) \leftarrow \text{GetIdealKernelConfig}(\text{KernelMultiblockNode})$ 
3  $\text{maxGridSize} \leftarrow \text{idealGridSize} \times \text{params}_1$ 
4 for stage  $\leftarrow 1$  to  $\dim(S) - 1$  do
5    $\text{numNodes} \leftarrow S_{\text{stage}+1} - S_{\text{stage}}$ 
6    $\text{numNodeBlocks} \leftarrow \min(\lfloor \text{maxGridSize}/\text{numNodes} \rfloor, \lceil M_{\text{stage}}/\text{idealBlockSize} \rceil)$ 
7   if  $M_{\text{stage}} > 1024 \times \text{params}_2$  and  $\text{numNodeBlocks} \geq 2$  then
8      $\text{gridDim} \leftarrow \{x = \text{numNodes}, y = \text{numNodeBlocks}, z = 1\}$ 
9      $\text{blockDim} \leftarrow \{x = \text{idealBlockSize}, y = 1, z = 1\}$ 
10    for step  $\leftarrow 0$  to  $P_{\text{stage}} - 1$  do
11       $\text{KernelMultiblockNode} \langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle(L,$ 
12         $x, \mathcal{S}, \text{stage}, \text{step})$ 
13    end
14  else
15     $\text{blockSize} \leftarrow \max(1, \min(1024, 2^{\lceil \log_2 M_{\text{stage}} \rceil - \lfloor \text{params}_3 \rfloor}))$ 
16     $\text{blockDim} \leftarrow \{x = \text{blockSize}, y = 1, z = 1\}$ 
17     $\text{gridDim} \leftarrow \{x = \text{numNodes}, y = 1, z = 1\}$ 
18     $\text{KernelDefault} \langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle(L, x, \mathcal{S}, \text{stage})$ 
19  end
20 end

```

---

---

**Algorithm 10:** Solution Step Kernel: adaptative
 

---

```

1 Function KernelMultiblockNode(stage, step):
2   cacheldx  $\leftarrow$  threadIdx.x
3   cacheSize  $\leftarrow$  blockDim.x
4    $t \leftarrow$  threadIdx.x + blockDim.x  $\times$  blockIdx.y
5    $T \leftarrow$  blockDim.x  $\times$  gridDim.y
6   cache[cacheSize]  $\leftarrow$  0
7   node  $\leftarrow$   $S_{\text{stage}}$  + blockIdx.x
8   row  $\leftarrow$   $N_{\text{node}}$  + step
9   if row  $\geq N_{\text{node}+1}$  then
10    | return
11  end
12   $i \leftarrow R_{\text{row}}$ 
13  for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
14    |  $j \leftarrow L_p^{(\text{col})}$ 
15    | cache[cacheldx]  $\leftarrow$  cache[cacheldx] -  $L_p^{(\text{val})} \times x_j$ 
16  end
17  __syncthreads()
18  Reduce(cache, cacheldx, cacheSize)
19  if cacheldx = 0 then
20    | AtomicAdd( $x_i$ , cache[0])
21  end
22 end

```

---

## 2.3 Solution Step with Multiple Right-Hand Sides

In this section, we will review multiple strategies to solve the system:  $LX = B$ , where  $L \in \mathbb{C}^{n \times n}$  is still a triangular matrix,  $X \in \mathbb{C}^{n \times m}$  and  $B \in \mathbb{C}^{n \times m}$ . The  $X$  and  $B$  matrices are stored in a dense matrix format as one-dimensional array. Each method will come in two variants: one where the two dense matrices are stored in column-major order, and one where they are stored in row-major order (see Section 2.2.3). Column-major order is the most natural order because each right-hand side is stored in one contiguous block as for the single right-hand side case.

### 2.3.1 Naive Method

We can easily extend the `default` method to multiple right-hand sides by launching a kernel with a two-dimensional grid instead of a one-dimensional one. The number of blocks in the  $y$ -dimension would be the number of right-hand sides  $m$ . Algorithm 11 shows how the algorithm could be implemented. This approach is called the **naive** method.

**Algorithm 11:** Solution Step: naive

---

**Input:**  $\mathcal{S} := (S, N, R, M)$   
 $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $B \in \mathbb{C}^{n \times m}$   
 $\text{params} \in \mathbb{R}^1$

**Output:**  $X \in \mathbb{C}^{n \times m}$

```

1  $X \leftarrow B$ 
2 for stage  $\leftarrow 1$  to  $\dim(S) - 1$  do
3    $\text{gridDim} \leftarrow \{x = S_{\text{stage}+1} - S_{\text{stage}}, y = m, z = 1\}$ 
4    $\text{blockSize} \leftarrow \max(1, \min(1024, 2^{\lceil \log_2 M_{\text{stage}} \rceil - \lfloor \text{params}_1 \rfloor}))$ 
5    $\text{blockDim} \leftarrow \{x = \text{blockSize}, y = 1, z = 1\}$ 
6    $\text{KernelMultiRHSNaive} \langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle (L, X, \mathcal{S},$ 
    $\text{stage})$ 
7 end
8 Function  $\text{KernelMultiRHSNaive}(L, X, \mathcal{S}, \text{stage})$ :
9    $t \leftarrow \text{threadIdx}.x$ 
10   $T \leftarrow \text{blockDim}.x$ 
11   $r \leftarrow \text{blockIdx}.y + 1$ 
12   $\text{node} \leftarrow S_{\text{stage}} + \text{blockIdx}.x$ 
13  for row  $\leftarrow N_{\text{node}}$  to  $N_{\text{node}+1} - 1$  do
14     $\text{cache}[t] \leftarrow 0$ 
15     $i \leftarrow R_{\text{row}}$ 
16    for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
17       $j \leftarrow L_p^{(\text{col})}$ 
18       $\text{cache}[t] \leftarrow \text{cache}[t] - L_p^{(\text{val})} \times X_{jr}$ 
19    end
20     $\_\_\text{syncthreads}()$ 
21     $\text{Reduce}(\text{cache}, t, T)$ 
22    if  $t = 0$  then
23       $X_{ir} \leftarrow X_{ir} + \text{cache}[0]$ 
24    end
25     $\_\_\text{syncthreads}()$ 
26  end
27 end

```

---

**2.3.2 Block Methods**

When the number of right-hand sides (RHS)  $m$  is large, the number of threads launched in the naive method can exceed the parallel processing capabilities of the GPU. To avoid this, new methods have been developed, where the number of blocks along the  $y$ -dimension of the grid can be smaller than  $m$ . These methods are called block methods

because instead of assigning one right-hand side to each thread block, we assign one block of right-hand sides (RHS block) to each thread block. By "block of right-hand sides", we mean a group of one or more right-hand sides; this should not be confused with a thread block. The number of RHS blocks at each stage is the minimum of  $m$  and the largest value for which the total number of blocks in the grid is less than or equal to `maxGridSize`. The `maxGridSize` variable is defined in the same way as for the `adaptive` method.

One advantage of grouping right-hand sides together is that we can choose to group those who are close in memory. This allows to increase the efficiency of the cache. To do so, the RHS blocks are defined as ranges of right-hand sides which are determined with the task distribution function  $d$  seen in Section 2.2.4.

Three block methods are proposed: **block-outer**, which can be seen in Algorithm 12; **block-middle**, which can be seen in Algorithm 13; and **block-inner**, which is shown in Algorithm 14 and Algorithm 15. The main difference between these methods is the location of the loop iterating over the right-hand sides. In the **naive** method, the kernel has two nested loops: the outer one iterating over the unknowns that need to be computed, and the inner one iterating over a portion of the nonzero elements of  $L$  in the row corresponding to the unknown being computed. In all of the block methods, there are three nested loops because there is an additional one to iterate over the right-hand sides in the RHS block. This loop can be either the outer loop (**block-outer**), the middle one (**block-middle**), or the inner one (**block-inner**).

The **block-inner** method is slightly more complex compared to the other two. In the other kernels, the shared memory was allocated such that each individual thread could own one 16-byte value representing a complex double precision number. Here, each thread needs as many 16-byte values as there are right-hand sides in its assigned RHS block. This is so because, contrary to the two previous kernels where we compute one segment at a time, here we compute concurrently multiple segments, one for each right-hand side in the RHS block. The outer loop iterates over the unknowns to be computed in the current node. At each iteration, one must first set to 0 the temporary values stored in the shared memory array `cache`. Each thread  $t$ , is responsible for initializing the values it owns, i.e., the `cache` array values at indices  $t + kT$ , with  $k \in \{0, 1, \dots, \text{rhsBlockSize} - 1\}$ . The loop performing this initialization starts at line 11. Once this is done, one can load  $R_{\text{row}}$  which contains the index of the unknown which needs to be computed. This value only needs to be loaded once for all the right-hand sides in the RHS block. In the two previous methods, this value had to be reloaded multiple times from global memory. Next, the middle loop starts at line 16, and at each iteration it load the column index  $L_p^{(\text{col})}$  from global memory. Once again, this can be loaded once in a register and be used for all the right-hand sides in the RHS block. The inner loop then starts and iterates over the right-hand sides in the RHS block. Once the computations in the inner loop are completed, the **Reduce** function is called once for each right-hand side in the RHS block. Note that, the `cacheldx` variable was always chosen such that the  $T$  segments that need to be summed together for the computation of each unknown are stored contiguously in shared memory. Finally, the

last step consists in updating the  $X$  matrix with the newly computed unknowns. In the previous methods, it was thread 0 which was responsible for copying `cache[0]` to  $X_{ir}$ . This time, since there are multiple values to update in  $X$  (one for each right-hand side in the RHS block), each thread  $t$  in the thread block will be responsible for copying the elements `cache[kT]` to  $X_{ir}$  for each  $k \in \{t + lT \mid l \in \mathbb{N} \text{ and } t + lT < \text{rhsBlockSize}\}$ , with  $r = k + \text{rhsStart}$ . As multiple threads use the results of the reductions, it is necessary to synchronize all threads after them.

Overall, the **block-inner** method should be the most efficient among the block methods. The only disadvantage of this method is that it requires more shared memory per thread which can reduce the maximum potential thread block size.

**Algorithm 12:** Solution Step: block-outer

---

**Input:**  $\mathcal{S} := (S, N, R, M)$   
 $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $B \in \mathbb{C}^{n \times m}$   
 $\text{params} \in \mathbb{R}^1$

**Output:**  $X \in \mathbb{C}^{n \times m}$

```

1  $X \leftarrow B$ 
2  $(\text{idealGridSize}, \text{idealBlockSize}) \leftarrow$ 
    $\text{GetIdealKernelConfig}(\text{KernelMultiRHSBlockOuter})$ 
3  $\text{maxGridSize} \leftarrow \text{idealGridSize} \times \text{params}_1$ 
4 for stage  $\leftarrow 1$  to  $\dim(S) - 1$  do
5    $\text{numNodes} \leftarrow S_{\text{stage}+1} - S_{\text{stage}}$ 
6    $\text{numRHSBlocks} \leftarrow \max(1, \min(m, \lfloor \text{maxGridSize}/\text{numNodes} \rfloor))$ 
7    $\text{blockSize} \leftarrow \min(\text{idealBlockSize}, 2^{\lceil \log_2 M_{\text{stage}} \rceil})$ 
8    $\text{gridDim} \leftarrow \{x = \text{numNodes}, y = \text{numRHSBlocks}, z = 1\}$ 
9    $\text{blockDim} \leftarrow \{x = \text{blockSize}, y = 1, z = 1\}$ 
10   $\text{KernelMultiRHSBlockOuter} \langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle(L, X,$ 
     $\mathcal{S}, \text{stage})$ 
11 end
12 Function  $\text{KernelMultiRHSBlockOuter}(L, X, \mathcal{S}, \text{stage})$ :
13    $t \leftarrow \text{threadIdx}.x$ 
14    $T \leftarrow \text{blockDim}.x$ 
15    $\text{rhsBlock} \leftarrow \text{blockIdx}.y$ 
16    $\text{numRHSBlocks} \leftarrow \text{gridDim}.y$ 
17    $(\text{rhsStart}, \text{rhsStop}) \leftarrow d(\text{rhsBlock}, m, \text{numRHSBlocks}) + (1, 1)$ 
18    $\text{node} \leftarrow S_{\text{stage}} + \text{blockIdx}.x$ 
19   for  $r \leftarrow \text{rhsStart}$  to  $\text{rhsStop} - 1$  do
20     for  $\text{row} \leftarrow N_{\text{node}}$  to  $N_{\text{node}+1} - 1$  do
21        $\text{cache}[t] \leftarrow 0$ 
22        $i \leftarrow R_{\text{row}}$ 
23       for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
24          $j \leftarrow L_p^{(\text{col})}$ 
25          $\text{cache}[t] \leftarrow \text{cache}[t] - L_p^{(\text{val})} \times X_{jr}$ 
26       end
27        $\_\_\text{syncthreads}()$ 
28        $\text{Reduce}(\text{cache}, t, T)$ 
29       if  $t = 0$  then
30          $X_{ir} \leftarrow X_{ir} + \text{cache}[0]$ 
31       end
32        $\_\_\text{syncthreads}()$ 
33     end
34   end
35 end

```

---



**Algorithm 13:** Solution Step: block-middle

---

**Input:**  $\mathcal{S} := (S, N, R, M)$   
 $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $B \in \mathbb{C}^{n \times m}$   
 $\text{params} \in \mathbb{R}^1$

**Output:**  $X \in \mathbb{C}^{n \times m}$

```

1  $X \leftarrow B$ 
2  $(\text{idealGridSize}, \text{idealBlockSize}) \leftarrow$ 
    $\text{GetIdealKernelConfig}(\text{KernelMultiRHSBlockMiddle})$ 
3  $\text{maxGridSize} \leftarrow \text{idealGridSize} \times \text{params}_1$ 
4 for stage  $\leftarrow 1$  to  $\dim(S) - 1$  do
5    $\text{numNodes} \leftarrow S_{\text{stage}+1} - S_{\text{stage}}$ 
6    $\text{numRHSBlocks} \leftarrow \max(1, \min(m, \lfloor \text{maxGridSize}/\text{numNodes} \rfloor))$ 
7    $\text{blockSize} \leftarrow \min(\text{idealBlockSize}, 2^{\lceil \log_2 M_{\text{stage}} \rceil})$ 
8    $\text{gridDim} \leftarrow \{x = \text{numNodes}, y = \text{numRHSBlocks}, z = 1\}$ 
9    $\text{blockDim} \leftarrow \{x = \text{blockSize}, y = 1, z = 1\}$ 
10   $\text{KernelMultiRHSBlockMiddle} \langle \text{gridDim}, \text{blockDim}, 16 \times \text{blockDim}.x \rangle(L,$ 
     $X, \mathcal{S}, \text{stage})$ 
11 end
12 Function  $\text{KernelMultiRHSBlockMiddle}(L, X, \mathcal{S}, \text{stage})$ :
13    $t \leftarrow \text{threadIdx}.x$ 
14    $T \leftarrow \text{blockDim}.x$ 
15    $\text{rhsBlock} \leftarrow \text{blockIdx}.y$ 
16    $\text{numRHSBlocks} \leftarrow \text{gridDim}.y$ 
17    $(\text{rhsStart}, \text{rhsStop}) \leftarrow d(\text{rhsBlock}, m, \text{numRHSBlocks}) + (1, 1)$ 
18    $\text{node} \leftarrow S_{\text{stage}} + \text{blockIdx}.x$ 
19   for row  $\leftarrow N_{\text{node}}$  to  $N_{\text{node}+1} - 1$  do
20      $i \leftarrow R_{\text{row}}$ 
21     for  $r \leftarrow \text{rhsStart}$  to  $\text{rhsStop} - 1$  do
22        $\text{cache}[t] \leftarrow 0$ 
23       for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
24          $j \leftarrow L_p^{(\text{col})}$ 
25          $\text{cache}[t] \leftarrow \text{cache}[t] - L_p^{(\text{val})} \times X_{jr}$ 
26       end
27        $\_\_\text{syncthreads}()$ 
28        $\text{Reduce}(\text{cache}, t, T)$ 
29       if  $t = 0$  then
30          $X_{ir} \leftarrow X_{ir} + \text{cache}[0]$ 
31       end
32        $\_\_\text{syncthreads}()$ 
33     end
34   end
35 end

```

---

---

**Algorithm 14:** Solution Step CPU: block-inner

---

**Input:**  $\mathcal{S} := (S, N, R, M)$  $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$  $B \in \mathbb{C}^{n \times m}$ params  $\in \mathbb{R}^1$ **Output:**  $X \in \mathbb{C}^{n \times m}$ 

```

1  $X \leftarrow B$ 
2 (idealGridSize, idealBlockSize)  $\leftarrow$ 
   GetIdealKernelConfig(KernelMultiRHSBlockInner)
3 maxGridSize  $\leftarrow$  idealGridSize  $\times$  params1
4 for stage  $\leftarrow 1$  to dim( $S$ )  $- 1$  do
5   numNodes  $\leftarrow S_{\text{stage}+1} - S_{\text{stage}}$ 
6   numRHSBlocks  $\leftarrow \max(1, \min(m, \lfloor \text{maxGridSize}/\text{numNodes} \rfloor))$ 
7   blockSize  $\leftarrow \min(\text{idealBlockSize}, 2^{\lceil \log_2 M_{\text{stage}} \rceil})$ 
8   blockDim  $\leftarrow \{x = \text{blockSize}, y = 1, z = 1\}$ 
9   maxRHSBlockSize  $\leftarrow \lceil m/\text{numRHSBlocks} \rceil$ 
10  sharedMem  $\leftarrow 16 \times \text{blockDim}.x \times \text{maxRHSBlockSize}$ 
11  maxSharedMem  $\leftarrow$  get maximum amount of shared memory per thread
    block
12  if sharedMem  $>$  maxSharedMem then
13    maxRHSBlockSize  $\leftarrow \max(1, \lfloor \text{maxSharedMem}/(16 \times \text{blockDim}.x) \rfloor)$ 
14    numRHSBlocks  $\leftarrow \lceil m/\text{maxRHSBlockSize} \rceil$ 
15    sharedMem  $\leftarrow 16 \times \text{blockDim}.x \times \text{maxRHSBlockSize}$ 
16  end
17  gridDim  $\leftarrow \{x = \text{numNodes}, y = \text{numRHSBlocks}, z = 1\}$ 
18  KernelMultiRHSBlockInner  $\langle \text{gridDim}, \text{blockDim}, \text{sharedMem} \rangle(L, X, \mathcal{S},$ 
    stage)
19 end

```

---

---

**Algorithm 15:** Solution Step Kernel: block-inner

---

```

1 Function KernelMultiRHSBlockInner( $L, X, \mathcal{S}, \text{stage}$ ):
2    $t \leftarrow \text{threadIdx}.x$ 
3    $T \leftarrow \text{blockDim}.x$ 
4    $\text{rhsBlock} \leftarrow \text{blockIdx}.y$ 
5    $\text{numRHSBlocks} \leftarrow \text{gridDim}.y$ 
6    $(\text{rhsStart}, \text{rhsStop}) \leftarrow d(\text{rhsBlock}, m, \text{numRHSBlocks}) + (1, 1)$ 
7    $\text{rhsBlockSize} \leftarrow \text{rhsStop} - \text{rhsStart}$ 
8    $\text{cacheSize} \leftarrow \text{rhsBlockSize} \times T$ 
9    $\text{node} \leftarrow S_{\text{stage}} + \text{blockIdx}.x$ 
10  for  $\text{row} \leftarrow N_{\text{node}}$  to  $N_{\text{node}+1} - 1$  do
11    for  $k \leftarrow 0$  to  $\text{rhsBlockSize} - 1$  do
12       $\text{cacheldx} \leftarrow t + kT$ 
13       $\text{cache}[\text{cacheldx}] \leftarrow 0$ 
14    end
15     $i \leftarrow R_{\text{row}}$ 
16    for  $p \leftarrow t + L_i^{(\text{row})}$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
17       $j \leftarrow L_p^{(\text{col})}$ 
18      for  $k \leftarrow 0$  to  $\text{rhsBlockSize} - 1$  do
19         $\text{cacheldx} \leftarrow t + kT$ 
20         $r \leftarrow k + \text{rhsStart}$ 
21         $\text{cache}[\text{cacheldx}] \leftarrow \text{cache}[\text{cacheldx}] - L_p^{(\text{val})} \times X_{jr}$ 
22      end
23    end
24     $\_\_\text{syncthreads}()$ 
25    for  $k \leftarrow 0$  to  $\text{rhsBlockSize} - 1$  do
26       $\text{Reduce}(\&\text{cache}[kT], t, T)$ 
27    end
28     $\_\_\text{syncthreads}()$ 
29    for  $k \leftarrow t$  to  $\text{rhsBlockSize} - 1$  step  $T$  do
30       $\text{cacheldx} \leftarrow kT$ 
31       $r \leftarrow k + \text{rhsStart}$ 
32       $X_{ir} \leftarrow X_{ir} + \text{cache}[\text{cacheldx}]$ 
33    end
34     $\_\_\text{syncthreads}()$ 
35  end
36 end

```

---

### 2.3.3 Flipped Methods

The last methods that will be discussed can only be used with schedulers that have unit-size nodes, such as those built using the `level-set` strategy. The idea is to shift from a block-based right-hand side distribution to a thread-based one and from a thread-based segment distribution to a block-based one. We essentially flip the parallelization layout of the right-hand sides and the segments. By doing so, consecutive threads will be responsible for consecutive right-hand sides. If the matrix  $X$  is stored in row-major order, it means that consecutive threads will load consecutive memory locations which will make the global memory accesses coalesced. This fact is the main motivation for these methods.

The first method is simply called `flipped` and it can be seen in Algorithm 16. Since each corresponding to a certain unknown  $X_{ir}$  is computed in a different thread block, we cannot use the block-level `Reduce` function to sum all the segments. We thus have to resort back to the `AtomicAdd` function.

If the number of right-hand sides is smaller than the ideal block size, it would be better to compute several segments within the same thread block.

The disadvantage of this method is that when  $m$  is smaller than the ideal block size, the occupancy of the kernel is not maxed out. We could increase the size of each thread block by assigning more than one segment per thread block such that the total block size is as close as possible to the ideal one. An other improvement would be to avoid having the total grid size exceed `maxGridSize` by grouping nodes in the same thread block when the total number of nodes exceeds `maxGridSize`. This is the idea of the `flipped-enhanced` method which is shown in Algorithms 17 and 18.

---

**Algorithm 16:** Solution Step: flipped

---

**Input:**  $\mathcal{S} := (S, N, R, M)$   
 $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $B \in \mathbb{C}^{n \times m}$   
 $\text{params} \in \mathbb{R}^1$   
**Output:**  $X \in \mathbb{C}^{n \times m}$

```

1  $X \leftarrow B$ 
2  $(\text{idealGridSize}, \text{idealBlockSize}) \leftarrow \text{GetIdealKernelConfig}(\text{KernelMultiRHSFlipped})$ 
3  $\text{maxGridSize} \leftarrow \text{idealGridSize} \times \text{params}_1$ 
4 for stage  $\leftarrow 1$  to  $\dim(S) - 1$  do
5    $\text{numNodes} \leftarrow S_{\text{stage}+1} - S_{\text{stage}}$ 
6    $\text{numSegments} \leftarrow \max(1, \min(\lfloor \text{maxGridSize} / \text{numNodes} \rfloor, M_{\text{stage}}))$ 
7    $\text{blockSize} \leftarrow \min(m, \text{idealBlockSize})$ 
8    $\text{gridDim} \leftarrow \{x = \text{numSegments}, y = \text{numNodes}, z = \lceil m / \text{blockSize} \rceil\}$ 
9    $\text{blockDim} \leftarrow \{x = \text{blockSize}, y = 1, z = 1\}$ 
10   $\text{KernelMultiRHSFlipped} \langle \text{gridDim}, \text{blockDim}, 16 \times m \rangle (L, X, \mathcal{S}, \text{stage}, m)$ 
11 end
12 Function  $\text{KernelMultiRHSFlipped}(L, X, \mathcal{S}, \text{stage}, m)$ :
13    $r \leftarrow \text{threadIdx}.x + \text{blockDim}.x \times \text{blockIdx}.z$ 
14   if  $r \geq m$  then
15     return
16   end
17    $t \leftarrow \text{blockIdx}.x$ 
18    $T \leftarrow \text{gridDim}.x$ 
19    $\text{cacheldx} \leftarrow \text{threadIdx}.x$ 
20    $\text{node} \leftarrow S_{\text{stage}} + \text{blockIdx}.y$ 
21    $\text{row} \leftarrow N_{\text{node}}$ 
22    $i \leftarrow R_{\text{row}}$ 
23    $\text{cache}[\text{cacheldx}] \leftarrow 0$ 
24   for  $p \leftarrow L_i^{(\text{row})} + t$  to  $L_{i+1}^{(\text{row})} - 2$  step  $T$  do
25      $j \leftarrow L_p^{(\text{col})}$ 
26      $\text{cache}[\text{cacheldx}] \leftarrow \text{cache}[\text{cacheldx}] - L_p^{(\text{val})} \times X_{jr}$ 
27   end
28    $\text{AtomicAdd}(X_{ir}, \text{cache}[\text{cacheldx}])$ 
29 end

```

---

---

**Algorithm 17:** Solution Step CPU: flipped-enhanced

---

**Input:**  $\mathcal{S} := (S, N, R, M)$   
 $L := (L^{(\text{row})}, L^{(\text{col})}, L^{(\text{val})})$   
 $B \in \mathbb{C}^{n \times m}$   
 $\text{params} \in \mathbb{R}^1$   
**Output:**  $X \in \mathbb{C}^{n \times m}$

```

1   $X \leftarrow B$ 
2   $(\text{idealGridSize}, \text{idealBlockSize}) \leftarrow$ 
    $\text{GetIdealKernelConfig}(\text{KernelMultiRHSFlippedEnhanced})$ 
3   $\text{maxGridSize} \leftarrow \text{idealGridSize} \times \text{params}_1$ 
4   $\text{numRHSThreads} \leftarrow \min(m, \text{idealBlockSize})$ 
5   $\text{numRHSBlocks} \leftarrow \lceil m / \text{numRHSThreads} \rceil$ 
6  for stage  $\leftarrow 1$  to  $\dim(S) - 1$  do
7     $\text{numNodes} \leftarrow S_{\text{stage}+1} - S_{\text{stage}}$ 
8     $\text{numNodeBlocks} \leftarrow \min(\text{numNodes}, \lfloor \text{maxGridSize} / \text{numRHSBlocks} \rfloor)$ 
9     $\text{numSegmentThreads} \leftarrow \min(\lfloor \text{idealBlockSize} / \text{numRHSThreads} \rfloor, M_{\text{stage}})$ 
10    $\text{numSegmentThreads} \leftarrow 2^{\lfloor \log_2 \text{numSegmentThreads} \rfloor}$ 
11    $\text{numSegmentBlocks} \leftarrow$ 
     $\max(1, \min(\lfloor M_{\text{stage}} / \text{numSegmentThreads} \rfloor, \lfloor \text{maxGridSize} / (\text{numNodeBlocks} \times$ 
     $\text{numRHSBlocks} \rfloor)))$ 
12    $\text{gridDim} \leftarrow \{x = \text{numNodeBlocks}, y = \text{numSegmentBlocks}, z =$ 
     $\text{numRHSBlocks}\}$ 
13    $\text{blockDim} \leftarrow \{x = \text{numRHSThreads}, y = \text{numSegmentThreads}, z = 1\}$ 
14    $\text{KernelMultiRHSFlippedEnhanced} \langle \text{gridDim}, \text{blockDim},$ 
     $16 \times \text{numRHSThreads} \times \text{numSegmentThreads} \rangle (L, X, \mathcal{S}, \text{stage}, \text{numNodes},$ 
     $m)$ 
15 end

```

---

---

**Algorithm 18:** Solution Step Kernel: flipped-enhanced

---

```

1 Function KernelMultiRHSFlippedEnhanced( $L, X, S, \text{stage}, \text{numNodes}, m$ ):
2    $r \leftarrow \text{threadIdx}.x + \text{blockDim}.x \times \text{blockIdx}.z$ 
3   if  $r \geq m$  then
4     return
5   end
6    $\text{thread} \leftarrow \text{threadIdx}.x + \text{threadIdx}.y \times \text{blockDim}.x$ 
7    $\text{cacheldx} \leftarrow \text{threadIdx}.y + \text{threadIdx}.x \times \text{blockDim}.y$ 
8    $\text{segment} \leftarrow \text{threadIdx}.y + \text{blockIdx}.y \times \text{blockDim}.y$ 
9    $\text{numSegments} \leftarrow \text{gridDim}.y \times \text{blockDim}.y$ 
10   $\text{nodeBlock} \leftarrow \text{blockIdx}.x$ 
11   $\text{numNodeBlocks} \leftarrow \text{gridDim}.x$ 
12   $(\text{nodeStart}, \text{nodeStop}) \leftarrow$ 
     $d(\text{nodeBlock}, \text{numNodes}, \text{numNodeBlocks}) + (S_{\text{stage}}, S_{\text{stage}})$ 
13  for  $\text{node} \leftarrow \text{nodeStart}$  to  $\text{nodeStop} - 1$  do
14     $\text{row} \leftarrow N_{\text{node}}$ 
15     $i \leftarrow R_{\text{row}}$ 
16     $\text{cache}[\text{cacheldx}] \leftarrow 0$ 
17     $\text{segmentSize} \leftarrow L_{i+1}^{(\text{row})} - L_i^{(\text{row})} - 1$ 
18     $(\text{segmentStart}, \text{segmentStop}) \leftarrow$ 
       $d(\text{segment}, \text{segmentSize}, \text{numSegments}) + (L_i^{(\text{row})}, L_i^{(\text{row})})$ 
19    for  $p \leftarrow \text{segmentStart}$  to  $\text{segmentStop} - 1$  do
20       $j \leftarrow L_p^{(\text{col})}$ 
21       $\text{cache}[\text{cacheldx}] \leftarrow \text{cache}[\text{cacheldx}] - L_p^{(\text{val})} \times X_{jr}$ 
22    end
23     $\_\_\text{syncthreads}()$ 
24     $\text{Reduce}(\text{cache} + \lfloor \text{thread} / \text{blockDim}.y \rfloor \times \text{blockDim}.y, \text{thread}$ 
       $\text{mod } \text{blockDim}.y, \text{blockDim}.y)$ 
25     $\_\_\text{syncthreads}()$ 
26    if  $\text{threadIdx}.y = 0$  then
27       $\text{AtomicAdd}(X_{ir}, \text{cache}[\text{cacheldx}])$ 
28    end
29  end
30 end

```

---

# Chapter 3

## Numerical Experiments

### 3.1 Hardware Setup

All the numerical experiments were executed on the Lucia HPC facility operated by Cenaero. For the simple tests, only the GPU partition, which contains 50 nodes, was used. Table 3.1 shows the details of this partition. Each node contains 4 NVIDIA A100 40 GB GPUs and one AMD EPYC 7513 32-core CPU with 240 GB of user available memory. For the advanced tests with preconditioned iterative methods, we compared the results obtained on the GPU partition with those on the CPU (`batch`) partition, the details of which are listed in Table 3.2.

Node Details	GPU Node
Num. of nodes	50
Node model	HPE XL645d
Processors	1× AMD EPYC 7513 32-core
Processors freq.	2.6 GHz, boost up to 3.65 GHz
Processor L3 cache	128 MB
Cores per node	32
Hyperthreading	Disabled
Memory	256 GB DDR4-3200
User available mem.	240 GB
Accelerators	4× NVIDIA A100 40 GB
Ethernet	2× 10 Gbps
Fast interconnect	2× Infiniband HDR-200
Local disk	SATA SSD 480 GB

Table 3.1: Node details of the GPU partition of the Lucia cluster. [16]



Node Details	CPU Node
Num. of nodes	270
Node model	HPE XL225n
Processors	2× AMD EPYC 7763 64-core
Processors freq.	2.45 GHz, boost up to 3.5 GHz
Processor L3 cache	256 MB
Cores per node	128
Hyperthreading	Disabled
Memory	256 GB DDR4-3200
User available mem.	240 GB
Ethernet	2× 10 Gbps
Fast interconnect	2× Infiniband HDR-100
Local disk	SATA SSD 480 GB

Table 3.2: Node details of the CPU (`batch`) partition of the Lucia cluster. [16]

## 3.2 Direct Method

### 3.2.1 Toy Problem

Before testing the algorithms on iterative methods, we first solve a toy problem with a direct method. The problem is an inhomogenous Helmholtz equation defined over a cubic domain  $\Omega$  with Robin boundary conditions:

$$\begin{cases} \nabla^2 u + k^2 u = -f, & \text{in } \Omega, \\ \alpha u + \beta \frac{\partial u}{\partial n} = g, & \text{on } \partial\Omega, \end{cases} \quad (3.1)$$

where  $k = \pi$ ,  $g(\mathbf{x}) = 0$ ,  $\alpha = ik$ ,  $\beta = 1$  and  $f(\mathbf{x}) = 1$ . The problem is solved using the finite element method. For generating the mesh, the open source-software Gmsh [13] was used. The script used to generate this mesh is shown in Listing 3.1. The Gmsh-Fem library [23] was used to assemble the matrix  $A$  and the right-hand side  $b$ .

```

1 SetFactory("OpenCASCADE");
2 LC = 0.1;
3 Box(1) = {0,0,0, 1,1,1};
4 Point(10) = {0.5, 0.5, 0.5, LC};
5 Point{10} In Volume{1};
6 MeshSize{:} = LC;
7 Physical Volume(1) = {1};
8 Physical Surface(1) = {1:6};
9 Physical Point(1) = 10;
```

Listing 3.1: Gmsh script for generating a mesh of a unit cubic domain with specified mesh size and physical entities.

An  $LU$  decomposition of  $A$  is performed by the CPU using UMFPACK [9], which produces two triangular matrices:  $L$  and  $U$  such that  $A = LU$ .  $L$  is lower triangular and  $U$  is upper triangular. The system  $Ax = b$  can then be solved by first solving  $Ly = b$  and then  $Ux = y$ . The triangular solver will thus be used twice to solve the problem. It is important to note that UMFPACK is a sequential direct solver, which is not as advanced as solvers like MUMPS [2]. UMFPACK was chosen specifically because it allows the retrieval of the  $L$  and  $U$  factors after the factorization step, whereas MUMPS does not provide any routine for this.

In what follows, we will focus solely on the time required for the forward and backward substitution steps (including the preprocessing step). The time for mesh generation, system assembly and factorization are not considered, as these processes were not ported to the GPU.

### 3.2.2 Comparison of Preprocessing Strategies

The preprocessing step required to build the scheduler is not free. We first compare the time it takes to build the scheduler across different strategies. Tests were conducted with multiple system sizes. To achieve this, the mesh size factor was adjusted through the `-clscale` runtime argument when running the Gmsh script.

Figure 3.1 shows the time needed to build the scheduler for each strategy and for multiple system sizes. One can conclude that each method has a similar execution time.

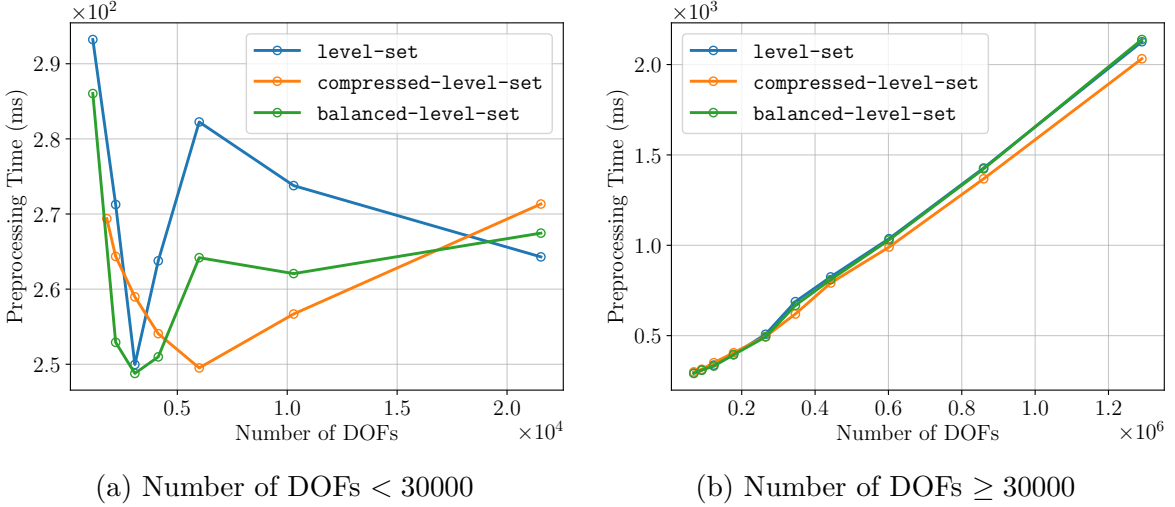


Figure 3.1: Preprocessing time for the  $L$  and  $U$  matrices for each strategy as a function of the number of DOFs in the system to solve.

It is necessary to assess the quality of the schedulers resulting from each scheduling strategy. Figure 3.2 shows the time it takes to solve the problem for each scheduler strategy. First, it is clear that using a scheduler is better than not. The **compressed-level-set** method is always significantly worse than the two other methods. The **balanced-level-set**

is always the best one with an average speedup of 1.87 over the `level-set` method. The `balanced-level-set` method should thus always be preferred over the others.

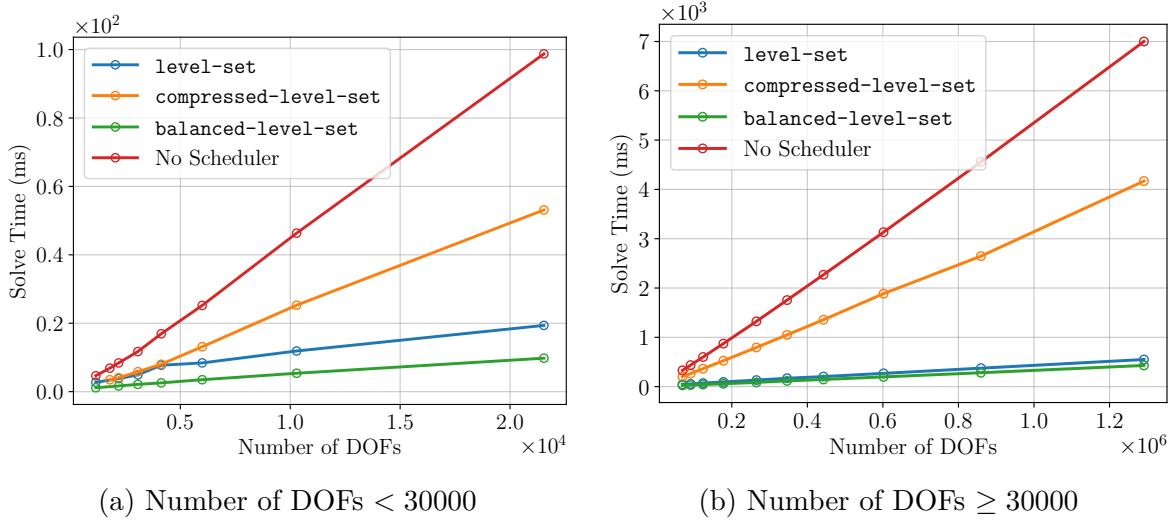
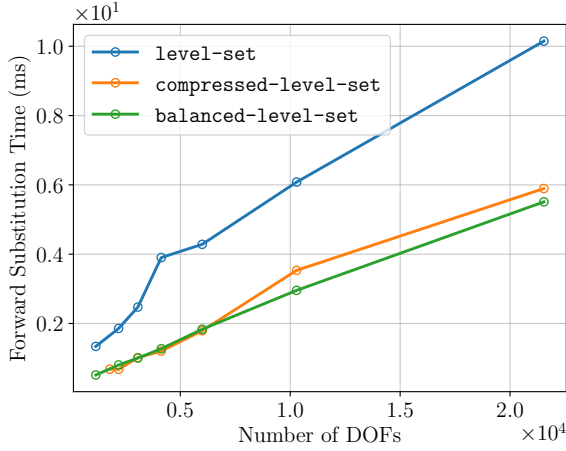
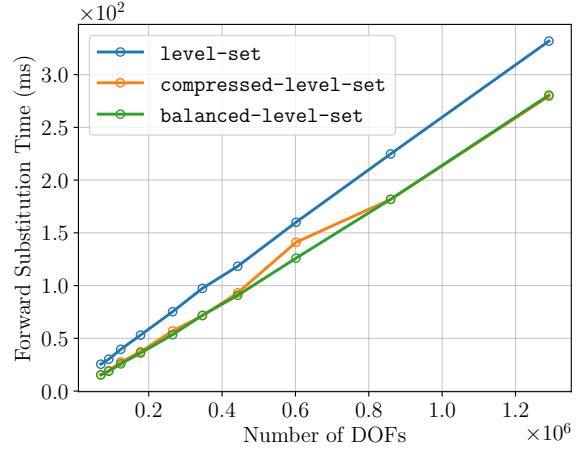
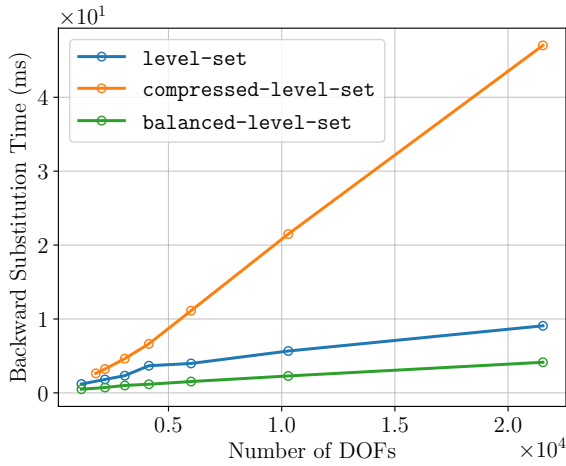


Figure 3.2: Solve time as a function of the number of DOFs for each scheduling strategy. The solution step is performed using the `default` method. The scheduler-free method uses the `sequential-singleblock` method for solving the problem.

One might wonder why the `compressed-level-set` method performs so much worse. Figure 3.3 shows how each part of the solve phase performs and one can notice that the `compressed-level-set` method is only consistently worse for the backward substitution. Figure 3.4 allows to understand why. The  $U$  matrix always contains a single row with a single nonzero element. This phenomenon occurs regardless of the system's size. This means that there is only one dependency-free row. As observed earlier, in this case the scheduler built by the `compressed-level-set` algorithm only contains one stage with a single node. This results in a serialized computation of unknowns. On the other hand, the `balanced-level-set` method works well for the forward substitution.



(a) Forward substitution with number of DOFs &lt; 30000

(b) Forward substitution with number of DOFs  $\geq 30000$ 

(c) Backward substitution with number of DOFs &lt; 30000

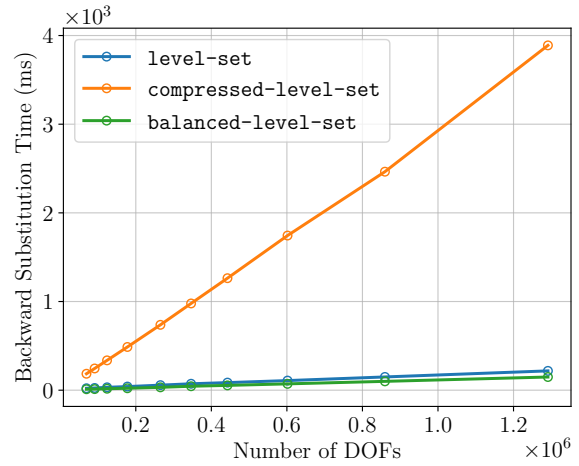
(d) Backward substitution with number of DOFs  $\geq 30000$ 

Figure 3.3: Forward substitution and backward substitution time as a function of the number of DOFs for each scheduling strategy. The solution step is performed using the `default` method. The scheduler-free method uses the `sequential-singleblock` method for solving the problem.

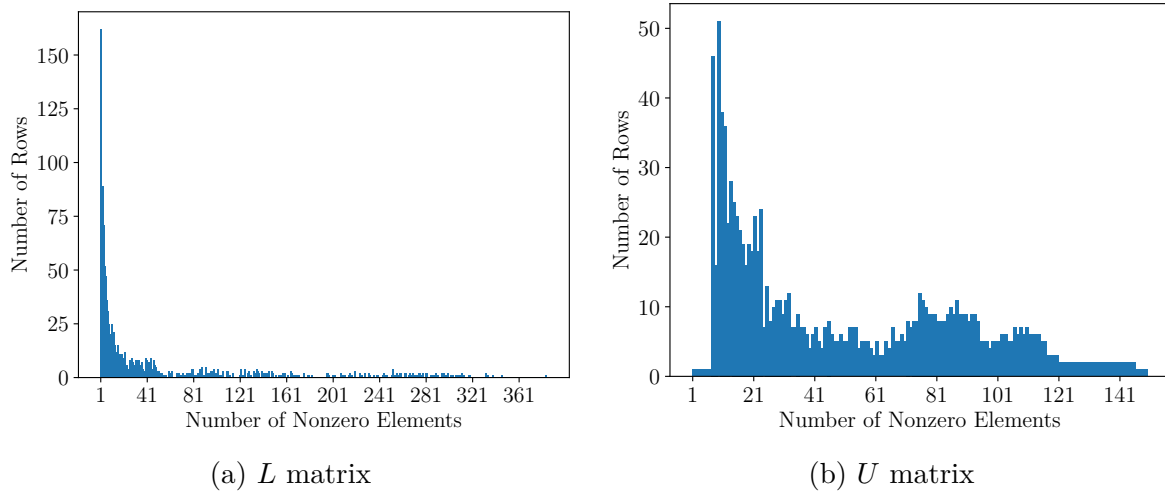


Figure 3.4: Bar plot representing the number of rows having a given number of nonzero elements for the  $L$  and  $U$  matrices corresponding to the problem with the most coarse mesh (`-clscale = 1`, i.e., the smallest system  $1156 \times 1156$ ).

### 3.2.3 Comparison of Solution Strategies

This section compares the different solution strategies seen in the previous chapter. These strategies are also compared to the cuSPARSE [22] sparse triangular solve routine. Figures 3.5 and 3.6 show how these strategies compare in terms of performance and solve time across multiple system sizes. It is clear that the **adaptative** strategy works the best and it scales better with the number of DOFs than the others. The two scheduler-free methods are by far the worst.

Note that the solve performance, measured in GFLOPS, is computed by dividing the fixed number of floating-point operations associated with solving the system using Equation 2.6 by the time it takes to solve the system. Although the actual number of FLOPs can vary depending on the solution strategy employed, it was decided to use a fixed number of FLOPs for the GFLOPS computation to ensure that the performance measurement is inversely proportional to the solve time. Plotting the GFLOPS instead of the solve time has the advantage of making the graphs more readable. The GFLOPS computation to solve the system is done as follows:

$$\text{GFLOPS} = (\mu + \alpha)(\text{nnz}(L) + \text{nnz}(U) - 2n), \quad (3.2)$$

where  $L \in \mathbb{C}^{n \times n}$ ,  $U \in \mathbb{C}^{n \times n}$ ,  $\alpha = 2$  is the number of FLOPs to perform a complex addition or subtraction, and  $\mu = 6$  is the number of FLOPs to perform a complex multiplication.

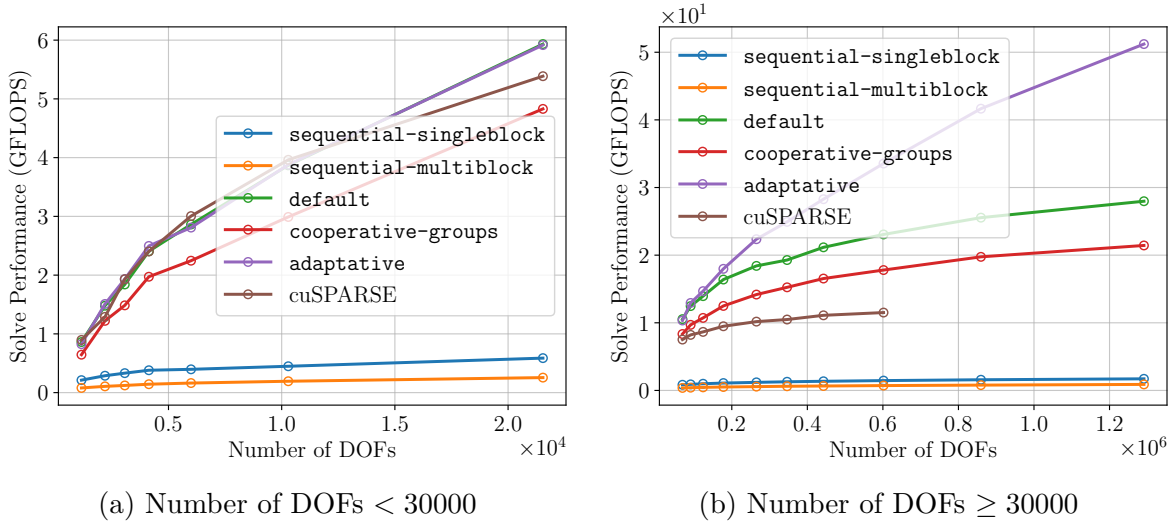
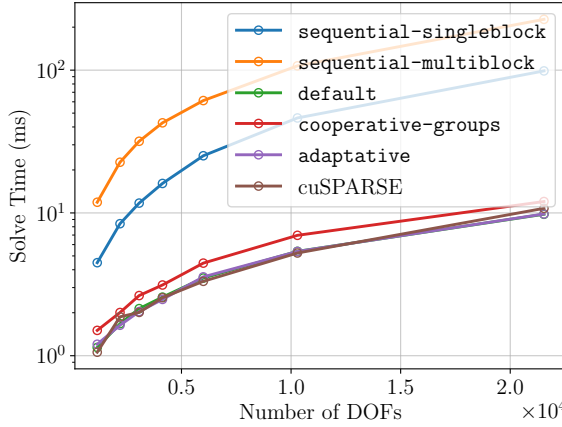


Figure 3.5: GPU solve performance as a function of the number of DOFs with several strategies, using the **balanced-level-set** method for constructing the scheduler when one is needed. The cuSPARSE strategy uses the `cusparseSpSV_solve` routine of the cuSPARSE library to perform the forward and backward substitution.

The **adaptative** method works best to solve the total problem which involves two triangular solves. However, as we have seen earlier, the structure of the two triangular



(a) Number of DOFs &lt; 30000

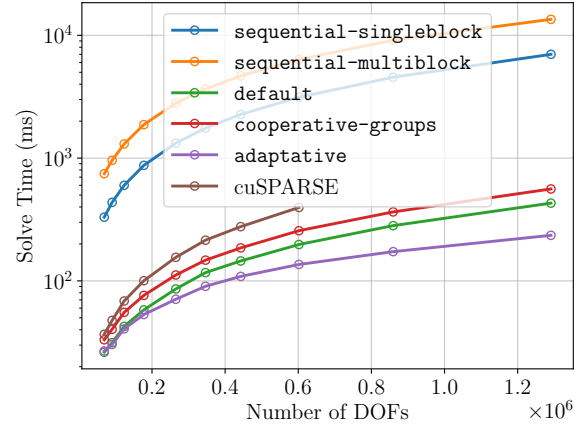
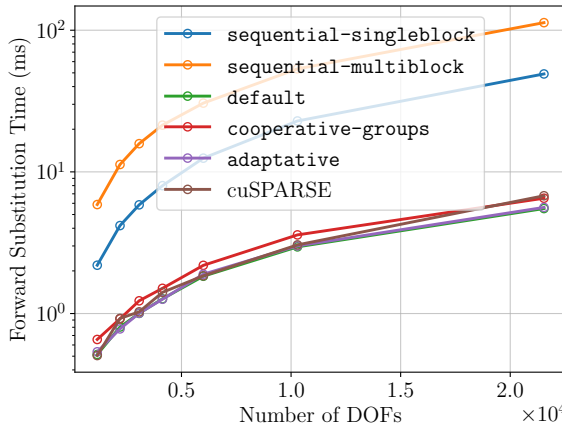
(b) Number of DOFs  $\geq 30000$ 

Figure 3.6: GPU solve time as a function of the number of DOFs with several strategies, using the **balanced-level-set** method for constructing the scheduler when one is needed.

matrices ( $L$  and  $U$ ) differ a lot. It is thus advisable to check if the **adaptative** method is the best for both the forward and the backward substitution. This is confirmed by Figure 3.7 and Figure 3.8 which show that the **adaptative** method is the best in both cases especially for large systems.



(a) Number of DOFs &lt; 30000

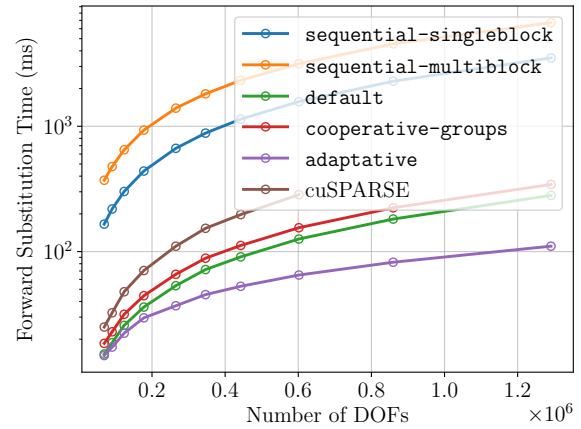
(b) Number of DOFs  $\geq 30000$ 

Figure 3.7: GPU forward substitution time as a function of the number of DOFs with several strategies, using the **balanced-level-set** method for constructing the scheduler when one is needed.

Figures 3.9 and 3.10 show the performance we can achieve on a CPU for solving the same problem. We first notice that parallelizing the computation on the CPU does not necessarily improve the performance especially for small systems where the

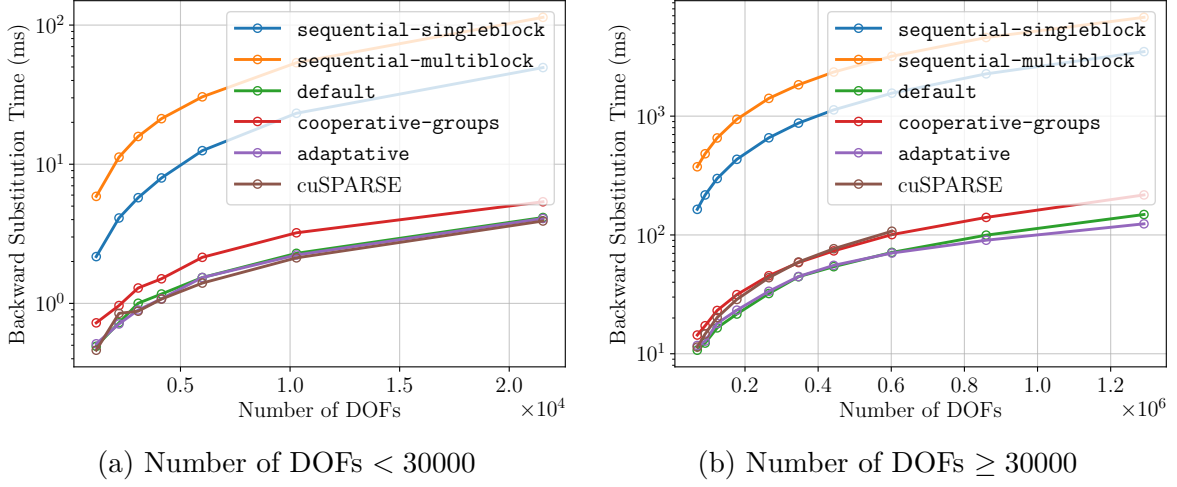


Figure 3.8: GPU backward substitution time as a function of the number of DOFs with several strategies, using the **balanced-level-set** method for constructing the scheduler when one is needed.

32-core method is slower than the single-core one. In Figure 3.11, we can see the speedup when using the GPU compared to the CPU. For small systems with less than 30000 unknowns, the GPU version is worse than the CPU one. However, when there are more than 30000 unknowns, the GPU version performs better, and the speedup increases with the size of the system. For the largest tested system (1291066 rows), the maximum speedup of 3.6 is reached.

We have only compared the performance of the solution step, but when we use a scheduler, we also need to take into account the cost of the preprocessing step. Assuming an iterative method is used to solve the system, the solution step is repeated multiple times with different right-hand sides. Therefore, the cost of the preprocessing step is amortized over the iterations. Figure 3.12 shows the number of iterations required to achieve a speedup greater than one, as well as the speedup achieved after 100 iterations compared to the single-core method which does not require any preprocessing step.



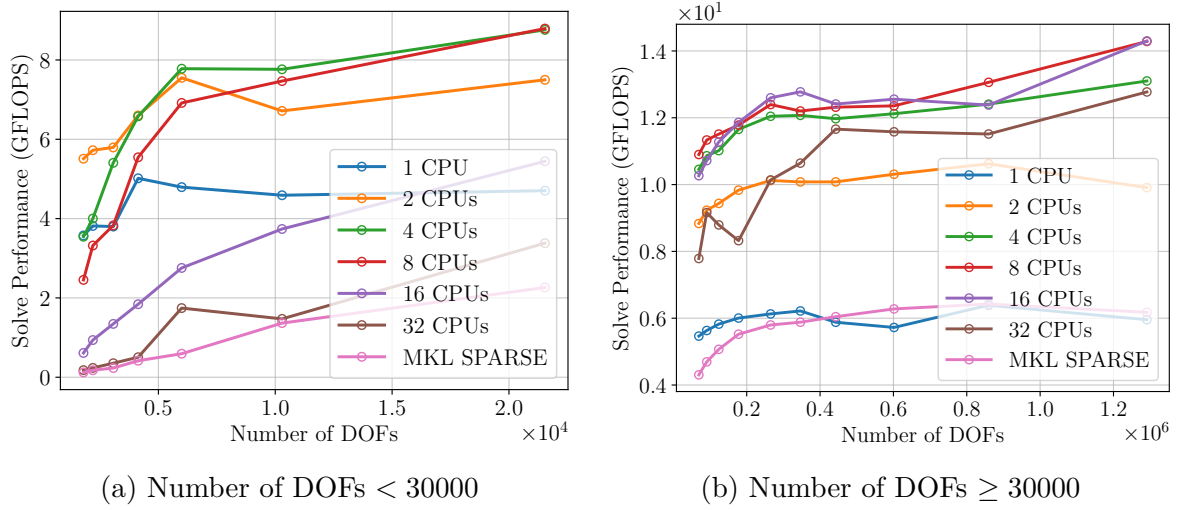
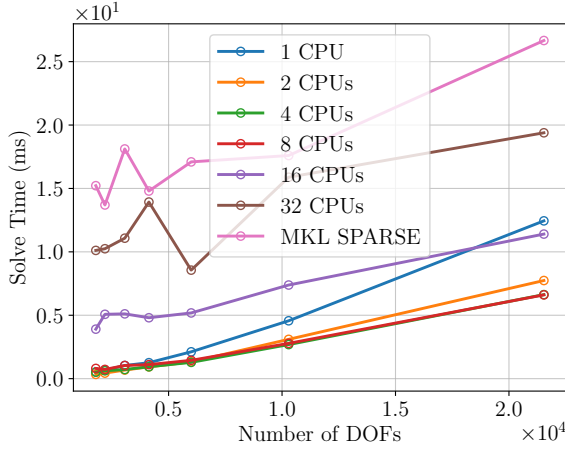


Figure 3.9: CPU solve performance as a function of the number of DOFs. We compare several computation methods. The “1 CPU” and “MKL SPARSE” correspond to single-core methods. The “1 CPU” method employs a simple implementation of the forward and backward substitution algorithm, while the “MKL SPARSE” method solves the system using the `mklsparse_z_trsv` routine from the Intel oneAPI Math Kernel Library [17]. For the multi-core methods, computations are parallelized using OpenMP similarly to the `default` method on GPU, utilizing a scheduler built with the `balanced-level-set` strategy.



(a) Number of DOFs &lt; 30000

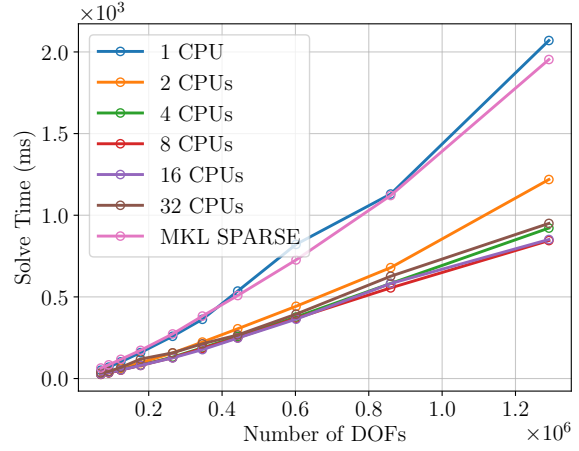
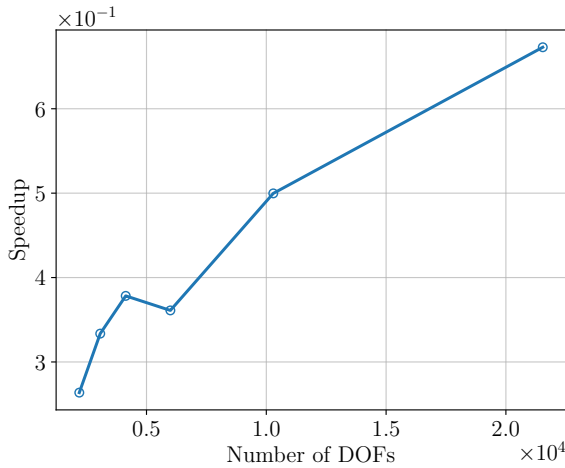
(b) Number of DOFs  $\geq 30000$ 

Figure 3.10: CPU solve time as a function of the number of DOFs. We compare several computation methods. The plots labeled “1 CPU” and “MKL SPARSE” correspond to single-core methods. The “1 CPU” method employs a simple implementation of the forward and backward substitution algorithm, while the “MKL SPARSE” method solves the system using the `mk1_sparse_z_trsv` routine from the Intel oneAPI Math Kernel Library [17]. For the multi-core methods, computations are parallelized using OpenMP similarly to the `default` method on GPU, utilizing a scheduler built with the `balanced-level-set` strategy.



(a) Number of DOFs &lt; 30000

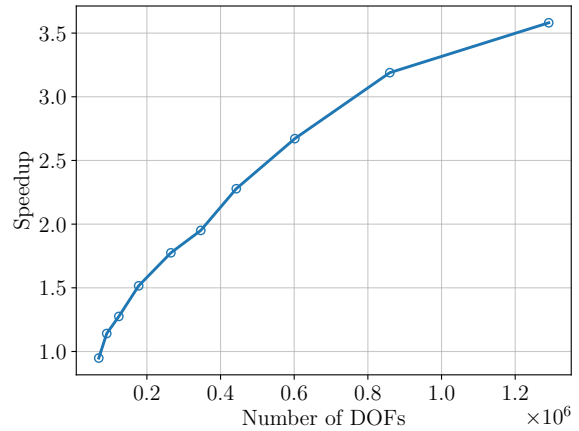
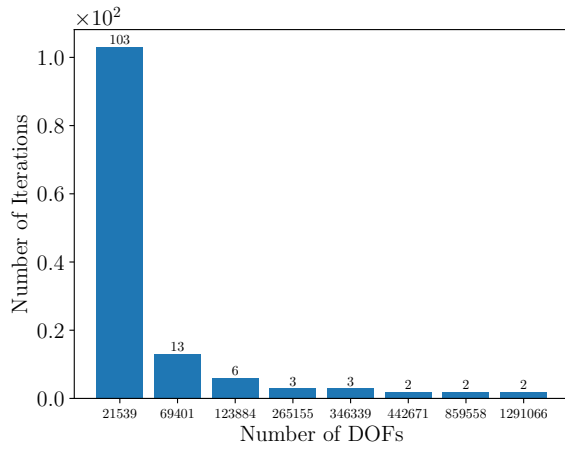
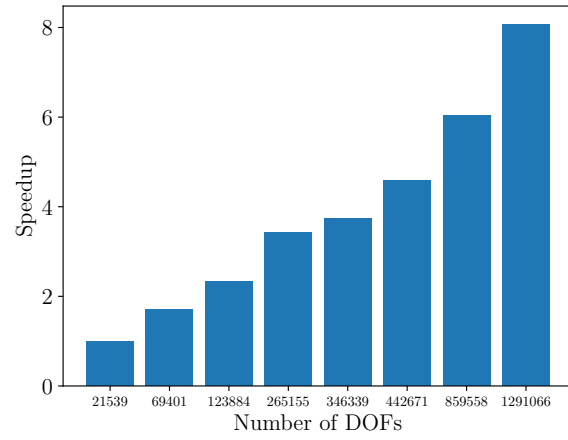
(b) Number of DOFs  $\geq 30000$ 

Figure 3.11: Speedup of the GPU solve using the `adaptative` strategy compared to the fastest CPU solve.



(a) Number of iterations required for the speedup of the GPU solve using the **adaptive** method to achieve a speedup greater than 1 compared to the single-core CPU solve, taking into account the preprocessing time.



(b) Speedup of the GPU solve using the **adaptive** method compared to the single-core CPU solve after 100 iterations, taking into account the preprocessing time.

Figure 3.12

### 3.2.4 Comparison of Multi-RHS Solution Strategies

In this section, we compare the different solution strategies when the system to solve has several right-hand sides.

First, the right-hand sides are stored in column-major order. Figure 3.13 shows how the number of right-hand sides affects performance for different system sizes across the various strategies. The performance of the cuSPARSE library is the worst in all cases. This comes from the fact that cuSPARSE is very slow for the backward substitution. When there are many right-hand sides, the **flipped-enhanced** method is the best for smaller systems, while the **block-inner** is faster for large systems. When the number of right-hand sides is low, all the block methods perform about the same. The **naive** method performs quite well when the number of right-hand sides is low in which case it manages to outperform the other methods by a small margin.

Figure 3.14 shows the same as Figure 3.13, but in this case the right-hand sides are stored in row-major order. Remember that this ordering allows to have coalesced memory accesses in the flipped methods. This is why the **flipped-enhanced** method is always the best when there are more than 128 right-hand sides and it peaks at more than 600 GFLOPS for large systems which is well above what was obtained when using column-major ordering. The **block-inner** method performs worse with this ordering.

Figure 3.15 summarizes the result by associating each pair of number of rows and number of RHS with the best strategy to solve the problem. Figure 3.16 shows the maximum GFLOPS obtained for each number of rows and RHS.

To compare these results with what can be obtained with CPUs, we implemented multiple methods on the CPU, namely **naive**, **block-outer**, **block-middle** and **block-inner**. We conducted the same tests as for the GPU with 1, 2, 4, 8, 16 and 32 CPU cores. Figure 3.17 shows the speedup of the best GPU method compared to the best CPU method.

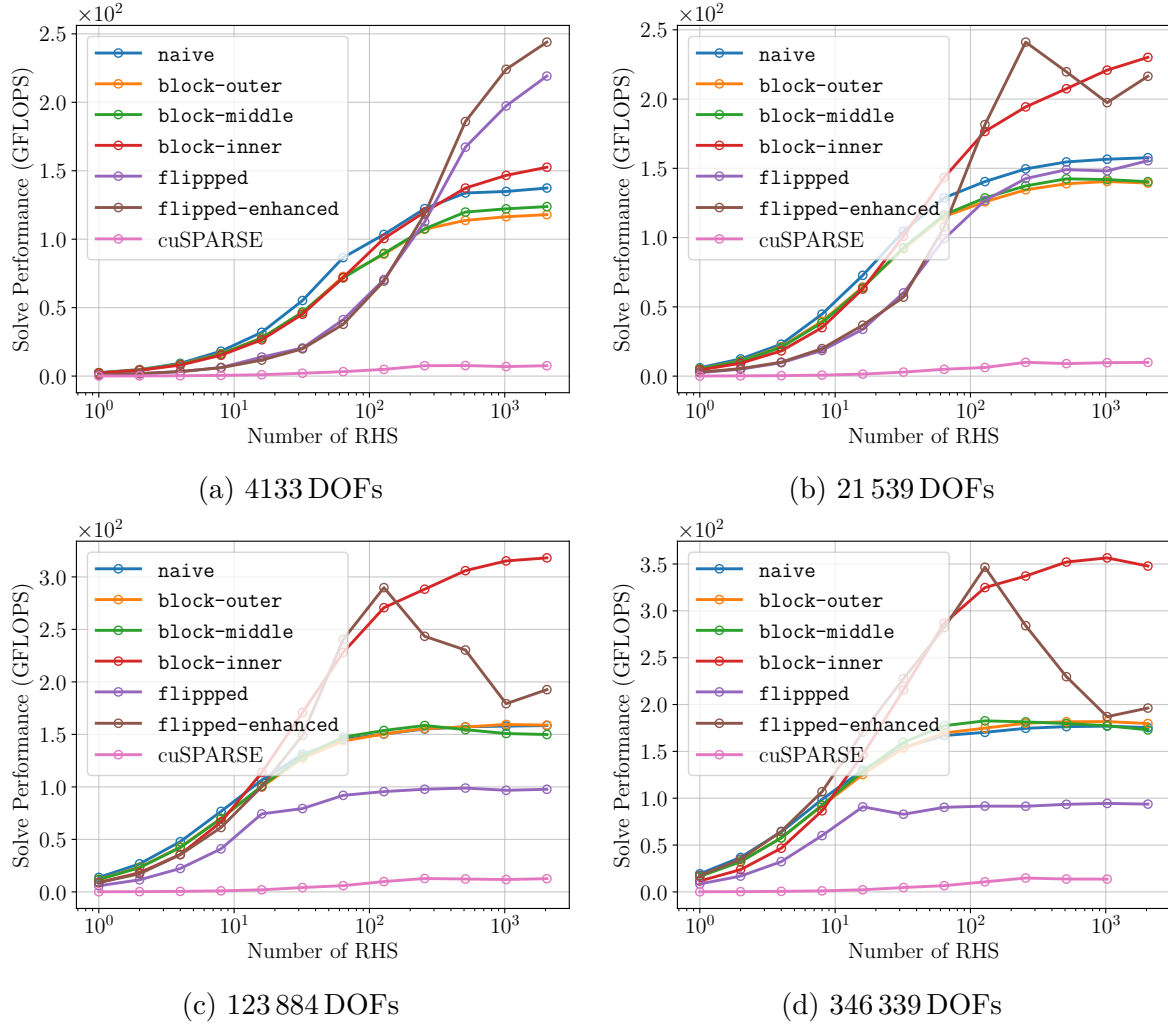


Figure 3.13: Solve performance as a function of the number of right-hand sides stored in column-major order for different system sizes and solution strategies. The cuSPARSE strategy uses the `cusparsespsm_solve` routine to solve the two triangular systems with multiple right-hand sides.

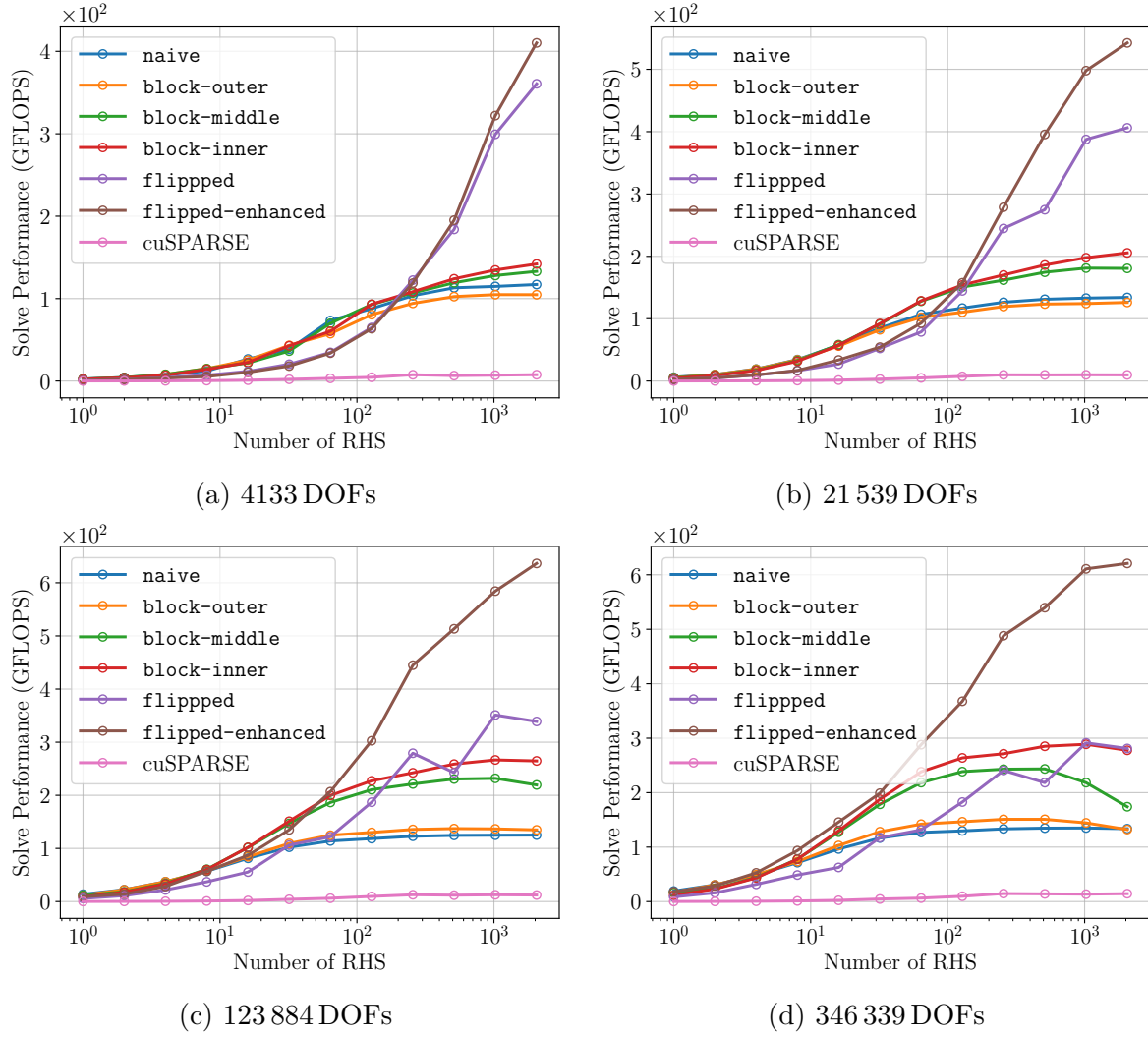


Figure 3.14: Solve performance as a function of the number of right-hand sides stored in row-major order for different system sizes and solution strategies. The cuSPARSE strategy uses the `cusparsespm_solve` routine to solve the two triangular systems with multiple right-hand sides.

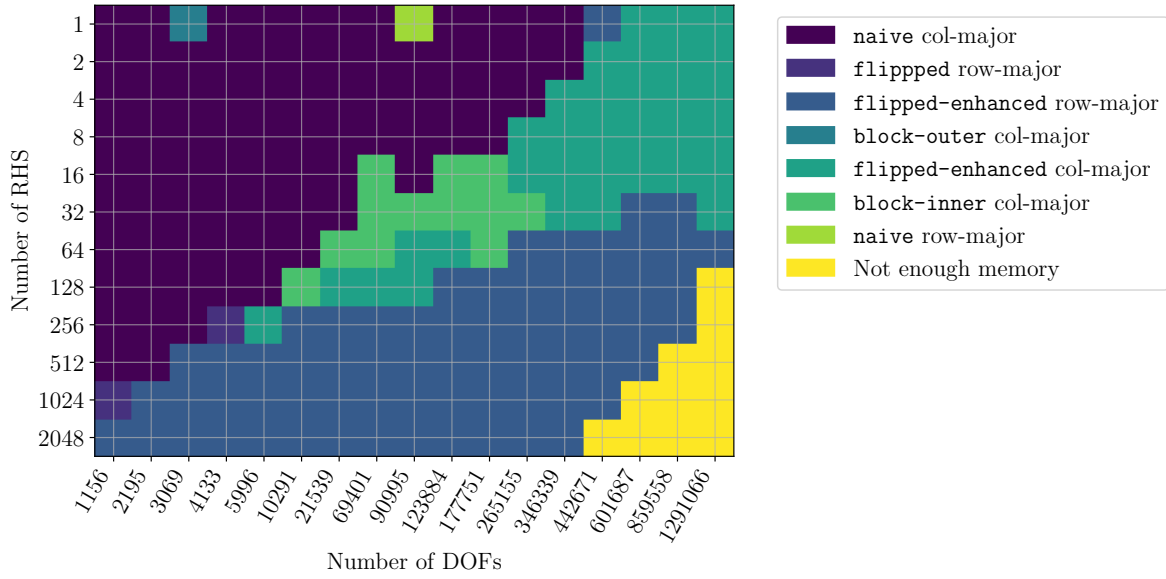


Figure 3.15: Colormap illustrating the optimal solution strategy for several system sizes and number of right-hand sides. Note that when there is only one right-hand side, this figure only shows the best multi-right-hand side strategy. However, in this case the best strategy would be the **adaptive** strategy.

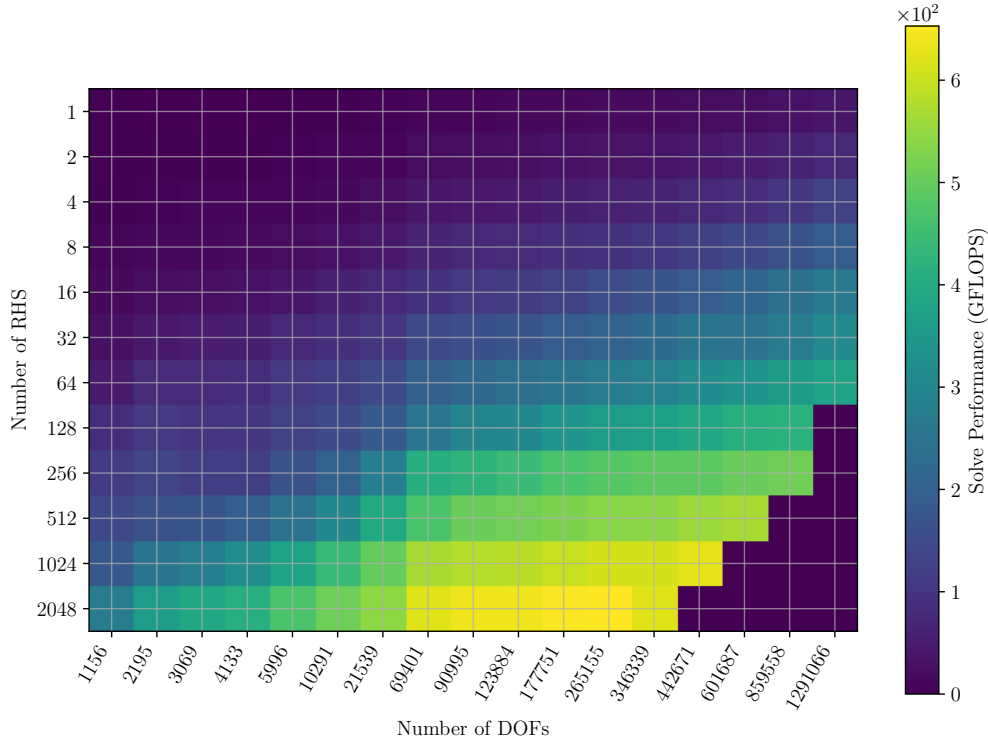


Figure 3.16: Heatmap illustrating the performance of the GPU solve when the best multi-right-hand-side strategy is used.

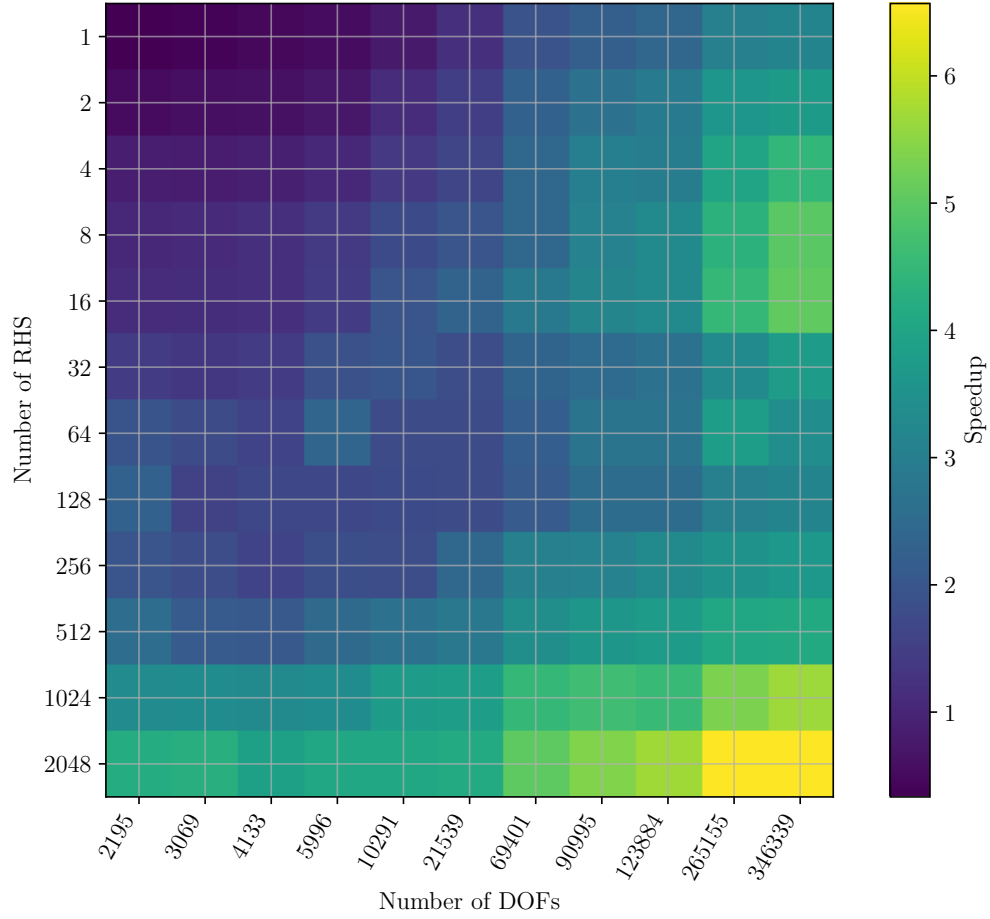


Figure 3.17: Heatmap representing the speedup of the best GPU multi-RHS solution strategy compared to the best CPU multi-RHS solution strategy.



### 3.3 ORAS

In this section, we test our triangular solver for use in applying the ORAS preconditioner at each iteration of the GMRES Krylov method running on the GPU (thanks to PETSc [20]). Recall that the ORAS preconditioner  $M_{\text{ORAS}}^{-1}$  is given by

$$M_{\text{ORAS}}^{-1} = \sum_{i=1}^{n_d} R_i^T D_i A_{\text{Robin},i}^{-1} R_i. \quad (3.3)$$

The triangular solves are required when applying the preconditioner to the residual  $r^k$ , where each process  $i$  needs to compute

$$A_{\text{ORAS},i}^{-1} R_i r^k = A_{\text{ORAS},i}^{-1} r^{k,i}, \quad (3.4)$$

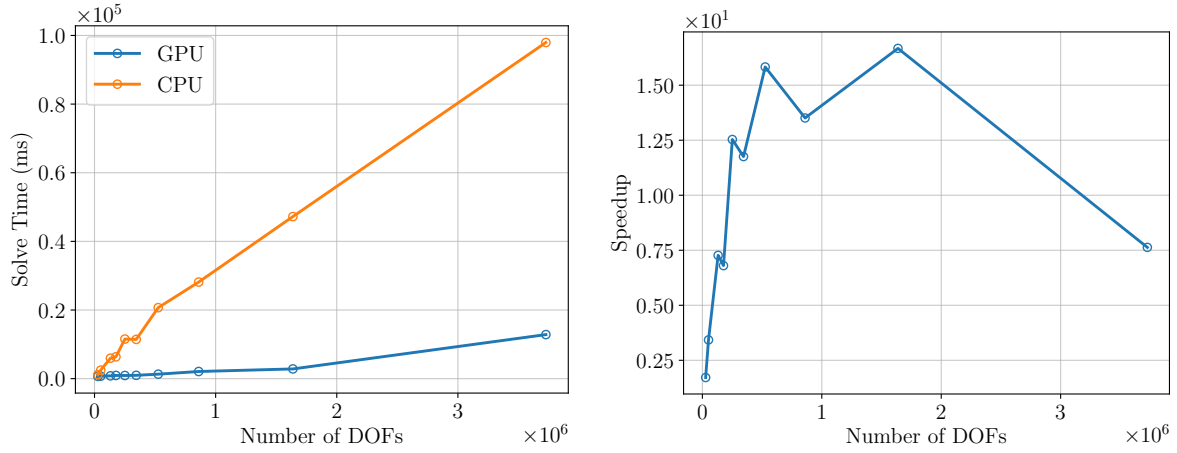
where  $r^{k,i}$  is the restriction of the residual to subdomain  $i$ . The forward and backward substitutions are performed using the *LU* decomposition of  $A_{\text{ORAS},i}$ . For this decomposition, we use UMFPACK [9] for the same reasons as earlier.

We once again solve a Helmholtz problem, but this time using 4 GPU nodes, which provides us with 16 GPUs (see Table 3.1). Consequently, the domain is decomposed into 16 overlapping subdomains, each assigned to its own MPI process and GPU. Each GPU node is equipped with one 32-core CPU, which allows each of the 4 MPI processes (per node) to utilize 8 CPU cores. We compare the results obtained with our GPU-based triangular solver with those obtained using MUMPS [2] and the CPU version of GMRES from PETSc [4] on the same number of CPU nodes (i.e., 4 nodes) (see Table 3.2) in the Lucia cluster. The test configuration is such that each node is fully utilized. A summary of the configuration of each test is shown in Table 3.3.

Test Configuration	CPU Test	GPU Test
Num. of nodes	4	4
Num. of subdomains	16	16
Num. of processes	16	16
Num. of CPU cores per process	32	8
Num. of GPUs per process	0	1

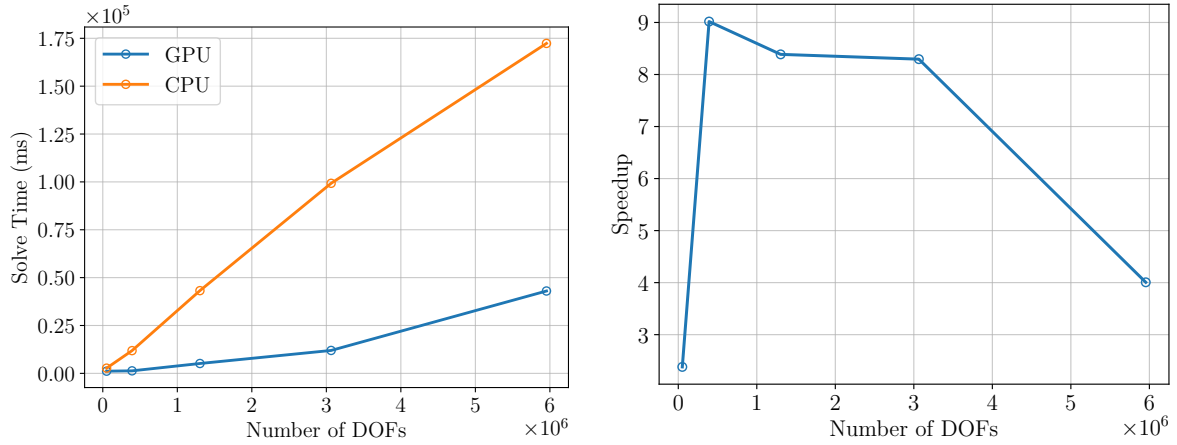
Table 3.3: ORAS test configuration

Figure 3.18 and Figure 3.19 show how the GPU version compares against the CPU one with varying mesh sizes and finite element orders. The results are also summarized in Table 3.4 and Table 3.5. The results clearly show that the GPU version is better. However, for large systems, the speedup of the GPU version compared to the CPU one decreases.



(a) Solve times of the GPU and CPU versions of the ORAS test with varying mesh element size. (b) Speedup of the GPU version compared to the CPU version of the ORAS test with varying mesh element size.

Figure 3.18: ORAS tests with varying characteristic length (CL) of the mesh elements.



(a) Solve times of the GPU and CPU versions of the ORAS test with varying mesh element order. (b) Speedup of the GPU version compared to the CPU version of the ORAS test with varying mesh element order.

Figure 3.19: ORAS tests with varying finite element order.

Order	CL	# DOFs	# Subdomain DOFs	# Iterations	Solve	Iteration
1	0.04	27 323	3272	19	663 ms	35 ms
1	0.03	51 220	5547	20	724 ms	36 ms
1	0.02	131 556	12 546	21	820 ms	39 ms
1	0.018	176 804	16 196	22	935 ms	43 ms
1	0.016	250 792	22 132	22	921 ms	42 ms
1	0.014	344 324	29 468	22	972 ms	44 ms
1	0.012	525 957	43 161	23	1307 ms	57 ms
1	0.01	859 989	68 056	23	2083 ms	91 ms
1	0.008	1 637 760	123 788	24	2833 ms	118 ms
1	0.006	3 726 197	269 303	24	12 836 ms	535 ms
1	0.03	51 220	5547	20	1145 ms	57 ms
2	0.03	392 478	40 119	21	1316 ms	63 ms
3	0.03	1 303 536	130 630	19	5151 ms	271 ms
4	0.03	3 064 155	303 996	18	11 967 ms	665 ms
5	0.03	5 954 096	587 129	18	43 016 ms	2390 ms

Table 3.4: Results of the ORAS experiment on GPU. The first column contains the order of the finite elements. The second is the characteristic length of one mesh element. The last column contains the average time of each GMRES iteration. The third column contains the average number of DOFs per subdomain. If one multiplies this number by the number of subdomains (16), then the result is larger than the total number of DOFs in the system, because the subdomains are overlapping.

Order	CL	# DOFs	# Subdomain DOFs	# Iterations	Solve	Iteration
1	0.04	27 323	3272	19	1138 ms	60 ms
1	0.03	51 220	5547	20	2480 ms	124 ms
1	0.02	131 556	12 546	21	5962 ms	284 ms
1	0.018	176 804	16 196	22	6362 ms	289 ms
1	0.016	250 792	22 132	22	11 544 ms	525 ms
1	0.014	344 324	29 468	22	11 433 ms	520 ms
1	0.012	525 957	43 161	23	20 689 ms	900 ms
1	0.01	859 989	68 056	23	28 144 ms	1224 ms
1	0.008	1 637 760	123 788	24	47 227 ms	1968 ms
1	0.006	3 726 197	269 303	24	97 947 ms	4081 ms
1	0.03	51 220	5547	20	2725 ms	136 ms
2	0.03	392 478	40 119	21	11 872 ms	565 ms
3	0.03	1 303 536	130 630	19	43 200 ms	2274 ms
4	0.03	3 064 155	303 996	18	99 265 ms	5515 ms
5	0.03	5 954 096	587 129	18	172 336 ms	9574 ms

Table 3.5: Results of the ORAS experiment on CPU. The first column contains the order of the finite elements. The second is the characteristic length of one mesh element. The last column contains the average time of each GMRES iteration. The third column contains the average number of DOFs per subdomain. If one multiplies this number by the number of subdomains (16), then the result is larger than the total number of DOFs in the system, because the subdomains are overlapping.

# Chapter 4

## Conclusion

The objective of this thesis was to accelerate the computation of large-scale time-harmonic problems using GPUs. We were particularly interested in solving these problems with iterative solvers, such as GMRES, preconditioned with the ORAS preconditioner. To achieve our objective, we used PETSc’s GPU implementation of GMRES [5, 20] and developed a GPU-accelerated sparse triangular solver used when applying the preconditioner at each GMRES iteration.

We evaluated multiple methods for efficiently solving triangular systems on the GPU, testing them on a simple Helmholtz problem. The most effective approach proved to be the **adaptative** method which achieved a speedup of up to  $3.6\times$  compared to the best CPU implementation. We also implemented several methods for solving systems with multiple right-hand sides, leading to more than  $6\times$  speedup compared to the best CPU implementation for the same Helmholtz problem. In the multi-right-hand side case, several methods were good depending on the size of the system and the number of right-hand-sides (see Figure 3.15). In general, the GPU version outperformed the CPU one for large systems, but performed worse for small ones.

Moreover, we tested the full iterative method with one right-hand side, using PETSc’s GPU-accelerated GMRES and the ORAS preconditioner. The GPU version performed quite well, achieving more than  $10\times$  speedup in certain cases.

However, we were not able to test the full method with multiple right-hand sides due to the absence of a GPU-accelerated iterative solver in PETSc that correctly parallelizes multiple right-hand sides.

Lastly, our study focused on the ORAS preconditioner, but other domain decomposition techniques exist. For example, GmshDDM [27] is a framework which solves the same types of problems, but using a matrix-free substructured non-overlapping domain decomposition method. It could also benefit from our triangular solver.

# Bibliography

- [1] Patrick R Amestoy, Alfredo Buttari, Jean-Yves L’excellent, and Theo Mary. Performance and scalability of the block low-rank multifrontal factorization on multi-core architectures. *ACM Transactions on Mathematical Software (TOMS)*, 45(1):1–26, 2019.
- [2] Patrick R Amestoy, Iain S Duff, Jean-Yves L’Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.
- [3] Edward Anderson and Youcef Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1(01):73–95, 1989.
- [4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc/TAO users manual. Technical Report ANL-21/39 - Revision 3.21, Argonne National Laboratory, 2024.
- [5] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc Web page. <https://petsc.org/>, 2024.
- [6] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

- [7] Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6-8):318–331, 2008.
- [8] NVIDIA Corporation. CUDA C++ Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [9] Timothy A Davis. Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):196–199, 2004.
- [10] Victorita Dolean, Pierre Jolivet, and Frédéric Nataf. An introduction to domain decomposition methods: algorithms, theory and parallel implementation. Master’s thesis, France, 2015. cel-01100932v4.
- [11] Iain S Duff and John K Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 1983.
- [12] Howard C Elman. *Iterative methods for large, sparse, nonsymmetric systems of linear equations*. Yale University, 1982.
- [13] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 79(11):1309–1331, 2009.
- [14] Anne Greenbaum. *Iterative methods for solving linear systems*. SIAM, 1997.
- [15] Mark Harris. Optimizing parallel reduction in cuda. <https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>. NVIDIA Developer Technology.
- [16] Cenaero HPC. Lucia system overview. <https://doc.lucia.cenaero.be/overview/>.
- [17] Intel Corporation. Intel oneapi math kernel library (onemkl). <https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-2/overview.html>, 2024. Version 2024.2.
- [18] Bruce M Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2(1):5–32, 1970.
- [19] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.
- [20] Richard Tran Mills, Mark F. Adams, Satish Balay, Jed Brown, and Alp Dener. Toward performance-portable PETSc for GPU-based exascale systems. *Parallel Computing*, 108:102831, 2021.

- [21] Andrea Moiola and Euan A Spence. Is the helmholtz equation really sign-indefinite? *Siam Review*, 56(2):274–312, 2014.
- [22] NVIDIA Corporation. Nvidia cusparse library user guide. <https://docs.nvidia.com/cuda/cusparse/index.html>, 2024. Version 12.6.
- [23] Anthony Royer, Eric Béchet, and Christophe Geuzaine. Gmsh-fem: an efficient finite element library based on gmsh. In *14th World Congress on Computational Mechanics (WCCM), ECCOMAS Congress 2020*. Scipedia, 2021.
- [24] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.
- [25] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.
- [26] Joel H Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM journal on scientific and statistical computing*, 11(1):123–144, 1990.
- [27] Bertrand Thierry, Alexandre Vion, Simon Tournier, Mohamed El Bouajaji, David Colignon, Nicolas Marsic, Xavier Antoine, and Christophe Geuzaine. Getddm: An open framework for testing optimized schwarz methods for time-harmonic wave problems. *Computer Physics Communications*, 203:309–330, 2016.