UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Ziya Mammadov

GPU-accelerated Domain Decomposition Methods for Helmholtz equation

Master's Thesis (30 ECTS)

Supervisor(s): Prof. Eero Vainikko

Tartu 2024

GPU-accelerated Domain Decomposition Methods for Helmholtz equation

Abstract:

The Helmholtz equation, used in various fields like acoustics, optics, and seismology, is a partial differential equation that describes how waves propagate in various physical systems. The Helmholtz matrix arises from the discretization of the Helmholtz problem when solving the Helmholtz equation numerically using finite difference or finite element methods. In practice, the numerical solution of the Helmholtz equation can be challenging due to the size of the discredited problem as well as the spectral properties of the matrix. The thesis explores iterative methods to solve the Helmholtz equation, speeding up the computations by using the power of GPUs. A special domain decomposition preconditioning technique, the Restricted Additive Average Schwarz method, is applied in a setup that allows using multiple subdomains to solve simultaneously in one go on a GPU. For this purpose a special implementation of the Conjugate Gradients iterative solver in PyOpenCL using complex arithmetics was developed, allowing to solve for multiple right-hand side vectors simultaneously. Performance evaluation for the overall solution of the discretized Helmholtz equation is performed experimentally to compare the efficiency of different subdomain solution techniques.

Keywords:

GPU programming, Conjugate Gradient method, Iterative methods, Domain Decomposition, OpenCL, PyOpenCL

CERCS:

P130 - Functions, differential equations, P170 - Computer science, numerical analysis, systems, control

GPU-kiirendatud alampiirkondadeks jagamise meetodid Helmholtzi võrrandi jaoks

Lühikokkuvõte:

Helmholtzi võrrand, mida kasutatakse erinevates valdkondades, näiteks akustika, optika ja seismoloogia, on osatuletistega diferentsiaalvõrrand, mis kirjeldab lainete tekkimist erinevates füüsikalistes süsteemides. Helmholtzi maatriks saadakse antud ülesande diskretiseerimisel numbriliseks lahendamiseks, kasutades lõplike diferentside või lõplike elementide meetodeid. Praktikas võib Helmholtzi võrrandi numbriline lahendamine olla keerukas nii probleemi suuruse kui ka maatriksi spektraalsete omaduste tõttu. Käesolev lõputöö uurib iteratiivseid meetodeid Helmholtzi võrrandi lahendamiseks kiirendades arvutusi kasutades GPU võimsust. Iteratiivses protsessis rakendatakse eelkonditsioneerijana spetsiaalset alampiirkondadeks jagamise meetodit, Restricted Additive Schwarz'i meetodit, mis võimaldab GPU-d kasutada samaaegselt mitme alampiirkonna lahendajana. Sel eesmärgil sai realiseeritud spetsiaalne Kaasgradientide kompleksarvuline blokk-lahendaja PyOpenCL-s mitme samaaegse parempoole vektori jaoks. Sooritatakse eksperimente diskretiseeritud Helmholtzi võrrandi lahenduse jõudluse hindamiseks, võrreldakse erinevate tehnikate tõhusust sõltuvalt alampiirkondade lahendamiseks kasutatavast meetodist.

Võtmesõnad:

GPU-programmeerimine, Kaasgradientide meetod, Iteratiivsed meetodid, Alampiirkondadeks jagamise meetod, OpenCL, PyOpenCL

CERCS:

P130 - Funktsioonid, diferentsiaalvõrrandid, P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

List of abbreviations and terms:

PDE	Partial Differential Equation
CG	Conjugate Gradient
GMRES	Generalized Minimal Residual Method
GMRES(m)	Restarted GMRES
FGMRES	Flexible Generalized Minimal Residual Method
ILU	Incomplete Cholesky Factorisation
DDM	Domain Decomposition Method
RAS	Restricted Additive Schwarz
RAAS	Restricted Additive Average Schwarz
GAS	Generalized Additive Schwarz
GPU	Graphics Processing Units
CPU	Central Processing Units
FPGA	Field Programmable Gate Arrays
BDD	Balancing Domain Decomposition
GPGPU	General-purpose computing on graphics processing units

List of Tables

List of Figures

List of Algorithms

Listings

Contents

1 Introduction

The solution of partial differential equations (PDEs) plays a fundamental role in numerous scientific and engineering applications. Helmholtz equations hold particular significance among the various types of PDEs due to their wide-ranging applicability in fields such as acoustics, electromagnetics, quantum mechanics, and seismology, which is the scientific study of earthquakes and elastic waves [JAET23]. However, obtaining accurate and efficient solutions for these equations can be challenging, especially when dealing with large-scale problems or complex geometries. Therefore, iterative Krylov subspace methods [LS13] have emerged as powerful techniques for solving linear systems arising from discretized PDEs. One of the simplest methods within this class is the Conjugate Gradient (CG) method [She94], renowned for its robustness and efficiency in the case of symmetric positive definite matrices. In the case of unsymmetric matrices, one of the fastest Krylov subspace methods is the GMRES Method (Generalized Minimal Residual Method) [Saa93], which can also be used in the case of more complex preconditioning techniques, like the Domain Decomposition Method (DDM), that we will describe later.

The Conjugate Gradient method is an iterative algorithm to solve symmetric positive definite systems of linear equations. It is beneficial for solving large, sparse systems, which arise in many scientific and engineering applications. Some properties of the CG method are given below.

- **Efficiency** - The Conjugate Gradient method is efficient, and it usually requires fewer iterations than other iterative methods to achieve a given level of accuracy.
- **Symmetry** - The Conjugate Gradient method is suitable for solving symmetric positive definite systems. The symmetric positive definite systems appear in many scientific and engineering applications. In such cases, the Conjugate Gradient method is often faster and more accurate compared to other methods not specifically designed for symmetric systems.
- **Parallelizability** - The Conjugate Gradient method is easily parallelizable, so it can leverage modern computer architectures and solve large systems efficiently. This parallelizability makes it a good candidate for GPGPU use.
- **Low memory footprint** - The Conjugate Gradient method requires less memory than other methods. That results in CG being ideal for solving large systems on computational devices with limited memory capacity that cannot store the whole system in memory simultaneously.

When discussing iterative methods for solving linear systems of equations, it is critical to consider using preconditioners. The preconditioners are used in iterative methods for solving linear systems of equations to improve their convergence properties. They are generally used when the coefficient matrix is large, sparse, and poorly conditioned. A

preconditioner is a matrix or an operator that approximates the inverse of the coefficient matrix and is applied to the system of equations before or during the usage of the iterative method. The goal of the preconditioner is to transform the system of equations into an equivalent system that is easier to solve iteratively by obtaining a new matrix with a lower condition number.

For instance, the Additive Schwarz Method [Dry89] can be a preconditioner for the Krylov subspace methods like GMRES (Generalized Minimal Residual) to improve their convergence properties. In this approach, the domain of the problem is divided into a set of overlapping subdomains. Each subdomain is solved independently with a direct or an inexact solver like ILU (Incomplete Cholesky Factorisation) or CG. At the same time, the outer iterative method is the Flexible GMRES (FGMRES) method, allowing the robust convergence of outer iterations with different preconditions on each iteration, see [GSV17a]. During the preconditioning step, the solutions obtained from each subdomain are combined using the Additive Schwarz method, which updates the global solution by adding up the local (possibly approximate) solutions from each subdomain. The outer iterative process is repeated until the global solution converges to the desired level of accuracy. The key advantage of this approach is that the computational cost of the local preconditioner is significantly reduced compared to solving the entire system using GMRES with direct subdomain solvers with sparse matrix factorization.

In this thesis, the background section (see Section 2) explores GPU programming and its benefits for computational tasks. It also provides an overview of Krylov subspace and domain decomposition methods. The data and methods section (see Section 3) details the implementation of GPU-accelerated solvers for the Helmholtz equations, while the results section (see Section 4) analyzes the solver's performance across various metrics. Finally, the discussion section (see Section 5) examines the results' interpretation, discusses the methods' limitations, and suggests potential improvements.

While writing this thesis, the author used ChatGPT [Ope23] to get feedback on the content and correct language errors. The feedback helped improve the text and fix the mistakes regarding the language.

1.1 Problem

The solution of Helmholtz equations presents numerous challenges, making it a complex and demanding task. These equations describe the behavior of wave phenomena, such as acoustic or electromagnetic waves, as they propagate through a medium. Wave propagation brings unique obstacles originating from diffraction, interference, and resonance. Accurate wave behavior representation necessitates meticulous treatment of boundary conditions, computational domain size, and discretization techniques.

The Helmholtz operator can also exhibit severe ill-conditioning, mainly when dealing with large wave numbers or complex geometries. Ill-conditioning refers to situations where slight variations in the input or data can lead to significant changes in the solution.

Furthermore, Helmholtz equations arise in various applications involving large-scale systems, such as wave propagation in realistic environments or the modeling of complex structures. Solving such large-scale problems calls for efficient computational techniques capable of managing vast amounts of data and computational operations while maintaining accuracy and reasonable time complexity.

Addressing these challenges requires developing and utilizing advanced numerical techniques, including iterative Krylov subspace methods like the Flexible Generalized Minimal Residual method (FGMRES) and the Conjugate Gradient (CG) method. These methods aim to overcome the limitations presented by direct solvers by efficiently approximating the solution through iterative means, making them well-suited for handling the complexities associated with Helmholtz equations.

1.2 Contribution

In this thesis, we study the solution to Helmholtz's problem using the FGMRES method as an outer Krylov subspace method preconditioned with Restricted Averaging Additive Schwarz (RAAS) to improve the convergence of FGMRES. The Conjugate Gradient (CG) is used as an inner method for equal-sized submatrices obtained by applying an RAAS preconditioner to the original domain. This makes it possible to solve for multiple subdomains on a single GPU simultaneously – using multiple right-hand side vectors since the matrices are precisely the same on each subdomain. The aim of the thesis is to shed light on the properties, strengths, and limitations of those methods.

In order to enhance the computational speed of these operations, advanced techniques are employed, such as utilizing the power of Graphical Processing Units (GPUs) to accelerate matrix operations using the OpenCL programming framework.

The choice of an appropriate preconditioner can significantly impact the performance of the CG method when solving Helmholtz equations. Preconditioners aim to transform the original system into a more well-conditioned form, thereby reducing the number of iterations required for convergence. By exploring and comparing different preconditioning strategies, this thesis aims to identify the most effective and efficient preconditioners tailored specifically for Helmholtz equations.

Furthermore, numerical experiments will be conducted to evaluate the performance of different solutions in various scenarios using the resources from UT HPC. This will involve systematically varying parameters such as the total number of domains, domain size, and number of inner iterations (and possibly the presence of heterogeneities - as future work) to assess the preconditioners' robustness and scalability. Comparisons will be made based on convergence behavior, computational efficiency, and overall solution accuracy. The outcomes of this investigation will provide the performance gains achieved by utilizing the parallel processing capabilities of GPUs and explore the integration of Domain Decomposition Methods to enhance the solution process's efficiency further. The

objective is to achieve significant speedup and reduce the time-to-solution for large-scale Helmholtz equation problems.

2 Background

This section will give an outlook on the application of GPGPU programming using OpenCL to accelerate the Krylov subspace methods for solving Helmholtz equations and provide background information about the separate components of the implementation.

2.1 General-purpose computing on graphics processing units (GPGPU) computations

Recently, the Graphical Processing Unit (GPU) has shown great potential for the future of computation. Initially designed for fast image rendering in graphical applications, GPUs are now widely used for non-graphical applications, a field known as GPU computing. The computational capability of GPUs is evolving into powerful parallel computing units. Hence, General-purpose computing on graphics processing units (GPGPU) is a technique that involves utilizing the processing power of graphics processing units (GPUs) for general-purpose computing tasks that were previously performed by the central processing unit (CPU).

The architecture of GPUs is designed to handle a large number of computations simultaneously. It makes GPUs more powerful than CPUs in certain types of computations. The main difference between CPU and GPU architecture is the number of cores. A CPU typically has a few cores optimized for sequential processing, while a GPU has thousands of smaller cores optimized for parallel processing. This makes GPUs more suitable for tasks that require big parallelisms, such as image and video processing, scientific simulations, and machine learning.

Furthermore, GPUs have much faster memory and memory bandwidth than CPUs. That allows GPUs to move data between the processor and memory quickly. This is critical for performance in data-intensive applications. Additionally, modern GPUs are designed to be highly programmable and can be programmed using a variety of programming languages, including CUDA, OpenCL, and others.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA. It allows developers to use GPUs for general-purpose computing tasks in addition to their traditional use in graphics processing. The CUDA is a parallel computing platform developed by NVIDIA and optimized only for NVIDIA GPUs. However, depending on the underlying hardware, it might bring some performance overhead and limitations. On the other hand, OpenCL is an open standard for parallel programming across different hardware architectures, including GPUs, CPUs, and FPGAs. This means that OpenCL can be used with various hardware vendors. It makes OpenCL more adaptable and practical compared to CUDA [DWL+12, FVS11].

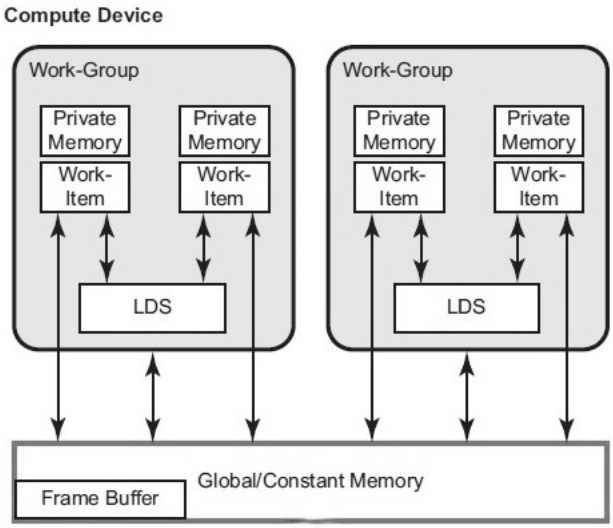

Figure 1. OpenCL memory model. LDS is an abbreviation for Local data storage

2.2 Execution of a program in OpenCL

In today's era of big data and complex computational tasks, the demand for high-performance computing solutions has never been greater. Traditional approaches to parallel computing often face challenges in achieving portability across different hardware platforms. OpenCL addresses these challenges by providing a standardized framework for parallel programming that enables developers to write efficient, portable code across a wide range of devices. It is used in a wide range of applications and use cases, including scientific simulations, image and video processing, machine learning, and computational finance. Real-world examples demonstrate the performance benefits and scalability achieved by using OpenCL in these applications.

OpenCL follows a host-device model, where the host CPU coordinates the execution of parallel kernels on compute devices such as GPUs and other accelerators. The architecture of OpenCL consists of compute devices, compute units, work-items, work-groups, and different memories. The memory hierarchy includes global, local, and private memory spaces; see Figure 1. This architecture enables fine-grained parallelism and efficient data movement between the host and compute devices. The programming model of OpenCL involves writing kernels, which are parallel functions written in the OpenCL C language. Kernels are executed on compute devices in parallel by a large number of work-items, with each work-item representing a single execution thread. Data parallelism is achieved by partitioning the computation across multiple work-items, while task parallelism is achieved by launching multiple kernels concurrently [KMSZ15, SGS10].

To start writing an OpenCL program, we must ensure that OpenCL SDK is correctly

installed for our platform. It includes the necessary libraries, header files, and tools for writing and building OpenCL programs. The term platform in OpenCL is the hardware and software environment in which OpenCL runs. There can be multiple platforms on a single computer. Each platform has its own set of devices. A compute device is the physical hardware on which OpenCL code runs. Examples of devices include CPUs, GPUs, and FPGAs. GPUs consist of a large number of compute units (CUs), and each CU has a set of cores that can execute instructions in parallel. These cores are also called processing elements. In addition to the processing cores, GPU architecture has several memory types, including global memory, shared memory, and registers. Global memory is accessible to all cores and is used to store data that needs to be accessed by all threads in a program. Shared memory is a fast, low-latency memory shared between threads within a thread block. Registers are fast, on-chip memory that stores data and intermediate values during computations. Additionally, each processing element has its own private memory, accessible only by one work item or thread [KSA+10].

In OpenCL programs, the host device communicates with the compute device by sending commands through a command queue. These commands can be for kernel execution or data transfers. Each command is executed in the same order as in the queue and asynchronously, meaning the host can add commands to the queue while the device executes previous commands. The compute device receives the command from the command queue to execute the kernel on a workgroup. A workgroup is a group of work items that execute, communicate, and synchronize using shared memory and barrier synchronization on the same compute unit. A workgroup is defined by its size, specified by the developer when the OpenCL kernel is launched. On the other hand, a wavefront is a group of work items that execute the same instruction in a lock-step fashion, and its size depends on the hardware architecture of the GPU, which is usually 32 or 64. Once all the workgroups have been executed, the compute device sends the results back to the host device [TS12].

There are several ways to develop OpenCL programs. The most renowned is C language-based development. While C provides a robust API for writing efficient parallel code across different computing devices, its complexity can cause some challenges. Hence, alternative options are explored and discussed in the following section.

2.3 GPU programming with PyOpenCL

Nowadays, we are witnessing the different complex algorithms that solve real-life applications. Therefore, an increase in the time complexity of algorithms has led to a need to accelerate them using GPUs. PyOpenCL is a Python library used to program heterogeneous systems with OpenCL. It provides a way for Python programmers to harness the power of OpenCL from within their Python scripts, allowing them to write high-performance code that runs on various hardware architectures. It also allows the use of multiple GPUs for a single Python program. There are many advantages to using the

PyOpenCL Python library over the C programming language. I will list some of them below:

- **Simplicity** - Python is a high-level programming language with a simpler syntax than C, which is also easy to debug. This can make development faster and more efficient.
- **Easy-to-use API** - PyOpenCL provides a simpler and more intuitive API than the low-level C API, making it easier to write and understand code.
- **Easy integration with different libraries** - PyOpenCL integrates well with the Python ecosystem, which includes libraries like NumPy, SciPy, and Matplotlib. This makes it a popular choice for scientific computing applications.
- **Cross-platform support** - PyOpenCL is designed to work on various operating systems and hardware platforms, including Windows, macOS, Linux, and others.

The PyOpenCL platform has different sections for both host code and device code. The host code encompasses typical Python operations like variable definition and regular code execution. In contrast, the device code is responsible for actual implementation. This device code operates within kernels. PyOpenCL views GPUs and accelerators as assemblies of multiple compute devices, each subdivided into units containing numerous processing elements. This organizational structure facilitates the execution of functions. Development in PyOpencCl starts with defining the context that binds all the objects inside the code. It can be created by providing the list of devices as `cl.Context(devices=[devices])`. The context contains all the information about devices, kernels, memory, and variables in the created context. The command queue defines the order in which the commands are executed. Queue is defined as `cl.CommandQueue(ctx)`, and used to enqueue commands for execution on a specific device associated with the context. We can pass data to kernels by buffers. It is defined as `cl.Buffer(ctx, mf.READ_WRITE, size=256, hostbuf=a_values)`. In the mentioned command, we define the read-write permission of the buffer and its size. Moreover, we pass the value `a_values` that is a numpy array. The most important part of PyOpenCL is defining the kernels. The kernel code is written as a string inside `cl.Program(ctx, kernel)` [PyO].

```
program = cl.Program(ctx, """
__kernel void mult_matrices(const unsigned int size, __global float *
    first, __global float * second, __global float * result) {
    int i = get_global_id(1);
    int j = get_global_id(0);
    result[i + size * j] = 0;
    for (int k = 0; k < size; k++) {
        result[j + size * i] += first[k + size * i] * second[j + size
            * k];
    }
} """).build()

program.multmatrices(queue, first.shape, None, np.int32(len(first)),
    first_buf, second_buf, result_buf)

cl.enqueue_copy(queue, result, result_buf)
```

Listing 1. PyOpenCL code for matrix multiplication

The provided implementation [1] demonstrates a Matrix Multiplication program utilizing PyOpenCL. It initializes two random matrices using NumPy arrays, passed to the kernel via buffers. The resulting matrix is stored in a destination matrix labeled 'result'. PyOpenCL uses the 'enqueue_copy' function to copy data between different memory objects. Specifically, it copies data from one buffer to another. This function allows you to efficiently transfer data between different memory regions, such as between the host and device memory or between different regions of device memory. In our code, `cl.enqueue_copy(queue, result, result_buf)` line copies the data from buffer 'result_buf' to the NumPy array.

```
for i in range(first[0].size):
    for j in range(second[0].size):
        for k in range(second[0].size):
            result[i][j] += first[i][k] * second[k][j]
```

Listing 2. Python code for matrix multiplication

The data presented in Table 1 shows that the PyOpenCL implementation considerably outperforms the regular Python implementation [2]. For instance, while the PyOpenCL implementation executes the multiplication in merely 0.183 seconds for a matrix size of 1024, the regular Python code requires a significantly longer time of 1379 seconds to accomplish the same task. Observations indicate that PyOpenCL demonstrates performance that increases exponentially as the size of the matrices increases. This phenomenon arises from the efficient method of passing an entire vector from the matrix as a single work

item to the kernel. It indicates that using PyOpenCL for matrix multiplication offers considerable performance improvements over regular Python code.

Table 1. Execution Time Comparison: PyOpenCL vs. Python for Matrix Multiplication

Size of Matrix	**PyOpenCL (sec)**	**Normal Python (sec)**
128	0.146	2.613
256	0.148	20.627
512	0.151	170.378
1024	0.183	1378.951

2.4 Krylov subspace methods

Solving large linear systems is a fundamental task in scientific computing, engineering, and other fields. Direct methods, while accurate, face scalability challenges for large-scale problems due to their computational complexity and memory requirements. Iterative solvers, particularly Krylov subspace methods, offer a promising alternative for efficiently solving large sparse linear systems. Krylov subspace methods are a family of iterative solvers used to solve large, sparse linear systems of equations. These methods are particularly effective when direct methods, such as LU decomposition or Cholesky factorization, become computationally unusable due to the size and sparsity of the coefficient matrix.

The fundamental idea behind Krylov subspace methods is to iteratively construct a sequence of approximations to the solution vector by generating a sequence of subspaces known as Krylov subspaces. These subspaces are formed by repeatedly applying the coefficient matrix of the linear system to an initial vector, see [Saa03].

Given a matrix A and a starting vector r_0, the Krylov subspace of dimension n, denoted as $\mathcal{K}_n(A, r_0)$, is the linear span of the vectors formed by repeatedly applying the matrix A to r_0 up to n times. Mathematically, it is defined as:

$$\mathcal{K}_n(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{n-1}r_0\}. \tag{1}$$

Krylov subspaces have several key properties:

- **Dimension**: The dimension of the Krylov subspace is at most n, where n is the number of iterations or the desired dimension of approximation.
- **Basis**: The vectors forming the basis of the Krylov subspace are obtained by repeatedly applying the matrix A to the initial vector r_0.

- **Approximation Space**: Krylov subspaces provide a way to approximate the action of A on any vector. By using a low-dimensional Krylov subspace, we can obtain an approximation to the action of A that can be computationally more efficient than directly computing A times a vector.
- **Convergence**: Iterative methods like the Conjugate Gradient method utilize Krylov subspaces to iteratively approximate the solution of linear systems. The convergence properties of these methods are related to the properties of the Krylov subspaces, such as their dimension and the properties of the matrix A.

Let's delve into a bit more detail on how the Krylov subspace is utilized in the Conjugate Gradient method. The vectors generated within the Krylov subspace, $K_n(A, r_0)$, are not only linear combinations of powers of the initial residual r_0 but also inherently orthogonal to each other concerning the matrix A. This orthogonality property is leveraged in the Conjugate Gradient method to ensure that the search directions at each iteration are conjugate to each other. In the context of the Conjugate Gradient method, when we say that vectors are conjugate to each other, we mean that they are orthogonal with respect to the matrix A being used in the linear system. Mathematically, two vectors u and v are considered conjugate with respect to a symmetric positive definite matrix A if their inner product (also known as the dot product) satisfies the following condition:

$$u^T Av = 0. \tag{2}$$

This condition indicates that the vectors u and v are perpendicular to each other in the space defined by the matrix A. In other words, they are orthogonal with respect to the inner product induced by A.

Therefore, if p_k and p_{k+1} are two consecutive search directions in the CG algorithm, then $p_k^T A p_{k+1} = 0$. Conjugacy ensures that the CG method moves efficiently through the search space, avoiding redundant directions and converging to the solution in fewer iterations.

The vectors generated in the Krylov subspace form a basis for constructing approximate solutions to the linear system. Krylov subspace methods iteratively refine these approximate solutions to converge towards the true solution. The iterative refinement process involves updating the current approximation to the solution based on the residual error of the linear system. The solution gradually converges towards the true solution by improving the approximation using information from the Krylov subspace.

The choice of initial vector and the size of the Krylov subspace can influence the convergence and efficiency of Krylov subspace methods. Additionally, the selection of the orthogonalization method and other algorithmic parameters can impact the performance and robustness of these iterative solvers.

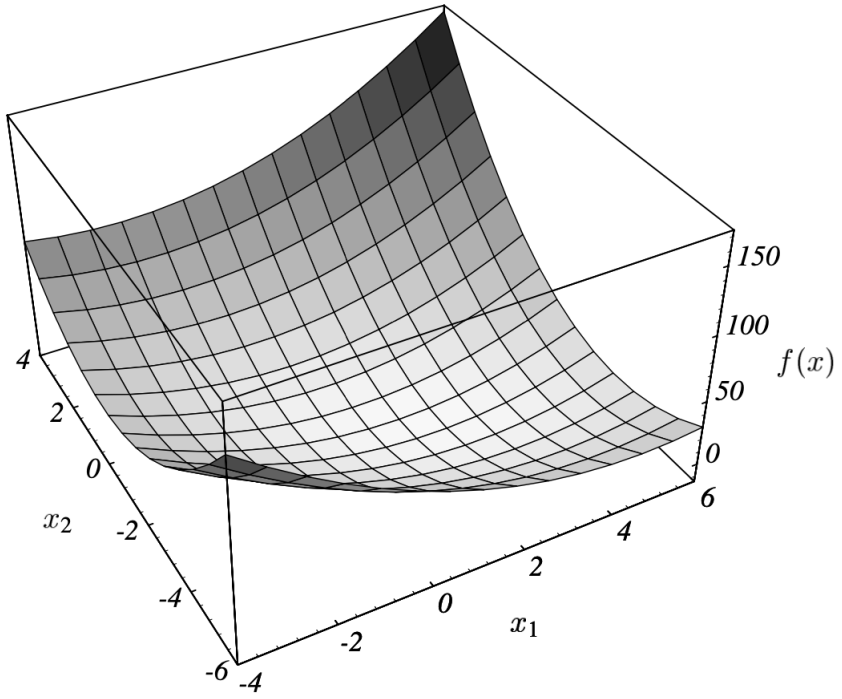

Figure 2. The graph of quadratic form $f(x)$. The solution of the $Ax = b$ gives the minimum point to the surface

2.4.1 The Conjugate Gradient method

Iterative methods play a crucial role in efficiently solving large systems of linear equations. One such iterative method is the Conjugate Gradient (CG) algorithm, which is widely used due to its desirable convergence properties and low memory requirements. It was first introduced by Hestenes and Stiefel in 1952. It has since become a widely used method in numerical linear algebra due to its efficiency and effectiveness on large and sparse matrices. The Conjugate Gradient method iteratively finds the solution by minimizing the residual error through a sequence of conjugate directions. However, solving large-scale systems using Conjugate Gradient can be computationally demanding and time-consuming.

The method works by minimizing a quadratic form (Figure 2) defined by the linear equation system. Instead of explicitly forming and manipulating the matrix A, the CG method operates within the Krylov subspace to iteratively refine the solution. It starts with an initial guess x_0 and iteratively updates the solution x by moving in the conjugate direction p within the Krylov subspace. At each iteration, the process computes a search

direction conjugate to the previous search direction and uses this direction to update the solution. The method converges in at most n steps, where n is the dimension of the matrix if no rounding errors occur.

The convex optimization problem in quadratic form can be written as:

$$f(x) = 1/2x^T Ax - b^T x + c. \tag{3}$$

Here A is matrix x, and b are vectors, and c is scalar. If we consider that A is symmetric and positive, define $f(x)$ can be minimized by the solution (Figure 2):

$$Ax = b. \tag{4}$$

In the algorithm 1, A is the coefficient matrix, b is the right-hand side vector, x is the n-th approximate solution, r is the n-th residual, d is the n-th search direction considering we have iteration n, and $alpha$ and $beta$ are scalar coefficients. The algorithm iteratively updates the approximate solution x by finding the optimal search direction d and the optimal step size α and then uses them to compute the new approximation x. This process continues until the residual r becomes sufficiently small, indicating that a good approximation has been found.[FR64, Lan52].

Algorithm 1 Conjugate Gradient algorithm

Ensure: A is a symmetric positive-define matrix
  $r \leftarrow b - Ax$
  $d \leftarrow r$
  $\delta \leftarrow r^T r$
  while not converged **do**
    $q \leftarrow Ad$
    $\alpha \leftarrow \dfrac{\delta}{d^T q}$
    $x \leftarrow x + \alpha d$
    $r \leftarrow r - \alpha q$
    $\delta_{old} \leftarrow \delta$
    $\delta \leftarrow r^T r$
    $\beta \leftarrow \dfrac{\delta}{\delta_{old}}$
    $d \leftarrow r + \beta d$
  end while
  return x

The Conjugate Gradient (CG) method theoretically converges to the exact solution of the linear system in at most n iterations for symmetric positive definite matrices. However, in practice, the number of iterations can be influenced by several factors:

- **Numerical Errors**: In real-world computations, numerical errors can accumulate during the iterative process, leading to deviations from the theoretical behavior. These errors can arise from finite precision arithmetic, round-off errors, and truncation errors, among others.
- **Ill-Conditioned Matrices**: Despite being designed for symmetric positive definite matrices, ill-conditioned matrices can pose challenges to convergence. In such cases, the number of iterations required for convergence may increase, and the convergence may not be as quick as for well-conditioned matrices.
- **Preconditioning Quality**: The effectiveness of preconditioning significantly affects the convergence behavior of iterative solvers like CG. If the preconditioner is not sufficiently effective in reducing the condition number of the matrix, the number of iterations needed for convergence may increase.
- **Convergence Criteria**: The choice of convergence criteria also impacts the number of iterations. If the convergence tolerance is set too tight, the iterative solver may require more iterations to meet the criteria, especially if the solution is close to the convergence threshold.
- **Matrix Structure and Eigenvalue Distribution**: The distribution of eigenvalues of the matrix A can influence the convergence behavior of the CG method. Clustering of eigenvalues near zero can lead to slow convergence, while widely spread eigenvalues can accelerate convergence.

Preconditioners can be applied to accelerate the algorithm's convergence, transforming the original matrix into a new matrix with a lower condition number. The condition number of a matrix measures how sensitive its solution is to perturbations in its coefficients or right-hand side. A matrix with a high condition number is more sensitive to perturbations, and thus, it is more difficult to solve accurately.

The Conjugate Gradient method is efficient in solving large and sparse systems of equations as it only needs the matrix-vector product and can be computed in parallel using OpenCL. This makes the method suitable for scientific and engineering applications such as finite element analysis, image processing, and machine learning.

The Conjugate Gradient has been extensively studied and optimized over the years, and many variations and improvements have been proposed. These include preconditioned Conjugate Gradient methods, which use a preconditioner to improve the convergence rate of the algorithm, and restarted Conjugate Gradient methods, which periodically restart the algorithm with a new initial guess to enhance convergence of difficult problems.

2.4.2 Generalized Minimal Residual Method

One of the successful Krylov methods designed to solve large nonsymmetric sparse linear systems of equations of the form $Ax = b$ is Generalized Minimal Residual Method (GMRES) method [WZ94]. GMRES iteratively constructs an orthogonal basis for the Krylov subspace generated by the matrix A and the initial residual vector $r_0 = b - Ax_0$, where x_0 is the initial guess. GMRES iteratively improves the approximation to the solution by minimizing the residual error in this subspace. The number of iterations required for convergence depends on factors such as the condition number of the matrix A, the choice of preconditioner, and the convergence criterion.

Orthogonalization is a key aspect of GMRES. It ensures that search directions are linearly independent. GMRES makes new search directions orthogonal to old ones. It does this during the iterative process to keep orthogonality and numerical stability. GMRES commonly uses Gram-Schmidt orthogonalization. It is used to make the basis vectors of the Krylov subspace orthogonal. However, this process can lead to loss of orthogonality. This is especially true for ill-conditioned matrices. In Gram-Schmidt orthogonalization [BG22], the method stores the residual vector at each iteration. This allows for cheaper re-initialization after each cycle of the restarted method GMRES(m). This is especially helpful for the restarted method GMRES(m). In this method, the process iterates. It is periodically restarted with a new guess. We need to re-initialize because we lose orthogonality. This loss can happen over long iterations. To fix the loss of orthogonality and save memory, GMRES often uses restarting. The iterative process is periodically restarted with a new initial guess.

One of the disadvantages of the GMRES method is that it might not perform well when applied to linear systems of ill-posed problems. Ill-posed problems contain error-contaminated data in the right-hand side vector [MRH14]. The problem is called an ill-posed problem if one of the following criteria is not met: the existence of the solution, the uniqueness of the solution, or the stability of the solution. Flexible GMRES performs better in such problems by allowing dynamic preconditioning. Another disadvantage of the GMRES method is memory usage. As all the previous residual vectors are stored to maintain the orthogonalization, it might be impractical for large systems. Despite some weaknesses in the GMRES method, it is still very well-suited and robust for non-symmetric matrices and for systems where the direct solvers are not effective.

2.4.3 Flexible Generalized Minimal Residual Method

Flexible GMRES (FGMRES), also introduced by Yousef Saad in 1993, is an extension of the GMRES (Generalized Minimal Residual) method. It is designed to handle the cases where the preconditioner may vary during the iterations. This characteristic of the FGMRES method makes it suitable for problems where a fixed preconditioner cannot be easily maintained. The algorithm follows the same processes as GMRES. The difference

is that, like GMRES, FGMRES builds a Krylov subspace incrementally but requires additional storage for the changing intermediate-step residuals. FGMRES stores each new search vector and the vector resulting from applying a preconditioner. This allows the solution to be restored to the original unpreconditioned system at the end of the algorithm. Because of high memory requirements, restarting the algorithm might be required for large-scale problems.

2.5 Domain Decomposition Methods (DDM)

Choosing between a direct solver or an iterative solver for large problems is not trivial. Iterative solvers are regarded as highly memory efficient, especially for large-scale problems; however, it is observed that the choice of an iterative solver is highly specific to the problem. Iterative solvers do not work very well for problems with highly ill-conditioned matrices. On the other hand, direct solvers are considered very robust and independent of the problem; however, they are seriously limited due to the large memory requirement [RK09]. In such cases, domain decomposition methods come into play to improve the following properties:

- **Parallel Computation**: Domain decomposition methods divide a large computational domain into smaller subdomains. This allows different processors to handle separate subdomains concurrently, reducing computation time significantly.
- **Memory Management**: Large problems typically require substantial memory, which might exceed the capacity of a single processor or node. By dividing the domain, each processor handles a smaller part, making the memory requirements more manageable.
- **Scalability**: These methods enhance the scalability of algorithms to large systems, allowing them to efficiently utilize the increasing number of processors in modern computing environments.
- **Convergence Rate Improvement**: In iterative methods, domain decomposition can be used as a preconditioner to improve the convergence rates of iterative solvers.

Domain Decomposition Methods are a class of numerical techniques used to solve PDEs on large and complex computational domains. The idea behind these methods is to divide the computational domain into smaller regions called subdomains. Each subdomain is then solved separately, and the solutions from each subdomain are combined to obtain the solution for the entire domain. Doing this would enable the preconditioner to improve the properties of the matrix system globally, in particular, eigenvalue distribution, and hence contribute toward speeding up the convergence of the global iterative method.

The most common types of domain decomposition methods are the Schwarz method, the substructuring method, and the mortar method.

In the context of Krylov subspace methods, domain decomposition methods can be used to improve the efficiency of the solver for large linear systems arising from PDEs. The GMRES, FGMRES, and CG mentioned above are iterative methods that require the solution of a linear system at each iteration. Domain decomposition methods can be used to parallelize the solution of the linear system, thereby reducing the time required to solve the problem.

There are two main types of Domain Decomposition methods: overlapping and non-overlapping [TW04]. The non-overlapping domain decomposition sees a large computational domain divided into a number of subdomains. All the same, the subdivisions share the boundaries but don't overlap. Interactions between the subdomains are so important that the interfaces require special treatment to ensure continuity and transfer of information in the right manner. It gives us high efficiency in parallel computation as subdomains can be solved independently. Also reduces complexity per subdomain, making it manageable to use direct solvers for each. Non-overlapping domain decomposition methods demonstrate slower convergence than overlapping domain decomposition methods.

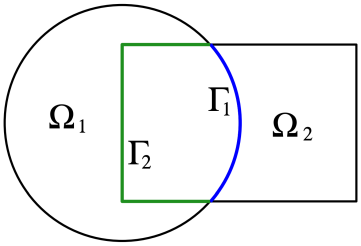

Figure 3. Overlapping domains

On the other hand, overlapping methods, also known as Schwarz methods, involve subdomains overlapping by some margin[Cai03]. The overlap region provides a better correction of errors and a smoother transfer of information for the boundaries of each subdomain. Every subdomain has part of its neighboring subdomains. In particular, the width of the overlap significantly impacts the convergence and stability of the method. In non-overlapping methods, the local problems are solved independently from other subdomains. However, the overlapping regions ensure that each subdomain solver would have more accurate information on the boundary, which would further improve the effectiveness of the solvers. We benefit from improved convergence due to a better exchange of information in the overlapping regions and better robustness to problem parameter variations or discretization errors.

A simple graphical visualization of overlapping domain decomposition methods is shown in Figure 3. It shows a domain formed by the union of a circle and a rectangle,

which is donated as $\Omega = \Omega_1 \cup \Omega_2$ The Additive Schwarz (AS), Restricted Additive Schwarz (RAS), and Restricted Additive Average Schwarz (RAAS) Schwarz methods are all examples of overlapping domain decomposition methods. The next section will describe RAAS in more detail because of its usage in our practical implementation.

2.5.1 The Restricted Additive Average Schwarz method

The standard Additive Schwarz method divides the computational domain into overlapping subdomains with minimal overlap. The minimal overlap consists of a single layer of nodes in the discretization grid. Having a larger extended overlap between subdomains in domain decomposition methods can improve the convergence rate of iterative solvers by reducing the number of iterations needed to reach a solution. In particular, this is known to be true in the case of the Helmholtz problem (see Subsection 3.4) due to the spectral properties of its stiffness matrix obtained through the discretization. The overlap allows for a smoother transition of information between adjacent subdomains, which can help to reduce the influence of interface errors. The minimal overlap is, therefore, extended. Then, each subdomain solves the local problem independently on the extended subdomain. However, the Restricted Additive Schwarz methods modify the Additive Schwarz method by introducing a restriction operator [EG03]. The restriction operator limits the updates of each subdomain after the subdomain solution to its interior nodes. The Additive Average Schwar (AAS) method updates the values on the overlap with averages from each neighboring subdomain instead of adding up the separate contributions. With the Restricted Additive Average Schwarz (RAAS) method, averaging is done only on the minimal overlap. Outside of the minimal overlap – the extended overlap values – are used only as an input for the extended subdomain problem solution.

This adjustment leads to improved performance in terms of both convergence and stability, particularly when dealing with certain large-scale parallel computations, like the discretization from the Helmholtz equation [GSV17a].

Figure 4 illustrates the partitioning of the unit square into four overlapping subdomains. The symbol Δ represents the extent of overlap between these subdomains, while red dashed lines demonstrate the minimal overlaps. Consequently, each extended subdomain will have a dimension of $0.5 + \Delta$. The solver will then be employed within these expanded regions. Upon completion of the subdomain solutions, the Restricted Additive Average Schwarz method will compute an average from the solutions within the minimal overlap areas while discarding the solutions from nodes located in the extended overlaps.

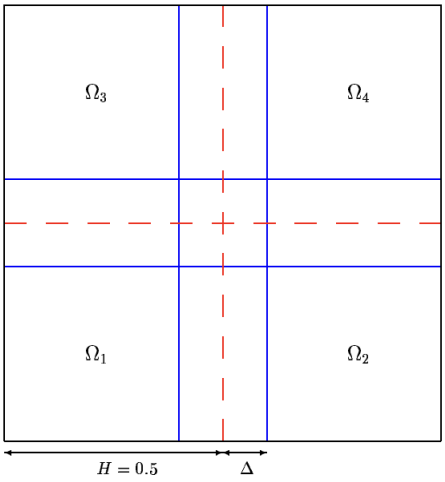

Figure 4. An example of partitioning the unit square into four overlapping subdomains with an extended overlap of width Δ.

3 Data and methods

This section will provide detailed insights into the implementation of the solvers. It will also introduce the experimental data, discuss the generation of Helmholtz matrices, and outline the specifications of the machines used for conducting these experiments.

3.1 Accelerating Mathematical Operations in the Conjugate Gradient method

Our primary aim is to develop a CG subsolver in Python using the PyOpenCL library, with the intention of surpassing other solvers implemented using NumPy and Scipy libraries. As previously mentioned, this approach enhances code readability, simplifying maintenance and potential future enhancements. In our application, CG serves as an inner solver to address subdomains generated from the Restricted Additive Average Schwarz method, with FGMRES being our outer Krylov subspace solver. The core of the Conjugate Gradient method involves several fundamental operations such as vector additions, scalar-vector multiplications, dot products, and matrix-vector multiplications. When these operations are performed on large data sets or high-dimensional spaces, they can become computationally intensive. Therefore, our implementation consists of five different kernels tailored to make the most of the GPU's computing power for various mathematical operations. As the Helmholtz matrices can be either real or complex, we have 2 different sets of kernels to support operations involving both real and complex values. These kernels enhance performance by enabling faster convergence and

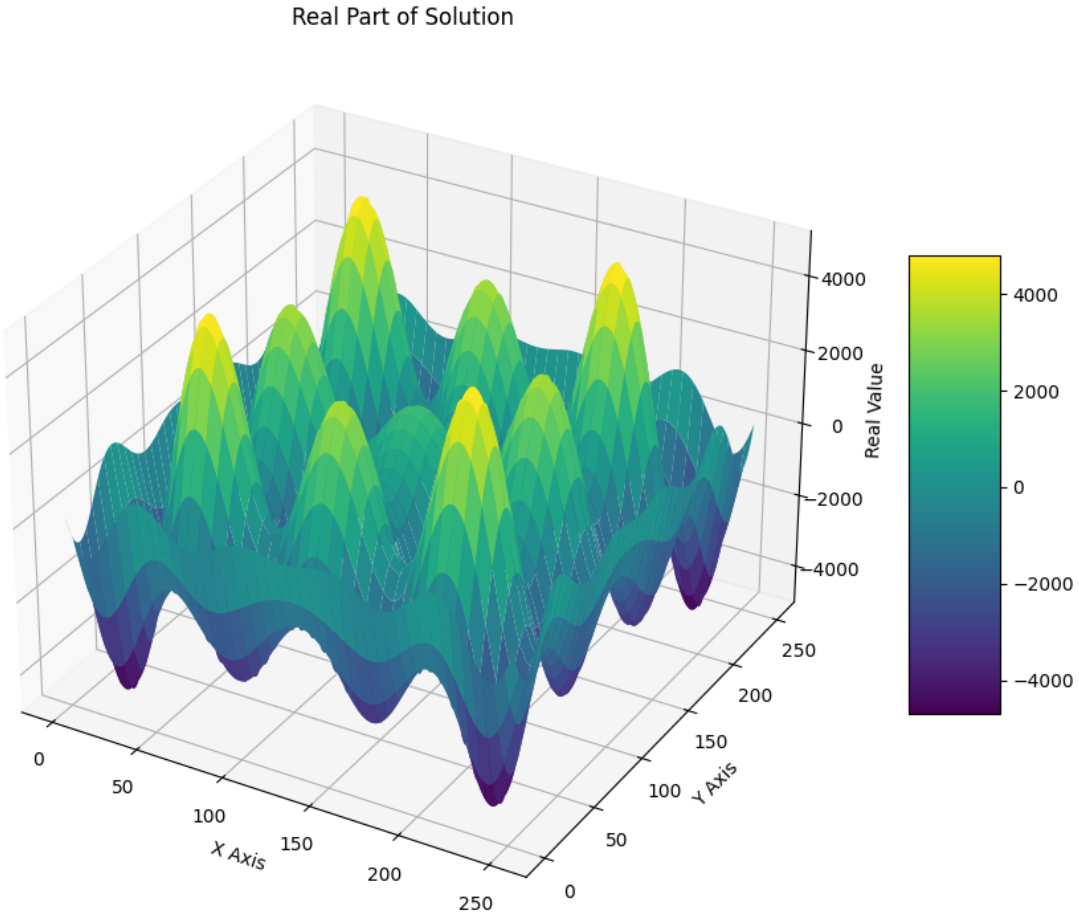

Figure 5. 3D Visualization of the Helmholtz Equation Solution

facilitating the handling of larger problems that are typical in scientific computing. It is also worth mentioning that the kernels have support for multiple RHSs where the actual performance improvement can be observed. Below, we list these kernels and explain where they are used in the Conjugate Gradient method:

- **Axpy** - This kernel is used to calculate $x + \alpha d$ and $r - \alpha q$ in the algorithm 1.
- **Aypx** - This kernel is used to calculate $r + \beta d$ in the algorithm 1.
- **Spmv** - This kernel is used to compute matrix-vector multiplication. In the CG algorithm 1, this kernel calculates Ax and Ad.

- **Sub** - This kernel calculates vector subtraction. In the algorithm 1, $b - Ax$ uses sub kernel where we already know the result of Ax.
- **VDot** - This kernel calculates the vector-vector multiplication. It is used to calculate $delta$ and dq dot operations in the algorithm 1.

The mentioned kernels are written in a language based on the C programming language. However, OpenCL C, the language used for writing kernels, has some differences and restrictions compared to standard C. These kernels can be utilized within different tools to leverage the computation power of GPU. Moreover, you can also see the kernel code of the Axpy operation for real numbers in the listing 3.

```
__kernel void axpy(__global float *x,
                   __global float *y,
                   __constant float *a,
                   const int aSign,
                   const int size) {

    const int __idx = get_global_id(0);
    if (__idx < size) {
        if (aSign)
            for (int r = 0; r < N_RHS; r++)
                y[__idx + r * size] += a[r] * x[__idx + r * size];
        else
            for (int r = 0; r < N_RHS; r++)
                y[__idx + r * size] -= a[r] * x[__idx + r * size];
    }
}
```

Listing 3. Kernel code of Axpy operation for real numbers

3.2 Host-device communication

Host-device communication involves transferring data between the host (typically the CPU) and the device (e.g., GPU) before and after kernel execution. PyOpenCL provides APIs for managing data transfer between the host and the device, as well as for queuing kernel execution commands. APIs provided by PyOpenCL allow us to create a context with a specified GPU. The chosen GPU is responsible for the execution of the kernel. I have separated the initialization of the context and loading of kernels from the actual code in order to avoid the initialization time of context and loading of kernels for each iteration of the FGMRES method.

3.3 GPGPU PyOpenCL solver for Block Conjugate Gradient method

Our implementation presents 2 solutions for the Block Conjugate Gradient algorithm with support for complex-valued inputs and multiple right-hand-side. One of them solves the linear equation of $Ax = b$ with a single right-hand side. Here, the x is a single vector that satisfies the equation. When multiple right-hand-sides (RHS) are involved, the problem extends to solving $AX = B$, where B is now a matrix of RHS vectors, and X is a matrix of solution vectors corresponding to each RHS vector. While solving the linear equation with multiple RHSs, computations are done by sending the whole RHS vector of the original equation with the submatrix, and the results are mapped into the whole solution vector. Our setup allows us to use multiple RHS solutions having the exact same subdomains. This allows us to get rid of the iterations through the subdomains.

3.4 Helmholtz matrices

Helmholtz matrices arise when discretizing the Helmholtz equation using numerical methods such as finite difference methods, finite element methods (FEM), or boundary element methods (BEM) [BDJT21]. The matrix represents the linear system that approximates the Helmholtz equation. Helmholtz matrices are an essential aspect of numerical methods related to the Helmholtz equation, a fundamental partial differential equation (PDE). In mathematical physics, these matrices represent wave propagation phenomena, such as sound and electromagnetic waves.

Helmholtz matrices are typically large and sparse because discretization of the domain involves many grid points or elements but has interaction with only those of its local neighbors. These matrices can be indefinite or even nearly singular, especially at higher frequencies, which makes them challenging for standard linear solvers [EG11].

We have a function implemented in Python using NumPy that discretizes Helmholtz Equation in a square domain $\Omega = (0, 1)^2$ using the Finite Element Method (FEM) with Robin boundary conditions on all sides before the start of the solution process. The Helmholtz equation is defined as the following [GSV17b]:

$$-\Delta u - (k^2 + i\epsilon)u = f \quad \text{on } \Omega, \tag{5}$$

where Δ is the Laplacian, i is the imaginary unit, ϵ is a damping parameter, and k wave number. On the boundary $\partial\Omega$, the condition is given by:

$$\partial_n u - iku = 0 \quad \text{on } \partial\Omega, \tag{6}$$

where $\partial_n u$ denotes the normal derivative of u.

The implemented function has the following inputs that can be modified to explore more about the Helmholtz equations and their convergence behavior in different setups:

- N: Number of grid points in each direction on the square domain Ω.

- k: Wave number, $k = \frac{\omega}{c}$ with ω representing the frequency of a wave and c the variable wave speed.
- ϵ: The damping parameter quantifies a wave's energy loss due to absorption, scattering, or other effects in the medium.

For simplicity, we are using uniform wave speed in our experiments. However, having a uniform wave speed is also quite common in both theoretical and applied physics. Some examples of Helmholtz problems where the wave speed is almost uniform are Underground Seismic Waves, Medical Ultrasound, and Astronomical Observations. How these changes in these parameters affect Helmholtz matrices' generation will be described later. The output of the function is a list of exact same-size sparse matrices. The reason for having exact same-size matrices is to ease the work of the domain decomposition method and to be able to solve the problem with multiple right-hand-sides.

Figure 5 illustrates the computed real part of the solution to the Helmholtz equation. The visualization highlights the dynamic behavior of wave propagation within the medium, showing variations in amplitude and phase that are critical to understanding the physical phenomena being modeled. Each peak and trough is represented with a color gradient, which corresponds to the real value intensity, ranging from -4000 to 4000, as detailed in the accompanying color bar.

3.5 Experimental Environments

Experiments and benchmarks are conducted at the High-Performance Computing (HPC) Center of Tartu University[Uni18]. For these tests, I utilized the Rocket cluster at the HPC. I specifically requested the Falcon GPU nodes for computational tasks, excluding the Falcon 2 node due to the errors while running OpenCL code. Also, each of these nodes is equipped with 24x NVIDIA Tesla V100 GPUs. The Tesla V100 GPUs come with 5120 CUDA cores, which are fundamental to its ability to handle parallel processing tasks efficiently. Below, you will find the SBATCH script used to submit the job:

```
#!/bin/bash
#SBATCH -Jp_helm
#SBATCH --partition=gpu
#SBATCH --exclude=falcon2
#SBATCH --gres=gpu:tesla:4
##SBATCH -N 2
##SBATCH --ntasks-per-node=1
#SBATCH -t 10:00:00
##SBATCH --exclusive
##SBATCH --mem=47000
####SBATCH --nodelist=falcon
```

```
S=4
N=128
NIT=512

module load py-numpy/1.22.0
module load py-mpi4py/3.1.4
module load cuda/11.1.0
module load py-scipy/1.8.1
module load py-pyopencl/2020.2.2
module load cmake/3.23.1

cd /gpfs/space/home/ziya/distributed-systems
srun hostname
./p_h-PY_C-CL.py ${S} ${N} 2 ${NIT}
```

Additionally, you can see the modules that have been loaded to establish an environment suitable for executing the code. The py-pyopencl library is an essential module for executing OpenCL (Open Computing Language) programs in Python.

3.6 Evaluation

Four different subsolvers, namely Exact Subsolver (SciPy), PyOpenCL-CG for a single RHS, PyOpenCL-CG for multiple RHSs, and NumPy-CG, are compared in different terms. The time complexity of different implementations was the first metric, varying with the number of subdomains and their sizes. The influence of extended overlaps obtained from applying the Restricted Additive Average Schwarz preconditioner was the second metric to observe the convergence of the solution. Additionally, we performed tests with different wavenumbers to see how well the Krylov subspace methods, with the inner iterative solver CG and the outer iterative solver FGMRES, perform under varying medium properties.

4 Results

In this section, we will take a look at how outer solver FGMRES preconditioned with either different implementations of Conjugate Gradient method solvers or exact subdomain solutions perform in different setups and also show a comparison between solvers using evaluation metrics mentioned in evaluation subsection 3.6.

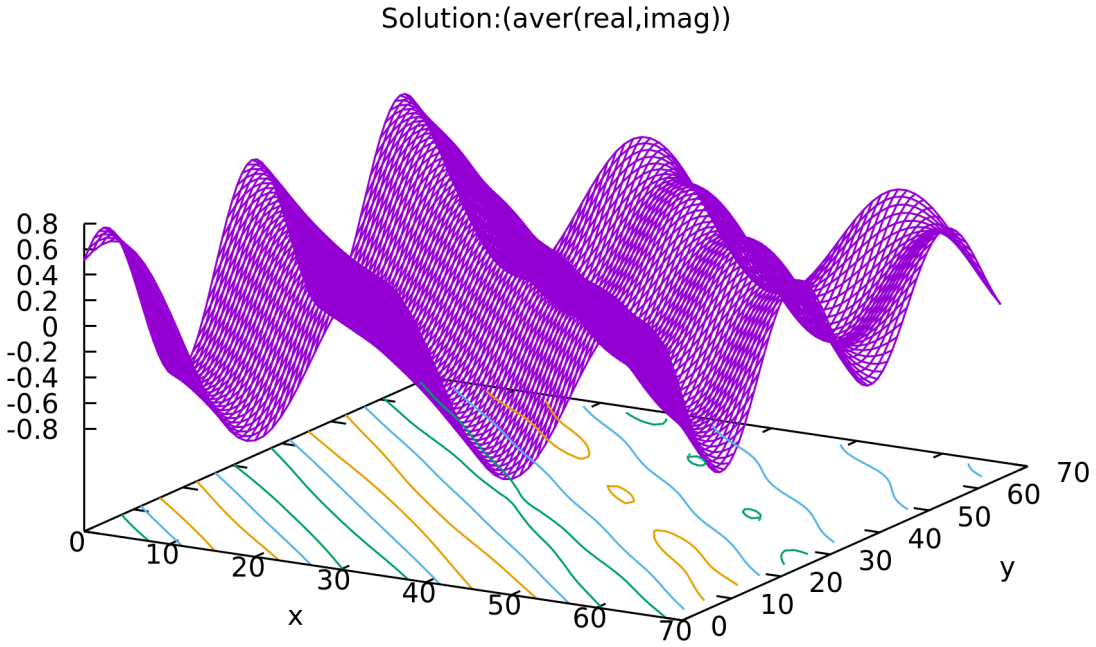

Figure 6. 3D Visualization of the Average Solution Showcasing the Propagation of a Frontal Wave Across the Domain

4.1 Time complexity of the solvers

As time complexity is one of the most useful indicators of performance, we have performed a series of experiments on the Falcon HPC cluster nodes to measure the execution times of various solvers. The graphs will display the comparison of execution times in different subdomain setups. These execution times are recorded in seconds. In the following experiments, we solve the Helmholtz problem with the FGMRES method as an outer iterative method that is preconditioned either by exact subdomain solutions in the RAAS method or different implementations of the CG method. The stopping criteria for the FGMRES method is the initial residual norm decreased by 6 orders of magnitude; we perform $4 * subdomain\_width$ CG iterations in subdomain solves. If not stated otherwise, we fix k and eps 20. The subdomain overlap, if not stated otherwise, is chosen

to be $(subdomain\_width/2 - 1)$. The initial guess of the CG method is chosen to be 0s for the experiments. Moreover, calculations are done for special RHS that produce a frontal wave moving across a medium. This wavefront starts at one corner and moves diagonally across to the opposite corner of the domain; see Figure 6. The graph also illustrates how the wave amplitude varies in a square domain.

4.1.1 Time complexity comparison in different subdomain setups

The first element in each tuple shows the number of subdomains, and the second element shows the width of each subdomain, which is the number of subdomain nodes in each direction. As the width of subdomains increases, particularly from 64 to 128, there is a substantial increase in the computational demands across all methods, see Figure 7.

The Scipy exact solver shows a steep exponential increase as the subdomain width increases. This dramatic rise suggests that the exact subsolves solution method's computational demands grow significantly with larger and more complex subdomain configurations. It starts with the lowest value but ends with the 2nd highest, indicating that it may not scale well with increasing problem size. A similar pattern is also observed in the CG solver written in Numpy. Although it shows good performance in small subdomain setups, the performance of the solver is decreased in bigger subdomains compared to other implementations.

The Performance curve for the PyOpenCL implementation with a single right-hand side starts with the worst performance in the case of 9 subdomains, having 27 subdomain nodes in each direction. However, it performs better as the subdomain number and size increase. That shows the scalability and efficiency of the solver in bigger subdomains which is typical for Helmholtz problems.

The best performance is obtained in PyOpenCL implementation of Block Conjugate Gradient method for multiple RHSs. We can see from the chart that multiple RHS implementations have always outperformed all the solvers.

4.1.2 Time complexity in a big subdomain setup

A similar experiment is performed in a bigger subdomain setup, having 256 subdomain nodes in each direction and 16 total subdomains, see Figure 8. The number of iterations for the CG method is set at 1000. This setting can impact performance, as too many iterations may waste resources. On the other hand, setting too few iterations might slow down convergence or, in some cases, prevent convergence completely. For example, while running the experiment within the same subdomain setup, having 500 CG iteration didn't converge to the solution because of a few iterations. The size of submatrices obtained in this setup was 260100, which means that the size of the original matrix is 4161600.

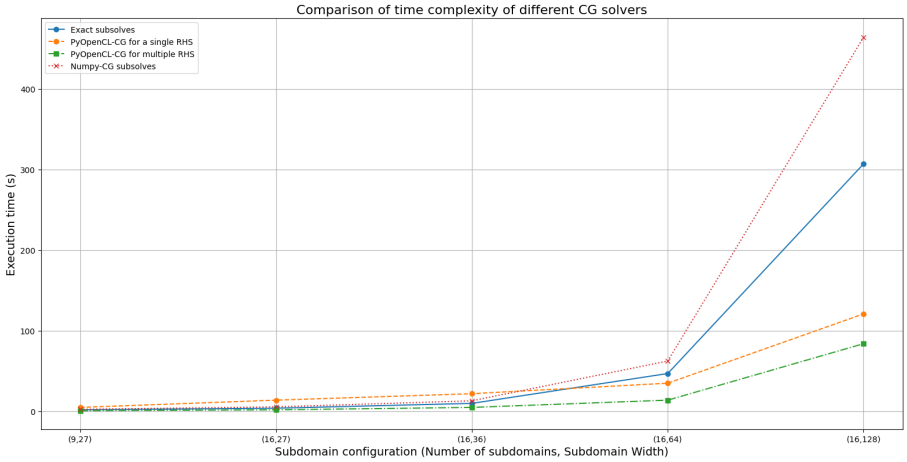

Figure 7. Comparison of time complexity of different CG solvers

The blue bar in the chart represents the execution time of the Scipy exact solver. In total, the exact solver took 5223 seconds (87 mins) to calculate the $Ax = b$ linear equation for 16 subdomains, which was 12 minutes faster than NumPy subsolves. Higher numbers are indicative of its computationally intensive nature. Exact subsolves often involve direct methods, which, while accurate, can be significantly slower, especially in complex or large-scale problems where approximation methods might be more efficient.

Numpy Solver solved the problem in 100 minutes, being the last in the ranking. It also took 21 iterations of the outer solver to converge to the true solution, while the exact subsolver only had 20 iterations to converge. Considering the Numpy CG solver is using approximate solutions, having almost the same number of outer iterations is quite impressive.

PyOpenCL solution for CG for multiple RHSs shows a significant reduction in execution time when solving for multiple RHS vectors simultaneously. With 21 outer iterations, it solved the problem in approximately 16 min that is more than 6 times faster than the Numpy solver.

PyOpenCL CG for a single RHS also performs considerably faster than exact subsolves and NumPy CG subsolves because of it is efficient usage of GPUs.

4.2 Subdomain overlap size

We have conducted 2 experiments with the size of subdomain overlaps. The first experiment illustrates how the number of outer iterations of the FGMRES method changes as subdomain overlap size increases. This will allow us to see how many iterations it takes to converge to the true solution. The second experiment shows how the execution

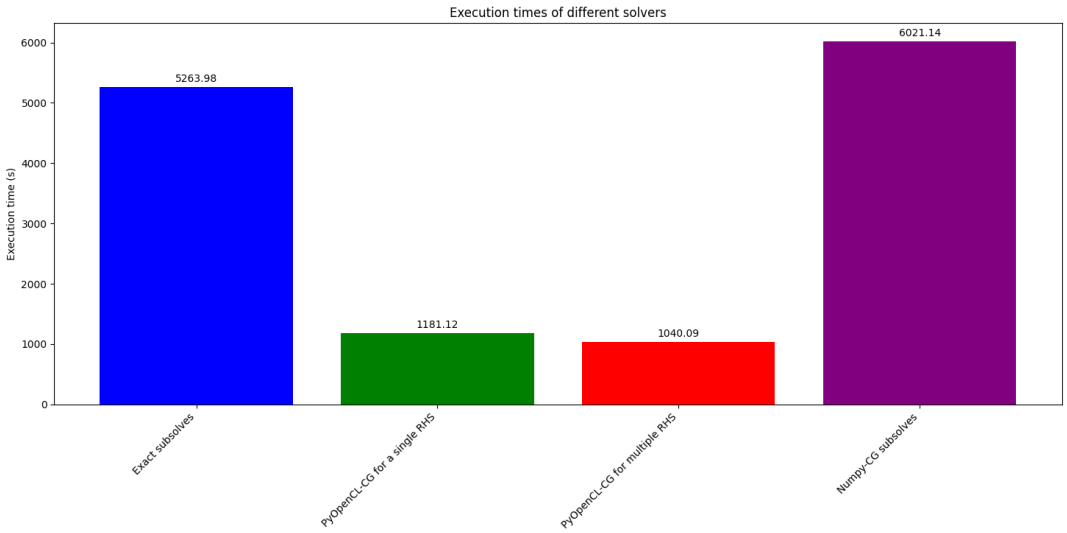

Figure 8. Execution times for CG solvers in bigger domain with 16 subdomains, each having the size of 256

time varies with different sizes of subdomain overlap for each solver. It will help us see if more overlaps lead to longer or shorter execution times for different solvers.

For these experiments, we have used 16 subdomains, each having a subdomain size of 64. The number of CG iterations was fixed to 256, wavenumber k and eps to 20.

4.2.1 The impact of subdomain overlaps on the convergence of solution

In figure 9, the number of outer iterations needed to converge across different subdomain overlaps is illustrated. The number of iterations needed to converge decreases as the size of subdomain overlap increases, stabilizing at 19 iterations from 21 subdomain overlap sizes onward. This suggests that having a higher subdomain overlap size can initially speed up the convergence. However, a bigger number of subdomain overlaps doesn't bring any benefit but increases the computational time to solve extended subdomains. Moreover, it is worth mentioning that none of the solvers converged when the number of overlap sizes was bigger than the subdomain size.

4.2.2 The impact of subdomain overlaps on the performance of solvers

The execution time for four different solvers was compared with the number of subdomain overlap sizes ranging from 1 to 60. The line plot displays the execution times for each solver as the number of subdomain layers increases; see Figure 10.

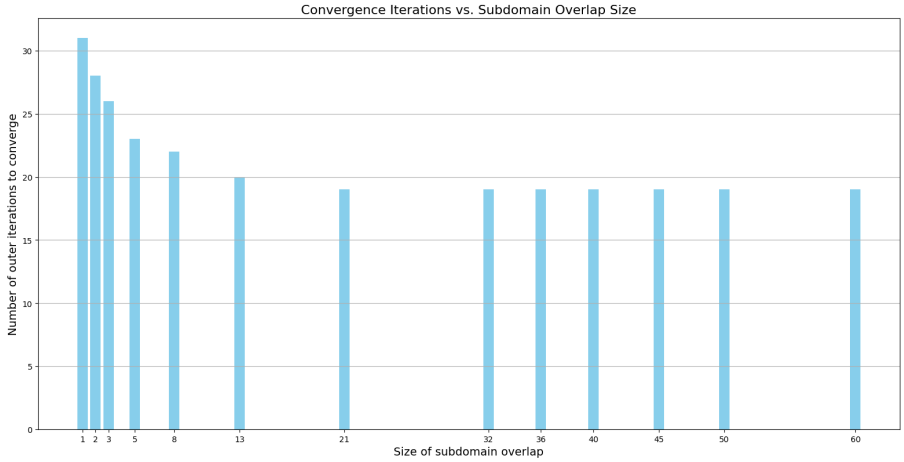

Figure 9. Convergence Iterations vs. Subdomain Overlap Size

Exact subsolves exhibited a relatively linear increase in execution time with increasing subdomain overlaps. The execution time was moderate at lower layer counts but escalated notably beyond 20 layers. For exact subsolves, one may expect that the computational overhead would grow in larger domains and, hence, be one of the reasons some efficiency gets lost with a larger overlap.

The execution time of PyOpenCL CG for multiple RHSs initially decreases when the number of subdomain overlap sizes changes from 1 to 13, indicating improved efficiency of the solver with moderate overlaps. As the width of the overlap increases, the number of outer iterations decreases, but the size of the subdomain problem grows. Beyond 13 layers, the execution time increased gradually, reflecting the diminishing returns in efficiency at higher overlaps.

Starting with execution times comparable to the Exact subsolves, Numpy CG's execution time increased significantly with more overlaps. The substantial rise in time after 20 layers was the steepest among the solvers, suggesting that while Numpy CG can handle lower overlaps efficiently, its performance degrades under larger computational loads typical of higher overlaps.

Overall, the data suggests that increasing the subdomain overlap size can initially reduce the computational complexity. Especially up to the point where the overlap doesn't overlap with the third subdomain overlap. In order to avoid such situations, we have a formula $(subdomain\_width/2-1)$ to calculate the maximum subdomain overlap without extending over the second subsequent subdomain. Also, excessive overlaps lead to a general increase in execution time across all solvers. This effect is most pronounced in solvers like Numpy CG, which do not inherently parallelize operations as effectively as PyOpenCL-based methods.

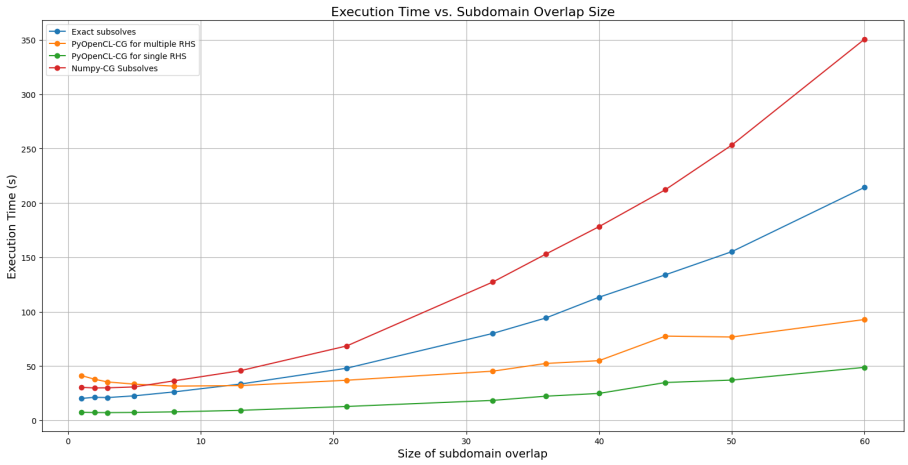

Figure 10. Execution Time vs. Subdomain Overlap Size

From the graphs, we can come to the conclusion that PyOpenCL CG implementation for multiple RHSs showed the best performance when the subdomain overlap size was 13, which is approximately one-fifth of the subdomain size.

4.3 Influence of the wave number to the convergence

As it was mentioned in earlier sections, we have a function to generate Helmholtz problems. One of the parameters of this function is wave number k, which influences the complexity of the domain. In this section, we are going to check the convergence of solvers while using different wave numbers. PyOpenCL-CG for multiple RHSs is the inner solver for this particular testing, FGMRES being the outer solver. For the experiment we have set the total number of subdomains to 16 and the size of each subdomain to 128. Moreover, the subdomain overlap size was set to 31, and the damping number ϵ to 20. The wavenumber varied from 5 to 150.

The execution time increases as the wavenumber increases from 5 to 50, which is expected due to the increasing complexity of the problem. Higher wavenumbers in Helmholtz equations often make the problems more oscillatory and harder to solve due to their impact on the eigenvalue distribution of the matrix. However, it is surprising that the solver computes the problem in less time when the wavenumber increases to 100 and 150. This behavior of the solver can be investigated further.

The trend for the number of outer iterations follows the same pattern as the execution time. Obviously, it takes more to solve the problem as the number of iterations increases.

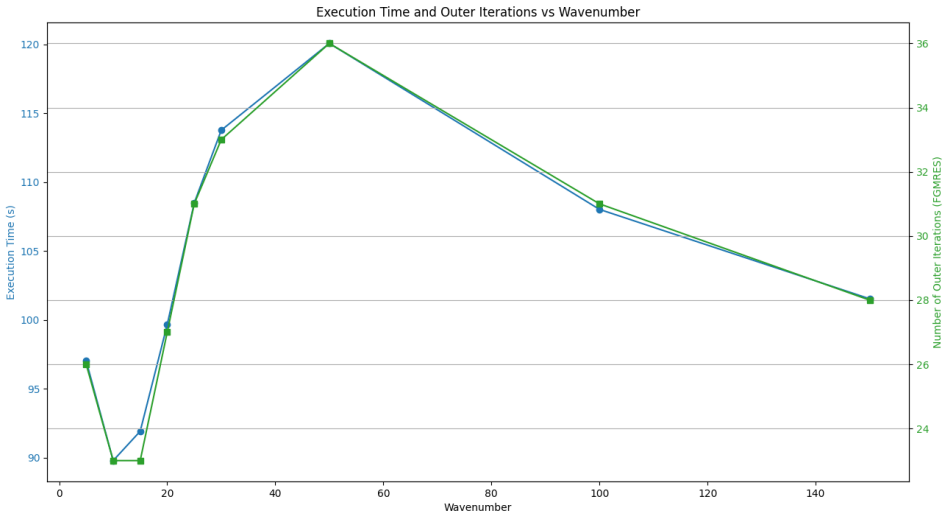

Figure 11. Wavenumber vs. Outer iterations and Execution Time

4.4 Weak scaling testing for PyOpenCL-CG for a single RHS

In preparation for weak scaling tests we have performed a series of experiments with keeping the subdomain size fixed while changing the number of subdomains. Note that here we still use a single node with a single GPU. Real weak scaling tests will be performed as a later contribution.

The weak scaling testing is performed on the implementation using the inner solver of PyOpenCL CG for the single RHS where the workload per subdomain remains constant, but the total size of the problem increases. For the testing, the subdomain size is fixed to 64, the number of CG iterations to 256, the number to 20, and the subdomain overlap size to the max, which is 31. You can also see the example of how the number of subdomains increases during the testing; see Figure 12. Total number of subdomains used for the testing are 16, 25, 36, 49, 64, 81, 100.

From the graph, we can see that execution time increases significantly as the number of subdomains increases. This increase appears to be more than linear, suggesting that adding more subdomains leads to disproportionately higher computation times. Some of the possible reasons for that increase are increased communication overhead, synchronization costs, or inefficient parallelization could contribute to this trend. As more subdomains are added, the complexity of managing them and the data exchange between them might increase. Also, it was surprising to see some of the iterations took significantly more time compared to others. For example, although the average time for iterations was 23 seconds while solving a 9x9 domain, some iterations took more than

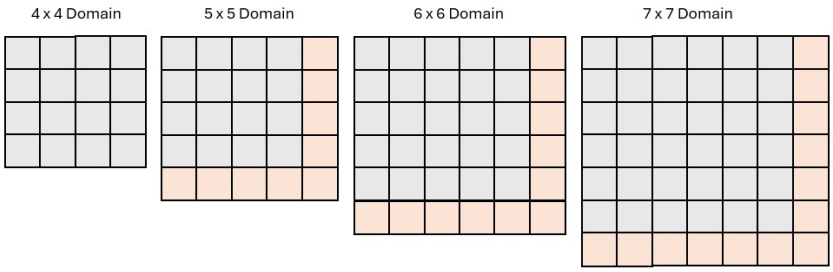

Figure 12. Example of subdomain size increase in weak scaling testing

100 seconds to compute.

Expectedly, the number of outer iterations (FGMRES iterations) also increases as the number of subdomains increases (Figure 13). This suggests that the problem becomes more challenging to solve iteratively as it is divided into more subdomains. It indicates that the preconditioner becomes less efficient as the domain is split into more parts. At the same time, of course, the potential for further parallelism is increasing, making it possible to add more nodes with GPUs to contribute to the solution process.

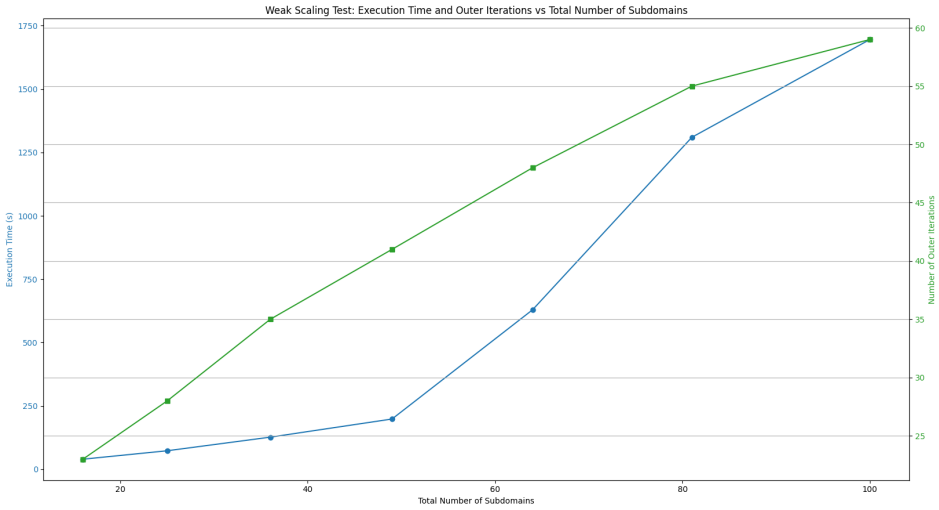

Figure 13. Number of Subdomains vs. Outer Iterations/ Execution time

5 Discussion

In this section, we compare our implementations using both exact subsolves and the Numpy CG solver with respect to time complexity. We also conducted experiments to examine the performance of these solvers under various subdomain configurations, as well as different characteristics of preconditioners and Helmholtz matrices. We will discuss the experiments that met our initial expectations and those that did not align with them. Additionally, we will provide insights into potential future research directions.

5.1 Performance of PyOpenCL implementations in terms of time complexity

As outlined in previous sections, our goal was to accelerate computational processes by harnessing the capabilities of GPUs. Our experiments demonstrated that the PyOpenCL implementation of the Conjugate Gradient method used as an inner solver for both single and multiple right-hand sides (RHSs), significantly outperformed traditional approaches. These results confirm that the computational power of GPUs was effectively utilized, resulting in substantial performance improvements, especially in bigger subdomain setups.

In those experiments, we have set different numbers for CG iterations considering the domain size. However, it is obvious that the number of iterations is not always precise. Setting a higher number of iterations for an algorithm might increase the

likelihood of convergence, but this approach can lead to resource wastage if the number is unnecessarily high. Conversely, setting too low a number may prevent the algorithm from converging at all, as it might not perform enough computations to reach an accurate enough solution. This balance requires careful tuning of the iteration count to optimize both computational efficiency and the reliability of results. One solution to that would be to have a stopping criteria for the algorithm. Each iteration of the CG algorithm typically involves a check to determine if the solution has converged sufficiently close to the true solution. This check usually requires calculating the norm of the residual (the difference between the left and right sides of the equation being solved), which is computationally intensive. Therefore, stopping criteria should be analyzed in detail to avoid performance issues.

5.2 Unexpected behavior of the solvers in higher wavenumbers

Algorithms were evaluated across various wavenumbers to assess their performance in different media. Surprisingly, higher wavenumbers resulted in fewer iterations and shorter execution times. This phenomenon can be explained as follows: Physically, higher wavenumbers correspond to shorter wavelengths, which may interact with the medium in ways that simplify the solution's structure, thereby reducing the complexity of the calculations required for convergence. Additionally, the matrix representation of the problem at higher wavenumbers might become better conditioned, facilitating faster convergence by reducing the number of iterations needed.

5.3 Future work

There are multiple directions in which further improvement could be done for the implemented solver. Currently, we are using even subdomains where the subdomain size is the same. However, an investigation can be done on how to adjust the solvers to work with uneven subdomains by adding a restriction and an interpolation operator, which translates between different sizes of subdomains into the setup used here for accelerated performance with multiple right-hand-sides solved in one go. For example, areas with higher complexity or finer details might require smaller subdomains to achieve more accurate results, whereas less complex areas can be represented with larger subdomains.

There can be enhancements to the code, such as using multiple GPUs per computation node. Each subdomain being computed independently and in parallel in different GPUs might result in better performance of the code.

Also, the implementation of the GPGPU CG solver can be enhanced by introducing stopping criteria based on the value of δ in the Algorithm 1. Within CG iterations, the value of δ (although non-monotonically) decreases. The value can be monitored for each right-hand side vector until all have reduced beneath a given threshold that guarantees the best performance for the overall (outer) iterative process.

The FGMRES solver can use different types of preconditioners at different iterations, greatly enhancing the adaptability and effectiveness of linear system solutions. In fact, it allows one to choose between different preconditioners suitably configured to the characteristics of different subdomains in the solution of a computational problem. In turn, using AI techniques, it is possible to conduct a dynamic choice or even design a preconditioner, which is grounded on real-time analysis of the solver's performance and on the changing conditions of the domain. AI will help automatically tune the strategy of preconditioning so that it optimizes the rates of convergence and computational efficiency. By selecting the most effective preconditioner for each iteration, AI can help minimize the number of iterations needed to reach convergence, thereby speeding up the solution process.

6 Conclusion

This thesis has demonstrated how GPU-accelerated Domain Decomposition methods efficiently solve the Helmholtz equation. More specifically, I have discussed the subsolvers, and it seems like there are huge benefits from the use of the Restricted Additive Average Schwarz (RAAS) method as a precondition. PyOpenCL implementation of the inner solver Conjugate Gradient method showed huge improvements in computational speed realized through GPU acceleration. The conducted experiments have mainly brought attention to PyOpenCL-CG implementation for multiple right-hand sides in terms of time complexity.

The solvers were also compared with different wave numbers and overlapping layers. The results obtained from solvers in different numbers of overlapping layers were helpful in finding the better values for the properties of the Restricted Additive Average Schwarz method.

This work underlines that, indeed, the remarkable potential is held by advanced GPU programming and domain decomposition in dealing with some of the complex computational challenges, and it opens up the door for numerical simulations to be furthered in this advanced skillset.

Overall, the presented results of this thesis prove the necessity of innovations in preconditioning strategies and the necessity of integration of GPU acceleration in computational science to advance toward more efficient and effective large-scale problem solvers.

7 Acknowledgements

I am grateful for the support and guidance I received during my studies at the University of Tartu. I would like to express my deepest appreciation to my supervisor, Professor Eero Vainikko, whose invaluable comments and unwavering support were essential to my work on this thesis. His expertise and insight have greatly enriched my learning experience and research.

I am also immensely thankful to the University of Tartu for providing access to the High-Performance Computing Center. This resource was crucial in allowing me to leverage the power of GPUs to conduct my experiments.

To all who supported me throughout my academic journey—thank you for your encouragement, for believing in my potential, and for helping me achieve this milestone.

References

[BDJT21] Niall Bootland, Victorita Dolean, Pierre Jolivet, and Pierre-Henri Tournier. A comparison of coarse spaces for helmholtz problems in the high frequency regime. *Computers & Mathematics with Applications*, 98:239–253, 2021.

[BG22] Oleg Balabanov and Laura Grigori. Randomized gram–schmidt process with application to gmres. *SIAM Journal on Scientific Computing*, 44(3):A1450–A1474, 2022.

[Cai03] X Cai. Overlapping domain decomposition methods. In *Advanced Topics in Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*, pages 57–95. Springer, 2003.

[Dry89] Maksymilian Dryja. An additive schwarz algorithm for two-and three-dimensional finite element elliptic problems. *Domain decomposition methods*, pages 168–172, 1989.

[DWL+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[EG03] Evridiki Efstathiou and Martin J Gander. Why restricted additive schwarz converges faster than additive schwarz. *BIT Numerical Mathematics*, 43(5):945–959, 2003.

[EG11] Oliver G Ernst and Martin J Gander. Why it is difficult to solve helmholtz problems with classical iterative methods. *Numerical analysis of multiscale problems*, pages 325–363, 2011.

[FR64] Reeves Fletcher and Colin M Reeves. Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154, 1964.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, 2011.

[GSV17a] Ivan Graham, Euan Spence, and Eero Vainikko. Domain decomposition preconditioning for high-frequency helmholtz problems with absorption. *Mathematics of Computation*, 86(307):2089–2127, 2017.

[GSV17b] Ivan G Graham, Euan A Spence, and Eero Vainikko. Recent results on domain decomposition preconditioning for the high-frequency helmholtz

equation using absorption. In *Modern solvers for Helmholtz problems*, pages 3–26. Springer, 2017.

[JAET23] Davron Aslonqulovich Juraev, Praveen Agarwal, Ebrahim Eldesoky Elsayed, and Nauryz Targyn. Applications of the helmholtz equation. *Advanced Engineering Days (AED)*, 8:28–30, 2023.

[KMSZ15] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, 2015.

[KSA+10] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of opencl programs. In *The fifth international workshop on automatic performance tuning*, volume 66, page 1, 2010.

[Lan52] Cornelius Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Standards*, 49(1):33–53, 1952.

[LS13] Jörg Liesen and Zdenek Strakos. *Krylov subspace methods: principles and analysis*. Numerical Mathematics and Scie, 2013.

[MRH14] Keiichi Morikuni, Lothar Reichel, and Ken Hayami. Fgmres for linear discrete ill-posed problems. *Applied Numerical Mathematics*, 75:175–187, 2014.

[Ope23] OpenAI. Chatgpt. Software available from OpenAI, 2023. Accessed: [insert date here].

[PyO] PyOpenCL. Pyopencl documentation.

[RK09] Mandhapati P Raju and Siddhartha Khaitan. Domain decomposition based high performance parallel computing. *arXiv preprint arXiv:0911.0910*, 2009.

[Saa93] Youcef Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

[Saa03] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[SGS10] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[She94] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. school of computer science. *Carnegie Mellon University, Pittsburgh, PA*, 15213:10, 1994.

[TS12] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 49:31, 2012.

[TW04] Andrea Toselli and Olof Widlund. *Domain decomposition methods-algorithms and theory*, volume 34. Springer Science & Business Media, 2004.

[Uni18] University of Tartu. Ut rocket, 2018.

[WZ94] Homer F Walker and Lu Zhou. A simpler gmres. *Numerical Linear Algebra with Applications*, 1(6):571–581, 1994.

Appendix

I. Access to Code

The code used to obtain the results can be found in this GitHub repository given below:
`https://github.com/ziyamammadov/conjugate-gradient-pyopencl`

II. Licence

Non-exclusive license to reproduce thesis and make thesis public

I, **Ziya Mammadov**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   GPU-accelerated Domain Decomposition Methods for Helmholtz equation,
   (title of thesis)

   supervised by Eero Vainikko.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Ziya Mammadov
15/05/2024