Minor Project Part B

TU Delft, Faculty of Electrical Engineering, Mathematics and Computer Science,
Minor computational science and engineering

GPU-Solvers for the Poisson and Helmholtz Equations

Authors:
Kaj Dockx
(4586425)
Matheus Coelho Silva
(4537823)
Julian Johnston
(4581075)

Supervisors:
Kees Vuik
Matthias Möller

Delft

February 20, 2019

Executive summary

In this report, a comparison is made between two different parallelizable solver packages for sparse linear systems, namely CuSolver [1] and Paralution [2]. The main objective is to create a clear overview of the strengths and weaknesses of the packages in respect to solving the linear systems arising from the discretization of differential equations, and in particular the Poisson and Helmholtz equations.

The Helmholtz equation arises naturally when looking at time harmonic solutions of the general wave equation. The inhomogeneous Helmholtz equation is given by [3] as:

$$-\nabla^2 u(\mathbf{r}) - k^2 u(\mathbf{r}) = w(\mathbf{r}),$$

where k is the wave number and w is a known function. In this work, the Helmholtz equation will be evaluated using Dirichlet boundary conditions [4] and a constant k on a two-dimensional domain. This equation is discretized using the finite difference method. For a square with a sidelength of 1, divided into N equidistant parts in each direction, the discretization yields the following result [5]:

$$-\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} - \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} - k^2 u_{i,j} = w(x_i, y_j).$$

Here, h equals $\frac{1}{N}$ and the indices i and j indicate the positions of respectively the x- and y-direction. Note that in order to ensure a good resolution, the product of k and h should be smaller than roughly 0.6. In order to make a one-dimensional vector of unknowns, the two indices i and j are replaced with a single index α using

$$\alpha = i + j(N-1).$$

Upon applying this to all $(N-1)^2$ vertex-centered unknown grid points, a system of $n = (N-1)^2$ linear equations is obtained which is denoted by $A\mathbf{u} = \mathbf{b}$. For high wave numbers, the Helmholtz equation becomes highly oscillatory and more complex numerical solvers are required to solve the system. For this purpose Krylov subspace methods are chosen [6].

Krylov subspace methods are based on extracting an approximation for $\mathbf{u}$ from the subspace $\mathbf{u}_0 + \mathcal{K}_m$ using another subspace $\mathcal{L}_m$ and the approximation $\mathbf{u} \approx \mathbf{u}_0 + q_{m-1}(A)$. Herein,

$$\mathcal{K}_m(A, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, ..., A^{m-1}\mathbf{r}_0\}$$

is the Krylov subspace and q_{m-1} is a polynomial of degree $m-1$. Different Krylov subspace methods depend on different choices for $\mathcal{L}_m$. In GMRES it is chosen to be $A\mathcal{K}_m(A, \mathbf{v}_1)$, with $\mathbf{v}_1 = \mathbf{r}_0/||\mathbf{r}_0||_2$. Using a preconditioning matrix M, an iterative method can be transformed to make it easier to solve. This is called preconditioning. The matrix can be applied in three ways: left, right and split preconditioning. Multigrid methods are also discussed, which rely on different grid sizes to recursively interpolate the solution until the solution for a small grid size is obtained.

Many of the iterative methods for solving large matrix equations can be sped up considerably through the use of parallelization. This is achieved by dividing computations between multiple processors, which can perform operations simultaneously. One possible implementation of such a parallel computer which has gained popularity in the scientific community in recent years is the Graphics Processing Unit (GPU). The architecture of a GPU makes it more efficient in dealing with compute bound problems than with memory bound problems. In other words, a program that runs on a GPU should ideally spend more time performing simultaneous calculations than transferring data between the GPU's many memory locations.

Writing code to run on an NVIDIA GPU is made easier with CUDA, NVIDIA's library for GPU programming. It introduces several abstractions for dealing with parallelism. A CUDA program consists of a grid of several blocks which execute independently from each other and can be scheduled on any of the available multiprocessors within a GPU. Each block is further divided into a three-dimensional array of threads. Threads within a block can all access the same Shared Memory but threads in different blocks can only share data through the global Device Memory.

Several high-level solver packages are available for doing sparse linear algebra with CUDA. The focus of this report is on solver packages which can be parallelized on NVIDIA GPUs with CUDA, such as cuSolver [1] and Paralution [2]. CuSolver includes only direct solvers, while Paralution has functions for direct solvers, iterative

solvers, and preconditioners. Performance comparisons are conducted using both packages to assess their usability for solving Poisson and Helmholtz problems. Both CuSolver and Paralution are used to solve the 2D Poisson problem via QR factorization, and then the same problem is also solved with Paralution's Algebraic Multigrid (AMG) solver. Finally, Paralution is used to solve the 2D Helmholtz problem for wave numbers 10, 50, and 100 with the GMRES solver (first unpreconditioned then preconditioned with Symmetric Gauss-Seidel). CuSolver is not included in the Helmholtz comparison because direct solvers are not enough for this problem. Instead, GPU run times are compared to CPU run times with the same Paralution code for insight on the speedup achieved.

CuSolver's QR solver outperforms that of Paralution for sparse matrices, because CuSolver's direct solvers are optimized to reduce fill-in of sparse matrices. However, Paralution's AMG solver is nonetheless able to solve the same 2D Poisson problem up to 57 times faster than CuSolver's QR method. Paralution's unpreconditioned GMRES is inefficient for solving the Helmholtz problem on a fine grid, but preconditioning with Symmetric Gauss-Seidel (SGS) greatly improves performance. Run times are mostly found to increase with increasing gridsize and wave number for the same stopping criteria, as expected. However, the wave number 10 case takes much longer than expected to solve when compared to the wavenumber 50 case. From the comparison between GPU and CPU run times, great speedup is achieved for larger problem sizes (as high as 108.5 for 2D Helmholtz with gridsize 501) but no improvement is found for smaller problems (as low as 0.2 for 2D Poisson with gridsize 51). This can be explained by the data transfer overhead incurred by parallelization.

Abstract

In this report, the Poisson and Helmholtz equations are solved in 2D using the GPU solver packages CuSolver and Paralution. The goal is firstly to compare the different available GPU solver packages and secondly to assess the added value of using a GPU instead of a CPU for such applications. The partial differential equations are discretized by way of the Finite Difference method and solved in a square domain with Dirichlet boundary conditions and a pulse source function. The Poisson problem is solved first via QR factorization with both packages, then by the Algebraic Multigrid (AMG) method with Paralution only. Next, the Helmholtz problem is solved for wavenumbers 10, 50, and 100 using Paralution's GMRES solver (first unpreconditioned then preconditioned with Symmetric Gauss-Seidel). Error analysis is conducted via the Method of Manufactured Solutions, and the two packages' results are compared in terms of run time, normalized L_2 norm of the final error, and code complexity.

CuSolver's direct solvers can solve the 2D Poisson problem up to 578 times faster than those of Paralution with the same final error. This occurs because CuSolver's direct solvers are optimized for sparse matrices, whereas Paralution's direct solvers internally convert sparse to dense matrices. However, Paralution's AMG method (which is unavailable on CuSolver) is nonetheless able to solve the same problem up to 57 times faster than CuSolver's QR method. Paralution's unpreconditioned GMRES is inefficient for solving the Helmholtz problem on a fine grid, but preconditioning with Symmetric Gauss-Seidel (SGS) greatly improves performance. Run times are mostly found to increase with increasing gridsize and wave number for the same stopping criteria, as expected. However, the wave number 10 case takes much longer than expected to solve when compared to the wavenumber 50 case. From the comparison between GPU and CPU run times, great speedup is achieved for larger problem sizes (as high as 108.5 for 2D Helmholtz with gridsize 501) but no improvement is found for smaller problems (as low as 0.2 for 2D Poisson with gridsize 51). This can be explained by the data transfer overhead incurred by parallelization.

In order to improve results in the future, a preconditioner that is better suited to the Helmholtz equation must be used. The method used requires a very large number of iterations and shows poor convergence behaviour. Since CuSolver has a limited amount of built-in solvers, not all of Paralution's solvers could be compared to other packages. It is therefore recommended to include other available GPU packages such as AMGCL or Rocalution in future comparisons.

Contents

1 Introduction

Over the past years, GPUs have become increasingly popular in the scientific community. Because of their highly parallel structure, they are well-suited for carrying out repetitive independent operations on large data sets. One application for which these properties prove very valuable is finding the numerical solution of partial differential equations, as solving these equations often utilizes an iterative process.

Two of these equations in particular - the Poisson and Helmholtz equations - arise frequently in analysis involving PDEs. Because of this, knowing the most efficient way of solving these problems is very valuable information. Larger problem sizes and increasing the wave number generally lead to more complications when searching for an approximate solution, so efficient solving is a must. Another sought-after factor is the ease of implementation of the solvers, as also less experienced programmers sometimes need to find numerical solutions to these problems. The goal is firstly to compare the different available GPU solver packages and secondly to assess the added value of using a GPU instead of a CPU for such applications.

In the next chapter, the Poisson and Helmholtz equations are first analyzed theoretically. Some theoretical background on discretization, matrix properties, and solving methods is explored in chapter 3. In chapter 4, GPU architecture is explained in detail. Chapter 5 explains methods of error analysis. In the last chapter, the results are given and compared.

This report is completed as part of the minor program Computational Science and Engineering from the Delft University of Technology.

2 Physics of the Helmholtz equation

2.1 Inhomogeneous wave equation

In classical physics, the (inhomogeneous) wave equation is used to describe the propagation of waves in a medium. The wave equation is a hyperbolic partial differential equation of a time- and space-dependent scalar function $f(\mathbf{r}, t)$. The general form of the inhomogeneous wave equation without damping is as follows [3]:

$$\nabla^2 f - \frac{1}{c^2}\frac{\partial^2 f}{\partial t^2} = G(\mathbf{r}, t) \qquad \text{in } \Omega, \tag{1}$$

where c is the propagation speed of the wave, $G(\mathbf{r}, t)$ is the known source function and ∇^2 is the Laplace operator. The function $G(\mathbf{r}, t)$ is called the source function because it usually describes the effect of sources in the medium through which the waves propagate.

The problems addressed in this paper are all two-dimensional. The spatial vector $\mathbf{r}$ will therefore consist of an x- and a y-coordinate. The three-dimensional wave equation is used to model a wide variety of problems. Examples are waves in a fluid, where f represents small displacements to a uniform mass density and pressure. Another example can be found in electrodynamics, where each component of vector fields satisfies the 3D wave equation (1) [4]. However, these will not be discussed in this report.

2.2 Derivation

The Helmholtz equation can be derived from the general wave equation by looking for time harmonic solutions of the wave equation [3]. Time harmonic waves are periodic waves for which the time variation is sinusoidal. A general form of a harmonic wave in the space/time domain is given by

$$f(\mathbf{r}, t) = U(\mathbf{r})\cos(\omega t + k\mathbf{r} + \varphi_0) = U(\mathbf{r})\cos(\omega t + \varphi(\mathbf{r})), \tag{2}$$

where $U(\mathbf{r})$ is the amplitude, ω the frequency, k the wave number and $\varphi(\mathbf{r})$ the phase of the wave. When working with harmonic waves, computations can be simplified by eliminating the time dependent part of the wave. In order to do so, a complex notation consisting of a time and space dependent part is used. The complex notation of the harmonic wave from equation (2) is given by

$$f(\mathbf{r}, t) = \mathrm{Re}\{U(\mathbf{r})e^{-i\varphi(\mathbf{r})}e^{-i\omega t}\} = \mathrm{Re}\{u(\mathbf{r})e^{-i\omega t}\}, \tag{3}$$

where $u(\mathbf{r})$ is the complex amplitude. As said earlier, the Helmholtz equation can be obtained by substituting the complex time harmonic function from equation (3) into the wave equation. Substituting the complex function gives

$$e^{-i\omega t}\nabla^2 u(\mathbf{r}) - \frac{1}{c^2}(-i\omega)^2 u(\mathbf{r})e^{-i\omega t} = g(\mathbf{r})e^{-i\omega t}$$

which results in

$$-\nabla^2 u(\mathbf{r}) - k^2 u(\mathbf{r}) = w(\mathbf{r}), \tag{4}$$

where $w = -g$. Usually, ∇^2 is replaced with the Δ symbol, which is called the Laplacian. Now, equation (4) is the (inhomogeneous) Helmholtz equation. Note that this equation is time-independent.

2.3 Physical meaning

The Helmholtz equation is a time-independent version of the general wave equation and is often called the reduced wave equation. When eliminating the time dependent part, as shown in the previous section, a switch is made from the time domain towards the frequency domain. The Helmholtz equation is thus a spatial differential equation for mono frequency waves [7]. A real-life signal, however, consists of multiple frequencies. When modelling such a signal using the Helmholtz equation, the signal is decomposed into a sum of time harmonic waves. For each of these harmonic waves, the Helmholtz equation can be evaluated. To return to a solution of the original signal all the results are again combined. Since the goal of this paper is to study the efficiency of different GPU-solvers, solutions for a single harmonic wave suffice.

The behaviour of the Helmholtz equation strongly depends on the wave number k. For low wave numbers, the Helmholtz equation behaves much like the Laplace equation. For high wave numbers however, the solution's behaviour becomes highly oscillatory. This drastically increases the complexity of the analytic and

numerical methods required to solve the problem [7]. In order to properly compare the different GPU-solvers, the case with high wave numbers is analyzed. Note that the wave number does not have to be constant but can vary within the control volume. In this paper however, the assumption is made that k is a real constant.

2.4 The Poisson Equation

The two-dimensional Poisson equation can be seen as the Helmholtz equation with $k = 0$, and is therefore a good starting point for solving the Helmholtz equation. The equation is given by

$$-\Delta u = f(x, y), \quad \text{in } \Omega. \tag{5}$$

2.5 Boundary conditions

In order to solve differential equations, they must be completed by compatible boundary conditions. As the goal of the project is to compare different solvers, the choice is made to use only one type of boundary condition: Dirichlet boundary conditions. I.e., $u = 0$ on $\partial\Omega$.

3 Matrix Properties and Iterative Solvers

3.1 Discretization of the Poisson and Helmholtz Equations

The domain in which the Poisson and Helmholtz equations will be evaluated is a square with side length 1. In each direction, the square is subdivided into N equidistant parts and therefore $N+1$ vertex-centered nodes. As a result, the spacing between each grid node is constant for both directions:

$$\Delta x = \Delta y = h,$$

where $h = \frac{1}{N}$. In this 2-dimensional domain, the Poisson and Helmholtz equations are discretized using the finite difference method. In order to discretize the equations, the second derivatives in each direction are required to evaluate the Laplacian. The second derivatives are approximated using a Taylor expansion which results in the following expression with respect to the x-direction [5]:

$$\frac{\partial^2 u(x,y)}{\partial x^2} = \frac{u(x-h,y) + 2u(x,y) - u(x+h,y)}{h^2} + \mathcal{O}(h^2).$$

Note that the error of this approximation is $\mathcal{O}(h^2)$. To indicate the positions of each grid node within the square, two indices are used: i for the x-direction, and j for the y-direction. The resulting discretization of the Helmholtz equation for an internal grid node (i,j) thus equals:

$$-\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} - \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} - k^2 u_{i,j} = f(x_i, y_j), \tag{6}$$

where here f denotes the source function. Note that the discretization of the Poisson equation is equal to that of the Helmholtz equation when substituting $k = 0$. Since in this paper only Dirichlet boundary conditions are used, the discretization of boundary points will not differ a lot from equation (6). The only difference is that the known value at the boundary has to be substituted for the correct point in equation (6).

By choice, the source function $f(x,y)$ represents a pulse in the middle of the domain. The discrete representation of $f(x,y)$ is therefore a vector that equals 0 everywhere except in the middle, where it equals $\frac{1}{h^2}$. Note that this function is a discrete approximation of Dirac's δ-function.

To avoid aliasing and to ensure good resolution, h should be sufficiently small. As the solutions are highly influenced by the wave number, the maximum grid size will directly depend on k. Since the time-harmonic solutions are sinusoidal, each wavelength, as a rule of thumb, should contain at least ten grid points. This results in the following relation between h and k:

$$k\,h < \frac{2\pi}{10}. \tag{7}$$

When applying the finite difference method for all grid points, a system of $n = (N-1)^2$ equations is obtained. Due to the Dirichlet condition at the boundaries, the equations corresponding to boundary nodes are not included. This system is denoted by $A\mathbf{u} = \mathbf{b}$, where $\mathbf{u}$ is the vector with the discrete solution at every node, $\mathbf{b}$ is the vector with the function $f(x_i, y_j)$ and possible boundary values at every node, and A is an $n \times n$ so-called sparse matrix. That is, it contains more zero than non-zero elements. Sparse matrices have many nice properties, one of which is that they can be stored occupying a minimal amount of space, because of the many zeroes. One way to store a sparse matrix is by using the Compressed Sparse Row (CSR) format [8]. Herein, three arrays are used to store the elements and order of the elements of the matrix. The first simply lists all the non-zero elements. The second array lists the column indices of where these non-zero elements lie. The third contains the pointers to the beginning of each row in the previously mentioned arrays. Suppose the matrix A, of size $(N \times N)$, equals:

$$A = \begin{bmatrix} 1 & 0 & 6 & 0 & 0 \\ 0 & 9 & 0 & 6 & 0 \\ 0 & 0 & 4 & 0 & 2 \\ 2 & 0 & 0 & 3 & 0 \\ 8 & 1 & 0 & 0 & 3 \end{bmatrix}.$$

Storing this matrix using the CSR format results in the following 3 vectors:

$$AA = \begin{bmatrix} 1 & 6 & 9 & 6 & 4 & 2 & 2 & 3 & 8 & 1 & 3 \end{bmatrix}$$
$$JA = \begin{bmatrix} 0 & 2 & 1 & 3 & 4 & 4 & 0 & 3 & 0 & 1 & 4 \end{bmatrix}$$
$$IA = \begin{bmatrix} 0 & 2 & 4 & 6 & 8 & 11 \end{bmatrix}$$

Where AA contains all the non-zero entries of A, JA contains the column indices of the non-zero entries and IA contains the pointers to the beginning of each row. Note that the array IA consists of $N+1$ elements while the matrix A only has N rows. The last element points to the beginning of row $N+1$ (which does not exist) to indicate the end of the matrix.

Besides the CSR format, compressed diagonal storage can be used to store square sparse matrices. This format utilizes the fact that the non-zero elements of A are located on diagonals. Compressed diagonal storage uses two (smaller) matrices to store the sparse matrix A. In the columns of the first matrix, each diagonal of A that has at least one non-zero element is stored. All the elements of the diagonal are stored in n adjacent locations in the matrix. However, not every diagonal contains the same amount of elements, therefore the empty places in the columns are filled with zeros. These extra zeros are added at the bottom of the columns for super-diagonals and are added at the top of the columns for sub-diagonals. The second matrix is one-dimensional and contains the numbers of the diagonals stored in the other matrix. [9]
Using diagonal storage to store the matrix A as stated above results in the following two matrices:

$$AD = \begin{bmatrix} 1 & 6 & 0 & 0 \\ 9 & 6 & 0 & 0 \\ 4 & 2 & 0 & 0 \\ 3 & 0 & 2 & 0 \\ 3 & 0 & 1 & 8 \end{bmatrix} \qquad LA = [0,\ 2, -3, -4]$$

Where AD contains the entries and LA the numbers of the non-zero diagonals. Apart from these two formats, other formats such as ELLPACK [10], HYBRID ELLPACK [10] and SELL-C-Sigma [11] have been developed specifically for efficient GPU implementation.

In order to have a single vector of unknowns, it is required to use a single index notation instead of the two indices i and j. For this purpose, the horizontal notation is used which, in 2D, is given by

$$\alpha = i + (j-1)(N-1), \tag{8}$$

where α is the new, single index.

Because the matrix A can easily become very large, solving this equation using a direct method can take a long time, or even be impossible. For example, directly computing the inverse of a matrix is a very time-consuming operation. More so, the inverse of a irreducible sparse matrix is proven to be full [12]. Computing the inverse of a very large sparse matrix is therefore not a good option. A common direct method which could be used is LU-decomposition or Gaussian elimination. However, applying this method to a sparse matrix can lead to fill-ins. Depending on the bandwidth of the sparse matrix, these fill-ins can dramatically decrease efficiency [6]. Figure 1 shows an example of such fill-ins when applying LU-decomposition on a sparse matrix [13].

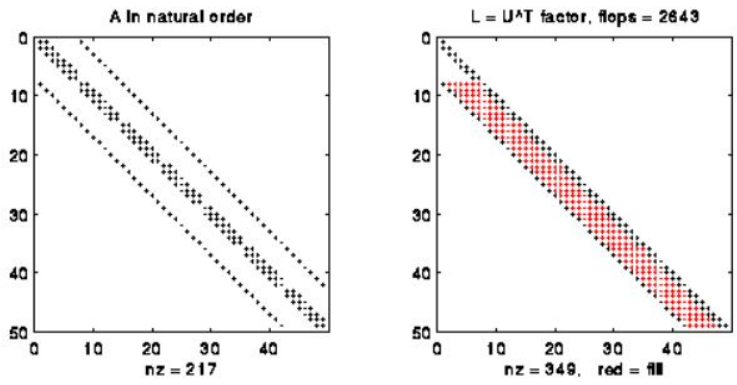

Figure 1: Fill-ins arise when performing a LU-decomposition of a sparse matrix [13]

Therefore, different methods are used to approximate $\mathbf{u}$ efficiently and accurately. This chapter will describe general Krylov subspace methods and preconditioned iteration methods, where the GMRES method is examined specifically, as this method will be the main focus point of this report.

3.2 Krylov Subspace Methods

Throughout this part of the chapter, the book *Iterative Methods for Sparse Systems* by Yousef Saad [6] is referenced. Krylov subspace methods are based on projection methods, which are iterative. In the m-th iteration of a projection method, an approximation $\mathbf{u}_m$ is extracted from a subspace $\mathbf{u}_0 + \mathcal{K}_m$. The vector $\mathbf{u}_0$ is an arbitrary initial guess for $\mathbf{u}$. In a Krylov subspace method, $\mathcal{K}_m$ is chosen to be

$$\mathcal{K}_m(A, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, ..., A^{m-1}\mathbf{r}_0\},$$

and $\mathbf{r}_0 = \mathbf{b} - A\mathbf{u}_0$. The dependency of $\mathbf{u}_m$ on the subspace $\mathbf{u}_0 + \mathcal{K}_m$ can be shown by writing:

$$\begin{aligned}\mathbf{u}_m &= \mathbf{b} + (I - A)\mathbf{u}_{m-1} \\ &= \mathbf{u}_{m-1} + \mathbf{r}_{m-1} \\ &= \mathbf{u}_0 + \mathbf{r}_0 + \mathbf{r}_1 + ... + \mathbf{r}_{m-1},\end{aligned}$$

Then from $\mathbf{b} - A\mathbf{u}_m = b - A\mathbf{u}_{m-1} - A\mathbf{r}_{m-1}$ we find $\mathbf{r}_m = (I - A)\mathbf{r}_{m-1}$. Therefore, the iterative solution $\mathbf{u}_m$ of the problem (which is an approximation of the real solution) can be expressed as:

$$\mathbf{u}_m = \mathbf{u}_0 + [\mathbf{r}_0, A\mathbf{r}_0, ..., A^{m-1}\mathbf{r}_0]\mathbf{c}, \tag{9}$$

where $\mathbf{c}$ is a vector of constants. Therefore, all Krylov subspace methods depend on the following approximation of $\mathbf{u}$:

$$\mathbf{u} = A^{-1}\mathbf{b} \approx \mathbf{u}_0 + q_{m-1}(A)\mathbf{b},$$

where $q_{m-1}(A)$ is an as of yet undetermined polynomial of degree $m - 1$.

The goal is then to find the best approximate solution to $A\mathbf{u} = \mathbf{b}$ in the subspace $\mathbf{u}_0 + \mathcal{K}_m$. This can be done by setting the following condition:

$$\mathbf{b} - A\mathbf{u}_m \perp \mathcal{L}_m,$$

where $\mathcal{L}_m$ is another subspace of dimension m. The different Krylov methods depend on different choices of the subspace $\mathcal{L}_m$.

3.2.1 GMRES

In the Generalized Minimum Residual Method - GMRES for short - the subspaces are chosen to be $\mathcal{K} = \mathcal{K}_m(A, \mathbf{v}_1)$ and $\mathcal{L} = A\mathcal{K}_m(A, \mathbf{v}_1)$, where $\mathbf{v}_1 = \mathbf{r}_0/||\mathbf{r}_0||_2$, the normalized version of the vector $\mathbf{r}_0$. To explain the way in which the approximation $\mathbf{u}_m$ of $\mathbf{u}$ is found, a new matrix needs to be defined.

The Hessenberg matrix $\bar{H}$ is defined iteratively, where in the j-th step the following four equations are computated:

$$\begin{aligned}h_{ij} &= (A\mathbf{v}_j, \mathbf{v}_i), \text{ for } i = 1, 2, ..., j; \\ \mathbf{w}_j &= A\mathbf{v}_j - \textstyle\sum_{i=1}^{j} h_{ij}\mathbf{v}_i; \\ h_{j+1,j} &= ||\mathbf{w}_j||_2; \\ \mathbf{v}_{j+1} &= \mathbf{w}_j/h_{j+1,j}.\end{aligned}$$

Continue this until $h_{j+1,j}$ is sufficiently small or $j = m$.
Then, V_m is the matrix with column vectors $\mathbf{v}_1, ..., \mathbf{v}_m$, which are computed in the algorithm above. In order to find an approximation, the following function needs to be minimized:

$$J(\mathbf{y}) = ||\mathbf{b} - A(\mathbf{u}_0 + V_m\mathbf{y})||_2. \tag{10}$$

Then the approximation is found quickly by $\mathbf{u}_m = \mathbf{u}_0 + V_m\mathbf{y}_m$, where $\mathbf{y}_m$ minimizes the function in (10). $\mathbf{y}_m$ is found by solving an $(m + 1) \times m$ least-squares problem, which is inexpensive to do if $m \ll n$.
If A is positive definite - which is the case with the Poisson equation - this method will definitely converge. When solving the Helmholtz equation, other tricks that will be discussed later can be used to make GMRES more suitable to use.

3.3 Multigrid Methods

Although Krylov Subspace Methods seem very efficient, they tend to lose this efficiency and slow down in terms of convergence as the problem becomes larger. This problem is solved with Multigrid Methods. These methods are in a sense independent of the grid size and are designed specifically for solving discretized elliptic PDEs, such as the Poisson equation. Multigrid Methods make use of a hierarchy of grids. Usually, this hierarchy is used to recursively interpolate solutions until a small enough grid size is obtained. Then, so-called smoothers are used to "smooth out" the solution.

Algebraic Multigrid Methods are a more general form of Multigrid Methods. They rely solely on knowledge of the matrix A, rather than of the whole problem. This makes it easier to implement them, but can also lead to loss of efficiency.

3.4 Preconditioned Methods

Preconditioning of an iterative method means that the linear system is transformed to a system that is easier to solve, but still has the same solution. This is done using a preconditioning matrix M. This M must satisfy some properties. Firstly, the system $M\mathbf{x} = \mathbf{b}$ should be inexpensive to solve. Secondly, M should be similar to A and lastly, it should be non-singular. This matrix can then be applied in three ways: with left, right and split preconditioning. These are explained hereafter using GMRES.

3.4.1 Preconditioned GMRES

The left-conditioned GMRES algorithm is applied to the system:

$$M^{-1}A\mathbf{u} = M^{-1}\mathbf{b}.$$

The left-conditioned Krylov subspace $\mathcal{K}_m(M^{-1}A, \mathbf{r}_0)$ is used. The computation of the Hessenberg matrix $\bar{H}$ is more or less the same as in the usual GMRES method, except that $M^{-1}A$ is used, rather than A. A disadvantage of this type of preconditioning is that the unpreconditioned residuals are not easily obtained.

In right-preconditioned GMRES the system

$$AM^{-1}\mathbf{v} = \mathbf{b}$$

is solved. $\mathbf{u}$ is then extracted from $\mathbf{v} = M\mathbf{u}$. Once the initial residual $\mathbf{r}_0$ is known, all following vectors of the Krylov subspace can be computed without any reference to $\mathbf{v}$. At the end, the approximation of $\mathbf{v}$ is given by:

$$\mathbf{v}_m = M\mathbf{u}_0 + \sum_{i=1}^{m} v_i\eta_i,$$

where the η_i are weight factors, constructed via a recurrence relation that is beyond the scope of this report. An essential difference to the left-conditioned version of GMRES is that the residual vectors are now calculated implicitly.
Right-preconditioned GMRES also gives rise to another form of this method, namely Flexible GMRES. Here, in every iteration the preconditioner is changed by computing $z_j = M_j^{-1}v_j$. Then the final approximation is found by

$$\mathbf{u}_m = \mathbf{u}_0 + Z_m\mathbf{y}_m,$$

where Z_m has the vectors $z_1, ..., z_m$ as its columns.

Split preconditioning is based on a factorization of M in the form of $M = LU$. Then, the system that is used is

$$L^{-1}AU^{-1}\mathbf{v} = L^{-1}\mathbf{b}, \quad \mathbf{u} = U^{-1}\mathbf{v}.$$

3.4.2 Examples of Preconditioners

One of the most simple preconditioning techniques is Jacobi Preconditioning, also known as Diagonal Preconditioning. It uses the diagonal of A as the preconditioner. This technique is not very accurate and therefore will not be used in our tests. Another example is Incomplete LU factorization, in which A is approximately factorized into lower and upper diagonal matrices L and U, which have the same non-zero structure as A. ILU is also not very accurate and thus will also not be used.

Another commonly used preconditioner is SSOR. This stands for Symmetric Succesive Over Relaxation. It is based on splitting the matrix A into

$$A = D - E - F,$$

where D is the diagonal of A and $-E$ and $-F$ are the strict lower and upper part of A, respectively. The SSOR preconditioner is defined as

$$M = (D - \omega E)D^{-1}(D - \omega F),$$

where ω is a scalar. Taking $\omega = 1$ gives the Symmetric Gauss-Seidel (SGS) preconditioner.
Something that is important to mention is that Multigrid Methods can not only be used as standalone methods. They can also be used as a preconditioner for other methods.

4 GPU Solvers

4.1 GPU Architecture

A modern Graphics Processing Unit (GPU) is a Single Instruction Multiple Data (SIMD) processor with a highly parallel structure. It consists of many multiprocessors which can perform operations simultaneously, using both global device memory and memory that is local to each multiprocessor. Such a device is especially well-suited for carrying out repetitive calculations on a large data set (or "stream"), as tasks can be split between several processing units instead of being done sequentially or in a loop. However, transferring data between multiprocessors through e.g. a PCI-E bus can sometimes take longer than the computations themselves and often presents a bottleneck to parallel algorithms.

To understand memory management in a GPU, one must first understand the difference between bus latency and bandwidth. Latency is the amount of time it takes data to travel from one point to another, whereas bandwidth is the rate of data transfer. The time it takes to send n real numbers can be written as

$$t = \alpha + \beta n, \tag{11}$$

where α is the latency (start-up time) and β is the time to send 1 real number. Usually $\beta \ll \alpha$.

It generally takes time to transfer information between a processor and a memory location. This depends on physical factors such as the medium through which information travels (i.e. the type of connection) and the distance travelled. As a rule of thumb, the latency for the data transfer between a given processor and some memory location increases as the physical distance between the two increases. The bandwidth of the transfer, however, stays about constant with distance and instead just depends on the type of bus used for the connection.

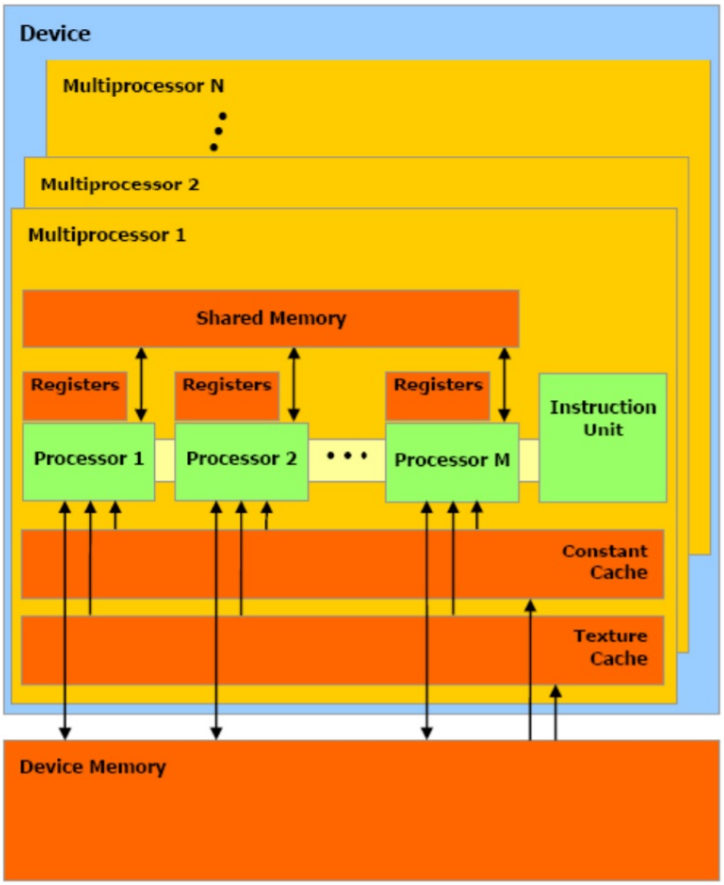

Figure 2: A schematic of a typical GPU architecture. [14]

An NVIDIA GPU [1] typically consists of several Streaming Multiprocessors (SMs), each with eight Scalar

Processors (SPs) within. Each SM has some low-latency Shared Memory, which only its own processors can directly access. All SMs also have access to the global Device Memory, but access to this memory has a high latency; it is therefore often preferable to transfer all necessary data from the Device Memory to local Shared memory only once, perform the instructions locally, then transfer the result back to Device Memory. The only problem with this approach is that each SM only has a limited amount of Shared Memory, so it is sometimes not possible to store all necessary data locally.

When dealing with spare matrix solvers, it is also important to note the distinction between memory bound and compute bound problems. A large number of data accesses relative to the number of calculations performed on each data value could potentially result in a parallel algorithm that is slower than its sequential counterparts, given the GPU's memory latency constraints.

4.2 CUDA Programming

CUDA [14] is NVIDIA's programming model and library for GPU computing. It introduces a series of abstractions to the hardware architecture itself to allow for different levels of data parallelism.

A multithreaded CUDA program consists of a grid of several blocks which execute independently from each other and can be scheduled on any of the available multiprocessors within a GPU [1]. Each block is further divided into a three-dimensional array of threads, which provide a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. As with the hardware processors themselves (figure 2), threads within a block can all access the same Shared Memory but threads in different blocks can only share data through the global Device Memory. It is important to note that there can be more blocks in a program than SMs within a GPU and more threads in a block than scalar processors within an SM, because these can be scheduled to the hardware independently and in a dynamic way.

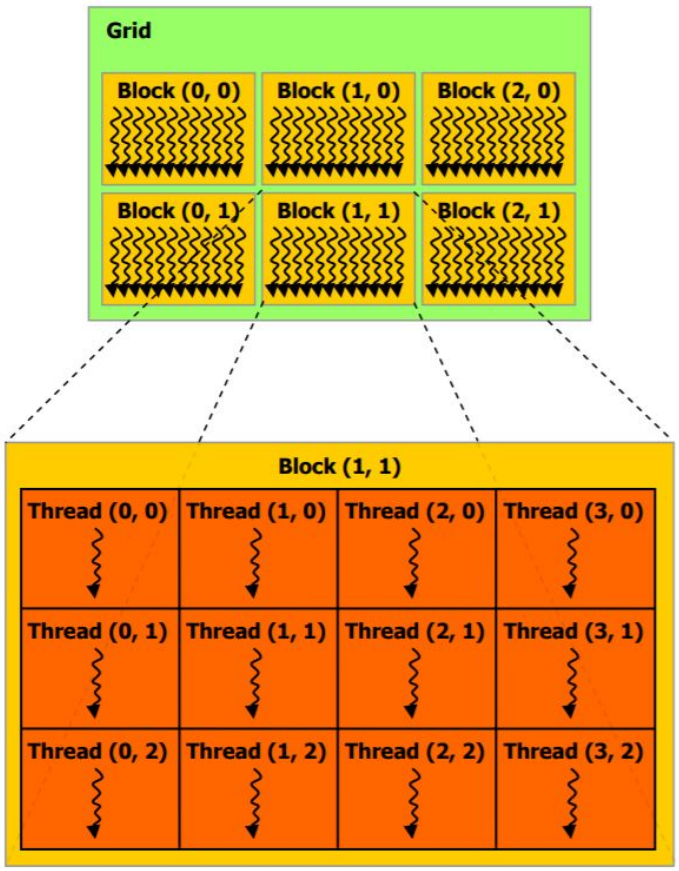

Figure 3: A schematic of CUDA's parallel model. [1]

4.3 Overview of Solver Packages

Several high-level solver packages are available for doing sparse linear algebra with CUDA. The focus of this report is on cuSolver [1] and Paralution [2]. AMGCL [15] is another upcoming package for GPU computing which is discussed here, but not included in the comparison. It is also possible to use the (very efficient) basic linear algebra routines of low-level libraries such as cuBLAS and cuSPARSE to develop solvers from scratch, but this is significantly more time-consuming and might not be faster than the solvers found in packages already available and will not be discussed here.

As mentioned in Chapters 2 and 3, the focus of this report is on using iterative methods to solve a finite-difference discretized Helmholtz equation. Krylov methods, among others, were tested with and without preconditioners for a holistic comparison of AMGCL and Paralution in this application. CuSolver, however, does not include iterative solvers but was nonetheless compared with Paralution in solving the Poisson equation using direct solvers. An overview of each of the three packages follows below.

CuSolver [1] is a C-language package based on the cuBLAS and cuSPARSE libraries, made specifically for CUDA. It combines three separate libraries, of which cuSolverSP is the most relevant to this study's purposes. The cuSolverSP library itself was designed to solve the sparse system $A\mathbf{u} = \mathbf{b}$. The core algorithm is based on sparse QR factorization, and the matrix A is accepted in CSR format. CuSolverSP provides a few high-level functions for direct solvers, but does not have built-in functions for any iterative (e.g. Krylov) methods. These must instead be implemented by the user with the help of cuSparse and cuBlas routines. CuSolver code is written at a lower level than Paralution code, as there are less built-in functions and CUDA's memory management functions must be called much more often. Unfortunately, it must also be noted that CuSolver's documentation is minimal and sometimes cryptic, and that this package was noted to be the least user-friendly of the three.

Paralution [2] is a library for iterative sparse methods with a focus on supporting multi-core CPU and GPU platforms. As such, it supports CUDA as well as OpenMP, OpenCL, and MPI. Paralution provides a variety of preconditioners and solvers (both iterative and direct) which can be accessed via simple function calls [2]. Paralution has recently been made open-source, as the original developers have moved on to work on rocALUTION [16] with a focus on AMD GPUs.

AMGCL [15] is a C++ library for solving large sparse linear systems with the algebraic multigrid (AMG) method. As previously described in Chapter 3, the AMG method can be used as a standalone solver but is also very useful as a preconditioner within an iterative solver such as CG, BiCGStab, or GMRES. Algorithms for these iterative solvers are included in the package, as well as preconditioners and backends. A backend in AMGCL is a class that defines matrix and vector types along with operations on them (e.g. creation, matrix-vector products, dot products). In order to solve a system of equations, an instance of the `amgcl::makesolver` class must be initiated with the desired backend, preconditioner, and solver. The solution phase then uses types and operations defined in the backend. This is useful for users looking to accelerate the solution through parallelization with CUDA, OpenMP, or OpenCL. The main code can be left mostly unchaged while the backend is switched between a regular sequential one and one which uses CUDA to perform its array operations in parallel. This makes AMGCL the most flexible package in this comparison in terms of backend support.

4.4 Hardware & Software Specifications

For the the comparisons done in this study, access to the INSY cluster at the Delft Technical University was obtained. Unless stated otherwise, all GPU calculations are performed on one Nvidia GeForce GTX 1080 Ti. Its specifications are:

- Architecture: Pascal
- Compute Capability[17]: 6.1
- Memory: 11 Gb
- Cores: 3584

Paralution 1.1.0 code was compiled with g++ and ran using the Cuda 8.0 module. CuSolver v10.0.130 code was compiled with nvcc and ran using Cuda 10.0. The exact compilation commands will be provided in the appendix.

5 Accuracy Analysis

Several metrics were used for measuring the accuracy of each solution. Residuals, L_2 norms, and L_∞ norms can be used for an arbitrary source function, such as the Delta source function that was used for most of the simulations. However, these methods alone only represent the error in the solution relative to the solution of the discrete system $A\mathbf{u} = \mathbf{b}$. The method of manufactured solutions was used in order to also calculate the error relative to a manufactured exact solution to the original PDE.

5.1 The residual, L_2 norm, and L_∞ norm

In numerical analysis, the residual of an approximate solution $\hat{\mathbf{u}}$ of $A\mathbf{u} = \mathbf{b}$ is defined as:

$$\mathbf{r} = \mathbf{b} - A\hat{\mathbf{u}} \tag{12}$$

The residual can be calculated with each iteration, and is often used as the stopping condition for iterative methods. In order to avoid ambiguity in potentially different implementations of the residual in the different packages, a standard algorithm was written and utilized across different platforms.

Because the residual is a vector, the (scaled) L_2 or L_∞ norm of $\mathbf{r}$ can be taken to provide simple comparisons between scalar values. These are defined here as follows (for $\mathbf{r}$ of size m):

$$||\mathbf{r}||_2 = \frac{1}{m}\sqrt{r_1^2 + r_2^2 + ... + r_m^2} \tag{13}$$

$$||\mathbf{r}||_\infty = \max_{1 \le j \le m} |r_j| \tag{14}$$

Such metrics are useful for determining the accuracy of a solution to the discrete $A\mathbf{u} = \mathbf{b}$ system, and a residual norm of zero indicates an exact solution to the linear system of equations. However, the residual does not allow for comparisons to the exact solution of the original PDE to be made.

The residual is instead used as the stopping criterion throughout the report. More precisely, the iterative algorithms are set to stop when the scaled L_2 norm becomes smaller than 10^{-6}.

5.2 The Method of Manufactured Solutions

Calculating the actual error in a numerical solution can be difficult because many differential equations do not have exact analytical solutions for arbitrary source function and boundary conditions. The method of manufactured solutions (MMS) bypasses this by first assuming an exact solution, then substituting this in to the PDE, which yields the corresponding source function and boundary conditions required. This of course does not easily work for arbitrary source functions, but is useful for calculating the exact error in the approximation of a given PDE.

For the Poisson equation, assume the solution $u(x, y) = \sin(x)\cos(y)$. From this it follows that:

$$\frac{\partial u}{\partial x} = \cos(x)\cos(y),$$

$$\frac{\partial u}{\partial y} = -\sin(x)\sin(y),$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial y^2} = -\sin(x)\cos(y) = -u(x, y).$$

Substituting this into the 2D Poisson equation yields:

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = S(x, y),$$

$$2\sin(x)\cos(y) = S(x, y).$$

The required boundary conditions then are: $u(0, y) = 0$, $u(x, 0) = \sin(x)$, $u(1, y) = \sin(1)\cos(y)$, and $u(x, 1) = \sin(x)\cos(1)$.

For the Helmholtz equation, assume the same $u(x, y) = \sin(x)\cos(y)$ and corresponding first and second derivatives. Substituting this into the 2D Helmholtz equation:

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - k^2 u(x, y) = S(x, y),$$

$$u(x, y) + u(x, y) - k^2 u(x, y) = (2 - k^2) \cdot \sin(x)\cos(y) = S(x, y).$$

Note that the vector $\mathbf{b}$ is then made up by the function $S(x, y)$ evaluated at each node. The corresponding boundary conditions again are: $u(0, y) = 0$, $u(x, 0) = \sin(x)$, $u(1, y) = \sin(1)\cos(y)$, and $u(x, 1) = \sin(x)\cos(1)$.

5.3 Timing and Comparison

The question of which program is faster is generally not a straight-forward one. A distinction can be made, for example, between compile and run times. Furthermore, one may choose to measure the entire length of the code or only a subsection. For the purposes of this study, focus is placed on the run time of the solver algorithms themselves. This is because the assembly of the matrix and vectors to be solved is carried out in the host CPU, and is done in the same way regardless of the solver package being utilized. However, it is also important to account for the data transfer overhead of the GPU. This is accomplished by starting the timer immediately before the assembled arrays are copied from the host to the device and stopping the timer after the final solution vector has been transferred back to the host for output.

6 Numerical Results

6.1 2D Poisson Equation with Direct Solvers

The results of solving the 2D Poisson problem by QR factorization in both Paralution and cuSolver are given in Table 1. In the table, "Size" refers to the amount of grid nodes used in one direction. These sizes are preferably made odd for convenience, as the middle grid node will then be exactly in the middle of Ω. All times are in seconds.

Size	Paralution				cuSolver		
-	CPU time	GPU time	L_∞-residual	L_2-residual	GPU time	L_∞-residual	L_2-residual
51	18.771	11.743	6.82e-12	2.56e-14	0.738	7.73e-12	1.74e-14
101	1081.13	659.388	4.73e-11	6.26e-14	1.141	5.09e-11	4.30e-14
201	-	-	-	-	3.127	2.66e-10	1.11e-13
301	-	-	-	-	8.786	6.33e-10	1.81e-13
401	-	-	-	-	19.213	1.98e-9	2.97e-13

Table 1: Comparing QR runtimes and final residuals between Paralution and cuSolver using different gridsizes

CuSolver performs much better than Paralution here. As the grid size is increased, the Paralution program's runtime increases much faster than the runtime of the CuSolver program. Besides, the Paralution program runs out of memory at a grid size of 201 (and larger). This is because Paralution's direct solvers are not optimized for sparse matrices. According to the user manual: "the user can pass a sparse matrix, internally it will be converted to dense and then the selected method will be applied". The documentation further warns that, due to the fact that the matrix is converted in a dense format, these methods should be used only for very small matrices. CuSolverSP, on the other hand, has functions specifically built with sparse matrix algebra in mind. Note that CuSolver also runs out of memory for large grid sizes; this happens at a grid size of 501 (and larger). The final approximation is plotted with a gridsize of 101 in Figure 4. It matches the expected physical representation of the solution.

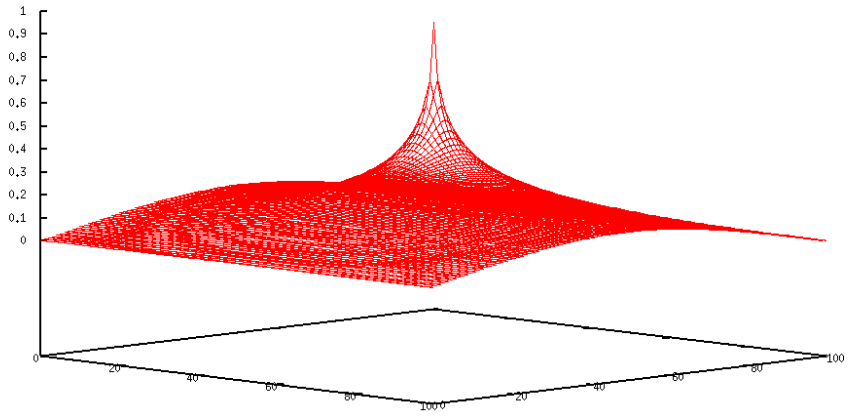

Figure 4: Approximation of the Solution of the Poisson equation using cuSolver and a grid size of 101

As mentioned in the previous chapter, the accuracy of the computations can also be analyzed using manufactured solutions. The results of this analysis are represented in Table 2. The given errors represent the difference between the exact and approximated solution.

Size	Paralution		cuSolver	
-	L_∞-error	L_2-error	L_∞-error	L_2-error
51	1.95e-6	1.92e-8	1.95e-6	1.92e-8
101	5.08e-7	2.60e-9	5.10e-7	2.60e-9
201	-	-	1.29e-7	3.37e-10
301	-	-	5.78e-8	1.01e-10
401	-	-	3.28e-8	4.32e-11

Table 2: Accuracy comparison between Paralution and cuSolver using the method of manufactured solutions for the Poisson equation in 2D. Solved with the QR direct solver

From the data above, it is determined that the cuSolver program converges with order $\mathcal{O}(h^{2.983\pm.008})$ and the Paralution program converges with order $\mathcal{O}(h^{2.838\pm.104})$. This is very close to $\mathcal{O}(h^3)$ convergence, as opposed to the expected $\mathcal{O}(h^2)$ from theory.

The Paralution code was also forced to run on the host backend (without a GPU device) for comparison. This was found to be faster only for very small matrix sizes (up to approximately 400 unknowns), which can be explained by the communication overhead caused by parallelization. For larger matrices, the sequential program performed much slower than the GPU-enabled program.

6.2 2D Poisson Equation with Multigrid Methods

The 2D Poisson equation was solved again, this time using the unpreconditioned Algebraic Multigrid iterative solver. The Paralution code was run on CPU and GPU. The results are given in Table 3. The stopping criteria are met very quickly with this method when compared to previous solvers.

Size	Paralution						
-	GPU time	CPU time	Speedup	L_∞-residual	L_2-residual	iterations	levels
51	0.042	0.010	0.238	1.52e-4	4.71e-7	7	3
101	0.068	0.040	0.588	3.24e-4	4.07e-7	7	3
201	0.181	0.156	0.862	1.09e-3	7.29e-7	9	4
401	0.334	0.531	1.590	4.00e-3	6.88e-7	9	5
501	0.312	0.924	2.962	2.9e-3	6.09e-7	10	5

Table 3: Comparing runtimes and final residuals between GPU and CPU in Paralution when using AMG as a solver.

The data from Table 3 is plotted in figure 5 in order to visualise the difference in runtime between the CPU and GPU. From this graph, it becomes apparent that a CPU is only faster for small grid sizes. In this case, a CPU is faster up to a grid size of around 225.

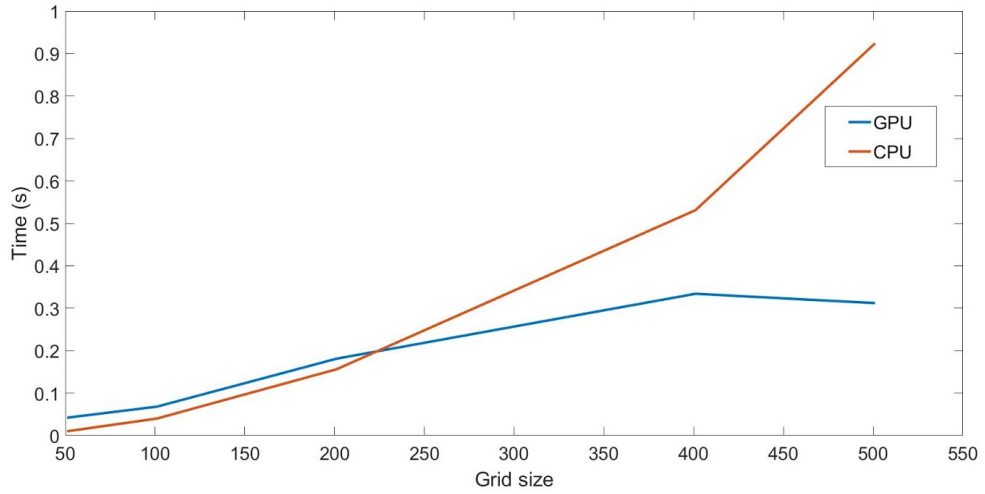

Figure 5: Comparing GPU and CPU runtimes when solving the 2D Poisson equation with AMG.

The results of the accuracy analysis using the method of manufactured solutions are given in Table 4.

Size	Paralution	
-	L_∞-error	L_2-error
51	1.95e-6	1.91e-8
101	5.08e-7	2.59e-9
201	1.30e-7	3.37e-10
401	3.23e-8	4.16e-11
501	2.33e-8	2.50e-11

Table 4: Accuracy comparison with Paralution using the method of manufactured solutions for the Poisson equation in 2D. Solved with the AMG solver.

These results converge with order $\mathcal{O}(h^{2.936\pm.089})$ when using AMG as a solver.

As multigrid methods can also be used as a preconditioner instead of a solver, AMG was used as a preconditioner to GMRES in order to solve the two-dimensional Poisson equation again. Table 5 shows the results. The times seem very similar to those in Table 3, in that the turning point at which the GPU starts performing better than the CPU is around the same gridsize.

Size	Paralution						
-	GPU time	CPU time	Speedup	L_∞-residual	L_2-residual	Iterations	Levels
51	0.069	0.016	0.232	1.52e-4	4.71e-7	2	3
101	0.297	0.061	0.205	3.24e-4	4.07e-7	2	3
201	0.177	0.246	1.390	1.09e-3	7.29e-7	2	4
401	0.399	0.869	2.178	4.00e-3	6.88e-7	2	5
501	0.404	1.588	3.931	2.9e-3	6.09e-7	2	5

Table 5: Comparing runtimes and final residuals between GPU and CPU in Paralution when using AMG as a preconditioner to GMRES.

Again, the times from the table are graphed. The graph is shown in Figure 6.

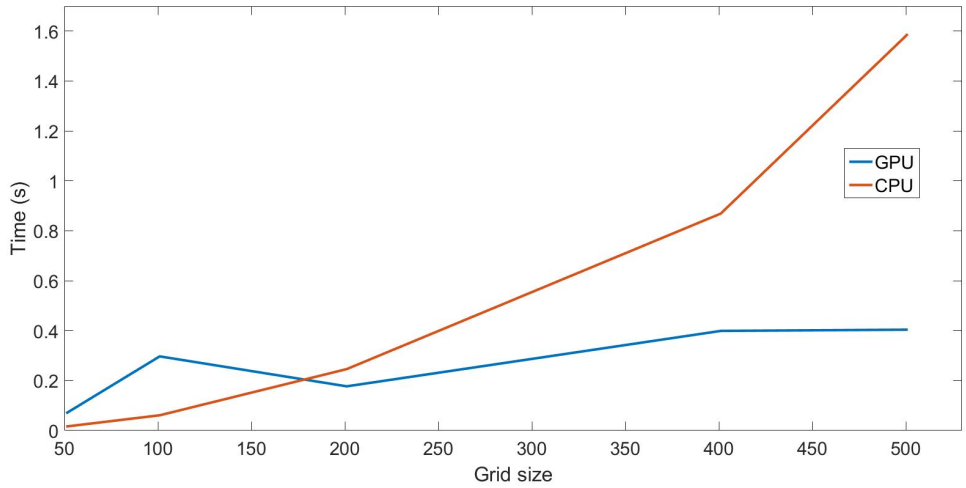

Figure 6: Comparing GPU and CPU runtimes when solving the 2D Poisson equation with GMRES preconditioned with AMG.

In this case, solving the system with a CPU is faster until grid sizes of around 175 grid points. The accuracy analysis of GMRES preconditioned with AMG is given in Table 6.

Size	Paralution	
-	L_∞-error	L_2-error
51	1.96e-6	1.93e-8
101	5.09e-7	2.60e-9
201	1.25e-7	3.19e-10
401	3.34e-8	2.53e-11
501	4.05e-8	2.67e-11

Table 6: Accuracy comparison with Paralution using the method of manufactured solutions for the Poisson equation in 2D. Solved with GMRES preconditioned with AMG.

The results converge with order $\mathcal{O}(h^{3.030\pm.442})$ when using AMG as a preconditioner to the GMRES solver. The convergence of both multigrid methods is more than the expected $\mathcal{O}(h^2)$ but they do comply with the convergence order obtained from QR factorization.

Clearly, the Algebraic Multigrid method (either as solver or preconditioner to another iterative solver) can solve the 2D Poisson problem studied here much faster than the QR direct solver studied before. And the order of convergence and L_2-residuals found here are very similar to those obtained from QR factorization. This gives Paralution an edge over CuSolver, even though CuSolver's direct solvers performed better than those of Paralution.

6.3 2D Helmholtz Equation with Iterative Solvers

The 2D Helmholtz equation with homogeneous Dirichlet boundary conditions is described by the PDE below:

$$\begin{cases} -\Delta u - k^2 u = f(x,y), & \text{in } \Omega, \\ u = 0, & \text{on } \partial\Omega, \end{cases} \tag{15}$$

In this case, after discretization, $f(x,y)$ equals $\frac{1}{h^2}$ in the centre node of Ω and 0 everywhere else, and the region is the square $\Omega = (0,1)\times(0,1)$ with homogeneous Dirichlet conditions at the boundaries.
Here the 2D Helmholtz equation is solved using iterative solvers, as direct solvers do not suffice for the Helmholtz equation. Because cuSolver only contains direct solvers and multigrid methods do not suffice for the Helmholtz equation, only Paralution is used in this section. The Helmholtz equation is solved using varying grid sizes and wave numbers, as well as different solvers and preconditioners, on both a GPU and CPU.

The first method that is analyzed is GMRES without any preconditioner. Table 7 shows the results for different values of k and varying grid sizes. Note that when evaluating with large wave numbers, a large grid size is required in order to satisfy $kh < 0.6$.

k	Size	kh	Paralution		
-	-	-	GPU time	L_∞-residual	L_2-residual
10	101	0.098	23.89	3.02e-4	-
	201	0.050	535.48	5.56 e-4	-
	301	0.033	-	-	-
	501	0.020	-	-	-
50	101	0.490	6.178	3.95 e-4	-
	201	0.248	34.62	9.42 e-4	-
	301	0.166	-	-	-
	501	0.100	-	-	-
100	201	0.495	-	-	-
	301	0.331	-	-	-
	501	0.199	-	-	-

Table 7: Comparing GMRES runtimes and final residuals using Paralution with different gridsizes

It quickly becomes apparent that unpreconditioned GMRES is not an efficient method of evaluating this problem. Runtimes increase drastically with increasing problem size, and already at a grid size of 301 the runtimes exceed acceptable limits. Furthermore, for a wave number of 100 even the smallest grid sizes could not be

evaluated.

Next, the Multi-Colored Symmetric Gauss-Seidel preconditioner is used in conjunction with the GMRES method. The resulting run times are shown in table 8. The program was first ran using the GPU device, then forced to use the host CPU for comparison.

k	Size	kh	Paralution					
-	-	-	GPU time	CPU time	Speedup	L_∞-residual	L_2-residual	Iterations
10	101	0.098	6.720	8.314	1.237	4.07e-4	1.02e-6	133
	201	0.050	48.520	219.513	4.524	8.30e-4	1.01e-6	1187
	301	0.033	445.469	2556.31	5.738	1.21e-3	1.01e-6	551 898
	501	0.020	4296.51	-	-	1.98e-3	1.00e-6	-
50	101	0.490	1.515	1.507	0.995	4.28e-4	1.02e-6	25
	201	0.248	5.392	18.049	3.347	1.27e-3	1.01e-6	92
	301	0.166	16.103	111.399	6.918	1.95e-3	1.01e-6	242
	501	0.100	97.243	10 552.4	108.516	3.50e-3	1.00e-6	78 194
100	201	0.495	22.833	20.283	0.888	9.27e-4	1.01e-6	37 499
	301	0.331	698.99	4877.64	6.978	1.41e-3	1.01e-6	1 033 669
	501	0.199	1604.26	-	-	2.82e-3	1.00e-6	-

Table 8: Comparing GMRES, preconditioned with Multicolored SGS, runtimes and final residuals using Paralution with different gridsizes

The CPU is only faster for the case where $k = 50$ and grid size is 101 (i.e., the case with the least iterations). This again supports the theory that data locality is the bottleneck in GPU computing. Note that the turning point at which the GPU becomes faster than the CPU occurs at a lower grid size here than in the Poisson Multigrid case; this can be explained by the much larger number of iterations accross the board when solving the Helmholtz equation as opposed to Poisson.

The approximate solutions at different wave numbers are presented in the surface plots below. Figure 7 shows the solution for a wave number of 10, Figure 8 shows the solution for a wave number of 50, and Figure 9 shows the solution for a wave number of 100. Notice the increase in detail and wave behaviour as the wave number gets larger. This is one of the reasons why, in general, problems with larger wave numbers are harder to solve.

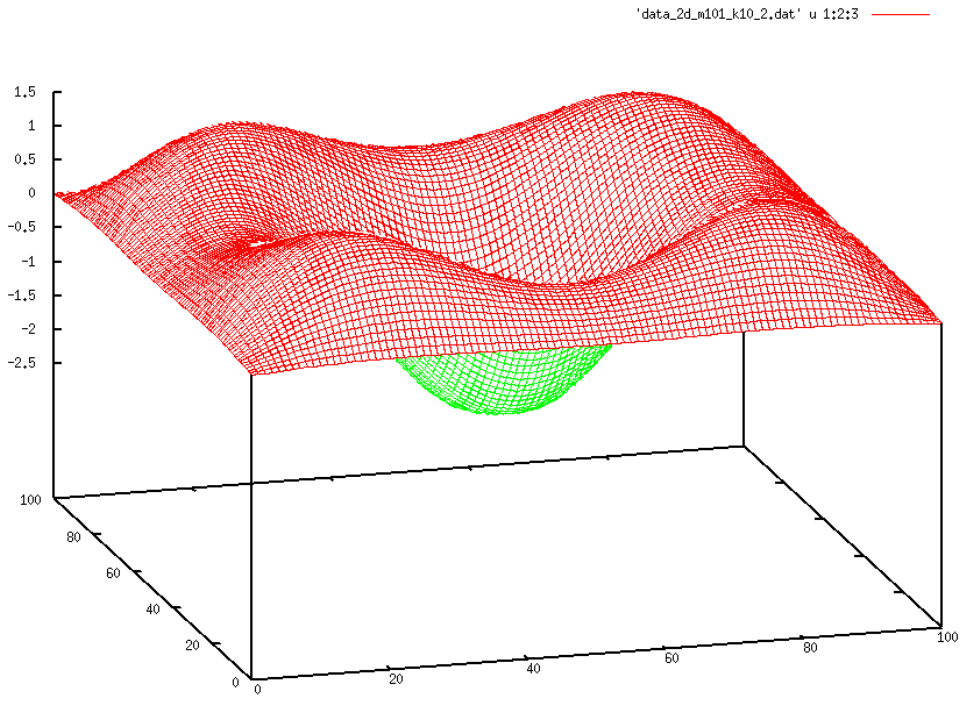

Figure 7: Approximation of the Solution of the 2D Helmholtz equation using Paralution with a grid size of 101 and wave number 10.

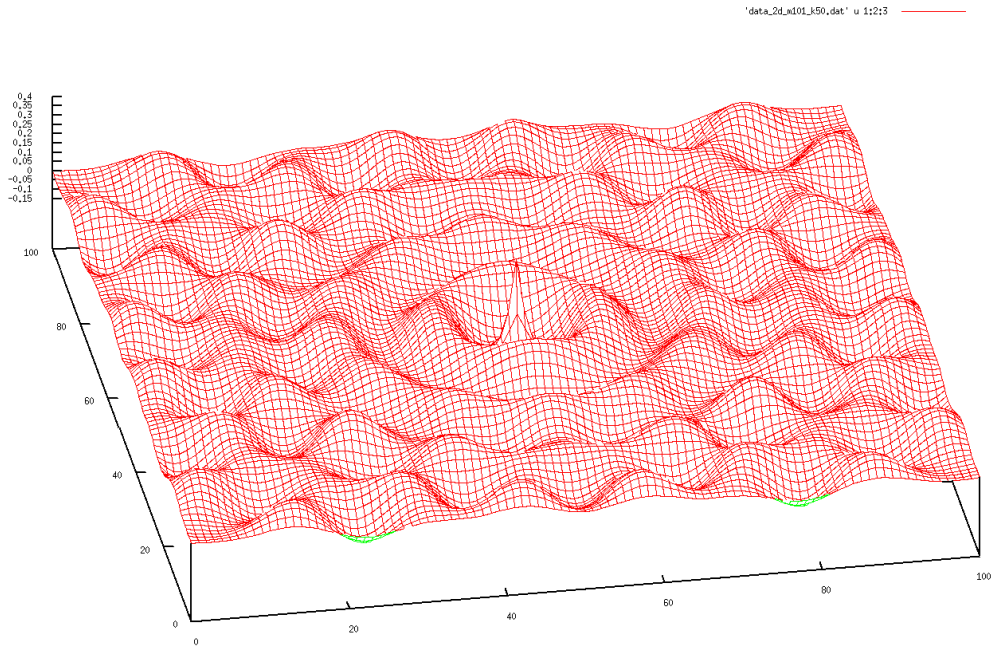

Figure 8: Approximation of the Solution of the 2D Helmholtz equation using Paralution with a grid size of 101 and wave number 50.

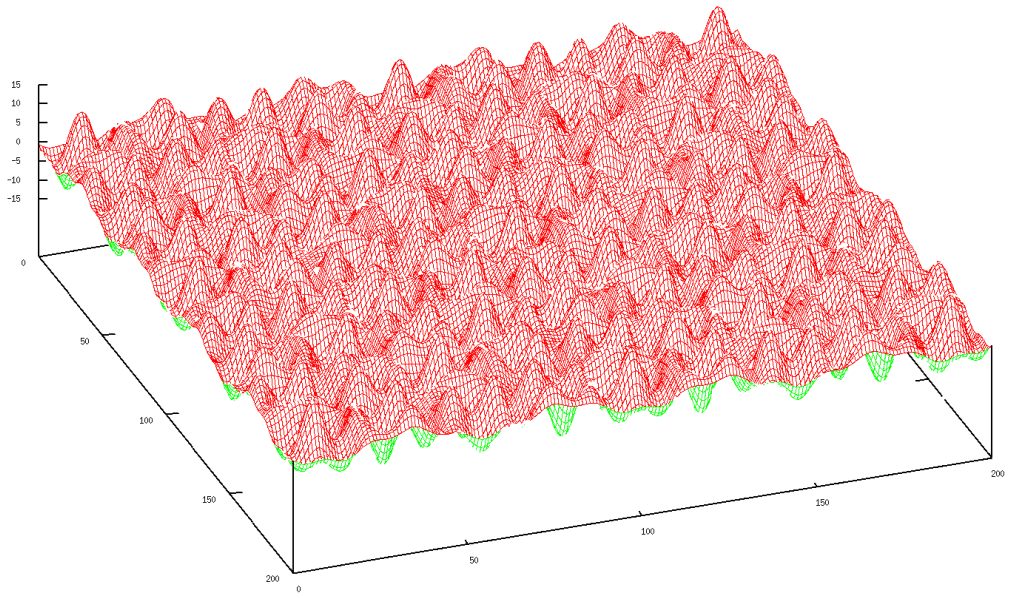

Figure 9: Approximation of the Solution of the 2D Helmholtz equation using Paralution with a grid size of 201 and wave number 100.

In order to analyze the convergence behavior of different numerical methods, the normalized residual can be plotted against the number of iterations. A logarithmic plot was used here to make the order of the normalized residual more clear. For example:

$$\frac{||\mathbf{r}_k||_2}{||\mathbf{b}||_2} \leq 10^{-4} \Rightarrow \log\left(\frac{||\mathbf{r}_k||_2}{||\mathbf{b}||_2}\right) \leq -4$$

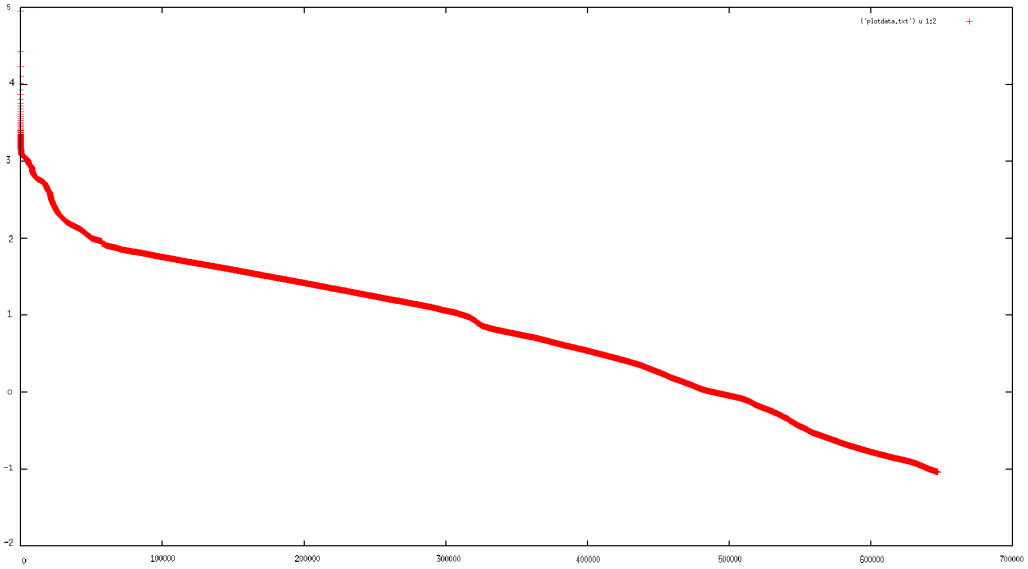

Figure 10: Logarithmic plot of the normalized residual against number of iterations, for the Helmholtz equation with wave number 10 solved with GMRES preconditioned with Multicolored SGS. The grid size is 301.

For a wave number of 10, as seen above, the residual initially decreases very quickly but then flattens out to a small negative slope.

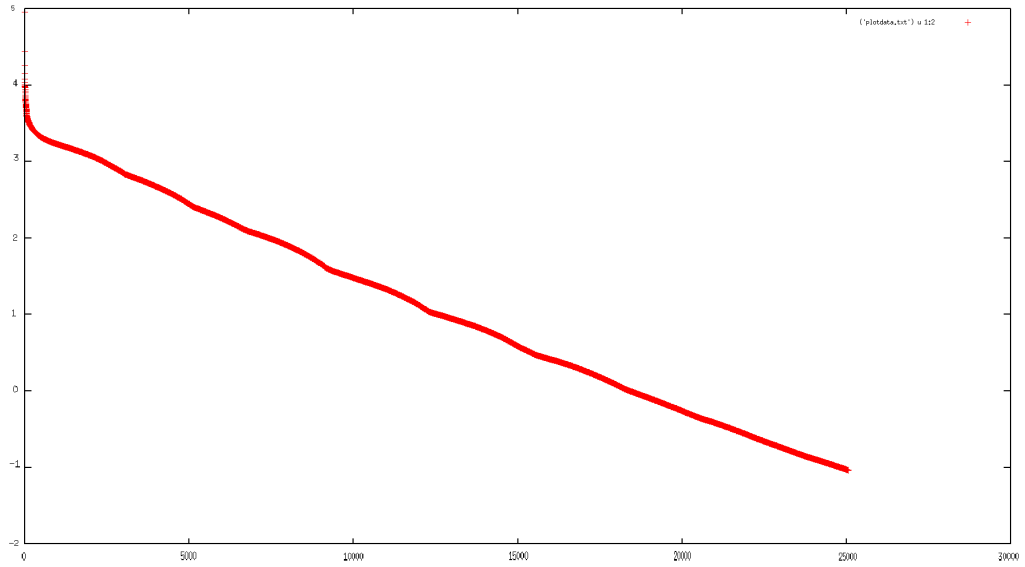

Figure 11: Logarithmic plot of the normalized residual against number of iterations, for the Helmholtz equation with wave number 50 solved with GMRES preconditioned with Multicolored SGS. The grid size is 301.

The case above (wave number 50) shows faster convergence behavior, with a steep linear slope that continues until the stopping criteria are reached.

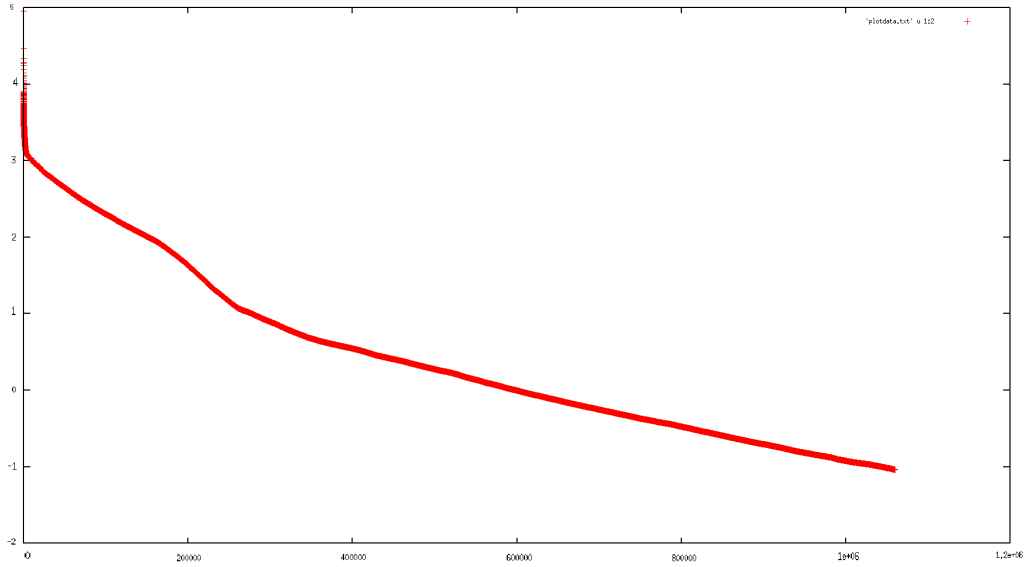

Figure 12: Logarithmic plot of the normalized residual against number of iterations, for the Helmholtz equation with wave number 100 solved with GMRES preconditioned with Multicolored SGS. The grid size is 301.

The wave number 100 case, like the wave number 10 case, shows a flattening of the slope after an initial fast convergence.

Again, the accuracy of the computations is analyzed using the method of manufactured solutions. The results of this analysis are represented in Table 9. The given errors represent the difference between the exact and approximated solution.

k	Size	kh	Paralution	
-	-	-	L_∞-error	L_2-error
10	101	0.098	9.01e-5	3.07e-7
	201	0.050	-	-
	301	0.033	-	-
	501	0.020	-	-
50	101	0.490	1.70e-4	5.08e-7
	201	0.248	8.54e-5	1.44e-7
	301	0.166	8.02e-5	7.84e-8
	501	0.100	7.99e-5	4.55e-8
100	201	0.495	1.18e-3	1.84e-6
	301	0.331	2.15e-3	2.42e-6
	501	0.199	6.22e-4	3.15e-7

Table 9: Accuracy of Paralution using the method of manufactured solutions for the Helmholtz equation in 2D. Solved with the GMRES solver, preconditioned with Multicolored SGS.

The Paralution calculations converge with order $\mathcal{O}(h^{1.587\pm.262})$ for $k = 50$ and $\mathcal{O}(h^{0.455\pm2.382})$ for $k = 100$ when using the GMRES solver preconditioned with SGS. For $k = 100$, the results don't seem to truly converge with increasing grid size but instead fluctuate around a mean of 1.67e-6. The error analysis for $k = 10$ and large grid sizes did not finish within a reasonable amount of time as the script ran for more than 6 days without converging. Subsequently, the error analysis was done with small grid sizes only (21 - 101). From this analysis an order of convergence of $\mathcal{O}(h^{0.317\pm.317})$ is found. The error thus seems to stagnate instead of converging.

7 Discussion

A first remarkable result was found when evaluating the 2D Helmholtz equation for different wave numbers. It is expected that for large values of k, the Helmholtz equation becomes increasingly more difficult to solve as the matrix A has more negative eigenvalues. The results however show that the equation was more easily solved for $k = 50$ than for $k = 10$, and a wave number of $k = 100$ proved the most difficult to solve. A possible explanation is that for specific wave numbers, some of the eigenvalues of A become zero or close to zero. When this is the case, the problem becomes ill-posed and difficult to solve as A is close to singular [18]. To this end, the eigenvalues of the matrix A corresponding to the problem with $k = 10$ were calculated. However, none of the eigenvalues turned out to be zero or close to zero. Therefore, another way to explain this behaviour should be sought.

Another thing that stands out is the high number of iterations required to solve the Helmholtz equation using the chosen solver and preconditioner. A lot of iterations imply a large amount of operations that are executed to arrive to the solution. This is a possible explanation for the large speedups observed between the CPU and GPU times. As more operations are performed, the data overhead of a GPU becomes insignificant and the highly parallel structure of a GPU is used optimally. However, an iterative solver that needs up to 1 million iterations to solve a system of linear equations does not solve the problem efficiently. Besides the large number of iterations, the results for the Helmholtz equation show an inconsistent convergence behaviour. Only for a wave number of $k = 50$ do the results converge while for the other wave numbers the errors stagnate. Future researchers are therefore recommended to try other solvers and preconditioners that need less iterations and are therefore better suited to the Helmholtz problem.

In order to make an honest comparison between packages and solvers, the matrix A is stored in CSR-format for all cases. However, due to the sparse properties of the matrix it is likely other storage formats yield better results. In future research, experimenting with other formats such as ELLPACK and Diagonal storage is advised to further improve efficiency.

Because of the many built in solver functions, good documentation and the ease of use Paralution proved to be a good GPU solver package. However, no real comparison between other available packages is be made except between CuSolver using only direct solvers. For future research it is recommended to compare Paralution with other GPU solver packages that include iterative solvers, such as AMGCL. Especially since Paralution is no longer being maintained it is important to know whether there are other alternatives available.

8 Conclusion

The direct solver results clearly show CuSolver to be more efficient than Paralution for sparse systems. Its algorithms are specialized for sparse matrices and cause little fill-in, which yielded faster results for the same accuracy. However, CuSolver's lack of iterative methods and sufficient documentation make it attractive only for those who are determined to utilize direct methods.

Paralution's suite of iterative methods and ease of use are what make it the more suitable package for solving the Poisson and Helmholtz equations in 2D. Its AMG solver solved 2D Poisson over 20 times faster than CuSolver's QR solver, at the same order of accuracy. And its preconditioned GMRES solver was able to solve the 2D Helmholtz equation accurately at wave numbers as high as 100, at the expense of a very high number of iterations. Future researchers are recommended to try the other preconditioners available in Paralution, which may be better suited to the 2D Helmholtz problem. Multi-Colored Symmetric Gauss-Seidel was found to be computationally expensive and show inconsistent convergence behavior for some wave numbers.

A comparison of GPU and CPU run times of the Paralution programs further illustrates the advantage of GPU computing. CPU times were found to be faster only for problems with a low ratio of operations to data transferred (data locality). For large problem sizes, a speedup of up to 100 was measured in this study.

Appendix

A CuSolver

In this appendix, part of the source code used for solving the 2D Poisson equation with QR in CuSolver is presented and explained.

In order to run the C code provided here, the header files cuda.h, cusolverSp.h, and cusparse.h must be included. Compilation can be done (on a Linux machine) with the following command:

```
nvcc filename.cu -o filename -lcusolver -lcusparse
```

In the main function, the `A` matrix is first stored on the CPU in CSR format in the form of the three dynamic arrays `I`, `J`, and `V`. The $\mathbf{b}$ vector, the solution vector $\mathbf{x}$, and the residual vector $\mathbf{r}$ are also declared as dynamic arrays and initialized to their respective values (the initial guess $\mathbf{x}_0$ in the case of the solution vector).

Next, these 6 arrays must be copied to the GPU device. Two functions are used to this aim: `cudaMalloc` and `cudaMemcpy`. Their signatures follow below:

```
cudaError_t cudaMalloc (void** devPtr, size_t size)
```

The `cudaMalloc` function takes a null pointer and a size in bytes, allocates `size` bytes of linear memory on the device, and writes in `*devPtr` a pointer to the allocated memory. Its return type is `cudaError_t`, which can be checked for debugging purposes.

```
cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
```

The `cudaMemcpy` function copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`. The `kind` argument specifies the direction of the copy, and can be one of: `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`.

An excerpt of the source code is provided below. Note that `d_V` is used to represent the GPU device's version of the array that is called `V` in the host. This is a commonly used convention in CUDA programming. The variable `nnz` is simply an integer representing the number of non-zeros in the matrix A. The variable `sizeA` is an integer representing the number of columns in the matrix A.

```
double* d_V = NULL;
cudaMalloc(&d_V, sizeof(V));
cudaMemcpy(d_V, V, sizeof(V), cudaMemcpyHostToDevice);
```

In order to solve this system on the device, the `cuSolverSpDcsrLSvQR` function is used. It stands for **cuSolver Sp**arse, for **d**ouble data type and **CSR** matrix format, **l**inear **s**olver, with **v**ector output format, based on **QR** factorization. An example of its use can be seen below.

```
int singularity = 0;
int reorder = 0;
double tol = 1.0e-11;
cusolverSpHandle_t handle;
cusolverStatus_t status;
status = cusolverSpCreate(&handle);
cusparseMatDescr_t descrA;
cusparseStatus_t sparse_status;
sparse_status = cusparseCreateMatDescr(&descrA);

cudaDeviceSynchronize();
```

```
status = cusolverSpDcsrlsvqr(
              handle,
              sizeA,
              nnz,
              descrA,
              d_V,
              d_I,
              d_J,
              d_b,
              tol,
              reorder,
              d_x,
              &singularity);

cudaDeviceSynchronize();
```

Lastly, `cudaMemcpy` is used again (this time with `cudaMemcpyDeviceToHost` as the kind) to copy the final solution vector back to the host. The function `cudaFree` is then used to free all the arrays in the device, while the standard C `free` function can be used to free arrays on the host.

```
cudaMemcpy(x, d_x, sizeA*sizeof(double),cudaMemcpyDeviceToHost);
cudaFree(d_x);
```

CuSolver does not have built-in timing functionality, but any regular C implementation of a timer can be used. In this case, the struct `timespec` was used.

B Paralution

Here the source code used for solving the 2D Helmholtz equation using Paralution is presented and explained.

In order to run the C++ code with Paralution, the header paralution.hpp is used together with standard C++ libraries such as cmath. Besides this, it is convenient to use the namespace paralution. The compilation of a Paralution code is done in three steps:

```
g++ -O3 -Wall -I../inc -c filename.cpp -o filename.o
g++ -o filename filename.o -L../lib -lparalution
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/paralution-1.1.0/build/lib/
```

The program was ran using CUDA 8.0. Since Paralution code is no longer being mantained, it is incompatible with the latest versions of CUDA.

Before writing actual code, Paralution has to be initialized using the command `init_paralution()`. At the end of the code, Paralution has to be ended using the command `stop_paralution()`.

In the main function, the matrix `A` is first constructed and stored in the COO format in the form of three vectors `I`, `J`, and `V`. These arrays are dynamically allocated and contain the row-numbers, column-numbers, and values of the nonzero elements respectively. These arrays are given as input to the Paralution assemble function which transforms them internally to a matrix. Note that in order to transform them into a single matrix `A`, the matrix has to be initialized using Paralution types. The initialization and assembling is done as follows:

```
LocalMatrix<double> A;
A.Assemble(I,J,V,nnz,"A", sizeA, sizeA);
```

Note that the `Assemble` function also needs the number of nonzero elements (`nnz`), the name of the assembled matrix (`A`) and the size of the matrix in each dimension (`sizeA`).

Now, the other vectors `b` (the right-hand side), `x` (the unknowns), and `r` (the residual) are initialized, allocated and set to zero using Paralution types and built-in Paralution functions. This process is shown below for a single vector `x`:

```
LocalVector<double> x;
x.Allocate("x", sizeA);
x.Zeros();
```

Next, the matrix `A` and all the vectors are sent to the GPU device. Because of a convenient built-in Paralution function, the memory allocation in the GPU device and the data transfer to the GPU are all done at once using a single command for each matrix or vector:

```
A.MoveToAccelerator();
```

When this is done, the system of linear equations is solved. This can be done either with or without a preconditioner. First, the case without a preconditioner is covered (QR direct solver). Each Paralution Solver has a unique acronym, which is placed in front of the initialization of the solver (`QR`). At the end of the initialization, a user defined variable (`ds`) is given which is used later on to apply functions. The actual solving part always consists of the same three functions. The entire solver is shown below:

```
QR<LocalMatrix<double>,LocalVector<double>,double>ds;

ds.SetOperator(A);
ds.Build();

ds.Solve(b,&x);
```

Adding a preconditioner requires one additional initialization and function. The initialization of the preconditioner works in exactly the same way as the solver, but is always placed after the solver initialization. In the case of the GMRES iterative solver preconditioned with SGS, the code looks as follows:

```
GMRES<LocalMatrix<double>,LocalVector<double>,double>it;
SGS<LocalMatrix<double>,LocalVector<double>,double>p;

it.SetPreconditioner(p);
it.SetOperator(A);
it.Build();

it.Solve(b,&x);
```

Paralution offers extra functions that can be added to the solver. Some of these extra functions that turned out to be useful in this report are listed below. Note that these are added below the solver and preconditioner initialization but above the actual `Solve(b,&x)` function.

```
it.RecordResidualHistory();
it.InitMaxIter(10000);
it.RecordHistory("residuals.txt")
```

The `RecordResidualHistory()` writes residual details to the terminal. With `InitMaxIter()` the maximum amount of iterations can be adjusted. `RecordHistory(file.txt)` writes the residual after each iteration into a seperate text file.

Once the system has been solved, the residual vector $\mathbf{r}$ can be calculated as shown below. The `Apply` method performs the matrix multiplication $A\mathbf{x}$ and stores it at `&t`. The `AddScale` method subtracts $\mathbf{b}$ from this value. L_∞ and L_2 norms can then be caculated via the `Amax` and `Norm` methods respectively.

```
A.Apply(x,&r);
r.AddScale(b,-1);
double residual, maxres;
r.Amax(maxres);
residual = r.Norm();
```

The desired vectors $\mathbf{r}$ and $\mathbf{x}$ can then be moved back to the host CPU for output.

```
    r.MoveToHost();
    x.MoveToHost();
```

Lastly, the arrays on the device can be freed via the `Clear` method.

```
    b.Clear();
    A.Clear();
    x.Clear();
    r.Clear();
```

```
LocalMatrix<double> A;
A.Assemble(I,J,V,nnz,"A", sizeA, sizeA);

LocalVector<double> x;
x.Allocate("x", sizeA);
x.Zeros();

A.MoveToAccelerator();

QR<LocalMatrix<double>,LocalVector<double>,double>ds;
    ds.SetOperator(A);
    ds.Build();
    ds.Solve(b,&x);

x.MoveToHost();
x.Clear();C
```

References

[1] NVIDIA Corporation. *CUDA Programming Guide Version 4.2*, 2012.

[2] PARALUTION Labs. Paralution user manual v1.1.0, 2018.

[3] William C. Elmore and Mark A. Heald. *Physics of Waves (Dover Books on Physics)*. Dover Publications, 1985.

[4] R. Haberman. *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems: Pearson New International Edition*. Pearson Education Limited, 2013.

[5] J. van Kan, A. Segal, and F. Vermolen. *Numerical Methods in Scientific Computing*. Delft Academic Press / VSSD, 2014.

[6] Yousef Saad. *Iterative Methods for Sparse Systems (2nd Edition)*. SIAM, 2003.

[7] Tilo Arens. Helmholtz equation, 2001. Retrieved on the 9th of November 2018 from `https://people.maths.ox.ac.uk/trefethen/pdectb/helmholtz2.pdf`.

[8] Scipy. Compressed sparse row format (csr), n.d. Retrieved on the 5th of November 2018 from `https://www.scipy-lectures.org/advanced/scipy_sparse/csr_matrix.html`.

[9] IBM. Compressed-diagonal storage mode, n.d. Retrieved on the 22nd of October 2018 from `https://www.ibm.com/support/knowledgecenter/SSFHY8_5.4.0/com.ibm.cluster.essl.v5r4.essl100.doc/am5gr_cdst.htm`.

[10] Nathan Bell. Sparse matrix representations & iterative solvers, n.d. Retrieved on the 16 th of November 2018 from `https://www.bu.edu/pasi/files/2011/01/NathanBell1-10-1000.pdf`.

[11] S. Tomov H. Anzt and J. Dongarra. Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus. 2014. Retrieved on the 16th of November 2018 from `https://www.icl.utk.edu/files/publications/2014/icl-utk-772-2014.pdf`.

[12] Patrick R. Amestoy, Iain S. Duff, Jean-Yves Lexcellent, Yves Robert, François-Henry Rouet, and Bora Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. *SIAM Journal on Scientific Computing*, 34(4), 2012.

[13] Berkeley University of California. Sparse gaussian elimination, 1996. Retrieved on the 9th of November 2018 from `https://people.eecs.berkeley.edu/~demmel/cs267/lectureSparseLU/lectureSparseLU1.html`.

[14] C. Vuik & C. W. J. Lemmens. *Programming on the GPU with CUDA (Version 6.5)*. Delft University of Technology, 2015.

[15] Denis Demidov. Amgcl documentation release 1.2.0.post185, 2018. Retrieved on the 9th of November 2018 from `https://media.readthedocs.org/pdf/amgcl/latest/amgcl.pdf`.

[16] rocalution documentation, 2018. Retrieved on the 16th of November 2018 from `https://rocalution.readthedocs.io/en/latest/`.

[17] Cuda toolkit documentation, 2018. Retrieved on the 16th of November 2018 from `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[18] J. Vermeer. *Dictaat TN2244WI: Lineaire Algebra, deel 2 voor Technische Natuurkunde*. Faculteit Elektrotechniek, Wiskunde en Informatica, TU Delft, June 2017.