A GPU-based Laplacian Solver for Magnetostatic Boundary Value Problems

by

Welitharage Piyumi Madhubhashini

A Thesis submitted to the Faculty of Graduate Studies of
The University of Winnipeg
in partial fulfillment of the requirements of the degree of

Master of Science

Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada

Abstract

Modern graphics processing units (GPUs) have more computing power than CPUs, and thus, GPUs are proposed as more efficient compute units in solving scientific problems with large parallelizable computational loads. In our study, we present a GPU algorithm to solve a magnetostatic boundary value problem, which exhibits parallel properties.

In particular, we solve the Laplace equation to find the magnetic scalar potential in the region between two coaxial cylinders. This requires discretizing the problem domain into small cells and finding the solution at each node of the generated mesh. The smaller the cell size is the more accurate the solution will be. More accurate solution leads to a better estimation of the surface current needed to generate a uniform magnetic field inside the inner cylinder, which is the final goal. Although solving a mesh with a large number of smaller cells is computationally intensive, GPU computing provides techniques to accelerate performance.

The problem domain is discretized using the finite difference method (FDM) and the linear system of equations obtained from the FDM is solved by the successive over relaxation (SOR) method. The parallel program is implemented using CUDA framework. The performance of the parallel algorithm is optimized using several CUDA optimization strategies and the speedup of the parallel GPU implementation over the sequential CPU implementation is provided.

Keywords: Graphic processing units (GPUs), Parallel programming, Magnetostatics, Finite difference method (FDM), Successive over relaxation (SOR) method.

Acknowledgment

Above all, I would like to express my very profound gratitude to my supervisors, Dr. Christopher Bidinosti and Dr. Christopher Henry for their invaluable guidance, encouragement, and all the support at each step of my thesis and during the master's degree program. I am indebted to them for sharing their knowledge and experience with me, not only to complete the thesis successfully but also towards my future success.

Besides my advisors, I am most grateful to all the members of my thesis committee, Dr. Simon Liao and Dr. Blair Jamieson for reading my thesis and for their valuable comments.

A very special gratitude goes out to all of my friends and the staff at the department of computer science for their cheerful support throughout my master's degree life.

Finally, I must thank my husband and my parents for their countless love and encouragement.

Dedication

In dedication to my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A variety of natural phenomena such as fluid dynamics, electrostatics, heat, diffusion, and elasticity are explained using partial differential equations (PDEs). A PDE gives information about the behavior of the spatial and temporal changes of a system in the interior of the problem domain in terms of partial derivatives. There are several important PDEs in physics. Some examples, discussed in the following section, are the wave equation, heat equation, Poisson equation, and Laplace equation.

One of the most important partial differential operators which appears in most of the PDEs is called the *Laplacian operator*, written as ∇^2. In a Cartesian coordinate system, the Laplacian of a function u is given by the sum of second order derivatives of u with respect to each independent variable. In three dimensions, it appears as

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}. \tag{1.1}$$

In other coordinate systems, the Laplacian will have different forms, and the Laplacian in cylindrical coordinates will be discussed explicitly in later chapters. Some general PDEs that involve the Laplacian operator are described in the following section.

By solving a PDE, we find the unknown function u that satisfies the PDE within the given problem domain. In order to solve a PDE, the behavior of the function u on the boundaries of the problem domain needs to be given. A differential equation with a set of boundary conditions on the boundaries of the problem domain is called a *boundary value problem*, as discussed further below. The work in this thesis will focus on numerical solutions of the Laplace equation for magnetostatic applications, as described in Section 1.4. Analytic solutions to PDEs typically exist only for rather simple geometries with high symmetry, and in general, one must rely on numerical methods to find the solution. To make our solution as fast and accurate as possible, we have developed a parallel solver that runs on a graphics processing unit (GPU).

1.1 Examples of PDEs Involving the Laplacian Operator

The wave equation is

$$\nabla^2 u = \frac{1}{v^2}\frac{\partial^2 u}{\partial t^2}. \tag{1.2}$$

Here, the function u may represent the displacement from equilibrium of a vibrating string or of the vibrating medium (gas, liquid or solid); in electricity, u may be the current or potential along a transmission line [1]. The quantity v is the velocity of wave propagation and t is the time variable.

The heat/diffusion equation is

$$\nabla^2 u = \frac{1}{\alpha^2}\frac{\partial u}{\partial t}. \tag{1.3}$$

The function u may be the temperature varying with time in a region with no heat source, or u may be the concentration of a diffusing substance [1]. The quantity α^2 is the diffusion coefficient and t is the time variable.

The Poisson equation is given by

$$\nabla^2 u = \rho, \tag{1.4}$$

where ρ is known as the source term. When there is no source term, this reduces to the Laplace equation

$$\nabla^2 u = 0. \tag{1.5}$$

Both the Poisson and Laplace equations describe steady state situations, meaning that the equations have no dependence on time. As an example, in electrostatics,

$$\nabla^2 u = -4\pi\rho, \tag{1.6}$$

where u is the electrostatic potential and ρ is the electric charge density. In a charge free region, $\rho = 0$ and the electrostatic potential satisfies the Laplace equation [1]. Another example can be found in Gravity. The gravitational potential u in a region containing mass with mass density ρ is given by the Poisson equation

$$\nabla^2 u = 4\pi G\rho, \tag{1.7}$$

where G is the gravitational constant. When there is no mass in the region, the gravitational potential satisfies the Laplace equation [1]. In magnetostatics, the magnetic scalar potential in a current free region satisfies the Laplace equation. In our thesis, we solve the Laplace equation to find the magnetic scalar potential values in a current free region. Our magnetostatic application is explained in detail in Section 1.4.

1.2 Classification of Boundary Conditions

In order to solve a boundary value problem, the behavior of the unknown function u on each boundary of the region of interest should be defined. Boundary conditions can be classified into the following three different types:

- **Dirichlet Boundary Conditions**
  Here, the value of the unknown function u on the boundary is given:
  $$u = f,$$
  where f is a given function which is defined on the boundary.
- **Neumann Boundary Conditions**
  Here, the value of the normal derivative of u on the boundary is given:
  $$\frac{\partial u}{\partial n} = f,$$
  where n is the unit normal to the boundary surface and f is a given function defined on the boundary.
- **Robin Boundary Conditions**
  Here, a linear combination of the values of u and its normal derivative on the boundary are given:
  $$au + b\frac{\partial u}{\partial n} = f,$$
  where a, b are non zero parameters, n is the unit normal to the boundary surface, and f is a given function defined on the boundary.

There are various analytical and numerical solution strategies for solving a PDE with given boundary conditions. Analytical methods provide exact solutions, but it is hard to use analytic strategies when problems are complex. Hence, numerical methods have become more popular for solving PDEs occurring in real world applications. Even though numerical methods cannot provide the exact solution as do analytical methods, they are able to give approximate solutions with sufficient accuracy to use in practice. The next section discusses the procedure of solving a PDE numerically and presents different methods for doing so.

1.3 Solution Methods for PDEs

Some of the analytic strategies available to solve PDEs can be named as transform methods, separation of variables, and characteristics methods. Even though these are powerful methods, only a limited

number of PDEs can be solved using these analytic strategies [2]. Hence, most of the PDE problems which occur in real-world applications are solved numerically.

The numerical solution of a PDE is an approximation to the unknown function in the PDE on a finite discrete grid which represents the original domain. The process of producing a numerical approximation to an original physical phenomenon is given in Fig. 1.1 [3].

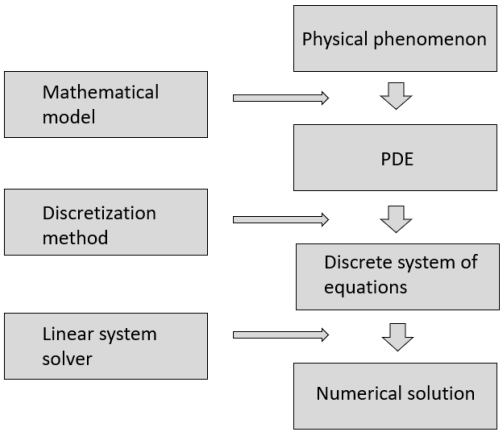


Figure 1.1: The flow of producing a numerical approximation to an original physical phenomenon. First, the real-world problem is converted into a mathematical model. The mathematical model consists of a PDE which describes the phenomenon, and the boundary conditions on the boundaries of the problem domain. If the problem is time-dependent, the model should also include the initial conditions of the system. Then, the PDE is discretized into a discrete system of linear equations using a discretization method. Each equation of the linear system represents a node on the problem space. Finally, the numerical solution corresponding to each node of the space is obtained by solving the linear system using a linear system solver.

The procedure begins by developing a mathematical model from the real physical application. The mathematical model consists of a PDE which describes the physical system within the region of interest, and the related boundary conditions. The unknown function u in the PDE can be 1D, 2D or 3D, according to the dimension of the problem space. As mentioned in Section 1.1, the same PDE can appear in various phenomena, but the boundary conditions and the solution domain are specific for each problem. If the problem depends on time, in addition to the boundary conditions, the behavior of the unknown function u at the initial state should also be given. In this thesis, we solve a magnetostatic application, which is a steady-state problem and there is no time variable.

The next step is choosing a discretization method which converts the continuous set of information given by the PDE into a discrete set of data, and that leads to a system of discrete equations. There are various discretization methods such as finite difference method, finite element method, finite volume method, and boundary element method. Among these, the most commonly used discretization methods are *finite difference method* (FDM) and *finite element method* (FEM). The finite difference approach is

suitable for the problems with simple geometric domains, which can be represented by uniform grids. Since our problem domain, as will discussed in Section 1.4, is not a complex geometric shape, FDM is suitable as the discretization method. More details about using FDM for our problem are given in Chapter 2. If the problem domain is very complicated, and when a non-uniform mesh is required to represent the problem domain, it is hard to formulate the problem with the finite difference method. In such cases, the more flexible finite element method is used.

In order for the numerical approximations to be useful, the error in the solution must be minimum. When increased accuracy is desired, finer meshes are used because finer meshes reduce the discretization error. However, this accuracy comes with the price of a large computational load and solving such a problem is time-consuming (see Fig. 1.2).

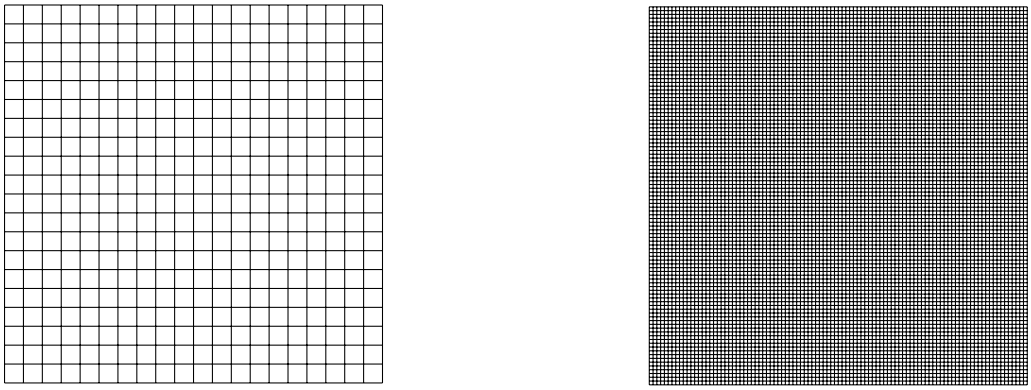


Figure 1.2: Overlaying a square problem domain with two different meshes. The grid in (a) represents a coarse mesh, which has 20 by 20 cells in the mesh. The grid in (b) is the same domain discretized into 100 by 100 cells, which is a finer mesh compared to the grid in (a). A finer mesh consists of a large number of nodes, therefore a large number of computations. Solving such a problem is expensive and time-consuming.

The final step of the numerical procedure is solving the large system of linear equations obtained in the previous step. Methods to solve such a linear system of equations fall under two categories as *direct* and *iterative*. Direct methods achieve the exact solution after a finite number of operations, while iterative methods achieve the exact solution after an infinite number of iterations [2]. Therefore, for iterative methods, there are some stopping criteria that can be used to stop the iterative procedure after a finite number of iterations, when a sufficient accuracy is reached.

Direct methods have some disadvantages over iterative methods when solving large systems. The number of operations required increases by the square of the number of nodes of the grid and the storage requirements are excessively large for large systems of equations. Hence, direct solvers are appropriate for solving small problems only. For large systems of equations such as what we get from

FDM, iterative solvers are more suitable [4]. There are two types of iterative methods, *stationary* and *non-stationary*. Stationary methods are older, simple, and easy to implement, but usually less effective. Non-stationary methods are recent developments and more complex than stationary methods [5]. Some examples of stationary iterative methods are the Jacobi method, Gauss-Seidel method, and successive over relaxation method (SOR). We chose the SOR iterative solver for our problem here, and a detailed discussion of the method is given in Chapter 2. Having a general idea about the numerical solution to a boundary value problem, in the following section, we describe our magnetostatic application, explain why we need to solve a boundary value problem, and give details of developing the boundary value problem from the known physical behavior of the system.

1.4 Problem of Interest

Our study is based on a uniform low field magnet design project proposed by Kyla Smith in her M.Sc. thesis [6]. Utilizing magnetic scalar potential Φ, her design method determines the required surface current distribution on a closed cylindrical surface that generates the desired uniform magnetic field inside the cylinder. The method rests on the fact that the difference in the magnetic scalar potential $\Delta\Phi$ across a surface boundary is directly related to the current distribution on the surface. Since the internal "target field" is known, and hence Φ_{inside}, the problem reduces to finding the scalar potential outside the cylindrical surface Φ_{outside} by numerically solving the Laplace equation in this region. The details are outlined below.

1.4.1 The Behaviour of Φ in Free Space

The behavior of Φ in free space is obtained as below. This analysis applies to any region where there is no current. According to one of Maxwell's equations,

$$\vec{\nabla} \times \vec{H} = \vec{J} + \frac{\partial \vec{D}}{\partial t}, \tag{1.8}$$

where $\vec{H}$ is the magnetic field strength, $\vec{J}$ is the current density, and $\vec{D}$ is the displacement field. In a region of free space, there is no current, so $\vec{J} = 0$. Also, for magnetostatic applications, there is no time dependence, so the second term on the right hand side in Eq. (1.8) is also zero. Therefore, one can write

$$\vec{\nabla} \times \vec{H} = 0. \tag{1.9}$$

Also, $\vec{H}$ can be expressed as the gradient of the scalar potential Φ as

$$\vec{H} = -\vec{\nabla}\Phi. \tag{1.10}$$

Another one of Maxwell's equations gives

$$\vec{\nabla} \cdot \vec{B} = 0, \tag{1.11}$$

where $\vec{B}$ is the magnetic field. The relationship between $\vec{B}$ and $\vec{H}$ when the magnetization is zero is given by

$$\vec{B} = \mu_0 \vec{H}, \tag{1.12}$$

where μ_0 is the permeability of free space. Using equations (1.11), (1.12), and (1.10), we can write

$$\vec{\nabla} \cdot \mu_0 \vec{H} = 0, \tag{1.13}$$

$$\vec{\nabla} \cdot (-\mu_0 \vec{\nabla} \Phi) = 0, \tag{1.14}$$

$$\nabla^2 \Phi = 0. \tag{1.15}$$

The Laplace equation, then, gives the behavior of Φ in any region where there is no current. We will use this result in particular to find Φ_{outside} by solving the Laplace equation using numerical methods as discussed in the previous sections. To solve this PDE, a boundary value problem is designed, and Section 1.4.4 gives more details about the mathematical model developed.

1.4.2 Choosing Φ Inside the Magnet

The magnetic scalar potential Φ inside the cylindrical volume obeys the Laplace equation as shown above. As a result, one can choose any solution, or sum of solutions, to this equation to obtain the magnetic field which needs to be generated. The desired magnetic field $\vec{B}$ here is a uniform field along the z-axis of the cylinder, which can be written as

$$\vec{B} = B\hat{z}. \tag{1.16}$$

Substituting Eq. (1.10) from equations (1.12) and (1.16) gives

$$\frac{B}{\mu_0}\hat{z} = -\vec{\nabla}\Phi. \tag{1.17}$$

In cylindrical coordinates, $\vec{\nabla}\Phi$ is given by

$$\vec{\nabla}\Phi = \hat{r}\frac{\partial \Phi}{\partial r} + \hat{\theta}\frac{1}{r}\frac{\partial \Phi}{\partial \theta} + \hat{z}\frac{\partial \Phi}{\partial z}. \tag{1.18}$$

According to Eq. (1.17), the gradient of Φ only has a component in $\hat{z}$ direction. Hence, Φ does not depend on θ or r. Therefore,

$$\vec{\nabla}\Phi = \hat{z}\frac{\partial \Phi}{\partial z}. \tag{1.19}$$

Substituting Eq. (1.17) from (1.19) gives

$$\frac{B}{\mu_0}\hat{z} = -\hat{z}\frac{\partial \Phi}{\partial z}, \tag{1.20}$$

$$-\frac{B}{\mu_0}\partial z = \partial \Phi, \tag{1.21}$$

$$-\frac{B}{\mu_0}\int \partial z = \int \partial \Phi, \tag{1.22}$$

$$\Phi_{\text{inside}} = -\frac{Bz}{\mu_0}. \tag{1.23}$$

One can easily verify that Eq. (1.23) is indeed a solution to the Laplace equation. Hence, Φ_{inside} has been set according to the desired magnetic field $\vec{B}$.

With Φ_{inside} specified as above and Φ_{outside} to be determined from numerical methods, one can now determine the discontinuity of the scalar potential at the boundary of the cylindrical surface of the magnet, which is related to a surface current density as described in the next section. The derivations given below are based on Section 8.5 from Ref. [7].

1.4.3 Relationship between $\Delta\Phi$ and Surface Current Density

Let us consider the current carrying surface shown in Fig. 1.3. According to Ampère's continuity condition, a surface current density $\vec{K}$ causes a discontinuity in the magnetic field intensity $\vec{H}$ and can be written as

$$\hat{n} \times (\vec{H}^a - \vec{H}^b) = \vec{K}. \tag{1.24}$$

Here, a and b denotes the two regions either side of the surface (see Fig. 1.3). The variables $\vec{H}^a$ and $\vec{H}^b$ are the magnetic field intensity on the region a and the region b respectively. The vector $\hat{n}$ is the unit normal vector pointing from region b to region a. For a region of free space where there is no current, the curl of $\vec{H}$ equals to zero. Therefore, we can write $\vec{H}$ using magnetic scalar potential Φ as

$$\vec{H} = -\vec{\nabla}\Phi. \tag{1.25}$$

On a current carrying surface, the curl of $\vec{H}$ is not equal to zero. Hence, the magnetic scalar potential Φ has no meaning on a such surface. Equation (1.24) states that the tangential component of $\vec{H}$ is discontinuous where there is a surface current density $\vec{K}$. Therefore, it is expected that Φ will also discontinuous on a current carrying surface.

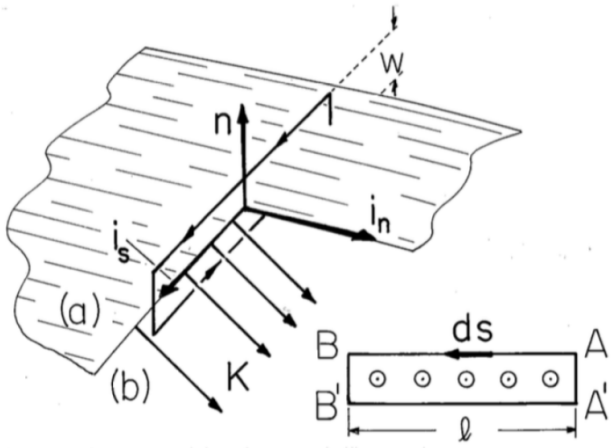


Figure 1.3: The figure from [7] shows a diagram of a current carrying surface. Here a and b refer to either side of the surface and w is the width of the surface. The vector $\hat{n}$ is a unit normal vector pointing from b to a.

According to the integral form of Ampère's law,

$$\oint_C \vec{H} \cdot d\vec{s} = \oint_S \vec{J} \cdot d\vec{a}. \tag{1.26}$$

By following the contour integral around the four segments $AB, BB', B'A', A'A$ of Fig. 1.3 as the width w goes to zero, the left hand side of Eq. (1.26) becomes

$$\int_A^B (\vec{H}^a \cdot \hat{i}_s)\, dl + 0 + \int_{B'}^{A'} (\vec{H}^b \cdot (-\hat{i}_s))\, dl + 0 = \int_A^B (\vec{H}^a - \vec{H}^b) \cdot \hat{i}_s\, dl. \tag{1.27}$$

Substituting $\vec{H}$ from Eq. (1.25) gives

$$\int_A^B (\vec{\nabla}\Phi^b - \vec{\nabla}\Phi^a) \cdot \hat{i}_s\, dl. \tag{1.28}$$

The current density $\vec{J}$ in Eq. (1.26) can be written as

$$\vec{J} = \vec{K}\delta(S), \tag{1.29}$$

where S refers to the surface and $\delta(S)$ is the Dirac delta function, which is zero outside the surface. The function $\delta(S)$ always returns one, when integrating over the width of the Amperian loop of the right hand side of Eq. (1.26), regardless of the value of the width w. As a result, the right hand side of Eq. (1.26) becomes

$$\int_A^B (\vec{K} \cdot \hat{i}_n)\, dl. \tag{1.30}$$

Now, Eq. (1.26) can be written as

$$\int_A^B (\vec{\nabla}\Phi^b - \vec{\nabla}\Phi^a) \cdot \hat{i}_s \, dl = \int_A^B (\vec{K} \cdot \hat{i}_n) \, dl,$$

$$\int_A^B \vec{\nabla}\Phi^b \cdot \hat{i}_s \, dl - \int_A^B \vec{\nabla}\Phi^a \cdot \hat{i}_s \, dl = \int_A^B (\vec{K} \cdot \hat{i}_n) \, dl. \tag{1.31}$$

The gradient integral theorem [7, Section 4.1] states that

$$\int_A^B \vec{\nabla}\Phi \cdot d\vec{s} = \Phi(B) - \Phi(A). \tag{1.32}$$

Now, we can evaluate the integrals on the left hand side of Eq. (1.31) to give

$$(\Phi_B^b - \Phi_A^b) - (\Phi_B^a - \Phi_A^a) = (\Phi_{B'} - \Phi_B) - (\Phi_{A'} - \Phi_A) = \Delta\Phi_B - \Delta\Phi_A \, . \tag{1.33}$$

Hence,

$$\Delta\Phi_B - \Delta\Phi_A = \int_A^B (\vec{K} \cdot \hat{i}_n) \, dl. \tag{1.34}$$

This implies that the amount of integrated surface current in the Amperian loop is the difference between the discontinuity of Φ at either end of the loop. That is, the difference of discontinuity in Φ between the locations A and B is the total current passing normal to the strip $AA'B'B$. As a result, evenly spaced contours of $\Delta\Phi$ give a discrete wire winding pattern that can be used to construct the desired magnet.

1.4.4 Boundary Value Problem

Now we need to develop a boundary value problem to solve for Φ_{outside} by defining the region where we want to solve the problem and determining the behavior of Φ on the boundaries of the region of interest. This section gives details about setting the problem domain and constructing boundary conditions as described in Chapter 3 of Ref. [6].

Here we need to solve the Laplace equation in the region outside the cylindrical magnetic surface. Therefore, the setup is two co-axial cylinders as shown in Fig. 1.4. The inner cylinder represents the surface of the magnet and the outer cylinder is to bound the problem domain. Since we intend to use the finite difference method, it requires the problem domain to be a closed region. Hence, this outer cylinder does not physically exist and is used only for the computational purpose. Now having a closed region, we can solve the Laplace equation in the region between two cylinders to find Φ_{outside}. Even though we solve the Laplace equation in a closed region, what we really need is a free-space solution for

Φ outside the inner cylinder. Therefore, we solve the numerical problem several times by expanding the outer boundary until the solution converges to a true free-space solution. The problem is solved with the cylindrical coordinate system (r, θ, z), where r is the radial coordinate, θ is the azimuthal coordinate, and z is the axial coordinate (see the first image of Fig. 1.5).

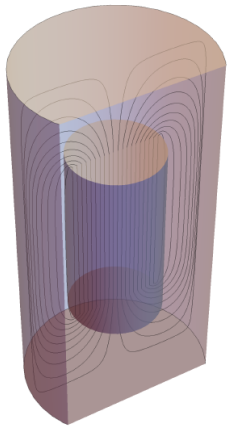


Figure 1.4: To solve the Laplace equation in the region outside the magnet, the problem domain is set using two co-axial cylinders. This figure from Ref. [6] shows the inner cylinder and the cross-sectional view of the outer cylinder. The inner cylinder represents the magnet and the outer cylinder is used to bound the problem domain. Laplace equation is solved in the region between two cylinders. Since we require a free-space solution to Φ_{outside}, the numerical problem is solved several times by expanding the outer cylinder until the solution converges to a free-space solution.

Boundary Conditions

By considering the symmetry of the problem, the 3D solving region can be reduced to a 2D region as shown in Fig. 1.5. The reduced domain D depends on only the coordinates r and z. As a result, the computational load and the memory requirement of the problem are greatly reduced.

The blue region of Fig. 1.6 shows the reduced problem domain. There are 6 boundaries and they are shown by red line segments. Boundaries 1 and 2 are on the outer cylinder; boundaries 3 and 6 are in between two cylinders; and boundaries 4 and 5 are on the inner cylinder. Now we need to define the behaviour of Φ on each of these boundaries and a detailed discussion on deriving the boundary conditions is given in Section 3.4 of Ref. [6]. The outer boundary is considered a superconductor, hence, it has zero flux passing through it. As a result, the normal derivative of Φ is zero all-over the surface of the outer boundary. The normal direction of Boundary 1 and Boundary 2 are $\hat{z}$ and $\hat{r}$ respectively. Hence, $\partial\Phi/\partial z = 0$ on Boundary 1 and $\partial\Phi/\partial r = 0$ on Boundary 2.

Since the medium is the same for either side of Boundary 3, the condition on this boundary is given by

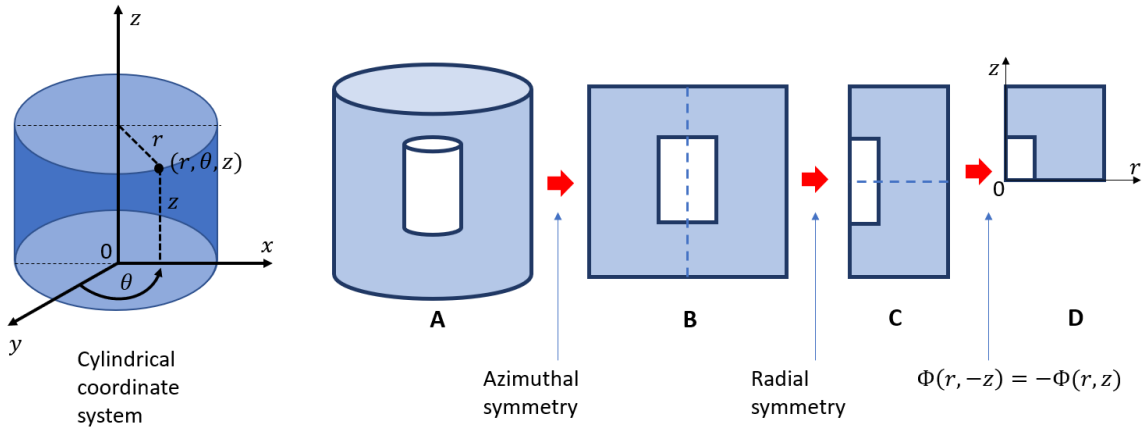


Figure 1.5: This figure illustrates the reduction of the problem space by symmetry. The blue area of region A is the full problem domain. Since the solution does not depend on θ, it reduces to the 2D blue area of region B. Because of the redial symmetry, we can further reduce the problem domain from B to C. Finally, C can be reduced to D, since the Φ values of the lower half of the domain C are simply the negative values of the upper half of C.

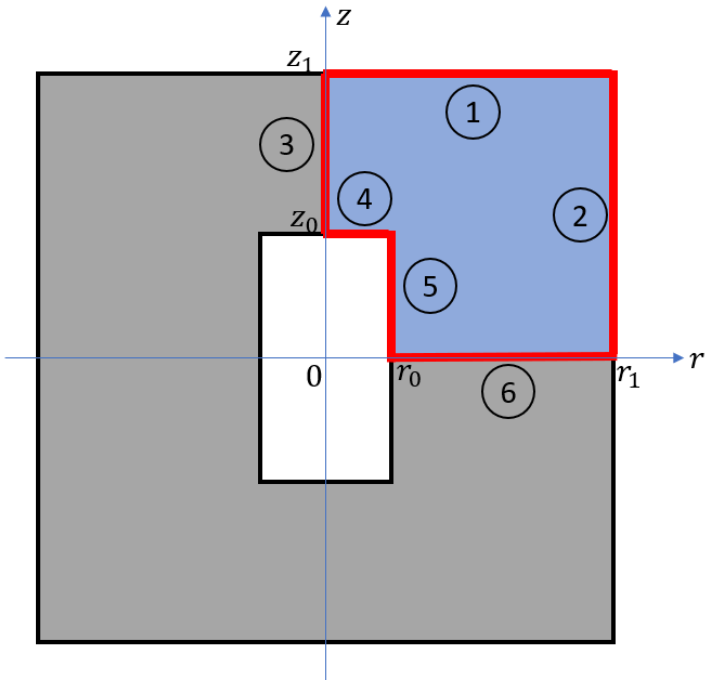


Figure 1.6: The blue region indicates the reduced problem domain. Six boundaries which enclose the problem domain are shown by red line segments. r_0 and z_0 are the radius and the half-height of the inner cylinder; r_1 and z_1 are the radius and the half-height of the outer cylinder. Boundary 1 and Boundary 2 are on the outer cylinder. Boundary 4 and Boundary 5 are on the inner cylinder. Boundary 3 and Boundary 6 are in the space between two cylinders.

$\partial\Phi/\partial r = 0$. Boundary 4 has magnetic flux as field lines enter and exit from the top and bottom caps of the inner cylinder. The condition on Boundary 4 is set according to the magnetic field B which we need to generate and is given as $\partial\Phi/\partial z = -B/\mu_0$, where μ_0 is the permeability of free-space. The magnetic field B inside the inner cylinder is along the $\hat{z}$ direction. Hence, Boundary 5 has zero magnetic flux and the condition on Boundary 5 can be written as $\partial\Phi/\partial r = 0$. The scalar potential Φ inside the inner cylinder at $z = 0$ is zero. Hence, the condition on Boundary 6 is set as $\Phi = 0$. Table 1.1 shows the derived conditions for all 6 boundaries and their types.

Table 1.1: Boundary conditions of the reduced problem domain.

Number	Boundary	Condition	Type
1	$0 \leq r \leq r_1$ and $z = z_1$	$\frac{\partial\Phi}{\partial z} = 0$	Neumann
2	$r = r_1$ and $0 \leq z \leq z_1$	$\frac{\partial\Phi}{\partial r} = 0$	Neumann
3	$r = 0$ and $z_0 \leq z \leq z_1$	$\frac{\partial\Phi}{\partial r} = 0$	Neumann
4	$0 \leq r \leq r_0$ and $z = z_0$	$\frac{\partial\Phi}{\partial z} = \frac{-B}{\mu_0}$	Neumann
5	$r = r_0$ and $0 \leq z \leq z_0$	$\frac{\partial\Phi}{\partial r} = 0$	Neumann
6	$r_0 \leq r \leq r_1$ and $z = 0$	$\Phi = 0$	Dirichlet

To make the solution of the developed boundary value problem as fast and accurate as possible, we solve the problem with the help of a GPU by developing a parallel numerical solver. Following section briefly explains serial vs parallel programming, high computational power, and the evolution of GPUs as computing devices.

1.5 GPUs for Performance Acceleration

Many scientific and engineering problems such as climate modeling, computer simulations, DNA analysis, data mining, and data visualization take a long time to execute, because of the heavy load of data and computation. When considering our magnetostatic problem, as discussed in Section 1.3 and 1.4, a sufficiently expanded problem domain (to achieve a free-space solution) with a finer mesh (to decrease the discretization error) gives better numerical approximations, but involves a large computational load. Therefore, more computational power is needed to efficiently solve such a compute-intensive problem.

Until recently, single processor performance has been increased by increasing the number of transistors in the integrated circuit. However, because of the heat generation, increasing power consumption as well as a rapidly approaching physical limit to transistor size, it has become impossible to continue to increase the speed of single processor systems [8]. As a result, the industry has decided to develop multi-processor systems and hence, the performance improvement is achieved by *parallelism* [9].

Traditionally, software has been written for serial computation. When executing a serial program, the computational problem is broken down to a discrete series of instructions. Then, these instructions are executed sequentially one after another on a single processing core (Fig. 1.7). Even the system consists of multiple processors, a serial program that is written to run on a single processor is unaware of the existence of multiple processors. Hence, the performance of a serial program on a system with multiple processors will be the same as its performance on a single processor of the multiprocessor system [9]. On the other hand, parallel programs are written to exploit the power of multiple processors of a multiprocessor system. When executing a parallel program, the problem is broken into discrete instructions which can be run concurrently. Instructions can be run concurrently if one does not consume the output of another (i.e. each instruction is independent). Each concurrent instruction is then executed simultaneously on different cores (Fig. 1.8). Therefore, with parallel computing, certain problems can be solved in less time than with serial computing.

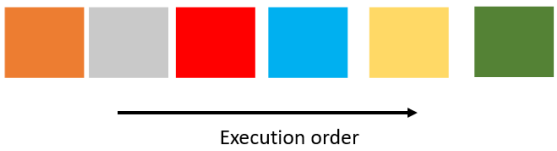


Figure 1.7: When executing a sequential program, the computational problem is divided into a discrete series of instructions. These instructions are executed sequentially one after another on a single processing core (e.g. a central processing unit). A sequential program does not run on more than one processor even the system consists of multiple processors.

If we need a serial program to run in parallel on a multi-processor system, the program should be re-written to divide the work load among multiple cores. However, not all algorithms can be accelerated through parallelism. The level of speedup that can be expected from parallelizing an application depends on the portion of the application that can be parallelized. If the parallel portion is very small, a sequential algorithm will perform well and it will be hard to get significant performance gain with parallelism [10]. Thus, before parallelizing an algorithm to run on a multi-processor system, the serial and parallel portion of the application should be well recognized.

High computational power is available in several forms. Some examples are high performance computing

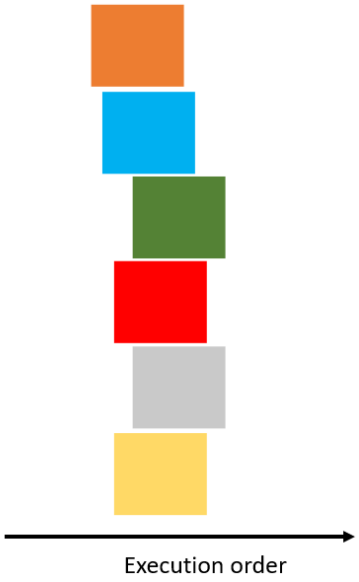


Figure 1.8: When executing a parallel program, the problem is broken into discrete instructions which can run concurrently. Instructions can run concurrently if the instructions are independent. These concurrent instructions can be further divided into a discrete series of sequential instructions. Each concurrent instruction is executed simultaneously on different cores (e.g. on a multi-processor system).

(HPC) clusters, supercomputers, HPC cloud computing, grid computing, and graphics processing units (GPUs). Nowadays, among these resources, GPUs have become economically attractive tools which consist of thousands of cores, hence, thousands of independent instructions can be executed simultaneously. Hence, GPUs have become popular among program developers and GPUs are used to reduce the runtime of many compute-intensive applications in both academic and industrial work [10].

GPUs were originally designed to accelerate only graphic tasks. Later, researchers explored the possibilities to use GPUs for general purpose computing. Since standard graphics APIs such as OpenGL and DirectX were still the only way to interact with a GPU, anyone who needs to use GPUs to perform general purpose computing would need to learn OpenGL or DirectX and to code the problem as a graphic task [8]. In 2006, NVIDIA introduced *CUDA*, a parallel computing platform that can be used for general purpose computing on NVIDIA GPUs. With CUDA, developers are able to parallelize programs which are written in popular programming languages such as C/C++, Python, Fortran, and MATLAB. With the introduction of the programming language *OpenCL* by Khronos Group in 2010, programmers can now develop GPU applications on hardware with GPUs from different vendors (e.g. AMD, Intel, ATI, NVIDIA). Today, there is a vast community of GPU programmers and many of them have reported 10 to 100 times speedup of their applications with GPU computing [11].

The most computationally intensive part of our numerical problem is updating each node of the grid with the iterative solver. In order to implement this compute-intensive part on a GPU, the iterative algorithm should exhibit parallel properties. Since the original SOR method does not exhibit parallel properties, we use the parallel version of the algorithm, which is called the Red-Black SOR method. Our GPU algorithm is implemented using the CUDA platform and tested with two NVIDIA GPUs, namely a GeForce GTX 960M and a Tesla P100.

1.6 Contribution of the Thesis

In her thesis [6], Kyla Smith successfully designed a magnet for low-field magnetic resonance studies, by solving Φ_{outside} using the Finite Element Method within the software package Mathematica. The code was run sequentially on her laptop and results were ultimately limited by compute time and memory. Future implementations of this design method could involve different geometries and different target field distributions. It might also involve an iterative approach to further refine and optimize coil winding patterns. As a result, it is very desirable to (i) develop a parallel Laplacian solver for increased speed, (ii) move the calculation to a server with more memory and compute capability, and (iii) use home-written rather than commercial software so the code can be run on any system. To achieve these goals, our research proposes a parallel GPU algorithm to solve the same numerical problem using the finite difference method. The main focus of this thesis is the development of a parallel algorithm to achieve performance acceleration over a CPU sequential algorithm, which ultimately allows the code to run faster, or with a finer mesh (for greater accuracy), or both. Thus, the contribution of this thesis is to parallelize the compute-intensive numerical part of the magnet design project in Ref. [6] using GPUs.

Chapter 2

Methodology

In this chapter, we explain the numerical methods used to solve the magnetostatic problem described in the previous chapter. In particular, we use the finite difference method (FDM) to discretize the problem into a system of linear equations and the successive over relaxation (SOR) iterative solver to solve the obtained system of equations. The chapter consists of two sections. The first section is about the FDM, and the second section provides details about the iterative solver. We review the fundamentals of the FDM, then describe how we apply the method to our magneto-static problem. We next discuss the SOR iterative solver, the determination of the optimum relaxation parameter, the convergence of the iterative procedure and the stopping criteria. Finally, we discuss the red-black SOR method, which is the parallel version of the original SOR solver.

2.1 Finite Difference Method

2.1.1 Finite Differences

For the simplicity, let us first consider the 1D case. Suppose that the behavior of an unknown function $u(x)$ on the interval $(0, L)$ is given by some PDE. As the first step, assuming equal mesh spacing h, we discretize the given problem interval as shown in Fig. 2.1. The interval is divided into a number of equal segments N and there the number of nodes is $N + 1$. Suppose that u_i is a node within the interval $(0, L)$. Using Taylor series expansion, we can write the value of the node u_{i+1} as

$$u_{i+1} = u_i + h\frac{\partial u_i}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 u_i}{\partial x^2} + \frac{h^3}{3!}\frac{\partial^3 u_i}{\partial x^3} + ... \tag{2.1}$$

That is, if we know the value of u_i and the values of its derivatives at the node i, then we can find an approximation to the nearby node u_{i+1}. Here we say approximation because the exact value of

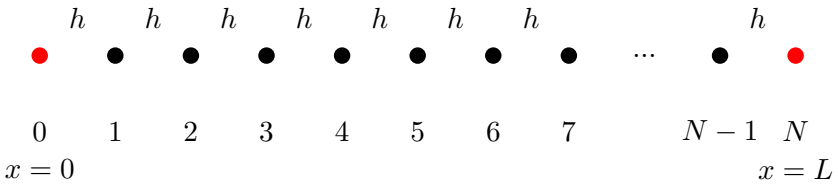


Figure 2.1: Discretizing the problem interval $[0, L]$ into an N number of equal segments. The mesh spacing between any two nodes is h. The total number of nodes along the interval is given by $(L/h) + 1$. The nodes from 1 to $N - 1$ are interior nodes of the problem domain. Node 0 and node N are boundary nodes.

u_{i+1} is given by the summation of an infinite number of terms. Now using Eq. (2.1), we can write the derivative $\frac{\partial u_i}{\partial x}$ in terms of u_i and u_{i+1} as

$$\frac{\partial u_i}{\partial x} = \frac{u_{i+1} - u_i}{h} - \frac{h}{2!}\frac{\partial^2 u_i}{\partial x^2} - \frac{h^2}{3!}\frac{\partial^3 u_i}{\partial x^3} - ...,$$

which can be written as

$$\frac{\partial u_i}{\partial x} = \frac{u_{i+1} - u_i}{h} + O(h). \tag{2.2}$$

Here, $O(h)$ is called the *leading error term* and it reflects the truncation error. For sufficiently small h, the truncation error will be dominated by the term $\frac{h}{2!}\frac{\partial^2 u_i}{\partial x^2}$, and all the other terms are negligible compared to this term [12]. Hence, the error is expected to behave roughly like a constant times h, where the constant has the value of $\frac{1}{2!}\frac{\partial^2 u_i}{\partial x^2}$, and therefore, we write this leading error using $O(h)$. In Eq. (2.2), the error is of order h and it can be said that the approximation to $\frac{\partial u_i}{\partial x}$ has first order of accuracy. This leading error decreases when we reduce the mesh spacing h [2].

Equation (2.2) gives a *forward difference approximation* to the first order derivative. Forward difference means that the approximation is obtained using u_i and the value next to it, u_{i+1}. We can also derive the *backward difference formula* for the first order derivative using Taylor series expansion of u_{i-1} as

$$\frac{\partial u_i}{\partial x} = \frac{u_i - u_{i-1}}{h} + O(h). \tag{2.3}$$

There is another form of approximation, which is called the *central difference approximation*. This can be generated using the expansions of both u_{i+1} and u_{i-1} as follows:

$$u_{i+1} = u_i + h\frac{\partial u_i}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 u_i}{\partial x^2} + O(h^3), \tag{2.4}$$

$$u_{i-1} = u_i - h\frac{\partial u_i}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 u_i}{\partial x^2} + O(h^3). \tag{2.5}$$

Subtracting the above two equations gives

$$\frac{\partial u_i}{\partial x} = \frac{u_{i+1} - u_{i-1}}{2h} + O(h^2). \tag{2.6}$$

This central difference approximation is second order correct, and therefore, better than Eq. (2.2) and (2.3). The central difference approximation for the second partial derivative $\frac{\partial^2 u_i}{\partial x^2}$ can be obtained using Taylor series expansion of u_{i+1} and u_{i-1}. The values u_{i+1} and u_{i-1} can be written as

$$u_{i+1} = u_i + h\frac{\partial u_i}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 u_i}{\partial x^2} + \frac{h^3}{3!}\frac{\partial^3 u_i}{\partial x^3} + O(h^4), \text{ and} \tag{2.7}$$

$$u_{i-1} = u_i - h\frac{\partial u_i}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 u_i}{\partial x^2} - \frac{h^3}{3!}\frac{\partial^3 u_i}{\partial x^3} + O(h^4). \tag{2.8}$$

Adding the above two equations produces

$$h^2\frac{\partial^2 u_i}{\partial x^2} = u_{i+1} - 2u_i + u_{i-1} + O(h^4),$$

and dividing both sides by h^2 gives

$$\frac{\partial^2 u_i}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + O(h^2). \tag{2.9}$$

The leading error involved in Eq. (2.9) is $O(h^2)$. This is a second order correct central difference approximation to the second order derivative. We can also derive higher order accurate formulas for partial derivatives by involving more terms. However, we solve our problem using more accurate second order central difference approximations.

Now, let us consider the finite difference approximations for nodes on a 2D mesh. In 2D, a node on the mesh is defined as $u_{i,j}$ as shown in Fig. 2.2. The finite difference formulas for the first and second order derivatives in 2D with respect to one independent variable are also generated as in the 1D case. They become

$$\frac{\partial u_{i,j}}{\partial x} = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + O(h^2), \tag{2.10}$$

$$\frac{\partial u_{i,j}}{\partial y} = \frac{u_{i,j+1} - u_{i,j-1}}{2h} + O(h^2), \tag{2.11}$$

$$\frac{\partial^2 u_{i,j}}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + O(h^2), \text{ and} \tag{2.12}$$

$$\frac{\partial^2 u_{i,j}}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} + O(h^2). \tag{2.13}$$

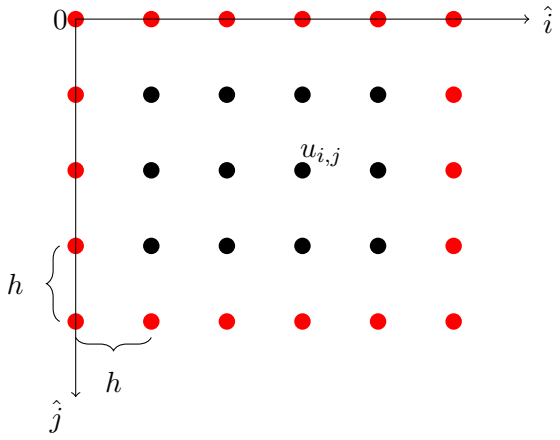


Figure 2.2: Discretizing a 2D square domain into equal cells. The mesh spacing is h on both directions. The red nodes indicate boundary nodes and all the black nodes are interior nodes. A node on the mesh is defined using two coordinates as $u_{i,j}$.

2.1.2 Finite Difference Approximation to the Laplace Equation

Let us consider our magnetostatic problem described in Section 1.4. The PDE involved in the boundary value problem is the Laplace equation. Here we first write the Laplace equation in a cylindrical coordinates as

$$\frac{\partial^2\Phi}{\partial r^2} + \frac{1}{r}\frac{\partial\Phi}{\partial r} + \frac{1}{r^2}\frac{\partial^2\Phi}{\partial\theta^2} + \frac{\partial^2\Phi}{\partial z^2} = 0. \tag{2.14}$$

Our problem does not depend on the azimuthal coordinate θ. Hence, the derivative with respect to θ is equal to zero. Therefore, the 2D Laplace equation for the reduced problem can be written as

$$\frac{\partial^2\Phi}{\partial r^2} + \frac{1}{r}\frac{\partial\Phi}{\partial r} + \frac{\partial^2\Phi}{\partial z^2} = 0. \tag{2.15}$$

Now suppose that i represents the points along the r direction and j represents the points along the z direction. The problem domain is discretized by making the grid spacing along the r direction δr, and the grid spacing along the z direction δz. Now, each derivative in Eq. (2.15) is replaced by the relevant second order correct central difference approximations as

$$\frac{(\Phi_{i-1,j} - 2\Phi_{i,j} + \Phi_{i+1,j})}{\delta r^2} + \frac{1}{r_i}\frac{\Phi_{i+1,j} - \Phi_{i-1,j}}{2\delta r} + \frac{(\Phi_{i,j-1} - 2\Phi_{i,j} + \Phi_{i,j+1})}{\delta z^2} = 0, \tag{2.16}$$

where $r_i = i \times \delta r$. Rearranging the terms gives

$$(\Phi_{i-1,j} - 2\Phi_{i,j} + \Phi_{i+1,j}) + \frac{1}{2i}(\Phi_{i+1,j} - \Phi_{i-1,j}) + \frac{\delta r^2}{\delta z^2}(\Phi_{i,j-1} - 2\Phi_{i,j} + \Phi_{i,j+1}) = 0. \tag{2.17}$$

Letting $\alpha = \frac{1}{2i}$ and $\gamma = (\frac{\delta r}{\delta z})^2$, we can write $\Phi_{i,j}$ as

$$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(\Phi_{i-1,j}(1-\alpha) + \Phi_{i+1,j}(1+\alpha) + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma)). \tag{2.18}$$

This is the finite difference formula to compute all the interior nodes of the problem domain. The value of $\Phi_{i,j}$ is given by the weighted sum of the four neighboring values of $\Phi_{i,j}$. These 5 nodes together is called a *five-point stencil* [13] (see Fig. 2.3). Computing one $\Phi_{i,j}$ value using its four neighboring nodes is called one stencil computation.

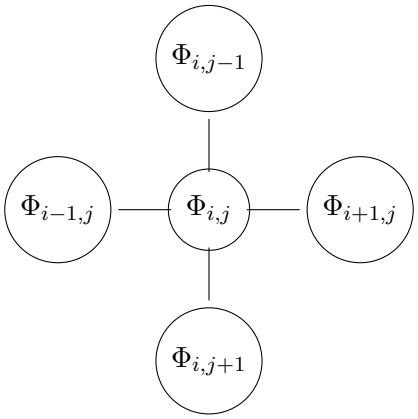


Figure 2.3: The five-point stencil. If $\Phi_{i,j}$ is a node on the mesh, the new value of the node is computed using the weighted sum of the four neighboring nodes of $\Phi_{i,j}$. Updating one node using the values of its neighbouring nodes is called a one stencil computation. If there are N number of nodes on the mesh, the number of stencil computations required to update the whole mesh is N.

2.1.3 Applying Boundary Conditions

Now we have a formula to compute all the interior nodes of the mesh. As the next step, we examine the behavior of Φ on the boundaries of the mesh. As shown in Fig. 2.4, there are six boundaries which enclose the problem domain. The 6^{th} boundary has a Dirichlet condition, and hence, we can apply the given values directly as the solution to Φ on Boundary 6. All the other five boundaries have Neumann conditions. Therefore, we derive finite difference formulas for each of these boundaries by applying given boundary conditions to Eq. (2.18) as below.

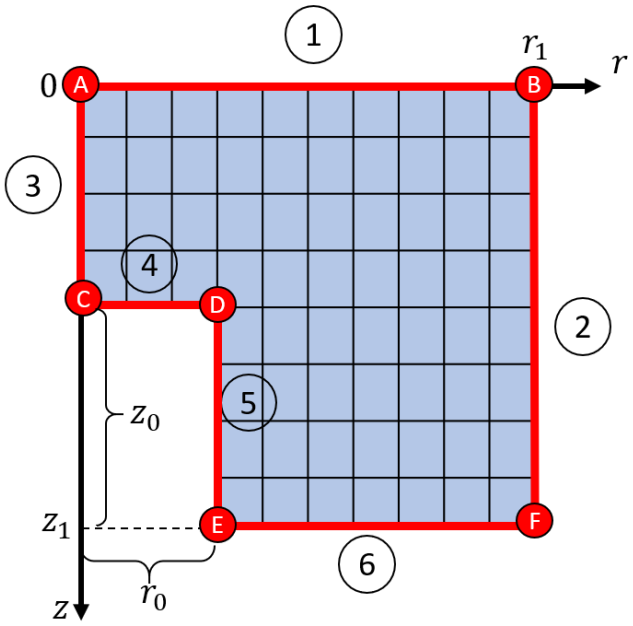


Figure 2.4: The discretized problem domain. Six boundaries which enclose the problem space are shown by red line segments, and 6 corner nodes A, B, C, D, E, and F are marked by red circles. The values r_0 and z_0 are the radius and the half-height of the inner cylinder. The values r_1 and z_1 are the radius and the half-height of the outer cylinder.

Boundary 1 – boundary condition: $\frac{\partial \Phi}{\partial z} = 0$.

Applying second order correct central difference formula for the first order derivative gives

$$\frac{\Phi_{i,j+1} - \Phi_{i,j-1}}{2\delta z} = 0, \tag{2.19}$$

which reduces to the equation

$$\Phi_{i,j-1} = \Phi_{i,j+1}. \tag{2.20}$$

When considering the nodes along Boundary 1, nodes $\Phi_{i,j-1}$ is outside the problem domain. Having the Eq. (2.20), we can replace $\Phi_{i,j-1}$ with $\Phi_{i,j+1}$ and write Eq. (2.18) as

$$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(\Phi_{i-1,j}(1-\alpha) + \Phi_{i+1,j}(1+\alpha) + 2\Phi_{i,j+1}(\gamma)). \tag{2.21}$$

This is the finite difference formula for the nodes on Boundary 1. The value of $\Phi_{i,j}$ is computed using the three neighboring nodes of $\Phi_{i,j}$ inside the problem domain.

Boundary 2 – boundary condition: $\frac{\partial \Phi}{\partial r} = 0$.

Applying the finite difference approximation gives

$$\frac{\Phi_{i+1,j} - \Phi_{i-1,j}}{2\delta r} = 0,$$

which reduces to the condition

$$\Phi_{i+1,j} = \Phi_{i-1,j}. \tag{2.22}$$

Replacing $\Phi_{i+1,j}$ by $\Phi_{i-1,j}$ in Eq. (2.18), we get the finite difference formula for the nodes on Boundary 2 as

$$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i-1,j} + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma)). \tag{2.23}$$

Boundary 3 – boundary condition: $\frac{\partial \Phi}{\partial r} = 0$.

Let us look again at our 2D Laplace equation in a cylindrical coordinates:

$$\frac{\partial^2 \Phi}{\partial r^2} + \frac{1}{r}\frac{\partial \Phi}{\partial r} + \frac{\partial^2 \Phi}{\partial z^2} = 0. \tag{2.24}$$

On Boundary 3, $r = 0$ and $\frac{\partial \Phi}{\partial r} = 0$. Then, the term $\frac{1}{r}\frac{\partial \Phi}{\partial r}$ assumes the indeterminate form $0/0$ on Boundary 3. Hence, we need to find a valid form of Eq. (2.24) to apply for the nodes on Boundary 3. As described in Ref. [14], we find the limit of the term $\frac{1}{r}\frac{\partial \Phi}{\partial r}$ as $r \longrightarrow 0$, using L'Hospitals rule as

$$\lim_{r \to 0} \frac{\partial \Phi / \partial r}{r} = \frac{\partial^2 \Phi}{\partial r^2}.$$

Now we can replace the term $\frac{1}{r}\frac{\partial \Phi}{\partial r}$ with $\frac{\partial^2 \Phi}{\partial r^2}$, and Eq. (2.24) becomes

$$2\frac{\partial^2 \Phi}{\partial r^2} + \frac{\partial^2 \Phi}{\partial z^2} = 0. \tag{2.25}$$

Eq. (2.25) is a valid form to apply on Boundary 3. By replacing these derivatives with second order accurate central difference approximations, we get a finite difference formula as

$$2\frac{(\Phi_{i-1,j} - 2\Phi_{i,j} + \Phi_{i+1,j})}{\delta r^2} + \frac{(\Phi_{i,j-1} - 2\Phi_{i,j} + \Phi_{i,j+1})}{\delta z^2} = 0. \tag{2.26}$$

Now let us consider the finite difference formula for the boundary condition $\frac{\partial \Phi}{\partial r} = 0$ on Boundary 3:

$$\frac{\Phi_{i+1,j} - \Phi_{i-1,j}}{2\delta r} = 0,$$

which reduces to the condition

$$\Phi_{i-1,j} = \Phi_{i+1,j}. \tag{2.27}$$

Applying Eq. (2.27) to (2.26) gives the finite difference formula for the nodes on Boundary 3 as

$$\Phi_{i,j} = \frac{1}{2(2+\gamma)}(4\Phi_{i+1,j} + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma)). \tag{2.28}$$

Boundary 4 – boundary condition: $\frac{\partial \Phi}{\partial z} = -\frac{B}{\mu_0}$.

Applying the finite difference approximation gives

$$\frac{\Phi_{i,j+1} - \Phi_{i,j-1}}{2\delta z} = -\frac{B}{\mu_0},$$

which can be written as

$$\Phi_{i,j+1} = \Phi_{i,j-1} - 2B\delta z/\mu_0. \tag{2.29}$$

Applying Eq. (2.29) to (2.18) produces the difference approximation to the nodes on Boundary 4 as

$$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(\Phi_{i-1,j}(1-\alpha) + \Phi_{i+1,j}(1+\alpha) + \gamma(2\Phi_{i,j-1} - 2B\delta z/\mu_0)). \tag{2.30}$$

Boundary 5 – boundary condition: $\frac{\partial \Phi}{\partial r} = 0$.

Applying the finite difference approximation gives

$$\frac{\Phi_{i+1,j} - \Phi_{i-1,j}}{2\delta r} = 0,$$

which reduces to the condition

$$\Phi_{i-1,j} = \Phi_{i+1,j}. \tag{2.31}$$

Applying Eq. (2.31) to (2.18) produces the difference approximation to the nodes on Boundary 5 as

$$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i+1,j} + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma)). \tag{2.32}$$

When considering the corner nodes A, B, C, D, E and F (see Fig. 2.4), the values of the nodes E and F can be set to $\Phi = 0$ as they are on Boundary 6. Finite difference formulas to compute the other four corner nodes are derived as below.

Corner node A

Node A satisfies both conditions on Boundary 1 and Boundary 3. Applying both conditions to Eq. (2.18) gives

$$\Phi_{i,j} = \frac{1}{2(2+\gamma)}(4\Phi_{i+1,j} + 2\Phi_{i,j+1}(\gamma)). \tag{2.33}$$

Corner node B

Node B satisfies both conditions on Boundary 1 and Boundary 2. Applying both conditions to Eq. (2.18) gives

$$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i-1,j} + 2\Phi_{i,j+1}(\gamma)). \tag{2.34}$$

Corner node C

Node C satisfies both conditions on Boundary 3 and Boundary 4. Applying both conditions to Eq. (2.18) gives

$$\Phi_{i,j} = \frac{1}{2(2+\gamma)}(4\Phi_{i+1,j} + \gamma(2\Phi_{i,j-1} - 2B\delta z/\mu_0)). \tag{2.35}$$

Corner node D

Node D satisfies both conditions on Boundary 4 and Boundary 5. Applying both conditions to Eq. (2.18) gives

$$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i+1,j} + \gamma(2\Phi_{i,j-1} - 2B\delta z/\mu_0)). \tag{2.36}$$

2.2 Iterative Solution

Having derived all the finite difference formula needed here, we now discuss the iterative procedure used to update each node of the mesh until convergence is reached. There are various iterative solvers. In this study, as mentioned in Chapter 1, we use the successive over relaxation (SOR) method to solve our problem. In this section, we first explain the matrix structure of the system of linear equations obtained from the finite difference method. Then we give a detailed description of the SOR solver and its parallel version.

Table 2.1: Derived finite difference formulas for interior nodes, boundaries, and corner nodes.

Interior nodes	$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(\Phi_{i-1,j}(1-\alpha) + \Phi_{i+1,j}(1+\alpha) + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma))$
Boundary 1	$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(\Phi_{i-1,j}(1-\alpha) + \Phi_{i+1,j}(1+\alpha) + 2\Phi_{i,j+1}(\gamma))$
Boundary 2	$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i-1,j} + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma))$
Boundary 3	$\Phi_{i,j} = \frac{1}{2(2+\gamma)}(4\Phi_{i+1,j} + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma))$
Boundary 4	$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(\Phi_{i-1,j}(1-\alpha) + \Phi_{i+1,j}(1+\alpha) + \gamma(2\Phi_{i,j-1} - 2B\delta z/\mu_0))$
Boundary 5	$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i+1,j} + \Phi_{i,j-1}(\gamma) + \Phi_{i,j+1}(\gamma))$
Boundary 6	$\Phi_{i,j} = 0$
Corner node A	$\Phi_{i,j} = \frac{1}{2(2+\gamma)}(4\Phi_{i+1,j} + 2\Phi_{i,j+1}(\gamma))$
Corner node B	$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i-1,j} + 2\Phi_{i,j+1}(\gamma))$
Corner node C	$\Phi_{i,j} = \frac{1}{2(2+\gamma)}(4\Phi_{i+1,j} + \gamma(2\Phi_{i,j-1} - 2B\delta z/\mu_0))$
Corner node D	$\Phi_{i,j} = \frac{1}{2(1+\gamma)}(2\Phi_{i+1,j} + \gamma(2\Phi_{i,j-1} - 2B\delta z/\mu_0))$
Corner node E	$\Phi_{i,j} = 0$
Corner node F	$\Phi_{i,j} = 0$

2.2.1 The Matrix Form of the Linear System

The system of linear equations obtained from the finite difference method can be converted into a matrix structure as

$$Ax = f, \tag{2.37}$$

where A is an $N \times N$ coefficient matrix, x is an $N \times 1$ column vector of N variables $\Phi_1, \Phi_2, \Phi_3, ..., \Phi_{\mathrm{N}}$, and f is a $N \times 1$ column vector which contains the constant terms that appear in the equations. Corresponding to each row of the matrix A is an equation of the form

$$a_{i,j}\Phi_{i+1,j} + b_{i,j}\Phi_{i-1,j} + c_{i,j}\Phi_{i,j+1} + d_{i,j}\Phi_{i,j-1} + e_{i,j}\Phi_{i,j} = f_{i,j}, \tag{2.38}$$

where a, b, c, d, e are the corresponding weight values and $\Phi_{i,j}$ represents the $(i,j)^{\mathrm{th}}$ node of the mesh. If the node is a boundary node or a corner node, some of these weights equal to zero and the equation

contains fewer terms than the general form. For most of the equations, $f_{i,j}$ values are zero except for the nodes on Boundary 4 and the corner nodes C and D (see Table 2.1). We use this general form of the finite difference formula in later sections to explain theories and to derive new formulas. In the following sections, before jumping into the SOR method, we first discuss the Jacobi and Gauss-Seidel iterative solvers. The SOR method is a variant of the Gauss-Seidel method which has faster convergence than both Jacobi and Gauss-Seidel.

2.2.2 Jacobi Iterative Solver

Jacobi method is a simple and easy to implement method to solve systems of linear equations. In the Jacobi method, we first guess an initial value for all the nodes of the mesh, then we find the new value of each $\Phi_{i,j}$ using the formula

$$\Phi^n_{i,j} = \frac{1}{e_{i,j}}\Big(f_{i,j} - (a_{i,j}\Phi^{n-1}_{i+1,j} + b_{i,j}\Phi^{n-1}_{i-1,j} + c_{i,j}\Phi^{n-1}_{i,j+1} + d_{i,j}\Phi^{n-1}_{i,j-1})\Big). \tag{2.39}$$

This is repeated until a suitable stopping criterion is satisfied. Equation (2.39) is obtained by rearranging the terms of Eq. (2.38), and n is the iteration number. The value of $\Phi_{i,j}$ of the current n^{th} iteration is obtained using the values of four neighbors of the previous $(n-1)^{\text{th}}$ iteration. We can also write the above Jacobi equation in matrix form, starting from $Ax = f$ as described next.

The matrix A can be written as

$$A = L + D + U, \tag{2.40}$$

where D is the diagonal part of A, L is the lower triangle of A with zeros on the diagonal, and U is the upper triangle of A with zeros on the diagonal. Then, the n^{th} iteration of the Jacobi method can be written using matrix structure as

$$Dx^n = -(L+U)x^{n-1} + f.$$

Multiplying both sides by D^{-1} gives

$$x^n = -D^{-1}(L+U)x^{n-1} + D^{-1}f. \tag{2.41}$$

The matrix $-D^{-1}(L+U)$ is called the *Jacobi iteration matrix*. This appears in different formulas in later sections when we discuss the SOR method.

Programming of the Jacobi method is easy as we can use the finite difference formula directly in the algorithm. Two arrays are needed to store $\Phi^n_{i,j}$ and $\Phi^{n-1}_{i,j}$ values. The order in which the nodes are updated is not important as all the $\Phi^n_{i,j}$ are computed before any of them are used [2]. However, because

of the slow convergence of the method, the Jacobi method is not used to solve large systems of equations which appear in real-world applications [15].

2.2.3 Gauss Seidel Method

This method is a modification of the Jacobi method [2]. Here we use the most recently updated values, on the right hand side of Eq. (2.39), as soon as they are available. If we are proceeding along the rows, by incrementing i for fixed j, the Gauss Seidel equation can be written as

$$\Phi^n_{i,j} = \frac{1}{e_{i,j}}\Big(f_{i,j} - (a_{i,j}\Phi^{n-1}_{i+1,j} + b_{i,j}\Phi^{n}_{i-1,j} + c_{i,j}\Phi^{n-1}_{i,j+1} + d_{i,j}\Phi^{n}_{i,j-1})\Big). \tag{2.42}$$

In this method, it is not necessary to store both vectors, Φ^n and Φ^{n-1}. Only one vector of Φ is used and the old value of $\Phi_{i,j}$ is overwritten by the new value of $\Phi_{i,j}$ as soon as it is computed. This method has faster convergence than the Jacobi method, but, still has slow convergence compared to other iterative methods [15].

2.2.4 SOR Method

In the SOR method, the Gauss Seidel process is accelerated by generating a weighted sum of the current and the previous iterates. If we suppose that $\tilde{\Phi}_{i,j}$ is the current Gauss Seidel iterate given by Eq. (2.42), then the SOR equation can be written as

$$\Phi^n_{i,j} = (1-\omega)\Phi^{n-1}_{i,j} + \omega\tilde{\Phi}^n_{i,j}. \tag{2.43}$$

The parameter ω is called the *over-relaxation parameter*. When $\omega = 1$, the SOR method reduces to the Gauss Seidel solver. For the stability of the method, ω should be $0 < \omega < 2$, but, only $\omega > 1$ can accelerate the convergence of the Gauss Seidel method. When $\omega < 1$, it is called under relaxation [15].

Determining Optimum Relaxation Parameter

As mentioned above, when ω is between 1 and 2, it accelerates the convergence of the Gauss Seidel method. Although a range of ω values could accelerate the convergence, there is one optimum value ω_b, which gives the fastest convergence. That is, with ω_b, the method converges with a minimum number of iterations. The rate of the convergence of the SOR method strongly depends on the relaxation factor ω [16]. Smaller differences in the relaxation parameter can cause larger variations of the convergence rate,

and hence, the SOR method is not efficient without a good estimation to ω_b. Hence, obtaining a good estimate for the optimum relaxation factor is an important step.

It can be shown [17] that the optimum value ω_b is given by

$$\omega_b = \frac{2}{1 + \sqrt{1 - \rho_{jacobi}^2}}, \tag{2.44}$$

where ρ_{jacobi} is the spectral radius of the Jacobi iteration matrix. The spectral radius of a matrix is the modulus of the largest eigen value of that matrix. Finding the spectral radius of a matrix is a computationally expensive task. Therefore, some formulas have been derived to approximate the value of ρ_{jacobi}, without going through all the expensive computations. It can be shown that the value of ρ_{jacobi} for a Poisson equation with homogeneous Dirichlet or Neumann boundary conditions over a rectangular domain of size $N \times M$ with the mesh spacing δx and δy [15] is given by

$$\rho_{jacobi} = \frac{\cos\frac{\pi}{J} + (\frac{\delta x}{\delta y})^2 \cos\frac{\pi}{L}}{1 + (\frac{\delta x}{\delta y})^2}, \tag{2.45}$$

where $J = N/\delta x$ and $L = M/\delta y$ are the width and height of the mesh.

Although Eq. (2.45) is very useful when computing rectangular grids, this equation cannot be used when the problem domain is non-rectangular as our problem. In Ref. [16], Kulsrud has experimented with different ways to estimate ρ_{jacobi} for non-rectangular domain problems. One of them was taking J and L of Eq. (2.45) as the width and height of the rectangular region which encloses the non-rectangular domain. For our particular problem, we found that using a rectangular region three times larger (in each direction) than the circumscribing rectangle worked best. This is described in greater detail in Appendix A. The method suggested there is used to determine the optimal omega ω_b when computing meshes throughout the rest of the thesis.

2.2.5 Stopping Criteria

In principle, iterative methods will only converge to the exact solution after an infinite number of iterations. As a result, some criteria are required to stop the iterative procedure when sufficient accuracy is reached. One way of stopping the iterative process is to terminate the procedure when the difference, or error, between the solution and the numerical approximation is small enough (i.e. when the error falls below a user-supplied tolerance). Since we do not know the solution a priori, the actual error is not

measurable. As a result, the error at each node must be approximated using some measurable quantity. In our problem, the relative residual is used to identify the error involved in the approximated solution.

To see this, let x^n be the approximation of the exact solution x of the system of linear equations $Ax = f$ obtained after n iterations. Then

$$r^n = f - Ax^n \tag{2.46}$$

is called the residual vector of the approximation x^n. Let the error vector associated with x^n be e^n. Then

$$e^n = x - x^n.$$

Since $A^{-1}f = x$ and $A^{-1}A = I$, where I is the identity matrix, e^n can be written as

$$e^n = A^{-1}f - A^{-1}Ax^n.$$

Rearranging the terms gives

$$e^n = A^{-1}(f - Ax^n),$$

which reduces to the equation

$$e^n = A^{-1}r^n. \tag{2.47}$$

Eq. (2.47) can be written using the Euclidean norms as

$$\|e^n\| \leq \left\|A^{-1}\right\| \|r^n\|. \tag{2.48}$$

The Euclidean norm of a vector $y = (y_1, y_2, ..., y_N)$ is denoted by $\|y\|$ and is defined as $\|y\| = \sqrt{\sum_{k=1}^{N}(y_k)^2}$.

Dividing both sides of Inequality (2.48) by $\|x\|$, we get the relative error as

$$\frac{\|e^n\|}{\|x\|} \leq \frac{\left\|A^{-1}\right\| \|r^n\|}{\|x\|}.$$

Multiplying the numerator and denominator of the right hand side by $\|A\|$ gives

$$\frac{\|e^n\|}{\|x\|} \leq \frac{\|A\| \left\|A^{-1}\right\| \|r^n\|}{\|A\| \|x\|}.$$

Since $\|f\| \leq \|A\| \|x\|$, the above inequality reduces to

$$\frac{\|e^n\|}{\|x\|} \leq \|A\| \left\|A^{-1}\right\| \frac{\|r^n\|}{\|f\|}, \tag{2.49}$$

where $\frac{\|r^n\|}{\|f\|}$ is the relative residual after n^{th} iteration and gives an indication of the relative error $\frac{\|e^n\|}{\|x\|}$. This relative residual can be measured during the iterative process and can be used as a stopping rule as

$$\frac{\|r^n\|}{\|f\|} < tolerance, \tag{2.50}$$

where *tolerance* is a user-supplied value.

We can write the SOR equation (2.53) using the residual $r^n_{i,j}$ as described next, so that $r^n_{i,j}$ can be calculated as a part of computing the new value of $\Phi_{i,j}$. To derive the formula, let us consider the general form of the finite different formula:

$$a_{i,j}\Phi_{i+1,j} + b_{i,j}\Phi_{i-1,j} + c_{i,j}\Phi_{i,j+1} + d_{i,j}\Phi_{i,j-1} + e_{i,j}\Phi_{i,j} = f_{i,j}. \tag{2.51}$$

The iterative procedure of solving for $\Phi_{i,j}$ in the Gauss Seidel method is

$$\tilde{\Phi}^n_{i,j} = \frac{1}{e_{i,j}}\Big(f_{i,j} - (a_{i,j}\Phi^{n-1}_{i+1,j} + b_{i,j}\Phi^{n}_{i-1,j} + c_{i,j}\Phi^{n-1}_{i,j+1} + d_{i,j}\Phi^{n}_{i,j-1})\Big). \tag{2.52}$$

Then the SOR equation is given by

$$\Phi^n_{i,j} = (1-\omega)\Phi^{n-1}_{i,j} + \omega\tilde{\Phi}^n_{i,j}. \tag{2.53}$$

The residual $r_{i,j}$ of the node $\Phi_{i,j}$ at the n^{th} iteration is

$$r^n_{i,j} = f_{i,j} - \Big(a_{i,j}\Phi^{n-1}_{i+1,j} + b_{i,j}\Phi^{n}_{i-1,j} + c_{i,j}\Phi^{n-1}_{i,j+1} + d_{i,j}\Phi^{n}_{i,j-1} + e_{i,j}\Phi^{n-1}_{i,j}\Big). \tag{2.54}$$

Substituting Eq. (2.54) to Eq. (2.52) gives

$$\tilde{\Phi}^n_{i,j} = \frac{1}{e_{i,j}}\Big(r^n_{i,j} + e_{i,j}\Phi^{n-1}_{i,j}\Big), \tag{2.55}$$

and Eq. (2.53) becomes

$$\Phi^n_{i,j} = (1-\omega)\Phi^{n-1}_{i,j} + \omega(\frac{r^n_{i,j}}{e_{i,j}} + \Phi^{n-1}_{i,j}),$$

which reduces to the equation

$$\Phi^n_{i,j} = \Phi^{n-1}_{i,j} + \omega\frac{r^n_{i,j}}{e_{i,j}}. \tag{2.56}$$

After each iteration, $\|r^n\|$ is calculated and check for the convergence with the stopping rule, $\|r^n\| \,/\, \|f\| < tolerance$. If the number of nodes on the mesh is N, $\|r^n\|$ and $\|f\|$ are computed as

$$\|r^n\| = \sqrt{\sum_{k=1}^{N}(r^n_k)^2} \quad \text{and} \tag{2.57}$$

$$\|f\| = \sqrt{\sum_{k=1}^{N}(f_k)^2}. \tag{2.58}$$

Algorithm 1 shows the pseudo-code of the SOR method.

Algorithm 1 SOR method

INPUT: the maximum number of iterations max_ite, the tolerance TOL, the coefficient vectors a, b, c, d, and e, the over relaxation parameter ω, the norm of the right hand side vector f, the vector $\Phi^{(0)}$, which is the initial guess for Φ.
Set $n = 1$
while n $\leq max\_ite$ **do**
  for i = 1,2,3,...M **do**
    for j = 1,2,3,... N **do**
      $r_{i,j} = f_{i,j} - \left(a_{i,j}\Phi_{i+1,j} + b_{i,j}\Phi_{i-1,j} + c_{i,j}\Phi_{i,j+1} + d_{i,j}\Phi_{i,j-1} + e_{i,j}\Phi_{i,j}\right)$
      $\Phi_{i,j} = \Phi_{i,j} + \omega\frac{r_{i,j}}{e_{i,j}}$
    end for
  end for
  if $\|r^n\| / \|f\| < TOL$ **then**
    STOP
  end if
  Set $n = n + 1$
end while
OUTPUT: $\Phi = (\Phi_1, \Phi_2, ...)$

2.2.6 Red-Black SOR Method

The original SOR method discussed above is a sequential process. The usual way of computing nodes is proceeding along the rows or the columns of the mesh. Since the current step consumes the output of the previous step, computation is serial. In order to run the SOR method in parallel on a GPU, the steps should be independent to run simultaneously. To achieve this parallelism, an alternative ordering is used and the method is called the *Red-Black SOR* method [18]. In this parallel version, the nodes on the mesh are divided into red and black as a checkerboard pattern. If $(i + j)$ is even, the node is red, if $(i + j)$ is odd, the node is black (see Fig. 2.5). In computation, the red nodes depend only on the black nodes and vice versa. As a result, the mesh can be computed in two passes. In the first pass, all the red nodes are updated in parallel; in the second pass, all the black nodes are updated in parallel with the newly calculated red nodes as Eq. (2.59) and (2.60) indicate:

$$\Phi^n_{i,j} = (1-\omega)\Phi^{n-1}_{i,j} + \frac{\omega}{e_{i,j}}\left(f_{i,j} - (a_{i,j}\Phi^{n-1}_{i+1,j} + b_{i,j}\Phi^{n-1}_{i-1,j} + c_{i,j}\Phi^{n-1}_{i,j+1} + d_{i,j}\Phi^{n-1}_{i,j-1})\right) \text{ if } (i+j) \text{ is even, } (2.59)$$

$$\Phi^n_{i,j} = (1-\omega)\Phi^{n-1}_{i,j} + \frac{\omega}{e_{i,j}}\left(f_{i,j} - (a_{i,j}\Phi^{n}_{i+1,j} + b_{i,j}\Phi^{n}_{i-1,j} + c_{i,j}\Phi^{n}_{i,j+1} + d_{i,j}\Phi^{n}_{i,j-1})\right) \text{ if } (i+j) \text{ is odd. } (2.60)$$

Algorithm 2 shows the pseudo-code of the Red-Black SOR method. Both serial and parallel versions of

this red-black SOR method were implemented in this thesis and their performance was compared.

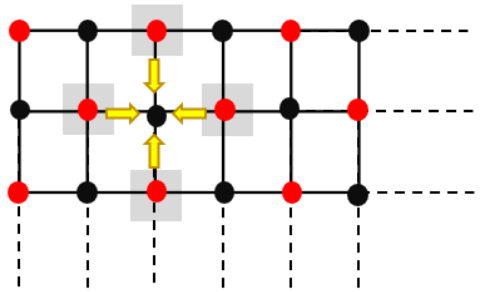


Figure 2.5: In the Red-Black SOR method, all nodes on the mesh are coloured as red and black as a checkerboard pattern. When updating nodes, the red nodes depend only on the black nodes and the black nodes depend only on the red nodes. Therefore, the same colour nodes can be updated in parallel on a GPU, with the entire mesh updated in two passes.

Algorithm 2 Red-Black SOR method

INPUT: the maximum number of iterations max_ite, the tolerance TOL, the coefficient vectors a, b, c, d, and e, the over relaxation parameter ω, the norm of the right hand side vector f, the vector $\Phi^{(0)}$, which is the initial guess for Φ.
Set $n = 1$
while n $\leq max\_ite$ **do**
  for all red nodes **do**
    $r_{i,j} = f_{i,j} - \left(a_{i,j}\Phi_{i+1,j} + b_{i,j}\Phi_{i-1,j} + c_{i,j}\Phi_{i,j+1} + d_{i,j}\Phi_{i,j-1} + e_{i,j}\Phi_{i,j}\right)$
    $\Phi_{i,j} = \Phi_{i,j} + \omega\frac{r_{i,j}}{e_{i,j}}$
  end for
  for all black nodes **do**
    $r_{i,j} = f_{i,j} - \left(a_{i,j}\Phi_{i+1,j} + b_{i,j}\Phi_{i-1,j} + c_{i,j}\Phi_{i,j+1} + d_{i,j}\Phi_{i,j-1} + e_{i,j}\Phi_{i,j}\right)$
    $\Phi_{i,j} = \Phi_{i,j} + \omega\frac{r_{i,j}}{e_{i,j}}$
  end for
  if $\|r^n\| / \|f\| < TOL$ **then**
    STOP
  end if
  Set $n = n + 1$
end while
OUTPUT: $\Phi = (\Phi_1, \Phi_2, ...)$

Chapter 3

Parallel Programming with CUDA

In this thesis, our parallel algorithms are implemented using CUDA computing platform which was introduced by NVIDIA. In order to fully utilize the available compute resources of GPUs, one should understand the hardware structure of CUDA enabled devices and the programming strategies of CUDA. In this chapter, we first provide a basic introduction of the CUDA programming model, the hardware structure, and different optimization techniques in CUDA. Next we discuss single-precision and double precision floating point operations and how they affect the performance and the accuracy of the application. In the last section, we provide details of one of the common and most important data parallel primitive called *parallel reduction*. Reduction operations are those which reduce a collection of values to a single value such as the sum, the minimum or the maximum. Since our parallel algorithms use reduction operations to get the sum out of an array of values, here we describe how to implement parallel reduction operations using CUDA.

3.1 CUDA Programming and Execution Model

CUDA (Compute Unified Device Architecture) [19] is a general-purpose parallel computing platform which was created by NVIDIA in 2006. CUDA C is an extension to the C/C++ programing language with new keywords to enable heterogeneous computing, which is a program that executes code both on the CPU as well as co-processors, such as a GPU. CUDA C is but one of the programming languages that can be used to write heterogeneous applications, and is solely used in this thesis. A CUDA program consists of *host code*, which runs on the CPU, and *device code* which runs on the GPU (Fig. 3.1). The host code is written in C/C++ and the device code, which contains data parallel functions called

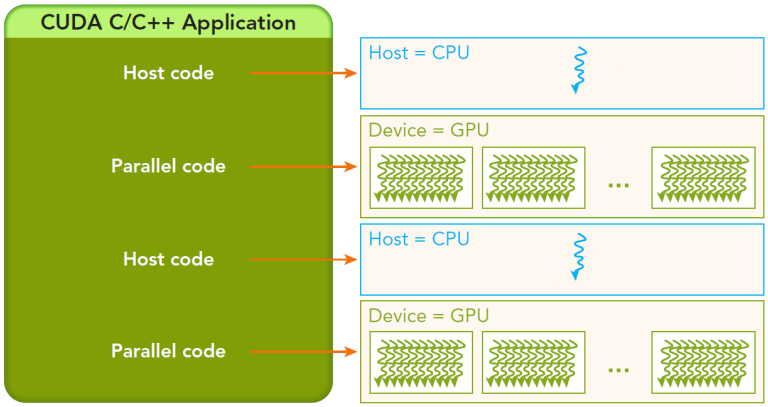


Figure 3.1: The figure from Ref. [20] shows the structure of a CUDA program. A CUDA program can have a mixture of both host and device code. Host code runs on the CPU and device code runs on the GPU. Device code is executed via data parallel functions called *kernels* and is written using CUDA C/C++. The blue and green arrows indicate the execution of instructions, or threads, on the CPU and GPU, respectively.

kernels, is written using CUDA C/C++. The NVIDIA C Compiler (`nvcc`) is used to compile both host and device codes in a CUDA program.

A CUDA program structure consists of the following three main steps:

1. Copy data from CPU memory to GPU memory.
2. Invoke one or more CUDA kernels.
3. Copy back data from GPU memory to CPU memory.

Copying data and invoking CUDA kernels are done from the host using specific CUDA functions. When launching a CUDA kernel from the host, it is required to specify how many threads we need to run simultaneously. These threads are organized into a two-level hierarchy as *block* and *grid*. A block consists of one or more threads and a grid consists of one or more blocks (Fig. 3.2). We can define these thread blocks and grids to be 1D, 2D, or 3D; however, since the memory space has a flat organization, all the multidimensional arrays are linearized into equivalent 1D arrays. For a given grid, the number of threads in a block is available in the `blockDim` variable. Each thread in a block and each block in a grid have a unique index and are available in the variables `threadIdx` and `blockIdx` respectively. For a 2D grid, the global index to access the element at the i^{th} column and the j^{th} row is computed as below:

```
i = blockIdx.x*blockDim.x + threadIdx.x;
j = blockIdx.y*blockDim.y + threadIdx.y;
index = i + j*width;
```

Here, `width` is the width of the grid. All threads in a grid execute the same kernel function. The number of threads which can reside in a block and the number of blocks which can reside in a grid are limited by the available compute resources of the device.

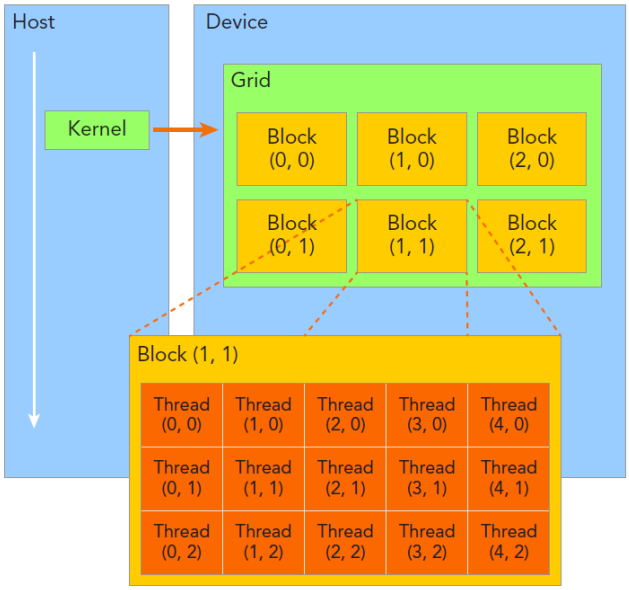


Figure 3.2: CUDA threads are organized into blocks, and blocks are organized into grids. These blocks and grids can be 1D, 2D or 3D, and this figure from Ref. [20] illustrates the 2D case. All threads in a grid execute the same kernel function, and the kernel is launched from the host.

The building block of the GPU architecture is known as a *streaming multiprocessor* (SM). Each SM has hundreds of CUDA cores which support concurrent execution of hundreds of threads. A GPU generally has multiple SMs. As a result, thousands of threads can be executed simultaneously in a single GPU. When a kernel is launched, thread blocks are distributed among available SMs. The number of thread blocks which can concurrently reside in an SM depends on the available capacity of the SM. When allocated thread blocks finish execution, new blocks are launched to fill the unoccupied positions in SMs.

Thread blocks in SMs are further divided into groups of 32 consecutive threads called *warps*. All threads in the same warp execute the same instruction at the same time. The number of clock cycles between an instruction being issued and being completed is defined as *latency* [20]. The latency of one warp can be hidden by switching execution to other warps if a sufficient number of active warps are available in an SM. Therefore, better performance can be achieved when a large number of threads are available for execution on the GPU.

3.1.1 Synchronization

Barrier synchronization is commonly used in parallel applications to coordinate parallel activities. In CUDA, threads within a block are synchronized using the CUDA function `__syncthreads()`. When a kernel function calls `__syncthreads()`, all the threads in a block wait at the calling location until every thread in the block completes its phase of execution and reaches the location. Thus, `__syncthreads()` can be used to ensure that every thread in the block has done with its current step of execution before it moves to the next step of execution. Although this block synchronization function is useful in many applications to coordinate communication among threads within a block, `__syncthreads()` can decrease the performance of the application as the function forces some warps to stay idle until others finish their job.

3.1.2 Warp Divergence

Flow-control constructs such as `if...then...else`, `for` and `while` are commonly used in many programming languages. CPUs include complex hardware to perform these control flow instructions and have a minimum effect on performance. However, GPUs do not include such complex hardware to perform CPU control-flow instructions [20]. In GPUs, all threads in a warp must execute the same instruction at the same time. If the threads in the same warp need to execute different instructions, the warp will execute each branch path serially, one after the other, which leads to what is called *warp divergence* (Fig. 3.3). Divergent branches negatively affect the performance of the application as they result in some threads of the warp remaining idle, while the other threads execute their branch path.

3.2 CUDA Memory Model

Unlike CPU memory, the CUDA memory model consists of many types of memory which programmers can explicitly control. Figure 3.4 illustrates the CUDA memory model. Each memory space has its own capacity, lifetime, and scope. Also, these memory spaces have different latencies, bandwidths (the amount of data transfers per unit time), and functionalities.

Global memory is the largest memory available and resides in the device memory. Global allocations are visible to all threads in all kernel functions and they exist for the lifetime of the application. Since global memory has the highest latency, accessing global memory variables is slow and expensive.

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

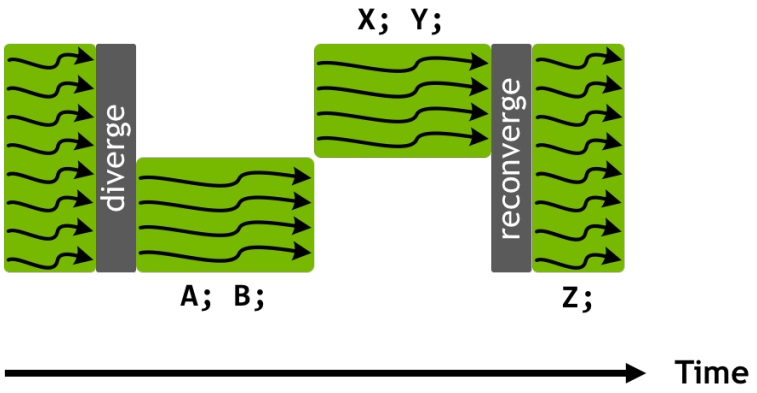


Figure 3.3: Diagram from Ref. [21] shows warp divergence. The threads within the same warp execute two different instructions given by the `if...else` statement. First, the threads which `threadIdx.x < 4` execute the instructions A and B while the other threads are idle. When it comes to the `else` statement, idle threads start executing the instructions X and Y, and threads from the first branch become inactive. After the branch statement, all threads within the warp re-converge and execute the instruction Z at the same time.

Constant memory is a read-only memory. This also resides in the device memory, but is cached in a per SM read-only cache to reduce the latency cost associated with reading from memory. The keyword `__constant__` is used when declaring a constant variable. These variables also can be accessed by all threads in all grids and the lifetime is for the entire execution of the application. Constant memory is used to store data which do not change over the kernel execution (e.g. mathematical constants). Memory access is efficient and fast when all threads read from the same memory location.

Texture memory is another read-only memory that resides in the device memory, which is cached in a per SM read-only cache. Texture memory provides higher bandwidth when the memory access pattern exhibits *2D spatial locality*. This roughly implies that a thread is likely to read from an address "near" the addresses that nearby threads read [8] as shown in Fig. 3.5. Since the four memory locations shown in Fig. 3.5 are not coalesced in one-dimensional memory, they will not cache together when loading from the global memory. Recall, memory is a linear array of bytes, thus the two locations pointed to by Threads 1 and 3 are stored very far in memory from the locations pointed to by Threads 0 and 2. Texture memory is designed to perform well when loading such data which are close together with respect to two-dimensions.

The keyword `__shared__` is used when declaring a shared memory variable. Shared memory is an on-chip memory (i.e. resides in SMs), which, therefore, has higher bandwidth and lower latency than global memory. The lifetime of a shared memory variable is the lifetime of the kernel function. Each thread block has its own shared memory. All threads in a block have access to the shared memory for

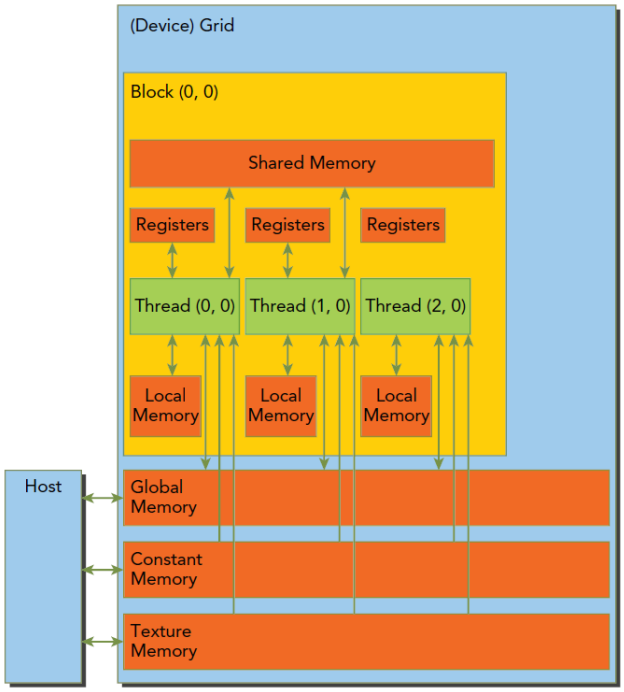


Figure 3.4: The CUDA memory model consists of many types of programmable memories as shown in this diagram from Ref. [20]. Global, constant, and texture memory are located in the device memory. Global memory has the highest latency, but texture and constant memory are cached in a per SM read-only memory for efficient memory loads. Shared memory, which is private to each block, is an on-chip memory, hence has lower latency. Registers, which are private to each thread, are the fastest memory available in CUDA. Variables which do not fit into registers are stored in local memory, which resides in the same location as global memory, and hence, has higher latency.

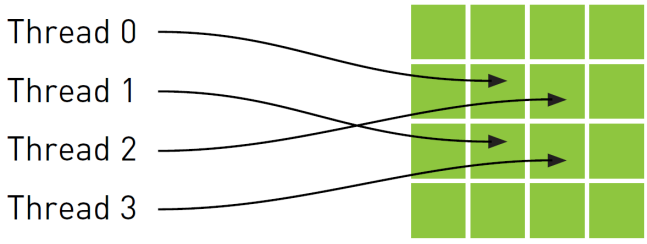


Figure 3.5: This figure from Ref. [8] shows an access pattern which exhibits 2D spatial locality. Here, four consecutive threads access four memory locations, which are spatially close with respect to two-dimension. Although these four locations are close together in 2D, the two locations pointed to by Thread 1 and Thread 3 are located very far in linear memory from the locations pointed to by Thread 0 and Thread 2. Since the locations are not coalesced in linear memory, they will not cache together when loading from global memory and hence, can affect negatively for the performance. Texture cache is optimized for accessing such data which exhibit 2D spatial locality.

that block and they cannot access the shared memory of another block. Threads within a block can cooperate by sharing data through shared memory.

Registers, which are private to each thread, are the fastest memory available in a GPU. All the automatic variables other than arrays that are declared within a kernel function are stored in registers. The lifetime of a register variable is the lifetime of the kernel function. Variables defined within a kernel function that do not fit into register space are stored in local memory. Local memory has higher latency and lower bandwidth as it resides in the same physical location as global memory.

Each memory space is optimized for different purposes. Better performance can be achieved by proper utilization of these memory spaces. Every memory type discussed above is limited by the amount available and varies from device to device.

3.2.1 Memory Access Patterns

In most applications, we need to access data which are stored in global memory. Since global memory has the highest latency and the lowest bandwidth, care must be taken not to waste bandwidth when accessing data from global memory. A proper access pattern maximizes bandwidth utilization and increases the performance of the application. In this section, we examine different memory access patterns and see how they affect the performance of the application.

Align and Coalesced Access

All accesses to global memory are cached through L2 cache. Depending on the hardware architecture of the GPU and the type of the memory access defined, accesses can also pass through L1 cache. In architectures like Kepler, Maxwell, and Pascal, L1 cache is not used by default to cache global memory loads. But we can enable or disable L1 cache at the compile time using compiler options. The L1 cache line is 128 bytes and the L2 cache line is 32 bytes. We say that an access pattern is *aligned* if the first address of a transaction is a multiple of 32 (128 if the L1 cache is used). The memory accesses are *coalesced* if all 32 threads in a warp access contiguous memory locations. Both aligned and coalesced memory accesses are ideal and it is important to arrange accesses to be aligned and coalesced in order to maximize the global memory throughput.

The figures below illustrate different scenarios of global memory accesses when only the L2 cache is enabled. Fig. 3.6 shows the ideal memory access pattern. Suppose that each thread in a warp accesses 4 bytes of data. Here, the addresses requested by all the threads in the warp falls within 4 cache lines of 32 bytes. There are no wasted loads and the bus utilization is 100 percent. Fig. 3.7 shows the worst case scenario, which is when the memory accesses are neither aligned nor coalesced. In this case, the 4-byte addresses requested by the warp is spread out across the global memory. The addresses can fall

across N cache lines where $4 \leq N \leq 32$. Suppose that the requested 32 addresses fall within 32 different cache lines. Then, the bus utilization is 128 bytes requested / 1024 bytes loaded = 12.5 percent. Strided memory accesses are also non coalesced memory accesses which result in poor performance. Fig. 3.8 shows adjacent threads accessing memory with a stride of two. As the stride increases, the effective bandwidth decreases.

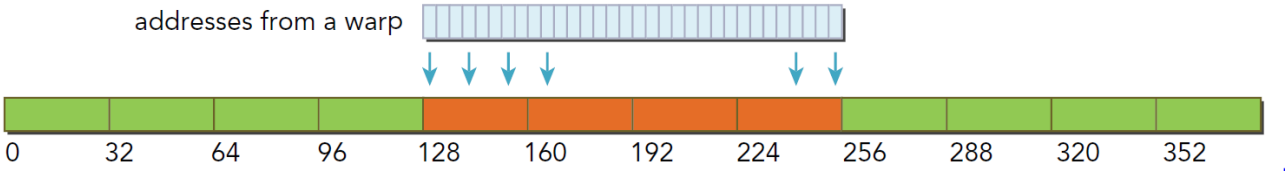


Figure 3.6: This figure from Ref. [20] illustrates the aligned and coalesced memory accesses. Each thread accesses 4 bytes of data and the total memory requested by the warp is 128 bytes. These 128 bytes fall within 4 cache lines of 32 bytes and all threads in the warp access a contiguous chunk of memory. The first address of the transaction is a multiple of 32.

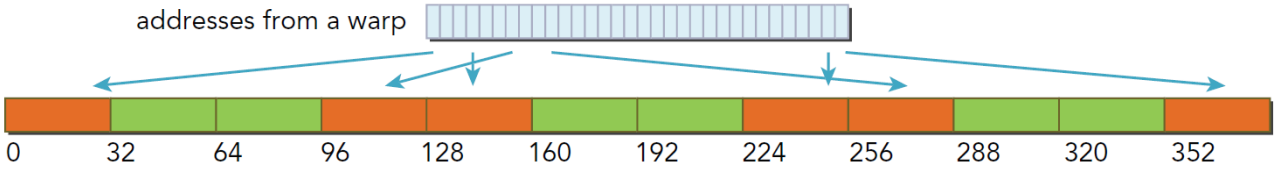


Figure 3.7: This figure from Ref. [20] illustrates a miss aligned and non coalesced memory accesses. The memory addresses requested by a single warp is scattered across the global memory. Requested memory addresses fall within many cache lines. Most of the data loaded is unused, and hence, the bus utilization is very low.

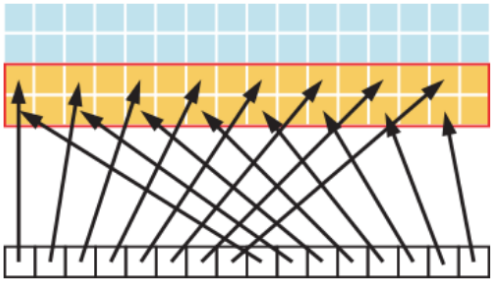


Figure 3.8: Figure from Ref. [22] illustrates adjacent threads accessing global memory with a stride of two. Each thread reads every other memory location and therefore, accesses are not coalesced. Strided accesses also cause memory replays, hence, negatively affecting the performance.

When accessing elements in a 2D array, the rows may not align properly and the first address from the series of addresses requested by a warp would not meet the alignment requirement [23]. Alignment in 2D arrays can be achieved using a special CUDA memory allocation function (or malloc) called `cudaMallocPitch()` :

```
cudaMallocPitch(void **devPtr, size_t *pitchBytes, size_t width, size_t height)
```

Here `devPtr` is the device memory pointer, the variable `pitchBytes` denotes the length in bytes of the padded row, `width` is the width of the array row in bytes, and `height` is the height of the array. Using this function, the size of the each row is padded by an amount necessary for the alignment requirement as shown in Fig. 3.9. The amount of bytes of the pitch is passed into the variable `pitchBytes`. Now, the index of each element of the 2D array is calculated using the pitch as follows:

```
int i = (blockIdx.x * blockDim.x + threadIdx.x);
int j = (blockIdx.y * blockDim.y + threadIdx.y);
int pitch = pitchBytes / sizeof(data type);
index = i + j * pitch;
```

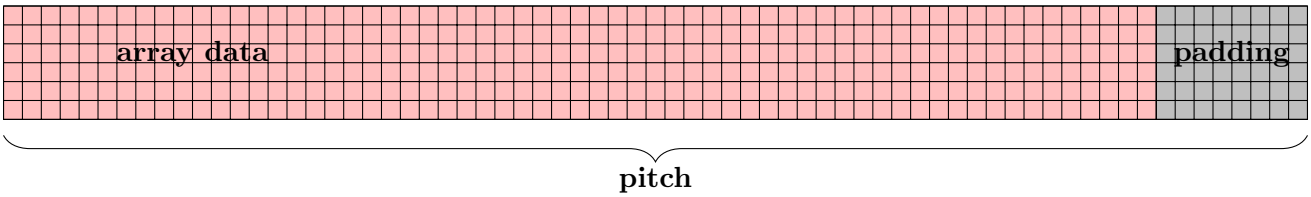


Figure 3.9: The function `cudaMallocPitch()` is used to allocated memory for 2D arrays in order to meet the alignment requirement. When `cudaMallocPitch()` function is called, each row of the array is padded by an amount of bytes necessary for the alignment. The function determines the best pitch for the given array and returns it to the program.

3.3 Single-precision and Double-precision Floating Point Operations

Since computer memory is finite, computers cannot store all the numbers with their complete precision. The precision of a number with infinite decimal places is limited by the number of bits used to represent the number. Nearly all hardware and programming languages use a common floating-point format to represent numerical values that is called the IEEE-754 floating-point standard. The IEEE-745 standard supports two levels of precision in floating-point representation. Single-precision is the 32-bit (4 bytes) representation of numerical values and corresponds to the data type `float` in C/C++. A numerical value represented by a 32-bit float usually yields 8 significant digits. Double-precision is the 64-bit (8 bytes) representation of numerical values and corresponds to the data type `double` in C/C++. A numerical value represented by a 64-bit double usually yields 16 significant digits.

A number represented as a double-precision value is more precise than a single-precision value as double-precision numbers use twice as many bits as single-precision numbers. Even though these double-precision numbers are more precise, this precision comes with both space and performance cost. A variable which stores as a double-precision value takes up to twice the memory space of a single-precision value. When considering performance, double-precision floating point operations are slower in both computation and communication (i.e. data transferring from host to device and device to host). Even though single-precision floating point operations would be quicker than double-precision operations, many scientific computations require more precise solutions and therefore must use the latter to minimize the round-off error. In this thesis, we tested the speed of our algorithms with both single and double precision operations. Final demonstrations were performed with double precision, in order to provide the most accurate solutions possible.

3.4 Parallel Reduction

Parallel reduction is a fundamental building block in parallel programming and appears in many algorithms, such as our Laplacian solver. Reduction methods are used to compute a single value from an array of values. The single value could be the sum, the maximal value, the minimal value, etc. Suppose that we need to find the sum of an array of values. This task can be done easily by sequentially visiting each element of the array and adding the current value to the running total. When doing this sum reduction in parallel, it can be done using an iterative pairwise implementation. First, the array is partitioned into small segments where each segment contains a pair of elements. Then, each thread computes the partial sum of each segment and these partial sums are then stored in-place in the original input array. These newly computed partial sums are the inputs to be summed in the next iteration. This iterative procedure stops when the input to the next iteration reduces to one. The most commonly used methods for parallel reduction in CUDA are shared memory reduction and reduction with shuffle instruction, both of which are described below.

Shared Memory Reduction

Figure 3.10 shows how we can do this sum reduction in parallel using shared memory. The reduction consists of two levels called (i) block level reduction and (ii) grid level reduction. We first explain how to do the reduction in block level. If there are N elements in the array, we use $N/2$ threads to compute the sum. First, we copy the original array from global memory to shared memory. At the first step, the stride is half of the block size. Each thread adds the value of its index to the value half of the block size

away of its index and replaces its value by the computed partial sum. Now, all the partial sums are stored in the first half of the array. At the second step, the stride is divided by two, then, the threads add elements that are one quarter a block size away from each other. At the final step, the total sum of the values of each thread block will be in element 0 of each block. The source code of the shared memory sum reduction is given in Listing 3.1. The `__syncthreads()` statement at line 10 is to ensure that all the partial sums in the previous iteration have been completed before go to the next iteration.

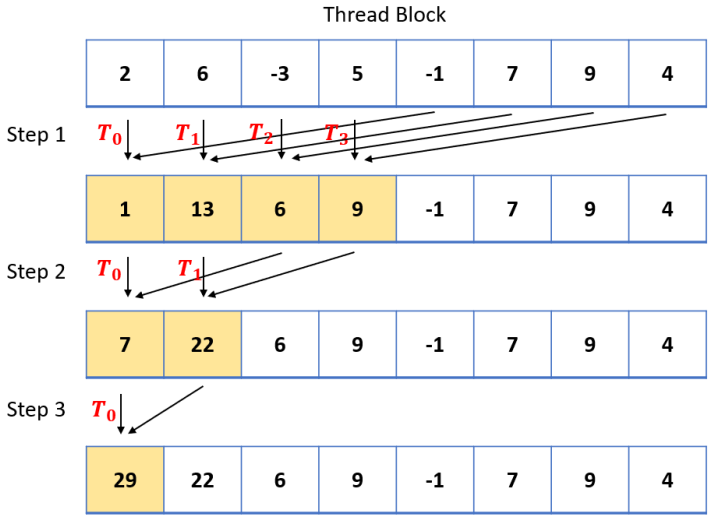


Figure 3.10: This figure illustrates parallel block sum reduction. Since the array contains 8 elements, we need 4 threads to compute the sum. At the first step, each thread reads the value of its index and the value at the half of the block size away (i.e. the stride is 4) from its index, adds them together and replaces its value by the partial sum. Then, the partial sums of the threads T_0, T_1, T_2 and T_4 are 1, 13, 6, and 9 respectively. At the second step, only the threads T_0 and T_1 are active and the stride is 2. Computed partial sums are stored at the index of T_0 and T_1. At the third step, T_0 computes the last partial sum and the total sum of the array is given by the value at the index of T_0.

```
1  __global__ void reduceSumShared (int *input, int *output, unsigned int n) {
2     // set thread ID
3     unsigned int tid = threadIdx.x; //local index within the block
4     unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x; //global index
5     __shared__ int S[];
6     S[tid] = input[idx];// copy data from global memory to shared memory
7
8     for (int stride = blockDim.x/2; stride > 1; stride /= 2) {
9        __syncthreads();
10       if (tid < stride) {
11          S[tid] += S[tid + stride];
12       }
13    }
14 }
```

Listing 3.1: Shared memory sum reduction.

After doing this block level reduction, we need to add these block partial sums together to get the total sum of the array. This final level of reduction is called the grid level reduction and can be done in two ways. One way is to copy all the partial sums from the device to host and do the sequential reduction to get the final result. The other way is to use the atomic add operation to do the grid level reduction on the device, which is discussed later in this section.

Reduction with Shuffle Instruction

In the previous reduction method, shared memory was used to exchange data between threads within the same thread block, without going through global memory. The shuffle instruction, which is available since Kepler GPU architecture, enables a thread to directly read a register from another thread in the same warp without going through shared memory or global memory. Since the shuffle instruction is a register operation, it has lower latency than shared memory and extra memory is not required to perform a data exchange [20]. Hence, it is a fast way to cooperate among threads within a warp.

Here we first introduce some terminologies related to the shuffle instruction. A single thread within a warp is called a *lane*. The *lane index*, which is in the range $[0, 31]$, refers to the thread index within the warp. Since a block can have more than one warp, more than one thread within a block can have the same lane index. In a 2D thread block, the lane index and the warp index for a given thread can be calculated via the following code:

```
laneID = (threadIdx.x + blockDim.x*threadIdx.y) % 32;
warpID = (threadIdx.x + blockDim.x*threadIdx.y) / 32;
```

There are different variants of shuffle instruction, but, in our reduction method, we use one variant called `__shfl_down()`. The `__shfl_down()` instruction is as follows:

```
int __shfl_down(int var, int delta, int width=warpSize)
```

When this instruction is given, each thread shifts its index down the warp by `delta` lanes, grabs the value of `var` at that position and assigns it to its index. Figure 3.11 shows how `x = __shfl_down(x,4,32)` works. When the instruction `x = __shfl_down(x, 4, 32)` is given, the source lane index is calculated by adding four lanes to the caller's lane index. As an example, if the caller's lane index is 0, the source lane index is calculated as 4. Now, the new value of `x` corresponding to lane 0 is the value of `x` held by lane 4. We can use this `__shfl_down` to do the sum reduction in parallel. The reduction with shuffle instruction consists of 3 levels of reduction as (i) warp level reduction, (ii) block level reduction, and (iii) grid level reduction. The source code for the shuffle warp level reduction is given in Listing 3.2. After warp reduction, the value corresponding to lane 0 holds the sum of all elements in its warp.

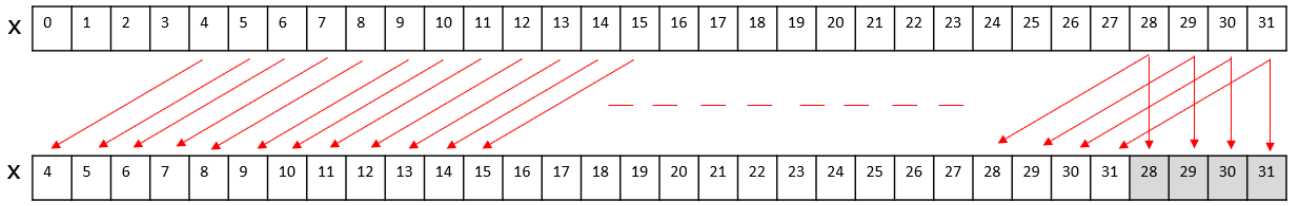


Figure 3.11: When the instruction `x = __shfl_down(x, 4, 32)` is given, each thread shifts its index down the warp by four lanes. Then, the new value of `x` corresponding to thread 0 is the old value of `x` corresponding to thread 4. The new value of `x` corresponding to thread 1 is the old value of `x` corresponding to thread 5 and so on. The last four values of the warp remain unchanged as shifting four lanes down exceeds the boundary of the warp.

```
1 __device__ __inline__ double warpReduceSum(double val) {
2   val += __shfl_down(val, 16);
3   val += __shfl_down(val, 8);
4   val += __shfl_down(val, 4);
5   val += __shfl_down(val, 2);
6   val += __shfl_down(val, 1);
7   return val;
8 }
```

Listing 3.2: Shuffle warp sum reduction

The next step is the block level reduction. Listing 3.3 gives the whole code for the block level reduction. First we compute the lane ID and the warp ID as in line 5 to line 7. Then, the warp reduction is done in line 9. Now, the lane 0 of each warp contains the partial sum of the warp. As the next step, each warp partial sum is written to the shared memory (line 11). Since the maximum number of warps which can reside in a thread block is 32, we have a maximum of 32 partial sums within a block. Therefore, the maximum amount of shared memory needed in this block reduction is that which can hold 32 elements. Then, each block is synchronized using `__syncthreads()` to ensure that all the memory copies are done before going to the next step. In line 16, the data in shared memory are copied back to the variable `var` of the first warp of each block. Finally, the same warp reduction function is used to reduce the first warp of each block and now the block partial sums reside in the thread 0 of each block.

```
1 __inline__ __device__ double blockReduceSum(double val) {
2
3   static __shared__ double shared_mem[32];
4
5   int id = threadIdx.x + blockDim.x*threadIdx.y;
6   int lane = id % warpSize; //index of thread within its warp
7   int warp_id = id / warpSize;
8
9   val = warpReduceSum(val);  // sum reduction in each warp
```

```
10
11    if (lane == 0) shared_mem[warp_id] = val; // write each partial sum to the shared memory
12
13    __syncthreads();
14
15    // write back to registers
16    val = (id < (blockDim.x*blockDim.y) / warpSize) ? shared_mem[lane] : 0;
17
18    if (warp_id == 0) val = warpReduceSum(val);
19    return val;
20 }
```

Listing 3.3: Shuffle Block Reduction

The shuffle instruction offers an efficient way of coordinating between threads within the warp. When considering the performance, the following are the main advantages of using shuffle reduction over shared memory reduction [24]:

- In shared memory reduction, we read from and write to shared memory multiple times. In shuffle reduction, warp level reduction is with register operations, which therefore, has lower latency than shared memory and does not require allocating additional resources. Shared memory is used only once when it comes to the block level reduction. Also, the amount of shared memory required is fixed and does not increase with the size of the array to be reduced.
- In shared memory block reduction, `__syncthreads()` should be used in each step to synchronize thread blocks, which adds some cost to the algorithm. In shuffle warp reduction, the synchronization within a warp is implicit in the instruction and there is no need to explicitly add `__syncthreads()`.

After block level reduction, thread 0 of each block holds the block partial sum. Finally, the reduction is done in grid level in which the block partial sums are added together to get the final total sum. The grid level reduction, as mentioned previously, can be done sequentially in the host or with atomic operations in the device, as described next.

Grid Level Reduction with Atomic Add Operation

Atomic operations can be used in device functions to perform read-modify-write operations safely on 32-bit or 64-bit words. Suppose that we need every thread in a kernel to read from a memory location, add one to that value, and write the modified value back to the same memory location. Since more than one thread is trying to read and write to the same memory location, there is competition

among threads, which is called a *race condition*. Because of the race condition, the final result is not deterministic. Atomic operations are used to allow threads to perform such read-modify-write operations safely, without being interrupted by another thread. The above mentioned read-modify-write operation can be done with `atomicAdd` as `atomicAdd(ptr, 1)`, where `*ptr` is the memory location in which we need to store the modified value.

In sum reduction, to get the total of block partial sums, we can use `AtomicAdd` as follows:

```
AtomicAdd(address, value);
```

When, this instruction is given, it reads the old value from the location pointed to by `address`, computes (old value + `value`), and stores the result back to memory at the same location.

Chapter 4

Development and Implementation of the Algorithm

In this chapter, we first review the literature to see the applications of the Red-Black SOR/Gauss-Seidel method in solving different numerical problems on GPFUs. Then we discuss implementing the serial Red-Black SOR method using C/C++. The output of this serial algorithm is used later to compare with the output of the parallel development to ensure that the parallel algorithm works correctly, and to benchmark the performance of the parallel algorithm. This serial code is then modified using CUDA C/C++, to run in parallel on a GPU. Since the Red-Black SOR method exhibits inherent parallel properties, implementing a baseline, non-optimized GPU parallel algorithm is straightforward. After implementing this baseline parallel algorithm, we then search for the performance inhibitors of this non-optimized development and improve the performance using several optimization strategies, such as improving memory access patterns and utilizing the fast memory caches available in GPUs. The speedup achieved after applying the optimization strategies are described in greater detail in Chapter 5. The development approach: (i) identifying the optimization opportunities, (ii) applying and testing the optimization, and (iii) identifying the speedup achieved is a typical approach in developing a well-performing CUDA application.

4.1 Applications of the Red-Black SOR/Gauss-Seidel Method in Literature

The Red-Black SOR/Gauss-Seidel method has already been studied by the research community and has been applied in solving linear systems of equations that occur in different numerical applications.

These studies have proposed different optimization strategies in order to accelerate the performance of the algorithm on GPUs. Following is a review of related GPU based Red-Black SOR or Gauss-Seidel method implementations.

The Red-Black Gauss Seidel method has been applied to fluid dynamics by Amador and Gomes in Ref. [25], where they solved the Navier-Stokes equations in parallel using CUDA. Liu at el. have applied the parallel Red-Black SOR method in solving computational fluid dynamics problems on a GPU with CUDA in Ref. [26], making use of domain decomposition methods and shared memory to accelerate the performance of the algorithm. The Red-Black SOR method has been applied by Itu in Ref. [27] in solving a steady state heat conduction problem with CUDA. They have applied memory padding and shared memory as the performance acceleration strategies. In Ref. [28], the parabolic equation has been solved by Foster with Red-Black Gauss Seidel method utilizing GPUs with CUDA. Here the algorithm has been optimized by changing the ordering pattern of the elements and this strategy was called "reordering by colour". The same reordering approach has been used by Konstantinidis and Cotronis in Ref. [29] and [30] to solve the Laplace equation in a 2D rectangular domain with CUDA. Here, shared memory and texture memory have been used to reduce the time associated with redundant global memory accesses. In Ref. [31], Cotronis at el. have used the local modified Red-Black SOR method to solve the convection diffusion equation on GPUs with CUDA. They have applied the reordering by colour approach with shared memory and texture memory to optimize the performance.

4.2 Development of the Algorithm

Now having the basic understanding of CUDA programming, memory management, and CUDA optimization strategies, we now describe the development of the algorithm used in this thesis to solve the Laplace equation.

The red-black SOR method described in Chapter 2 is implemented as both serial and parallel. The serial version is implemented with C++ and several parallel versions are implemented with CUDA C/C++. The relative residual is used as the stopping criterion and the *tolerance* is chosen to be 0.5×10^{-6}. The algorithm requires storage of several variables for the iterative procedure. One array is needed to store the initial scalar potential values Φ^0. Considering that the problem size is $n \times n$, we use an $n \times n$ array `U` to store the initial $\Phi_{i,j}$ values. The array `U` is initialized to zero as our initial guess for the $\Phi_{i,j}$ values is zero.

As discussed at the end of Section 2.1.3 in Chapter 2, the general form of the finite difference equation that we use to update the nodes on the mesh is

$$a_{i,j}\Phi_{i+1,j} + b_{i,j}\Phi_{i-1,j} + c_{i,j}\Phi_{i,j+1} + d_{i,j}\Phi_{i,j-1} + e_{i,j}\Phi_{i,j} = f_{i,j}. \tag{4.1}$$

Dividing both sides of Eq. 4.1 by $e_{i,j}$ gives

$$\frac{a_{i,j}}{e_{i,j}}\Phi_{i+1,j} + \frac{b_{i,j}}{e_{i,j}}\Phi_{i-1,j} + \frac{c_{i,j}}{e_{i,j}}\Phi_{i,j+1} + \frac{d_{i,j}}{e_{i,j}}\Phi_{i,j-1} + \Phi_{i,j} = \frac{f_{i,j}}{e_{i,j}}, \tag{4.2}$$

which allows us to avoid the division in each iteration. Now, four $n \times n$ weight arrays W^1, W^2, W^3, W^4 are used to store the weights of four neighbouring nodes of $\Phi_{i,j}$, where $W^1_{i,j} = \frac{a_{i,j}}{e_{i,j}}, W^2_{i,j} = \frac{b_{i,j}}{e_{i,j}}, W^3_{i,j} = \frac{c_{i,j}}{e_{i,j}}$, and $W^4_{i,j} = \frac{d_{i,j}}{e_{i,j}}$. These weights are pre-calculated and stored before beginning the iterative process and are not modified throughout the iterative procedure. The values $\frac{f_{i,j}}{e_{i,j}}$ are not stored in a separate array as most of the $f_{i,j}$ values are zero. Non-zero $f_{i,j}$ values appear only in the equations of the nodes on Boundary 4 and the corner nodes C and D. These non-zero $\frac{f_{i,j}}{e_{i,j}}$ values are added to the computation using a few conditional statements.

Even though our problem domain is L-shaped, our 2D arrays represent all the nodes on the rectangular domain as shown in Fig. 4.1. Each node is accessed by the index $[i + j \times width]$, where i is the column index, j is the row index, and $width$ is the number of nodes along the i axis of the grid. The blue region of Fig. 4.1 is our problem domain and the nodes outside the blue area do not belong to the problem. Hence, the weights $W^1_{i,j}, W^2_{i,j}, W^3_{i,j}, W^4_{i,j}$ corresponding to the nodes outside the domain are set to zero and as a result, the outside nodes remain zero throughout the iterative procedure. For all the grid implementations, $r_1 = z_1$, where r_1 and z_1 are the radius and the half-height of the outer cylinder respectively. In the end, the radius and the half-height of the inner cylinder (r_0 and z_0) are chosen to be 0.25 m and 0.5 m respectively, allowing us to benchmark results against the work of Ref. [6]. These two values remain the same for all the grid implementations.

Having the Eq. (4.2), the residual of each node is computed as

$$R_{i,j} = F_{i,j} - \Big(W^1_{i,j}\Phi_{i+1,j} + W^2_{i,j}\Phi_{i-1,j} + W^3_{i,j}\Phi_{i,j+1} + W^4_{i,j}\Phi_{i,j-1} + \Phi_{i,j}\Big), \tag{4.3}$$

where $R_{i,j} = \frac{r_{i,j}}{e_{i,j}}$ and $F_{i,j} = \frac{f_{i,j}}{e_{i,j}}$. As discussed in Section 2.2.5, the updated value of $\Phi_{i,j}$ is given by

$$\Phi_{i,j} = \Phi_{i,j} + \omega R_{i,j}, \tag{4.4}$$

where ω is the relaxation parameter. When computing the relative residual, we divide the R norm by the F norm, where

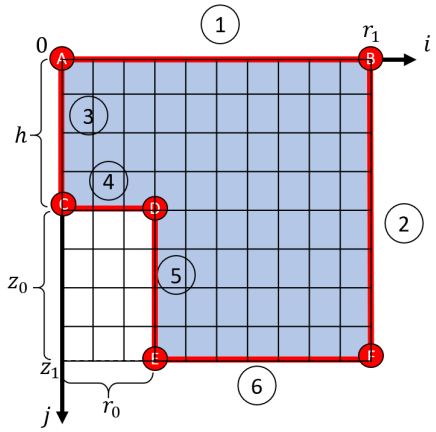


Figure 4.1: This figure illustrates overlaying the problem domain with a rectangular mesh. The blue region indicates the problem domain of our magnetostatic application. The boundaries are shown by the red line segments and the corner nodes are shown by the red circles. The nodes on the white region are outside the domain and do not belong to the problem space. We set the weights $W^1_{i,j}, W^2_{i,j}, W^3_{i,j}, W^4_{i,j}$, corresponding to those nodes to zero, and hence, they remain zero throughout the iterative procedure and do not affect the computation of the nodes in the problem region.

$$\frac{\|R\|}{\|F\|} = \frac{\sqrt{(\frac{r_1}{e_1})^2 + (\frac{r_2}{e_2})^2 + (\frac{r_3}{e_3})^2 + ... + (\frac{r_k}{e_k})^2}}{\sqrt{(\frac{f_1}{e_1})^2 + (\frac{f_2}{e_2})^2 + (\frac{f_3}{e_3})^2 + ... + (\frac{f_k}{e_k})^2}}. \tag{4.5}$$

Here, k is the total number of nodes on the mesh.

4.3 Serial Implementation

The serial red-black SOR method is implemented using two *for loops* to compute the red and black elements on the mesh. If $(i+j)\%2 = 0$, the node is red, if $(i+j)\%2 = 1$, the node is black. The source code of the serial version is given by Listing 4.1.

```
1  int repeat = 1;
2  int i_stride = 1;
3  int j_stride = width;
4  while (repeat <= MAX_ITE) {
5     double res_sum = 0;
6     for (int j = 0; j < height; j++) { // height is the # nodes along j direction
7        for (int i = 0; i < width; i++) { // width is the # nodes along i direction
8           if ((i + j) % 2 == 0) { // compute red nodes
9              double temp = 0;
10             double old_val = 0;
11             double res = 0;
12             index = i + j*width;
```

```
13
14            old_val = U[index];
15            temp += (i == 0) ? 0 : U[index - i_stride] * W1[index];
16            temp += (i == width-1) ? 0 : U[index + i_stride] * W2[index];
17            temp += (j == 0) ? 0 : U[index - j_stride] * W3[index];
18            temp += (j == height - 1) ? 0 : U[index + j_stride] * W4[index];
19
20            if (j == h && i == 0) {
21               res = F1 - (temp + old_val);} // F1 is the f_(i,j)/e_(i,j) of node C
22            else if (j == h && i > 0 && i <= r0) { // F2 is the f_(i,j)/e_(i,j) of boundary 4
                  and node D
23               res = F2 - (temp + old_val);}
24            else
25               res = - (temp + old_val); // f_(i,j) is zero for all other nodes
26
27            U[index] = old_val + omega*res; // replace the old value with new value
28            res_sum += (res*res);}}} //
29
30    for (int z = 0; z < z1_nodes; z++) {
31       for (int r = 0; r < r1_nodes; r++) {
32          if ((r + z) % 2 == 1) { // compute black nodes
33             //same code as for the red nodes goes here
34    }}}
35
36
37    if (sqrt(res_sum) / F_norm < TOL) { // F_norm is the norm of F
38       cout << "number of iterations :" << repeat << endl;
39       break;}
40
41    repeat++;
42 }
```

Listing 4.1: Red-black SOR method - serial version

4.4 Baseline Parallel Implementation

As discussed before, the Red-Black SOR method is inherently parallel and as a result it is straightforward to write a first-go, or baseline implementation that will run on a GPU. This baseline implementation is described below. It is then profiled using CUDA tools to identify the performance bottlenecks and subsequent improvements are made to remove or reduce these bottlenecks. This is a typical approach in developing a well-performing CUDA application.

In this baseline parallel implementation, an $n \times \text{ceil}(n/2)$ grid of threads is used to compute an $n \times n$ mesh. The mesh is computed with two kernel calls, one for the red and black elements respectively. The source code of the kernel function of the baseline parallel implementation is given in Listing 4.2.

The node $\Phi_{i,j}$ is red if $(i+j)$ is even; black if $(i+j)$ is odd. Since the red or black elements do not depend on the values of the nodes of their same color, all the red nodes can be computed in parallel and subsequently all the black nodes can be computed in parallel. In this baseline implementation, all the memory accesses are through global memory and no other fast memories available are used. In each kernel call, each thread reads from every other memory location of five arrays `W1, W2, W3, W4` and `U` (lines 18-21 and line 24). Since the adjacent threads read the global memory with a stride of 2, the memory accesses are not coalesced. In line 25, each thread reads corresponding four neighbouring nodes from `U` to compute the new value of $\Phi_{i,j}$, and replace the old $\Phi_{i,j}$ with the new value (line 36).

```
1  __global__ void solverKernel(double* U, double* __restrict__ W1, double* __restrict__ W2,
       double* __restrict__ W3, double* __restrict__ W4, bool red_black, int width, int height,
       int r0, int h, double* total_res_sum) {
2
3     int i = (blockIdx.x * blockDim.x + threadIdx.x);
4     int j = (blockIdx.y * blockDim.y + threadIdx.y);
5     int i_stride = 1;
6     int j_stride = width;
7
8     i *= 2;
9     if ((j % 2) != red_black)
10       r++;
11
12    int index = i + j*width;
13    double temp = 0;
14    double old_val = 0;
15    double res = 0;
16    double residual_sum = 0;
17
18    double left = W1[index];
19    double right = W2[index];
20    double top = W3[index];
21    double bottom = W4[index];
22
23    if (i < width && j < height) {
24       old_val = U[index];
25       temp = left * U[index - r_stride] + right * U[index + r_stride] + top * U[index -
             z_stride] + bottom * U[index + z_stride];
26
27       if (j == h && i == 0) {
28          res = F1 - (temp + old_val);} // F1 is the f_(i,j)/e_(i,j) of node C
29
30       else if (j == h && i > 0 && i <= r0) {
31          res = F2 - (temp + old_val);} // F2 is the f_(i,j)/e_(i,j) of boundary 4 and node D
32
33       else {
34          res = - (temp + old_val);}
35
```

```
36        U[index] = old_val + omega*res; //replace the old value by new value
37     }
38     residual_sum = res*res;
39     residual_sum = blockReduceSum(residual_sum);
40     if (threadIdx.x + blockDim.x*threadIdx.y == 0) {
41        atomicAdd(total_res_sum, residual_sum);}
42  }
```

Listing 4.2: Red-black SOR method - Baseline parallel version

Here, the boolean variable `red_black` is 0 when computing the red nodes and 1 otherwise. The variables `omega, F1` and `F2` are stored as constant variables. The square of the residual of each node is stored in a register variable called `residual_sum` (line 38). To add those square residual values together, the parallel reduction is used. In our algorithm, we use the parallel reduction with shuffle instruction to get the sum of square residual values. The function `blockReduceSum()` (line 39) is the block reduction function given in Listing 3.3 in Section 3.4. As in lines 40 and 41 in Listing 4.2, we use atomic add operation to sum up all the block partial sums of `residual_sum` and the total value is stored at the memory location pointed to by `total_res_sum`.

In the host, we check the stopping criterion and if the relative residual is less than the given tolerance, stop the iterative process. The host code of our baseline Red-Black SOR algorithm is given in Listing 4.3.

```
1  //in the host
2  int repeat = 1;
3  while (repeat <= MAX_ITE){
4     solverKernel <<<dimGrid, dimBlock >>> (...); // launch kernel to update red elements
5     solverKernel <<<dimGrid, dimBlock >>> (...); // launch kernel to update black elements
6     //copy total_res_sum from device to host
7     cudaMemcpy(total_res_sum_h, total_res_sum, sizeof(double), cudaMemcpyDeviceToHost);
8     //calculate the relative residual and check for convergence
9     if (sqrt(*total_res_sum_h) / F_norm < 0.5*pow(10, -6)) {
10       cout << "Number of iteration :" << repeat << endl;
11       break;
12    }
13    repeat++;
14 }
```

Listing 4.3: Host code of Red-Black SOR algorithm.

4.5 Improving Performance of the Baseline Parallel Implementation

The next step is to identify the optimization opportunities for our baseline parallel implementation. Here we use `nvprof`, the primary profiling tool for CUDA applications to recognize the performance inhibitors of the baseline `solverKernel`. The algorithm is executed and profiled on GTX 960M to evaluate the efficiency of the kernel. Some of the more important metrics related to our baseline `solverKernel` are the following:

```
branch efficiency : 99.9%
global load efficiency : 48.29%
global store efficiency : 47.77%
dram read throughput : 49.768 GB/s
dram write throughput : 10.977 GB/s
dram utilization : High
```

The metric `branch efficiency` is the percentage of the ratio of non divergent branches to total branches. Since the branch efficiency is almost 100%, there is no issue with branch divergence in the baseline `solverKernel`. The metric `global load efficiency` is defined as the ratio of the requested global memory load throughput to the required global memory load throughput. The metric `global store efficiency` is the same as `global load efficiency`, but for global memory stores. Lower load/store memory efficiency indicates that there are more memory transactions per request. If memory accesses are properly coalesced and aligned, there are no memory access replays, and load/store efficiency should be 100%. In the baseline `solverKernel`, both metrics are far less than 100%, meaning that the kernel has many non coalesced memory accesses. The total device memory bandwidth is the sum of the forth and fifth metrics and is roughly 60 GB/s here. Since the peak memory bandwidth of the device is 80 GB/s, the device memory utilization is 75%, resulting in a `dram utilization` that is indicated to be `High`. Overall, then, the poor global load/store efficiency, which indicates high bandwidth waste, requires the most attention. As a result, the most important first step in optimizing `solverKernel` is to improve the memory access pattern to optimize the device memory bandwidth utilization.

An optimization strategy which improves the memory access pattern of the red-black kernel is referred to as *element reordering by colour* in Ref. [29, 30, 31]. In this approach, the grid elements are split into two grids according to their colour (see Fig. 4.2). Now, adjacent threads can access consecutive memory locations when computing red or black nodes. This reordering method eliminates the strided access

pattern that occurred in the baseline implementation. As shown in Fig. 4.2, the computation requires two grids as `U_red` and `U_black` to store the red and black elements separately. In the first pass, the red nodes are updated by reading the neighboring nodes from the black grid. In the second pass, the black nodes are updated by reading the neighboring nodes from the previously updated red grid. In either pass, one grid is updated by reading from the other grid.

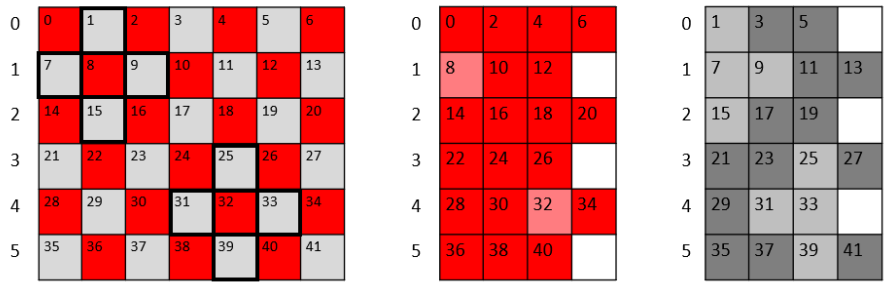


Figure 4.2: In the *element reordering by colour* approach, the nodes are separated into two grids according to their colour. The red elements are updated by reading neighbouring nodes from the black grid and vice versa. The positions of the left and right neighbouring nodes depend on the row number `j`. Suppose that we are computing the red nodes. If `j` is odd, the left and the right neighbours of $\Phi^{\text{red}}_{i,j}$ are $\Phi^{\text{black}}_{i,j}$ and $\Phi^{\text{black}}_{i+1,j}$ respectively (see element 8 and its left and right neighbours 7 and 9). If `j` is even, the left and the right neighbours of $\Phi^{\text{red}}_{i,j}$ are $\Phi^{\text{black}}_{i-1,j}$ and $\Phi^{\text{black}}_{i,j}$ respectively (see element 32 and its left and right neighbours 31 and 33). When computing the black nodes, it is the other way around.

Based on this reordering, accessing neighbouring nodes is not as simple as before. Suppose that we update the red node $\Phi^{\text{red}}_{i,j}$. The top and the bottom neighbours of $\Phi^{\text{red}}_{i,j}$ are $\Phi^{\text{black}}_{i,j-1}$ and $\Phi^{\text{black}}_{i,j+1}$. The positions of the left and right neighbours depend on the value of the row number j (see elements 8 and 32 in Fig. 4.2). If j is odd, the left neighbour of $\Phi^{\text{red}}_{i,j}$ is $\Phi^{\text{black}}_{i,j}$ and the right neighbour is $\Phi^{\text{black}}_{i+1,j}$. If j is even, the left neighbour is $\Phi^{\text{black}}_{i-1,j}$ and the right neighbour is $\Phi^{\text{black}}_{i,j}$. When updating the black elements, indexing the top and the bottom neighbours is the same, but it is the opposite when accessing the left and right neighbours. Since the positions of neighbouring nodes depend on the colour of the node and the row number, a new variable is required to keep track on the colour of the node and whether the row number is even or odd:

```
bool line_no = ((j + red_black) % 2 == 0) ? 1 : 0;
```

The boolean variable `red_black` is 0 if we update red elements and 1 otherwise. The variable `line_no` depends on both the row number `j` and the variable `red_black`. Since we read from four weight arrays `W1, W2, W3` and `W4,` we split these weight arrays also as red and black to ensure coalescing. In order to achieve the alignment requirement as discussed in Section 3.2.1, these 2D arrays are allocated to the

memory using `cudaMallocPitch()` and the new width of the arrays is stored into the variable `pitch`. Suppose that we are updating a red element. Then, the four neighbours can be accessed using `line_no` and `pitch` as below:

```
int i = (blockIdx.x * blockDim.x + threadIdx.x);
int j = (blockIdx.y * blockDim.y + threadIdx.y);

int index = i + j*pitch;

double temp = W1_red[index] * U_black[index - line_no] + W2_red[index] * U_black[index + 1
    - line_no] + W3_red[index] * U_black[index - pitch] + W4_red[index] * U_black[index +
    pitch];
```

Here, `W1_red, W2_red, W3_red, W4_red` are the weight arrays corresponding to red elements.

When computing the elements of one particular colour, reading from and writing to that particular array is contiguous, but reading neighbouring nodes from the other array is not coalesced. Although the coalesced access pattern is not possible with accessing neighbouring nodes, we can see that the accessing pattern exhibits data sharing properties. Most of these neighbour elements are used in the computations of four elements, meaning that the same node value is loaded four times from global memory. Therefore, it is advantageous to use the fast memories available in the GPU to reduce the time associated with redundant global memory accesses. Two strategies taken to achieve this are described in the following sections.

4.5.1 Shared Memory Implementation with Reordering

Here, the *reordering by colour* approach is implemented with shared memory. The neighbouring nodes are cached in shared memory as accessing shared memory is faster than accessing global memory. There are four neighbours per node and each thread needs to access these four values from the opposite colour grid. The four neighbours are $\Phi_{i,j}$, $\Phi_{i,j-1}$, $\Phi_{i,j+1}$ and $\Phi_{i+1,j}$ or $\Phi_{i-1,j}$ depending on the row number. For some threads, some of these values reside outside the boundary of the block and belong to some other thread blocks. Since the threads of a block cannot see the elements of the shared memory of another thread block, these values should be loaded into the respective shared memories for threads to access them. The elements that are involved in multiple shared memory tiles and loaded by multiple blocks are referred to as *halo elements* [10].

For an $M \times N$ thread block, an $(M + 2) \times (N + 2)$ shared memory tile is needed to hold the block interior nodes as well as the halo elements as shown in Fig. 4.3. There are various ways that we can copy data from global memory to shared memory. In this implementation, the interior nodes and the

halo elements are copied to shared memory as given in Listing 4.4. Each interior node is copied by each thread of its index (line 11). The threads which `threadIdx.x` is zero, load the left and right halos (line 15 to 17). The threads which `threadIdx.y` is zero, load the top and bottom halos (line 21 to 23). The corner halo elements are not loaded as they do not involve in any computation. After copying data, all the thread blocks are synchronized before moving to the next step.

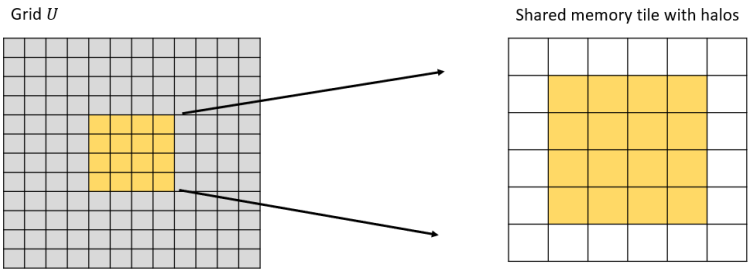


Figure 4.3: If the block size is $M \times N$, the required shared memory tile is $(M + 2) \times (N + 2)$. The yellow elements are the block interior nodes and the white elements around the block are called halo elements. These halo elements loaded by multiple blocks and therefore, are involved in multiple shared memory tiles.

```
1  int i = (blockIdx.x * blockDim.x + threadIdx.x);
2  int j = (blockIdx.y * blockDim.y + threadIdx.y);
3  int index = i + j*pitch; //global index
4
5  __shared__ double S[BLOCK_HEIGHT + 2][BLOCK_WIDTH + 2];
6  //index to access shared memory tile
7  int sr = threadIdx.x + 1;
8  int sz = threadIdx.y + 1;
9
10 //read block interior elements from U to S
11 S[sz][sr] = U[index];
12
13 //read halo elements
14 //read left and right halos
15 if (threadIdx.x < 1) {
16    S[sz][threadIdx.x] = U[index - 1];
17    S[sz][threadIdx.x + 1 + BLOCK_WIDTH] = U[index + BLOCK_WIDTH];
18 }
19
20 //read top and bottom halos
21 if (threadIdx.y < 1) {
22    S[threadIdx.y][sr] = U[index - pitch];
23    S[threadIdx.y + 1 + BLOCK_HEIGHT][sr] = U[index + BLOCK_HEIGHT*pitch];
24 }
25
```

```
26 __syncthreads();
```

Listing 4.4: Copying data from global memory to shared memory

4.5.2 Texture Memory Implementation with Reordering

Alternatively, the *reordering* approach is also implemented using texture memory. Texture memory is a device memory which is efficient when the memory accesses are spatially localized. Memory accesses exhibit 2D spatial locality if the threads in the same warp read memory addresses that are close together (Section 3.2). Since our algorithm processes four neighbouring nodes, they are spatially localized, and so the performance can be improved by loading neighbouring nodes through texture cache. Double precision textures are not supported in the current version of CUDA. Therefore, `int2` textures are used and converted them into double precision values using the user-defined function `texfetchDouble()` given in Listing 4.5. In order to use the texture memory, first we need to declare two 2D texture references for our input arrays `U_red` and `U_black` as follows:

```
texture<int2, 2> texRed;
texture<int2, 2> texBlack;
```

These texture references are then bound to the memory buffers using `cudaBindTexture2D()` as follows:

```
cudaChannelFormatDesc desc = cudaCreateChannelDesc(32, 32, 0, 0,
    cudaChannelFormatKindSigned);
cudaBindTexture2D(NULL, texRed, U_red, desc, (width + 1)/2, height, pitchBytes);
cudaBindTexture2D(NULL, texBlack, U_black, desc, width_half, height, pitchBytes);
```

where `cudaChannelFormatDesc` describes the format of the texture elements. In our case, it is `int2` and has two 32-bit components. The arguments of the function `cudaBindTexture2D()` are as below:

Offset in bytes - `NULL`

Texture reference to bind - `texRed`

2D memory area on device - `U_red`

Channel format - `desc`

Width of the array - `(width + 1)/2`

Height of the array - `height`

Pitch in bytes - `pitchBytes`

Here, the variable `width` is the number of nodes along the width of the mesh before separate it into red and black.

In the kernel function, the neighbouring nodes are loaded through texture cache using `texRed` and `texBlack`. We declare another texture reference called `texRead`. Then, `texRead` is `texRed` if we compute black nodes and `texBlack` if we compute red nodes. Textures are fetched using the function `texfetchDouble()` as given in Listing 4.6.

```
__inline__ __device__ double texfetchDouble(texture<int2, 2> t, int i, int j) {
   int2 val = tex2D(t, i, j);
   return __hiloint2double(val.y, val.x);
}
```

Listing 4.5: texfetchDouble() function.

```
...

int i = (blockIdx.x * blockDim.x + threadIdx.x);
int j = (blockIdx.y * blockDim.y + threadIdx.y);

int index = i + j*pitch;
bool line_no = ((j + red_black) % 2 == 0) ? 1 : 0;

//reading four weight values
dataType left = d_L[index];
dataType right = d_R[index];
dataType top = d_T[index];
dataType bottom = d_B[index];
//declare another texture reference and assign texRed or texBlack according to the colour
    of the nodes
texture<int2, 2> texRead = (red_black == 0) ? texBlack : texRed;

dataType temp = 0;
dataType old_val = 0;
dataType res = 0;
dataType residual_sum = 0;

if (i < ((width + 1) / 2) - (1 - line_no) && j < height) {

   old_val = write_U[index];

   temp = left * texfetchDouble(texRead, i - line_no, j) +
           right * texfetchDouble(texRead, i + 1 - line_no, j) +
           top * texfetchDouble(texRead, i, j - 1) +
           bottom * texfetchDouble(texRead, i, j + 1);

   ...

   write_U[index] = old_val + res*omega; //replace the old value by the new value

   ...
}
```

```
37 ...
```

Listing 4.6: Reading neighbouring nodes through texture memory using texfetchDouble() function within the solverKernel.

4.5.3 Computing More Than One Element Per Thread

In some CUDA applications, the performance can be further improved by increasing the work load of each thread. Therefore, both shared memory and texture memory versions are written so that each thread computes more than one node of the mesh. Performance of the algorithms are compared by varying the number of node computations per thread. All the performance comparison results are shown in the next chapter.

Chapter 5

Performance Comparisons and Analysis

In this chapter, we analyze the performance of the implemented serial and parallel algorithms. We compare different configurations of the parallel solvers on multiple problem sizes to identify the best set of configurations which give faster performance. Then, we obtain the throughput for each solver. The throughput is computed as the number of stencil computations per second. Finally, we compare the running time of the serial and parallel versions to identify the speedup achieved through parallelism. The performance comparison is carried out on two different systems. System descriptions are listed in Table 5.1.

5.1 Comparison of Different Parallel Solvers

Three main parallel solvers were implemented using different techniques. Kernel 1 is the baseline, which is the non-optimized version of the parallel algorithm. This uses only global memory and the memory access pattern is not coalesced. Kernel 2 uses *reordering by colour* technique with shared memory. Kernel 3 was also implemented with *reordering by colour* approach, but uses texture memory instead of shared memory. Kernel 2 and Kernel 3 were also tested by increasing the workload per thread. In our parallel solver, the workload can be increased by computing more than one stencil per thread. The number of stencil computations per thread is referred to as *thread granularity* in Ref. [29, 30, 31]. Kernel 2 and Kernel 3 were tested for different thread granularities to identify the optimum granularity which gives the fastest performance. In addition, all the kernels were tested with different thread block configurations. The parameter values which gives the fastest performance for each kernel were chosen for the next experiments. Figure 5.1 shows the computational time for each kernel implementation on both GPUs. For the kernel comparison, the problem size was chosen as 4001×4001 and the time

Table 5.1: System descriptions. We used two systems for testing performance, System 1 is an ASUS laptop with NVIDIA Geforce GTX 960M GPU and System 2 is a UW GPU server with a Tesla P100 card. Even though both of these GPUs can be used for general purpose computing, these GPUs are basically designed by NVIDIA for different purposes. NVIDIA Geforce series is basically designed for optimizing gaming performance and NVIDIA Tesla series is developed for accelerating compute intensive scientific problems such as simulations, data science problems and deep learning algorithms.

	System 1	**System 2**
CPU	Intel core i5 @ 2.30 GHz	Intel Xeon(R) CPU @ 2.60GHz
RAM	16 GB	125 GB
GPU	NVIDIA Geforce GTX 960M Compute capability 5.0 640 CUDA cores Peak memory bandwidth 80 GB/s	NVIDIA Tesla P100 GPU Compute capability 6.0 3584 CUDA cores Peak memory bandwidth 549 GB/s
GPU device memory	2 GB	12 GB
PCIe bus	v3.0 x16 (8.0 GT/s)	v3.0 x16 (8.0 GT/s)
C++ compiler	Microsoft Visual Studio C++ 2015 compiler	GCC C++ compiler
CUDA compiler	`nvcc`	`nvcc`
Operating System	Windows 10	Linux

required to solve the input problem for 1000 iterations was measured. All the time measurements were for double precision computations.

For the GTX 960M, the texture memory implementation (K3) with thread granularity 8 gave the fastest performance. The shortest compute time (only for the kernel function) is 12946 ms. For the Tesla P100, the best performance was given by the shared memory implementation (K2) with thread granularity 1, which had a compute time of 2161 ms. According to Fig. 5.1, the shared memory implementation (K2) with granularity 2 can be chosen as the parallel version which gives the better performance on both GPUs. All the parallel versions are around 5 or 6 times faster when running on the Tesla P100 than on the GTX 960M. The reason for this performance difference is that the Tesla P100 has around 5 times as many CUDA cores as in the GTX 960M. The best thread block configuration for Kernel 1 on the GTX 960M was obtained as 256×1 and on the Tesla P100, it was 64×16. The optimum configurations for Kernel 2 and Kernel 3 on each device are listed in Table 5.2.

The total device memory required by our parallel implementations for double-precision computations can be computed using the following formula:

$$\text{Total device memory in bytes} = \frac{r_1}{\text{mesh spacing}} \times \frac{z_1}{\text{mesh spacing}} \times 10(\text{ten such arrays}) \times 8 \text{ bytes},$$

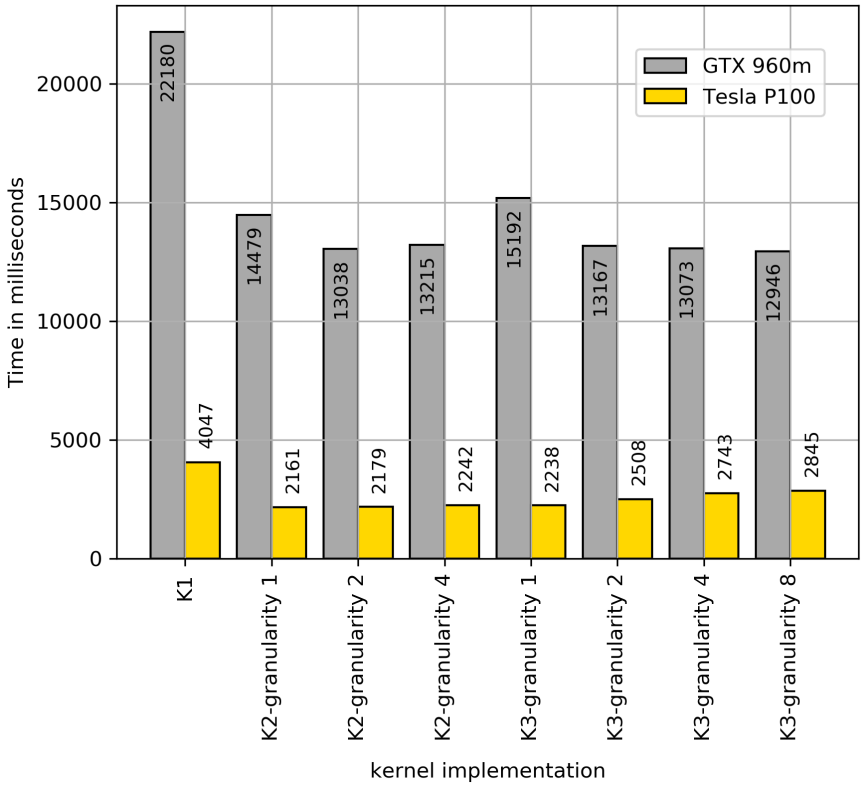


Figure 5.1: Performance of the three different parallel solvers for different thread granularities. K1: baseline implementation, K2: shared memory implementation, K3: texture memory implementation. Time measurements for all the kernel implementations were obtained after running the algorithm for 1000 iterations with the problem size 4001×4001. For the comparison, only the kernel execution times were measured. The minimum time achieved using the Tesla P100 is 2161 ms, which was for K2 with granularity 1. For the GTX 960M, the minimum time was obtained for K3 with granularity 8 and the time was 12946 ms. All the kernel computations were double precision computations.

where r_1 and z_1 are the radius and the half-height of the outer cylinder. We can compare the total memory required with the available device memory to check whether the problem fits within the memory limit of the device.

5.2 Data Transferring and Element Reordering Overhead

In our parallel solvers, host to device memory transfers for large arrays at the beginning of the computation are not required. Both input grids and weight arrays can be initialized on the device. Only four constant terms are transferred from host to device at the beginning using `cudaMemcpyToSymbol()` and the time overhead is negligible. At the end of the computation, the solution grid is needs to be transferred from device to host, but this data transfer overhead is also insignificant compared to the

Table 5.2: Configurations with best performance for each kernel implementation.

Device	Kernel	Granularity	Block size
Tesla P100	K2	1	64×2
	K3	1	64×4
GTX 960M	K2	2	256×1
	K3	8	128×1

computational time. Another data processing step includes separating and reordering data according to their colour. Data separation into red and black is needs to be done on four weights arrays at the beginning of the computation. At the end of the computation, two red and black solution arrays should be combined into one final solution. It was found that the time for this pre and post data ordering was also very small compared to the kernel compute time.

Table 5.3 shows the time taken by different GPU activities of the parallel algorithm for multiple problem sizes. The time measurements were obtained using `nvprof` with the option `--print-gpu-summary`. Here we used both systems for the analysis and all the time measurements were obtained using the K3 solver (K3 with granularity 8 on System 1 and K3 with granularity 1 on System 2). The algorithm consists of four kernel functions: `allocWeightsKernel()` is used to fill the arrays `W1`, `W2`, `W3`, and `W4`; `sepatareWeightsKernel()` is used to separate weight arrays into red and black; `solverKernel()` updates the value of each node of the mesh; and `reorderKernel()` combines the red and black solution arrays into a one. According to the data listed in Table 5.3, we can conclude that more than 98% of the time is taken by the computation (i.e. by the `solverKernel()`) and data transferring and reordering overhead is negligible.

5.3 Throughput and Speedup Comparison

Throughput was measured as the number of million stencil computations per second. Figure 5.2 and 5.3 illustrate the throughput for GPU and CPU implementations. Here, the kernel functions used for the experiment are with their optimum parameters corresponding to each GPU. Both single-precision and double-precision computations were considered. Higher throughput values were obtained for larger data sets on GPUs as when computing larger problem sizes, streaming multiprocessors are well occupied and GPUs can efficiently reduce latencies of warps. In both GPUs, single-precision computations were

Table 5.3: The summary of different GPU activities of the parallel solver K3 on System 1 and System 2. Here "Iterations" is the number of iterations required for convergence when TOLERANCE is given as 0.5×10^{-6}. The algorithm consists of four kernel functions: "Solver kernel" updates the nodes of the mesh; "Alloc weights kernel" is used to fill weights arrays with corresponding weights; "Separate weights kernel" separates the weights arrays into red and black; and "Reorder elements kernel" combines separated red and black elements into one array. The column "Memcpy DtoH" shows the time required to copy data from Device to Host and "Memcpy HtoD" shows the time required to copy data from Host to Device. The column "CUDA memset" shows the time required to set all CUDA arrays to zero at the beginning.

(a) Time for different GPU activities on System 1 (with the GTX 960M)

Mesh size	Iterations	Time						
		Solver Kernel (s)	Memcpy DtoH (ms)	CUDA memset (ms)	Alloc weights kernel (ms)	Separate weights kernel (ms)	Reorder elements kernel (ms)	Memcpy HtoD (μs)
1001×1001	8279	7.0	8.5	7.7	1.3	0.9	0.23	2.82
2001×2001	15832	51.4	20.9	19.4	5.4	3.7	0.92	2.72
3001×3001	23265	168.3	37.3	33.3	12.1	8.3	2.09	2.75
4001×4001	30888	396.5	58.8	45.3	21.5	14.7	3.72	2.75

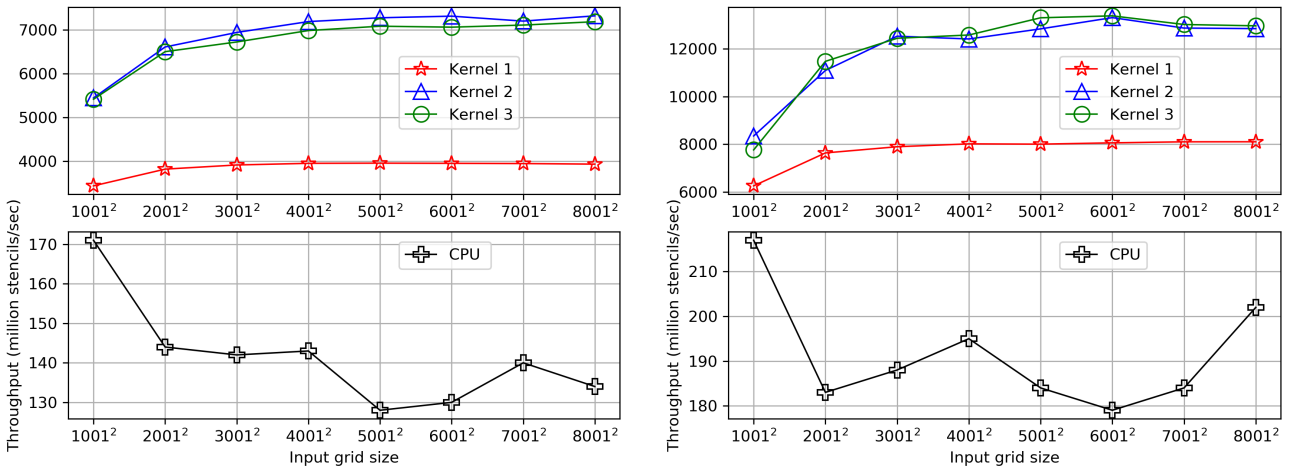
(b) Time for different GPU activities on System 2 (with the Tesla P100)

Mesh size	Iterations	Time						
		Solver Kernel (s)	Memcpy DtoH (ms)	CUDA memset (ms)	Alloc weights kernel (ms)	Separate weights kernel (ms)	Reorder elements kernel (ms)	Memcpy HtoD (μs)
1001×1001	8279	1.2	13.4	8.6	0.09	0.17	0.06	4.26
2001×2001	15832	8.7	34.9	17.1	0.38	0.64	0.22	4.45
3001×3001	23265	28.4	60.6	24.3	0.85	1.43	0.49	4.74
4001×4001	30888	66.8	95.6	32.0	1.44	2.53	0.87	4.74

faster than double-precision computations (see Section 3.3). In our case, using double-precision values reduces the throughput nearly by half. Since double-precision variables can represent values with wider range and they are twice as long as single-precision values, double-precision floating point operations are slower in both computation and communication. When considering CPU algorithms, the intel core i5 CPU with visual C++ compiler gives almost the same performance for both double-precision and single-precision calculations, while the intel Xeon(R) CPU with GCC C++ compiler performs better on single-precision calculations.

As the final experiment, the speedup of the GPU implementations were compared over the CPU sequential implementations. The speedup factors obtained for eight different problem sizes on the Tesla P100 and four different problem sizes on the GTX 960M are shown in Fig. 5.4 and Fig. 5.5 respectively. Here, both the baseline non-optimized parallel version and the optimized parallel algorithm which gives the fastest performance (K2 with granularity 1 on the Tesla P100 and K3 with granularity 8 on the GTX 960M) were considered for the comparison. The speedup was calculated by measuring the total run time required to finish 1000 iterations. The total computation time includes the data pre/post ordering and data transferring overhead. The speedup factor was computed as below:

$$speedup\ factor = serial\ algorithm\ time/parallel\ algorithm\ time.$$

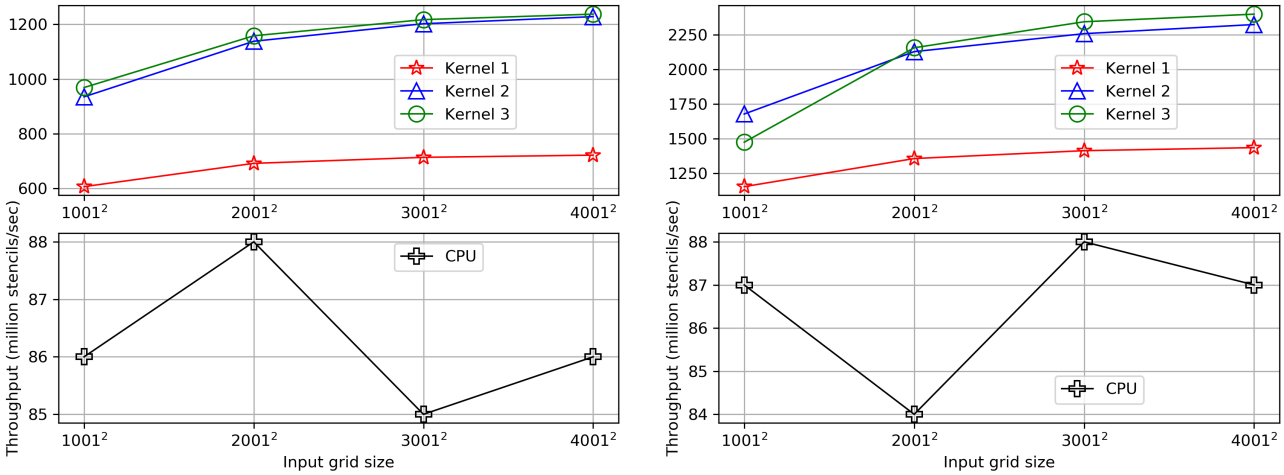


(a) Double precision computations.

(b) Single precision computations.

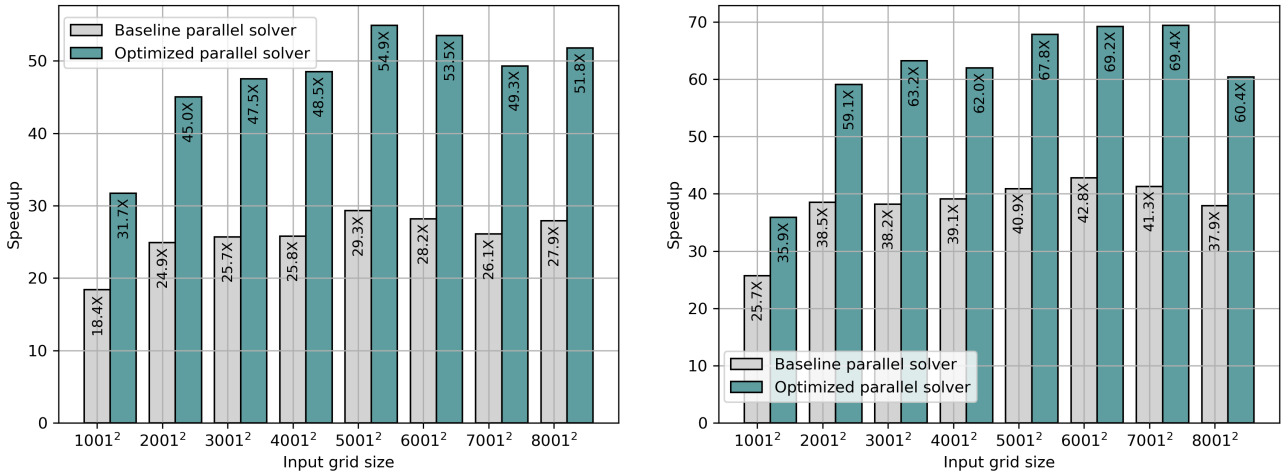
Figure 5.2: The throughput comparison for the CPU sequential implementation and the three different kernels on the Tesla P100 for different problem sizes. Kernel 1: baseline GPU implementation; Kernel 2: shared memory implementation with granularity 1; Kernel 3: texture memory implementation with granularity 1. Throughput was measured for both single and double precision computations.

According to the experiment data, even the unoptimized parallel version is significantly faster than the serial algorithm on both systems. If we consider the problem size 4001×4001, for the double precision computations, the baseline version is 8 times faster than the serial version on the GTX 960M, and it is 26 times faster on the Tesla P100. For the single precision computations, the speedup factor is 16 on the GTX 960M and on the Tesla P100, the baseline version is 39 times faster than the serial version. After optimizing the algorithm, for the double precision computations, the optimized version is 14 times faster than the serial version on the GTX 960M. On the Tesla P100, the speedup factor of the optimized version is 49. For the single precision computations, on the GTX 960M, the optimized version is 27 times faster and on the Tesla P100, it is 62 times faster.



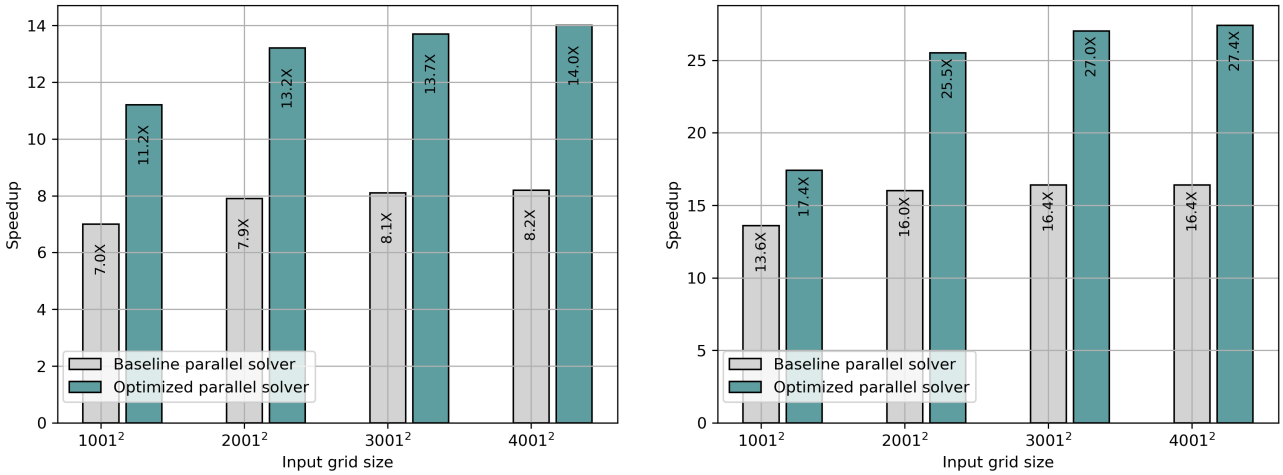
(a) Double precision computations. **(b)** Single precision computations.

Figure 5.3: Throughput comparison for the CPU sequential implementation and the three different kernels on the GTX 960M for different problem sizes. Kernel 1: baseline GPU implementation; Kernel 2: shared memory implementation with granularity 2; Kernel 3: texture memory implementation with granularity 8. Throughput was measured for both single and double precision computations.



(a) Double precision computations. **(b)** Single precision computations.

Figure 5.4: The speedup of the GPU implementation on the Tesla P100 over the CPU sequential implementation. Baseline (Kernel 1) is the unoptimized parallel version of the algorithm. Kernel 2 (with granularity 1) is the optimized parallel version which gives the fastest performance on the Tesla P100.



(a) Double precision computations.

(b) Single precision computations.

Figure 5.5: The speedup of the GPU implementation on the GTX 960M over the CPU sequential implementation. Baseline (kernel 1) is the unoptimized parallel version of the algorithm. Kernel 3 (with granularity 8) is the optimized parallel version which gives the fastest performance on the GTX 960M.

Chapter 6

Application to Magnet Design

After implementing a numerical solver, the next important step is to test whether the simulated results are correct regarding the behavior of the system. Therefore, it is necessary to use a validation method to analyze the correctness of the simulated data obtained from the implemented algorithm. There are various validation methods and one way of validating a numerical algorithm is comparing the solutions to be validated with the solutions through other numerical methods previously validated [32]. Since Kyla Smith has validated her results in her M.Sc. thesis, we use the results given in Ref. [6] as a reference to validate our parallel numerical solver.

As discussed in Section 1.4, our problem of interest is the design of a cylindrical electromagnet for low-field MRI. The design method requires that we solve the magnetic scalar potential between two nested cylindrical volumes, with the boundary conditions given in Table 1.1. Ideally, the outer cylinder is much larger than the inner cylinder, giving the free-space solution for the magnet windings on the latter. Since the memory requirements and run-time are proportional to the area between the cylinders, it is important that the outer cylinder be only as large as necessary to provide a solution with sufficient accuracy. As a result, an important step in the design process is to check the convergence of the solution as a function of the ratio of the outer to inner cylinder dimensions, as well as mesh size. As shown below, we recover the same results as in Ref. [6] and with similar rates of convergence, keeping in mind that different methods and mesh geometries were employed (FEM versus FDM).

6.1 Solving for the Magnetic Scalar Potential

As in Ref. [6], we first define α and β as the ratios of the radius and the half-height of the inner and outer cylinders:

$$\alpha = r_1/r_0, \text{ and} \tag{6.1}$$

$$\beta = z_1/z_0, \tag{6.2}$$

where r_1 and r_0 are the radii of the outer and inner cylinder, while z_1 and z_0 are the half-heights of the outer and inner cylinder. In all the experiments, r_0 and z_0 are fixed and have the values of 0.25 m and 0.5 m.

First, we compute Φ_{inside}, which is the scalar potential inside the inner cylinder. As discussed in Section 1.4.2, the magnetic scalar potential Φ inside the magnet is given by $\Phi_{\text{inside}} = -Bz/\mu_0$, where z varies within the range $[-0.5, 0.5]$ and $\mu_0 = 4\pi \times 10^{-7}$ is the permeability of free space. The desired uniform magnetic field B is chosen to be 0.008 T as in Ref. [6]. The contour plot of Φ_{inside} associated with B is shown in Fig. 6.1. As we can see, these contours of Φ_{inside} are horizontal and evenly spaced.

A solution for Φ in the region between two cylinders (i.e. Φ_{outside}) is obtained by solving the Laplace equation using the implemented algorithm and the contour plot of Φ_{outside} is shown in Fig. 6.2. Since zero flux passing through the boundaries 1,2,3, and 5, the contours of Φ are normal to these boundaries. Now, using the symmetry, we can map this solution to get the complete solution for the full 2D domain as in Fig. 6.3.

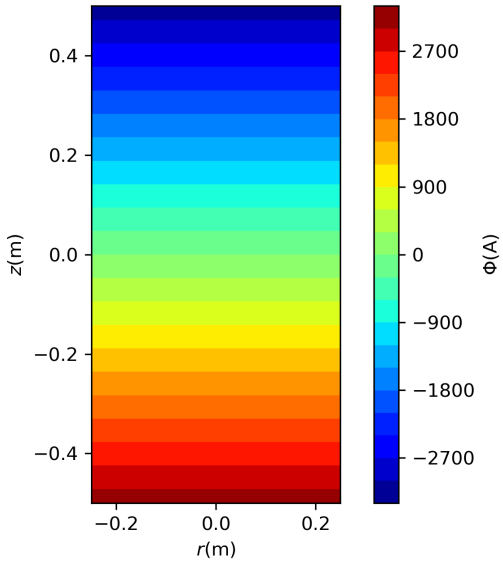


Figure 6.1: The contour plot of Φ_{inside}. The contours of Φ inside the inner cylinder (i.e. magnet) are horizontal and uniformly spaced. The quantity Φ is given in units of amperes (A).

Since we need a free space solution for Φ outside the magnet, we repeatedly solve the problem while increasing the outer boundary until Φ reaches its free space solution (i.e. until the effect of the outer

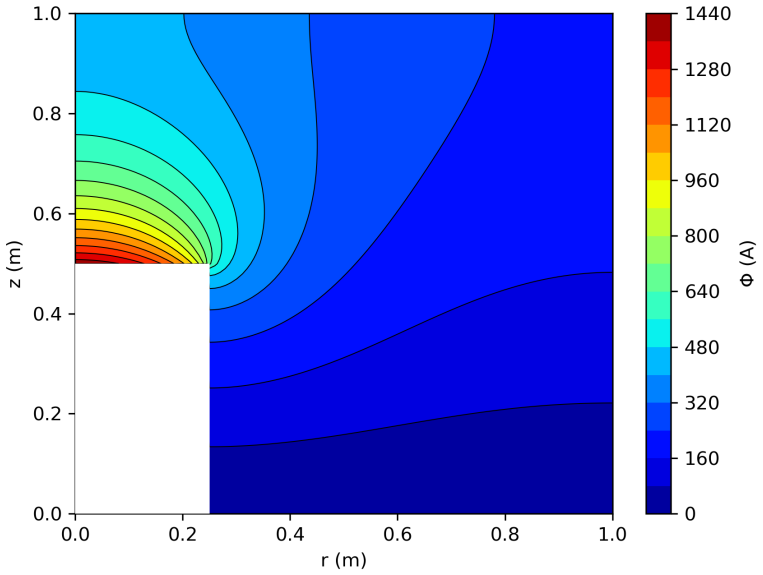


Figure 6.2: The contours of Φ within the solution region between the two cylinders. The solution for Φ is obtained using $\alpha = 4$, $\beta = 2$, and the mesh spacing of 0.002 m on both sides of the mesh. Φ is given in units of amperes.

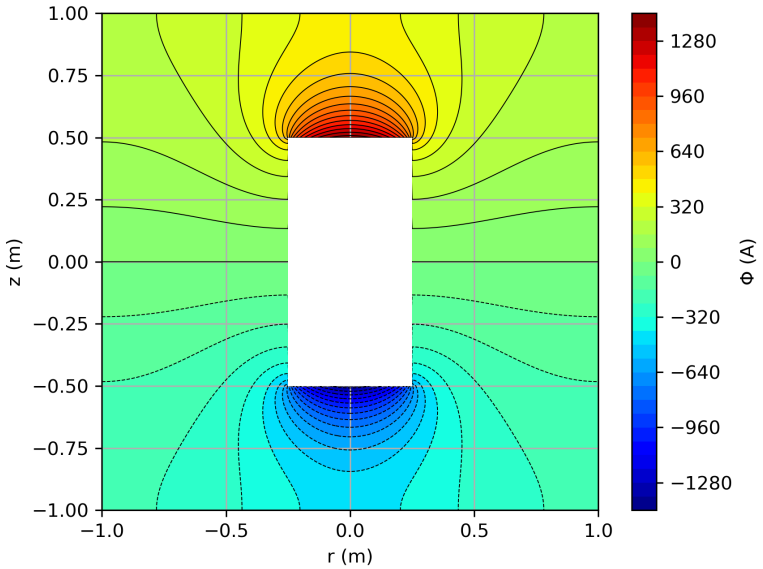
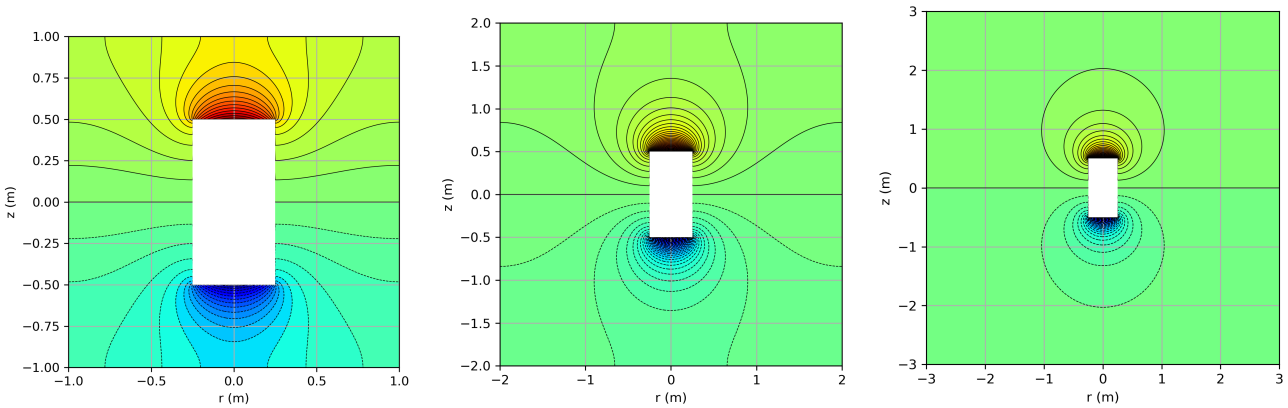


Figure 6.3: The contours of Φ within the full 2D domain between the region between two cylinders. Here, $\Phi(-z) = -\Phi(z)$ and the negative contours are shown by dashed lines. the results are with $\alpha = 4$, $\beta = 2$, and the mesh spacing is 0.002 m on both sides of the mesh.

boundary to the solution is minimum). Figure 6.4 shows the variation of the solution of Φ as outer boundary size increases.



(a) $\alpha = 4$ and $\beta = 2$. **(b)** $\alpha = 8$ and $\beta = 4$. **(c)** $\alpha = 12$ and $\beta = 6$.

Figure 6.4: The variations of contours of Φ_{outside} as the outer boundary size increases. Here, the inner cylinder size (r_0 and z_0) is fixed and the outer boundary size (r_1 and z_1) varies as the given α, β values.

6.2 Scalar Potential Difference Across the Surface of the Magnet

Having the values of Φ_{inside} and Φ_{outside}, now we compute the discontinuity of the scalar potential Φ at the boundary of the cylindrical surface of the magnet. As discussed in Section 1.4.3, this scalar potential difference $\Delta\Phi$ is related to a surface current density and is used later to design the wire placement of the magnet. The scalar potential difference $\Delta\Phi$ is given by

$$\Delta\Phi = \Phi_{\text{outside}} - \Phi_{\text{inside}}. \tag{6.3}$$

Figure 6.5 shows $\Delta\Phi$ along the body of the inner cylinder, where $|r| = r_0$ and $-z_0 \leq z \leq z_0$. The difference $\Delta\Phi$ along the caps of the inner cylinder is shown in Fig. 6.6, where $|z| = z_0$ and $-r_0 \leq r \leq r_0$. These figures showing the scalar potential difference across the magnet surface are the same as what was achieved by Kyla in Ref. [6].

6.3 Convergence of $\Delta\Phi$

Even though the implemented algorithm is used to solve for Φ_{outside}, the quantity that is important to determine the wire placement is $\Delta\Phi$. Since we are using a numerical method to obtain $\Delta\Phi$, we need to check for the convergence of $\Delta\Phi$ over two parameters: (i) mesh spacing and (ii) outer boundary size.

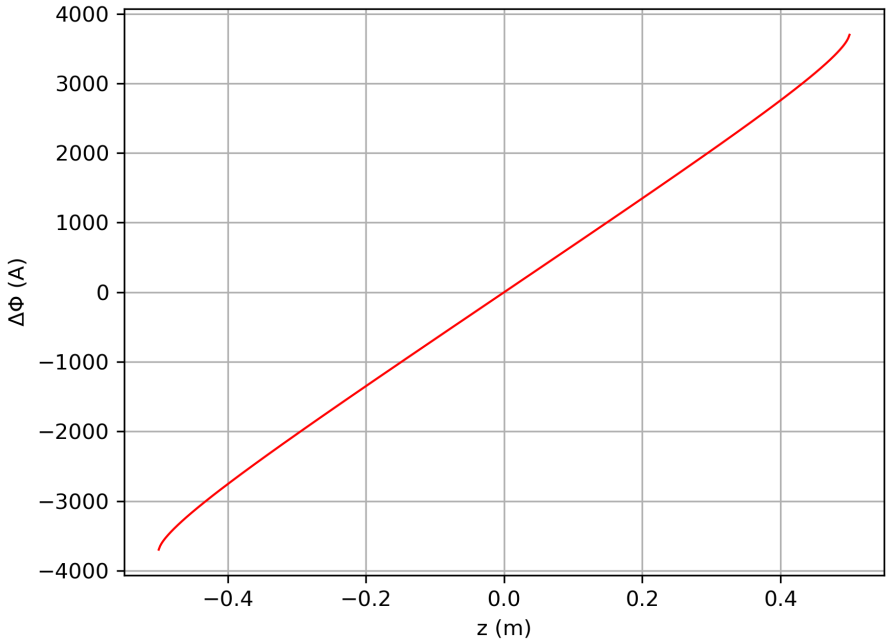


Figure 6.5: The scalar potential difference $\Delta\Phi$ across the body of the magnet for $\alpha = 28$, $\beta = 14$, and the mesh spacing of 0.002 m on both sides of the mesh.

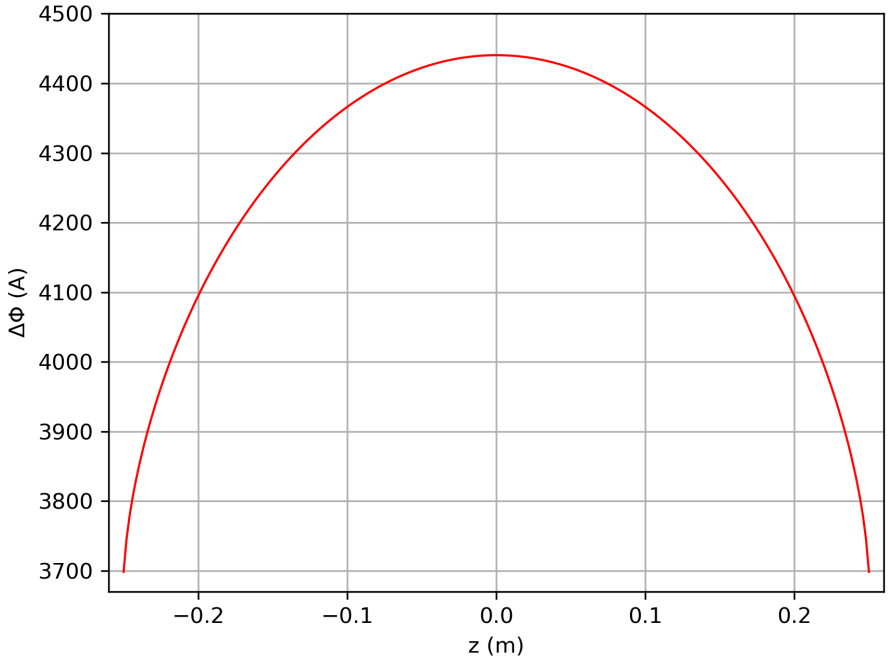


Figure 6.6: The scalar potential difference $\Delta\Phi$ across the cap of the magnet for $\alpha = 28$, $\beta = 14$, and the mesh spacing of 0.002 m on both sides of the mesh.

6.3.1 Mesh Spacing

As discussed in Section 1.3, coarse meshes involve higher discretization error and fine meshes involve lower discretization error. As a result, the solution is more accurate when the mesh spacing is smaller. In order to see the convergence of $\Delta\Phi$ as the mesh spacing decreases, we generate the solution for Φ_{outside} for different mesh spacings and compute the normalized difference in $\Delta\Phi$ as

$$\text{Normalized difference in } \Delta\Phi = \frac{\Delta\Phi - \Delta\Phi^{\text{smallest}}}{\Delta\Phi^{\text{smallest}}}, \tag{6.4}$$

where, $\Delta\Phi^{\text{smallest}}$ refers to the $\Delta\Phi$ corresponding to the mesh with the smallest mesh spacing. For this convergence test, the smallest mesh spacing was chosen as 0.002 m. Figures 6.7 and 6.8 shows the log normalized difference in $\Delta\Phi$ along the body and the cap of the inner cylinder respectively. Here, the log scale was used to visualize the normalized difference in $\Delta\Phi$, because with the log scale, a large range of data can be displayed without small values being compressed down into the bottom of the graph.

According to Fig. 6.7 and 6.8, we can see that the normalized difference in $\Delta\Phi$ decreases as the mesh spacing decreases. These results of convergence of $\Delta\Phi$ when varying the mesh spacing, are consistent with what Kyla has achieved in Ref. [6], noting that our method and the method in Ref. [6] were not identical.

6.3.2 Outer Cylinder Size

Since the finite difference method requires the problem space to be a closed region, the outer cylinder is used for the computational purpose to bound the problem domain. A true free space solution for Φ_{outside} can be obtained when the outer boundary is at infinity. Therefore, the outer cylinder must be sufficiently away from the inner cylinder, so that the changes of the outer cylinder size have a minimum effect on $\Delta\Phi$. We calculate $\Delta\Phi$ by varying the outer cylinder size and compute the normalized difference in $\Delta\Phi$ as

$$\text{Normalized difference in } \Delta\Phi = \frac{\Delta\Phi - \Delta\Phi^{\text{largest}}}{\Delta\Phi^{\text{largest}}}, \tag{6.5}$$

where $\Delta\Phi^{\text{largest}}$ refers to $\Delta\Phi$ corresponding to the problem with the largest outer cylinder size. The largest outer cylinder size was chosen as 7 m by 7 m, hence, $\alpha = 28$ and $\beta = 14$. Figures 6.9 and 6.10 shows how the log normalized difference in $\Delta\Phi$ changes along the body and the cap of the inner cylinder as the outer cylinder size increases (i.e. as $\alpha, \beta \to \infty$). For these experiments, the mesh spacing was chosen to be 0.002 m on both sides of the mesh. According to the graphs, the normalized difference in

$\Delta\Phi$ decreases as the outer cylinder size increases and the results are consistent with the results given in Ref. [6], again keeping in mind that two different methods (FEM versus FDM) were employed.

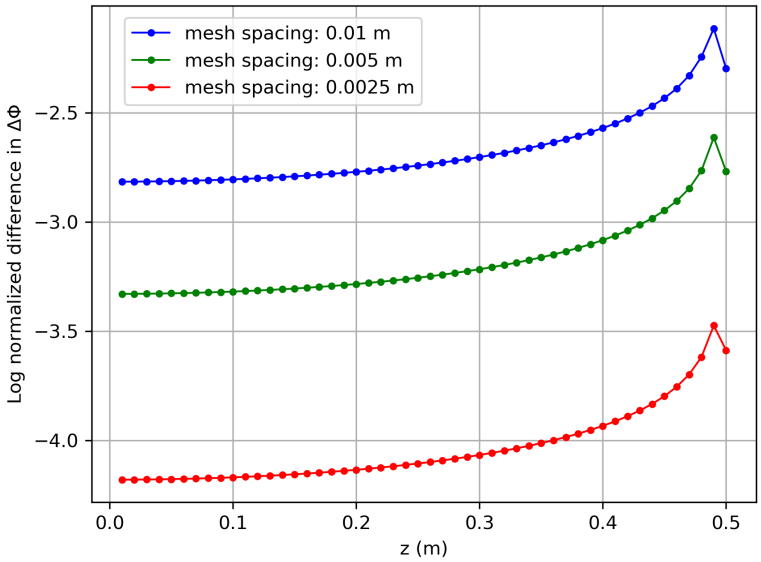


Figure 6.7: The convergence of $\Delta\Phi$ along the body of the inner cylinder as the mesh spacing decreases. The normalized difference in $\Delta\Phi$ was calculated using the smallest mesh spacing as 0.002 m on both sides of the mesh. All the results were obtained with $\alpha = 28$ and $\beta = 14$. The graph was plotted using 50 values within the region [0, 0.5].

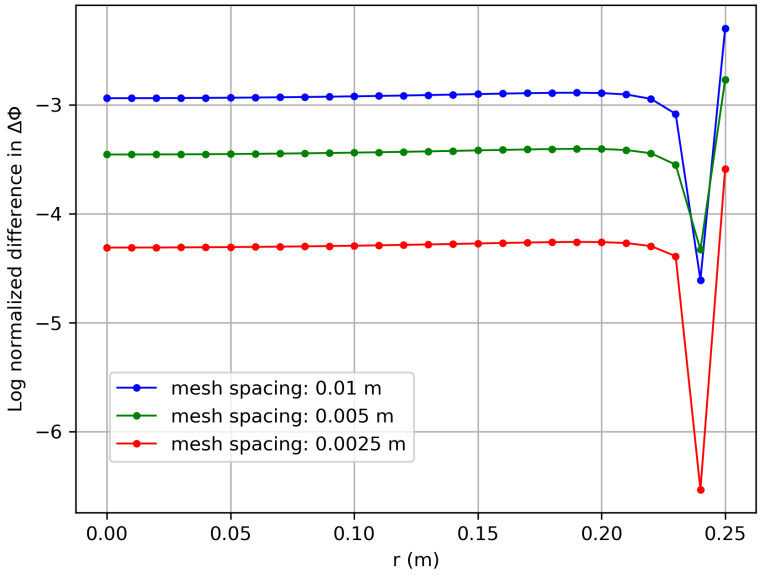


Figure 6.8: The convergence of $\Delta\Phi$ along the cap of the inner cylinder as the mesh spacing decreases. The normalized difference in $\Delta\Phi$ was calculated using the smallest mesh spacing as 0.002 m on both sides of the mesh. All the results were obtained with $\alpha = 28$ and $\beta = 14$. The graph was plotted using 25 values within the region [0, 0.25].

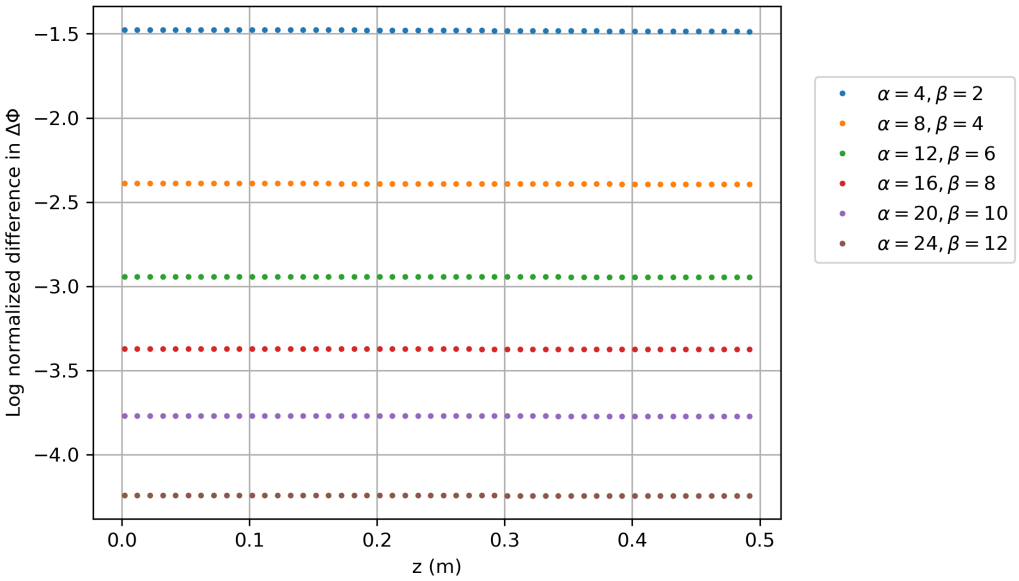


Figure 6.9: The convergence of $\Delta\Phi$ along the body of the inner cylinder as the outer cylinder size increases. To compute the normalized difference in $\Delta\Phi$, $\Delta\Phi^{\text{largest}}$ was computed for $\alpha = 28$ and $\beta = 14$. The mesh spacing was 0.002 m on the both sides of the mesh. For clarity, only every fifth point is shown.

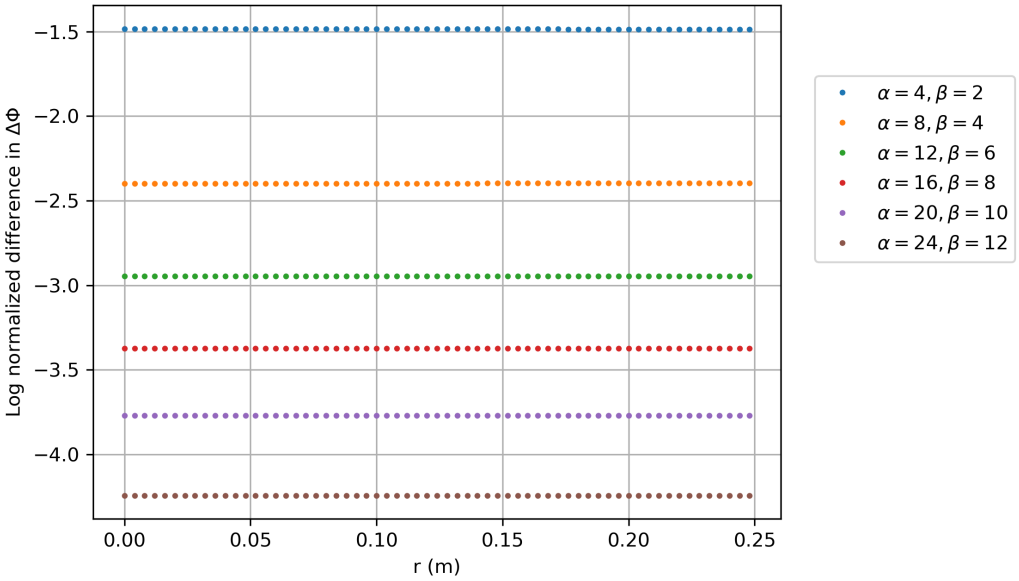


Figure 6.10: The convergence of $\Delta\Phi$ along the cap of the inner cylinder as the outer cylinder size increases. To compute the normalized difference in $\Delta\Phi$, $\Delta\Phi^{\text{largest}}$ was computed for $\alpha = 28$ and $\beta = 14$. The mesh spacing was 0.002 m on the both sides of the mesh. For clarity, only every second point is shown.

In this chapter, by plotting different graphs for different quantities, we could conclude that the results obtained from the implemented parallel solver are numerically correct compared to the results given by Kyla in Ref. [6]. Therefore, our algorithm can be used in practice to estimate the surface current distribution for a given target field. This surface current then can be discretized into wires to design the magnet. Since our main focus in this thesis is to implement an efficient parallel Laplace solver, in this study, we do not discuss wire placement methods in detail. In the following section, we only discuss designing a theoretical magnet.

6.3.3 Designing a Theoretical Magnet

After computing the magnetic scalar potential differences over the magnetic surface, these results were used to demonstrate a wire winding pattern for designing a conceptual magnet. The total current on the upper half of the cylinder was obtained using the $\Delta\Phi$ at the locations A and B (see Fig. 6.11) as follows:

$$\text{Total current on the upper half} = \Delta\Phi_B - \Delta\Phi_A$$

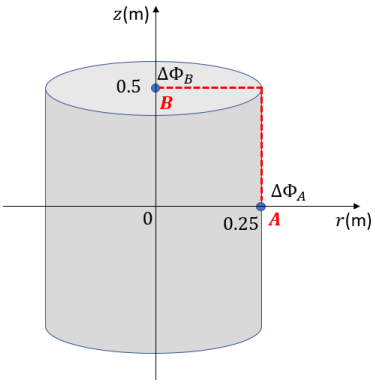


Figure 6.11: The difference of the discontinuity in Φ between the locations A and B gives the total current on the upper half of the cylinder.

As the next step, this total current was divided into N equal segments, where N is the number of wires selected for the upper half of the magnet. Now we can calculate the size of a current step ($\mathrm{I_{step}}$) as

$$\mathrm{I_{step}} = \frac{\Delta\Phi_B - \Delta\Phi_A}{N}.$$

Since the total number of wires on the full cylinder is $2N$, the total number of wires is an even number. Hence, starting at the center of the magnet, then we calculated the values $\Delta\Phi_i$ corresponding to each

wire on the cylindrical surface as follows:

$$\Delta\Phi_i = (i - 0.5) \times \mathrm{I}_{\mathrm{step}}. \tag{6.6}$$

Then, the (r_i, z_i) position on the cylindrical surface corresponding to each $\Delta\Phi_i$ was calculated using the graphs 6.5 and 6.6. Here, the interpolation was done using the `interpolate.interp1d()` function in Python. Figures 6.12 and 6.13 show an example of calculating the wire positions on the upper half of the cylinder for 10 wires. Figure 6.14 shows a sample magnet with 20 wires on its body and the caps.

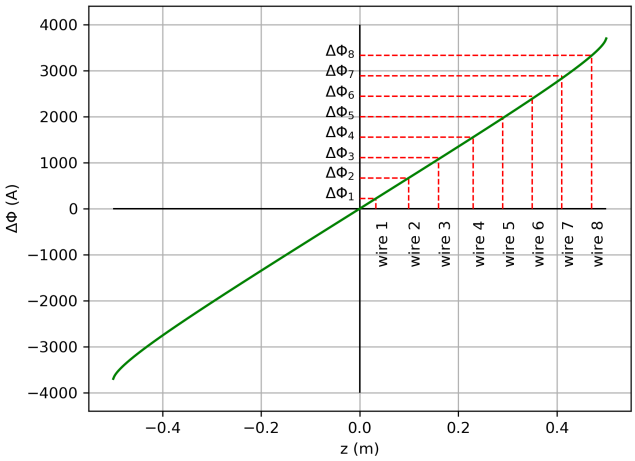


Figure 6.12: The process of calculating the wire positions (r_i, z_i) on the upper half of the body of the magnet. For the body wires, $r_i = 0.25$ m and z_i values are obtained using the interpolated function for $\Delta\Phi$ along the magnet body. In this example, 8 out of 10 wires are on the upper half of the body.

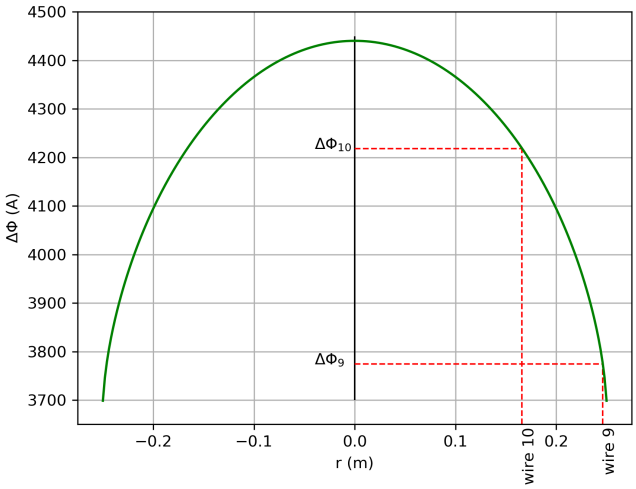


Figure 6.13: The process of calculating the wire positions (r_i, z_i) on the cap of the magnet. For the cap wires, $z_i = 0.5$ m or -0.5 m depending on whether it is the upper cap or the lower cap, and r_i values are obtained using the interpolated function for $\Delta\Phi$ along the cap of the magnet. In this example, 2 out of 10 wires are cap wires.

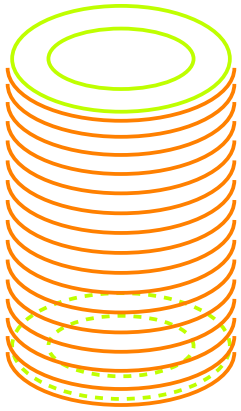


Figure 6.14: A sample magnet with 20 wires on its surface. There are 4 cap wires represented by green colour. The orange colour wires are body wires and this design has 16 body wires.

6.3.4 Magnetic Field Homogeneity

After computing the wire positions on the magnetic surface, we used these data to compute the magnetic field along the axis of the cylinder. The magnetic field per unit current at a point z on the axis due to i^{th} current loop can be computed using an application of the Biot-Savart law as follows:

$$\frac{B_{z_i}(z)}{I} = \frac{\mu_0 r_i^2}{2(r_i^2 + (z - z_i)^2)^{3/2}}, \tag{6.7}$$

where μ_0 is the permeability of free space and (r_i, z_i) is the radius and z-position of the i^{th} current loop on the magnet surface. The net magnetic field at point z was obtained by summing up the contribution from each current loop as follows:

$$\frac{B_z(z)}{I} = \sum_{i=1}^{N} \frac{B_{z_i}(z)}{I}, \tag{6.8}$$

where N is the total number of wires. Figure 6.15 shows the normalized field difference on-axis for different numbers of wires. According to the figure, we can see that the number of wires greatly affects the field homogeneity and the field homogeneity increases as the number of wires increases. The efficiency of the magnetic field was computed as

$$\chi = \frac{B}{I} \tag{6.9}$$

using B at the centre of the magnet. The field efficiencies for different numbers of wires are as below:

- for 50 wires : $\chi = 0.045$ mT/A

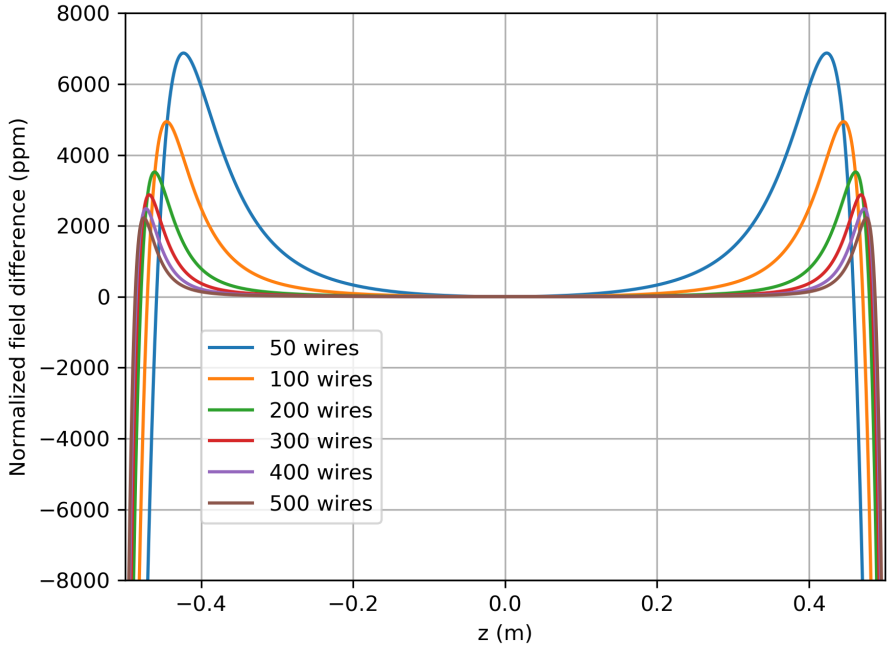


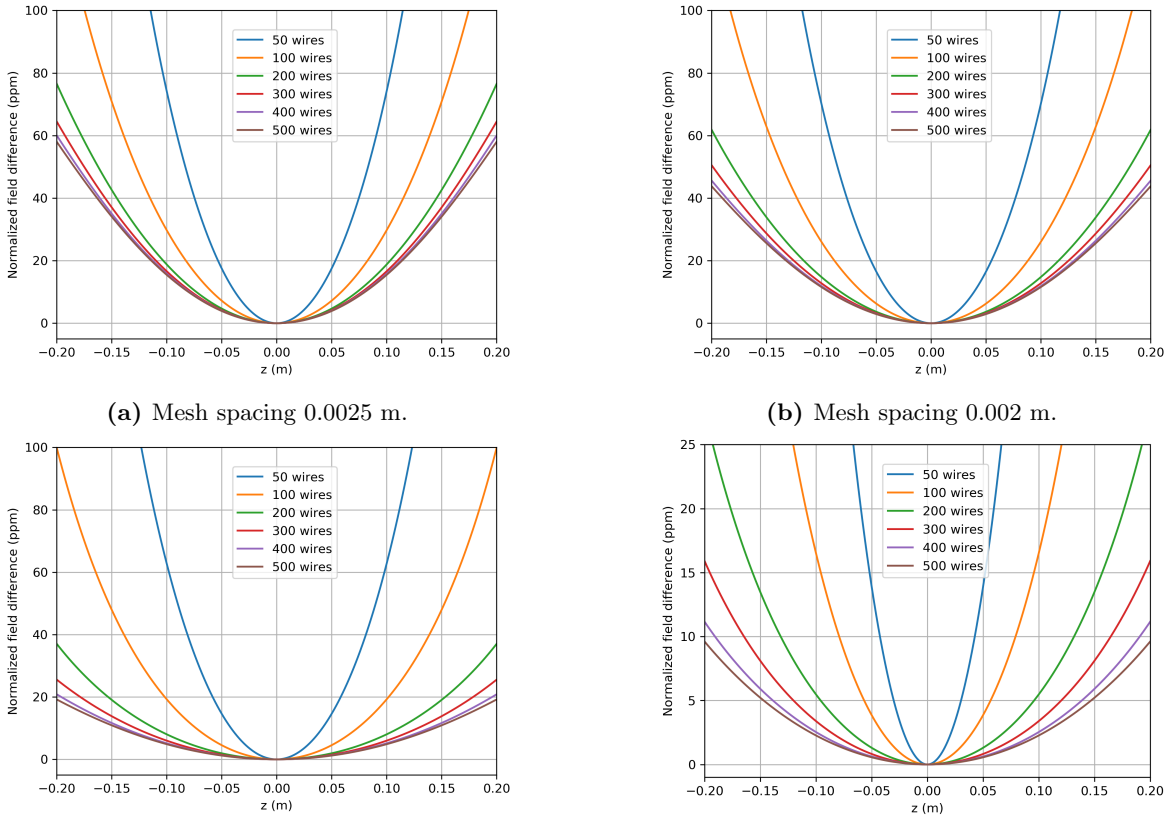
Figure 6.15: The normalized field difference on-axis for different numbers of wires. The field homogeneity increases as the number of wires increases. After 400 wires, the improvement of the homogeneity becomes less significant.

- for 100 wires: $\chi = 0.090$ mT/A
- for 200 wires: $\chi = 0.18$ mT/A
- for 300 wires: $\chi = 0.27$ mT/A
- for 400 wires: $\chi = 0.36$ mT/A
- for 500 wires: $\chi = 0.45$ mT/A

As we can see, the field efficiency also increases with the number of wires.

Figure 6.16 shows the normalized field difference for different numbers of wires only for the middle 40 cm range of the magnet. Here we have plotted the same graph by varying the mesh spacing of the grid. We considered four different mesh spacings, 0.0025 m, 0.002 m, 0.001 m, and 0.0005 m. According to the graphs, we can see that the ppm values of the normalized difference decrease as the mesh spacing decreases. By looking at the results, we can expect field difference to decrease by further reducing the mesh spacing. However, it cannot be further reduced because of the memory limitation.

When comparing our results with the results presented in Kyla's thesis, these middle-range ppm values are slightly higher than Kyla's results. There are several reasons that we can think of for these differences.



(a) Mesh spacing 0.0025 m.

(b) Mesh spacing 0.002 m.

(c) Mesh spacing 0.001 m.

(d) Mesh spacing 0.0005 m.

Figure 6.16: The normalized field difference in the middle 40 cm range on the axis of the cylinder for different mesh spacings. The field homogeneity improves as we decrease the mash spacing.

First of all, since two different discretization methods have employed (the finite difference method versus the finite element method), we cannot expect both methods to behave exactly the same way. Different numerical methods can work differently (in terms of the rate of convergence over the mesh spacing) on the same problem. Also, without implementing and experimenting with both methods, it is difficult to say in advance which method works better in terms of the rate of convergence over the mesh spacing. In addition, the interpolation function also may have an effect on our results. Tiny changes in the interpolated values can cause a large difference at the ppm-level. When the mesh is finer, there are more data points and then the interpolation algorithm can predict more accurately since more data points are available to interpolate. That might be one of the reasons for receiving better results for field values when the mesh is finer.

Chapter 7

Conclusion

As the concluding remark of our work, in this chapter we first summarize what has been presented in this thesis. Then we discuss what we have achieved and further improvements possible for this work.

7.1 Summary

We have implemented an efficient parallel solver for solving the magnetostatic applications described in Ref. [6]. The development of the solver consists of several steps and the flowchart depicting the process of solving the problem is shown in Fig. 7.1.

Our magnetostatic problem, the physical model, and the mathematical formulation of the problem were explained in Chapter 1 based on Ref. [6]. The magnetostatic problem was to design a uniform low-field magnet which was cylindrical in shape. The physical model described that the difference in the magnetic scalar potential across the surface of the cylinder is directly related to the current distribution of the surface which generates the desired magnetic field. Since the internal "target" field is known, the scalar potential inside the cylinder was set accordingly. As a result, the problem reduced to finding the scalar potential outside the cylinder by solving the Laplace equation in the region outside the cylinder. Therefore, we needed to create a mathematical model to solve the Laplace equation in the desired region. The model included defining the problem domain and determining the behavior of the magnetic scalar potential on each boundary of the domain. The goal of this thesis was to develop an efficient GPU-based tool to solve this boundary value problem and not to explore specific coil winding patterns.

The next step was to solve the Laplace equation within the defined domain using derived boundary conditions. Chapter 2 described the numerical methods that we used to solve the problem, which were the finite different method (FDM) and successive over relaxation (SOR) method. The FDM was used

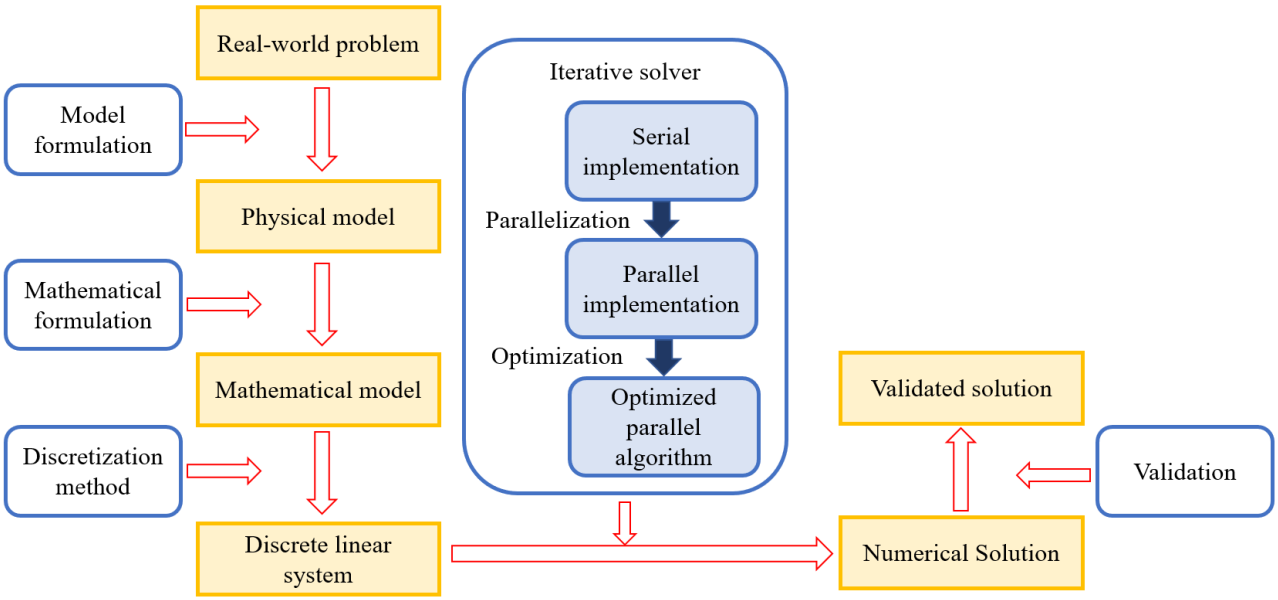


Figure 7.1: The flowchart presents the flow of the work which has been presented in this thesis. The first three steps until the mathematical model formulation of the flow are based on Ref. [6] and this thesis mainly focus on the proceeding steps.

to discretize the continuous information given by the Laplace equation into a discrete set of linear equations. The SOR method was then used to solve the obtained linear system iteratively until the convergence is reached.

Since our aim was to implement this iterative solver on a GPU, a brief description of parallel programming with CUDA was given in Chapter 3. Chapter 4 described the implementation details of the iterative solver. The serial algorithm was implemented first and then the code was modified to run in parallel in multiple cores. After identifying the optimization opportunities, this first-go, or baseline parallel implementation was accelerated using CUDA optimization strategies. The performance and the speedup achieved by using different optimization techniques were given in Chapter 5.

The last step was to test whether the simulated results given by the implemented algorithm are correct regarding the behavior of the system. The magnetic scalar potential inside and outside the cylinder, and other related results were shown in Chapter 6. To validate our solutions, we used the results given in Ref. [6] as a reference and we could recover the same results.

7.2 Conclusion and Discussion

We could implement an efficient and accurate GPU parallel solver which can be used in the magnet design project in Ref. [6]. The algorithm was approximately 13 times faster than the sequential CPU code even in my laptop (with a GPU GTX 960M) and it is approximately 50 times faster when running on a Testa P100. Because of the high computing power of GPUs, our parallel code has the ability to solve finer meshes (for greater accuracy) which consist of large computational load efficiently with less amount of time. Since we used C++ and CUDA C to implement the algorithm, the code can be run on any system which has an NVIDIA GPU.

The implemented algorithm can be used to solve cylindrical-shaped problems with different target field distributions, but the implementation is not able to handle the problems with different geometries. In order to use the algorithm for a different geometric-shaped problem, one should derive corresponding finite difference equations, fill the weight arrays W_1, W_2, W_3, W_4 with new values and make appropriate changes to the boundary limits of the code according to the corresponding geometric shape.

When using the SOR method as the iterative solver, the choice of the over-relaxation parameter ω strongly affects the rate at which the SOR method converges. Even though the optimal omega ω_b can be computed using the spectral radius of the Jacobi iteration matrix ρ_{jacobi}, calculating ρ_{jacobi} requires an impractical amount of computation. Since ω_b depends on the factors such as the geometric shape of the domain and the mesh spacing, the method to estimate ω_b described in this thesis works only for our particular problem. Therefore, as an improvement for the algorithm, one can try the *ad hoc SOR method* given in Ref. [33], in which a different relaxation factor ω is determined for each node of the mesh, depending on the coefficients of the finite difference equations, the nature of the problem domain, and the boundary conditions.

Future complementary work to this thesis could include the development of a GPU-based winding pattern optimizer and a GPU-based Biot-Savart solver. This would allow one to take the results of our Laplacian solver and quickly determine what might be the best winding pattern for the electromagnet based on calculations of the magnetic field homogeneity.

Appendices

Appendix A

Experimental Determination of ω_b

In our study, the SOR method is used to update each node of the mesh until the convergence is reached. The choice of the over-relaxation parameter ω significantly affect the rate at which the SOR method converges. Figure A.1 and A.2 shows how the number of iterations varies with the value of ω, when the mesh size is 401×401 and TOLARENCE is given as 0.5×10^{-6}. Figure A.1 shows the variation of the number of iterations when ω varies in the second decimal place. When ω is 1.91, the number of iterations is more than 50000, but when ω is 1.99, the number of iterations is less than 10000. Figure A.2 illustrates the variation of the number of iterations of the same problem when ω varies in the third decimal place. When ω changes from 1.991 to 1.995, the number of iterations required to converge has reduced by half. These results imply that the choice of ω strongly affects the rate of convergence of the SOR method. Even though the value of ω is a critical parameter, it is difficult to determine in advance the value of ω that is optimal with respect to the rate of convergence when the problem is non-rectangular. Also, the optimum omega ω_b varies with the size of the problem domain and the mesh spacing. Therefore, we ran some experiments to determine ω_b for our particular problem and the experimental details are described below.

One of the Kulsrud's experiments in Ref. [16] suggests to estimate ρ_{jacobi} with Eq. (2.45) for non-rectangular problems using the dimensions of the rectangular region which encloses the non-rectangular region. For our particular problem, we found that the ρ_{jacobi} obtained for a rectangle 3 times larger (in each direction) than the region which encloses our problem domain gives a better approximation for ω_b. That is, if our circumscribing region is $N \times M$, there is a rectangular problem of size $3N \times 3M$, whose ω_b is approximately equal to ω_b of our non-rectangular problem. Thus, if the circumscribing region of

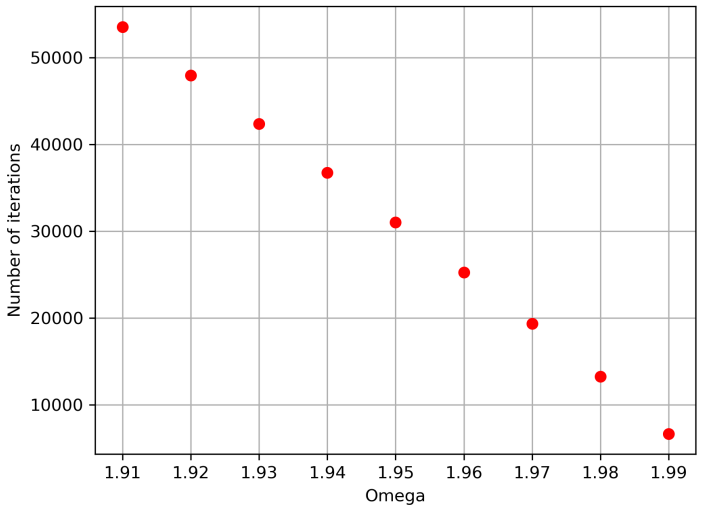


Figure A.1: This figure illustrates number of iterations required to converge as ω varies from 1.91 to 1.99. The results are obtained by solving a $r_1 = 4$ m by $z_1 = 4$ m problem with the mesh spacing of 0.01 m. When ω increases in the second decimal place, the number of iterations required decreases and when $\omega = 1.99$ the algorithm converges only after 6661 iterations.

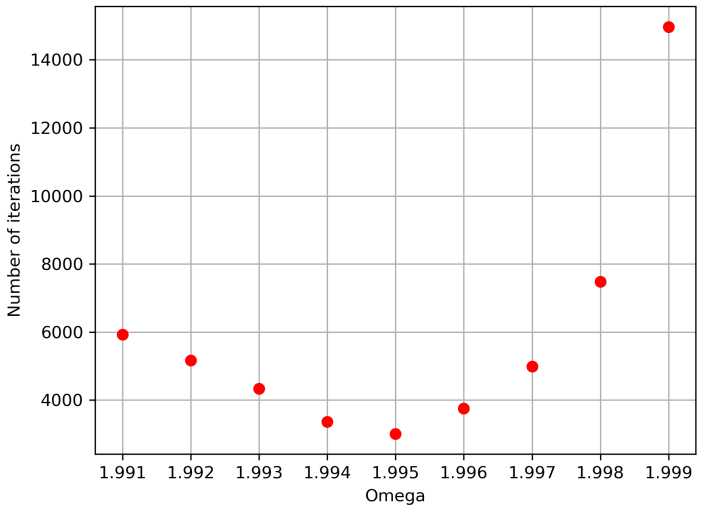


Figure A.2: This shows the variation of the number of iterations required to converge for the same problem illustrated in Fig. A.1, when ω varies in the third decimal place. When considering ω values up to three digits, the algorithm converges with a minimum number of iterations if ω is 1.995. The results are obtained by solving a $r_1 = 4$ m by $z_1 = 4$ m problem with the mesh spacing of 0.01 m.

our problem is $N \times M$ with the mesh spacing of δr and δz in each direction, ρ_{jacobi} is computed as

$$\rho_{jacobi} = \frac{\cos\frac{\pi}{3J} + (\frac{\delta r}{\delta z})^2 \cos\frac{\pi}{3L}}{1 + (\frac{\delta r}{\delta z})^2}, \tag{A.1}$$

where $J = (N/\delta r)$ and $L = (M/\delta z)$. After computing ρ_{jacobi}, Eq. (2.44) is used to compute ω_b. Table A.1

and Table A.2 show the actual ω_b values and ω_b values computed by Eq. (A.1) and Eq. (2.44). The actual ω_b values corresponding to each mesh size are obtained by trial and error for five significant digits. The radius and the half-height of the inner cylinder (r_0 and z_0 of Fig. 2.4) are set as 0.25 m and 0.5 m for all the computations, as they are the dimensions which need to be used in the magnet design project. Four problem domains are considered for the comparison; $N \times M$ as 1 m $\times$ 1 m, 2 m $\times$ 2 m, 3 m $\times$ 3 m and 4 m $\times$ 4 m. Here we use square domains, because we consider equal sizes for both r_1 and z_1, which are the radius and the half-height of the outer cylinder.

Table A.1: Comparison of ω_b values of our non-rectangular problem with ω_b values of the corresponding rectangular problem. Mesh spacing is 0.01 m on both sides.

Mesh size of the circumscribing region of the non-rect. domain $(N/0.01) \times (M/0.01)$	Mesh size of the corresponding rectangular domain $3(\frac{N}{0.01}) \times 3(\frac{M}{0.01})$	ρ_{jacobi} of the $3(\frac{N}{0.01}) \times 3(\frac{M}{0.01})$ rectangular problem by Eq. (A.1)	ω_b of the $3(\frac{N}{0.01}) \times 3(\frac{M}{0.01})$ rectangular problem by Eq. (2.44)	Actual ω_b of the non-rect. problem by trial & error for 5 sig. digits
100×100	300×300	0.99994516	1.97927172	1.9790
200×200	600×600	0.99998629	1.98958176	1.9892
300×300	900×900	0.99999390	1.99303862	1.9927
400×400	1200×1200	0.99999657	1.99477536	1.9945

Table A.2: Comparison of ω_b values of our non-rectangular problem with ω_b values of the corresponding rectangular problem. Mesh spacing is 0.0025 m on both sides.

Mesh size of the circumscribing region of the non-rect. domain $(N/0.0025) \times (M/0.0025)$	Mesh size of the corresponding rectangular domain $3(\frac{N}{0.0025}) \times 3(\frac{M}{0.0025})$	ρ_{jacobi} of the $3(\frac{N}{0.0025}) \times 3(\frac{M}{0.0025})$ rectangular problem by Eq. (A.1)	ω_b of the $3(\frac{N}{0.0025}) \times 3(\frac{M}{0.0025})$ rectangular problem by Eq. (2.44)	Actual ω_b of the non-rect. problem by trial & error for 5 sig. digits
400×400	1200×1200	0.99999657	1.99477536	1.9947
800×800	2400×2400	0.99999914	1.99738046	1.9973
1200×1200	3600×3600	0.99999961	1.99823520	1.9982
1600×1600	4800×4800	0.99999978	1.99867422	1.9986

Eq. (A.1) works better if the radius r_0 and the half-height z_0 of the inner cylinder is 0.25 m and 0.5 m respectively. If we change the dimensions of the inner cylinder by giving different values for z_0 and r_0, Eq. A.1 may not give a better approximation to ρ_{jacobi} to get an good approximation to ω_b.

Bibliography

[1] M. L. Boas, *Mathematical methods in the physical sciences*. John Wiley & Sons, 2006.

[2] D. R. Lynch, *Numerical partial differential equations for environmental scientists and engineers: a first practical course*. New York: Springer, 2005.

[3] Y. Shapira, *Solving PDEs in C++: Numerical Methods in a Unified Object-Oriented Approach*. Society for Industrial and Applied Mathematics, 2006.

[4] P. E. Rijtema and V. Elias, *Regional approaches to water pollution in the environment*, vol. 20. Springer Science & Business Media, 2012.

[5] V. N. Kaliakin, "Introduction to approximate solution techniques, numerical modeling, and finite element methods," 2001.

[6] K. M. Smith, "A homogeneous rf-shielded magnet for low-field magnetic resonance studies," Master's thesis, University of Manitoba, Manitoba, Canada, 2019.

[7] H. A. Haus and J. R. Melcher, *Electromagnetic fields and energy*, vol. 107. Prentice Hall Englewood Cliffs, NJ, 1989.

[8] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[9] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.

[10] D. Kirk and W.-m. Hwu, "Programming massively parallel processors: a hands-on approach," 2013.

[11] W. H. Wen-mei, *GPU Computing Gems Jade Edition*. Elsevier, 2011.

[12] R. J. LeVeque, *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, vol. 98. Siam, 2007.

[13] P. Knabner and L. Angerman, *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*. Texts in Applied Mathematics, Springer New York, 2006.

[14] E. H. Chao, S. F. Paul, R. C. Davidson, and K. S. Fine, "Direct numerical solution of poissons equation in cylindrical (r, z) coordinates," tech. rep., Princeton Univ., Princeton Plasma Physics Lab., NJ (United States), 1997.

[15] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical recipes-the art of scientific computing*. Cambridge University Press, Cambridge, 2007.

[16] H. E. Kulsrud, "A practical technique for the determination of the optimum relaxation factor of the successive over-relaxation method," *Communications of the ACM*, vol. 4, no. 4, pp. 184–187, 1961.

[17] G. D. Smith and G. D. Smith, *Numerical solution of partial differential equations: finite difference methods*. Oxford university press, 1985.

[18] I. Epicoco and S. Mocavero, "The performance model of an enhanced parallel algorithm for the sor method," in *International Conference on Computational Science and Its Applications*, pp. 44–56, Springer, 2012.

[19] N. Corporation, "Cuda zone." https://developer.nvidia.com/cuda-zone, August 2019.

[20] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA c programming*. John Wiley & Sons, 2014.

[21] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside volta: The world's most advanced data center gpu," *NVidia Parallel for All Blog, Available at: https://devblogs.nvidia.com/inside-volta/*, 2017.

[22] C. NVIDIA, "Cuda c best practices guide v. 4.0," 2011.

[23] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[24] J. Luitjens, "Faster parallel reductions on kepler," *Parallel Forall. NVIDIA Corporation. Available at: https://devblogs.nvidia.com/faster-parallel-reductions-kepler/*, 2014.

[25] G. Amador and A. Gomes, "Cuda-based linear solvers for stable fluids," in *2010 International Conference on Information Science and Applications*, pp. 1–8, IEEE, 2010.

[26] J. T. Liu, Z. S. Ma, S. H. Li, and Y. Zhao, "A gpu accelerated red-black sor algorithm for computational fluid dynamics problems," in *Advanced Materials Research*, vol. 320, pp. 335–340, Trans Tech Publ, 2011.

[27] L. Itu, C. Suciu, F. Moldoveanu, and A. Postelnicu, "Gpu optimized computation of stencil based algorithms," in *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pp. 1–6, IEEE, 2011.

[28] D. L. Foster, "Gpu acceleration of solving parabolic partial differential equations using difference equations," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, p. 1, The Steering Committee of The World Congress in Computer Science, 2011.

[29] E. Konstantinidis and Y. Cotronis, "Accelerating the red/black sor method using gpus with cuda," in *International Conference on Parallel Processing and Applied Mathematics*, pp. 589–598, Springer, 2011.

[30] E. Konstantinidis and Y. Cotronis, "Graphics processing unit acceleration of the red/black sor method," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1107–1120, 2013.

[31] Y. Cotronis, E. Konstantinidis, and N. M. Missirlis, "A gpu implementation for solving the convection diffusion equation using the local modified sor method," in *Numerical Computations with GPUs*, pp. 207–221, Springer, 2014.

[32] R. Jauregui and F. Silva, "Numerical validation methods," *Numerical analysis—theory and application*, pp. 155–174, 2011.

[33] L. W. Ehrlich, "An ad hoc sor method," *Journal of Computational Physics*, vol. 44, no. 1, pp. 31–45, 1981.