GPU Accelerated Iterative Solvers for Linear System of Equations

Bachelor of Technology Project Report

By

Vedant Chourasia **Kunal Garg**

190103102 **190103115**

Project Supervisor

Prof. Atul Kumar Soti

Department of Mechanical Engineering
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI, INDIA,
2023

Declaration

We declare that this written submission represents our ideas in our own words and we have cited and referenced the original sources where others' ideas or words have been included. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute.

Vedant Chourasia (190103102) **Kunal Garg (190103115)**

Chourasia

[illegible]

Department of Mechanical Engineering

Indian Institute of Technology, Guwahati-781039, India

Abstract

In this project, our aim is to develop GPU accelerated numerical solvers for linear systems of equations arising from discretization of partial differential equations such as heat equations. We used CUDA programming for this purpose. CUDA is a parallel computing platform that allows software to use certain types of graphics processing units. Firstly we developed a cuda code for solving some generic randomly generated Diagonally dominant coefficient matrices and then the idea was further expanded to get temperature distribution in a solid plate and cube using i) Jacobi's Iterative method for 2D steady state heat equation, ii) Red Black Gauss Seidel iii) Conjugate gradient method.We also learned to reduce error by increasing the iteration and plotted various graphs about the same. In this project we also analysed time for computation in CPU as well as GPU. Our results show significant speed up in the computational speed of our GPU code as compared to that of the serial CPU code. We then extended the idea of Jacobi's Iterative method for 3D problem and also devised the GPU code for the same.

Table of Contents:

Chapter 1
Introduction and Literature Review

Linear system Ax = b is widely used in applied mathematics. It is not necessary that this may occur directly but it may occur while solving numerical analysis. Linear equations (or linear system) comprises one or more linear equations containing the same set of variables. For example

$$\begin{aligned} 3x - 2y + z &= 1 \\ 2x - 2y + 4z &= -2 \\ -2x + y - 2z &= 0 \end{aligned}$$

is a system of three equations in the three variables *x*, *y*, *z*. A solution to the system above is given by

$$x = 1,\ y = -2,\ z = -2$$

We can verify the solution by substituting the values into each equation to see if the above solution satisfies all 3 equations.

There are two general categories of numerical methods for solving Ax = b:

- ❖ **Direct Methods**: These are methods with a finite number of steps and they end with the exact solution x, provided that all arithmetic operations are exact. The most used of these methods is Gaussian elimination.
- ❖ **Iteration Methods**: Solving systems of linear equations by iterative methods involves the correction of one searched-for unknown value in every step. These methods have a very large number of steps and generally require computational powers of the GPU.
    - ➢ Jacobi's method
    - ➢ Gauss-Seidel's method

- Parallel computing is used to attain high computation power to solve very large iterative equations. GPU as a device was used to perform multiple tasks simultaneously and decrease time taken by CPU to solve the same computation. Cuda programming was used as it provides a platform to work on GPU. Cuda code can be written in many languages, for example C, C++, Fortran, Python and MATLAB.

- The primary advantage of using CUDA for parallel computing is its ability to perform multiple tasks simultaneously, which significantly reduces the time taken to perform the computation. This is achieved by dividing the computation into smaller tasks and distributing them across multiple processing cores in the GPU.
- The most common approach to parallel computing using GPUs is to use the Single Instruction Multiple Data (SIMD) architecture. This approach involves executing the same instruction on multiple data elements simultaneously. This is achieved by dividing the data into multiple processing blocks, which are then executed in parallel on the GPU.
- CUDA programming provides a range of features that can be used to optimize the performance of parallel computations. These features include shared memory, which allows multiple processing cores to share data, and constant memory, which can be used to store read-only data that is accessed frequently.
- Numerical computations that can benefit from parallel computing using GPUs include matrix computations, image processing, and deep learning. Matrix computations, such as matrix multiplication and solving linear systems of equations, are highly parallelizable and can be efficiently implemented on GPUs using CUDA programming. Image processing tasks, such as convolution and edge detection, can also be parallelized using CUDA. Deep learning algorithms, which require large amounts of data and computation, can benefit significantly from GPU acceleration using CUDA programming.
- Parallel computing using GPUs and CUDA programming has shown significant potential in accelerating numerical computations and solving complex problems. The high parallelism and memory bandwidth of GPUs allow for efficient execution of computations, which can significantly reduce the time taken to perform the computation. CUDA programming provides a range of features that can be used to optimize the performance of parallel computations, making it a popular platform for developing parallel applications.

Chapter 2
Parallel Computing and Objective

Parallel computing is using multiple central processing units, or cores on a processor (which act as processors on their own), to work on problems that need a lot of computation. For example, mathematical or scientific problems. The primary goal of parallel computing is to increase available computation power for faster application processing and problem solving.

- For parallel computing we used the Cuda platform. CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).
- OpenMP is mostly famous for shared memory multiprocessing programming.
- MPI is mostly famous for message-passing multiprocessing programming.
- CUDA technology is mostly famous for GPGPU computing and parallelising tasks in Nvidia GPUs.

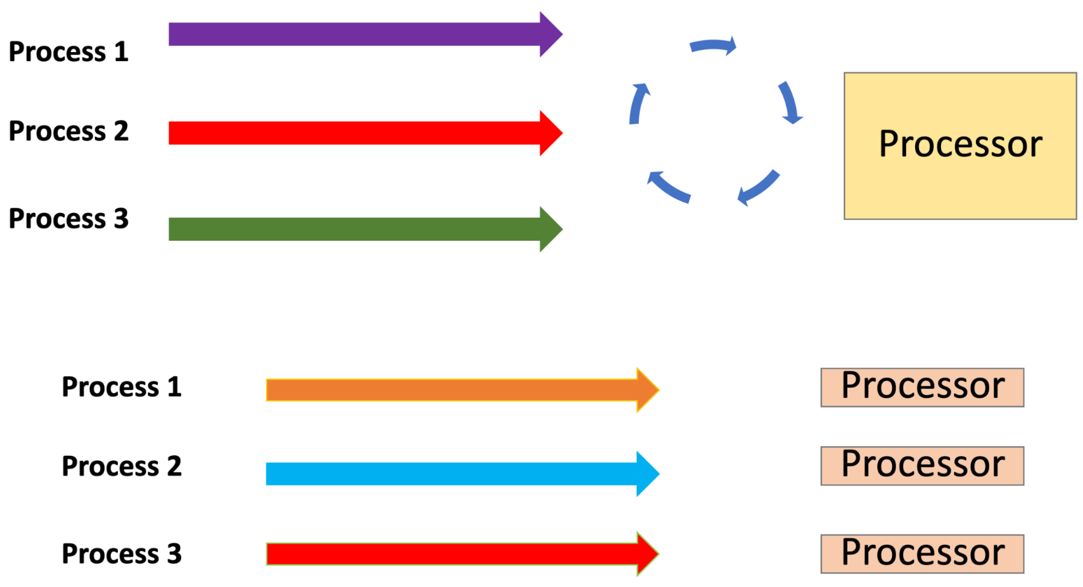


Figure 2.1 Processes of CPU and GPU respectively.

Reference:Jaegeun Han, Bharatkumar Sharma, *Learn CUDA Programming*

As seen in the figure 3.1 the above processes are executed on a single processor and the lower processes are simultaneously executed on different processors which in turn make the GPU processing much faster. Now we will discuss about the advantages,disadvantages of CUDA

- **CUDA Advantages**
    - Designed to run for non graphic purposes.
    - Its software development kit includes libraries, various debugging, profiling and compiling tools.
    - Programming task is simple and easy as kernel calls are written in C-like language.

- **CUDA Limitations**
    - CUDA is restricted to NVIDIA GPUs only.
    - CUDA runs its host code through a C++ compiler so it doesn't support the full C standard.

- **CUDA and Nvidia GPUs** have been adopted in many areas that need high floating-point computing performance, such as:
    - Computational finance
    - Climate, weather, and ocean modelling
    - Data science and analytics
    - Deep learning and machine learning
    - Defence and intelligence

The objective of this project was to develop efficient GPU-accelerated numerical solvers for linear systems of equations using CUDA programming. The main focus was to implement and compare different methods such as Jacobi's iterative method, Red-Black Gauss-Seidel, and Conjugate Gradient method for solving 2D steady-state heat equations. Additionally, the project aimed to extend Jacobi's iterative method for 3D problems and develop a GPU code for it. The primary objective was to demonstrate the superiority of GPU computation over the serial CPU code by analysing the computation time for both methods. Overall, the project aimed to showcase the potential of GPU acceleration in numerical computing and provide insights into its practical applications.

Chapter 3
Keywords used in CUDA

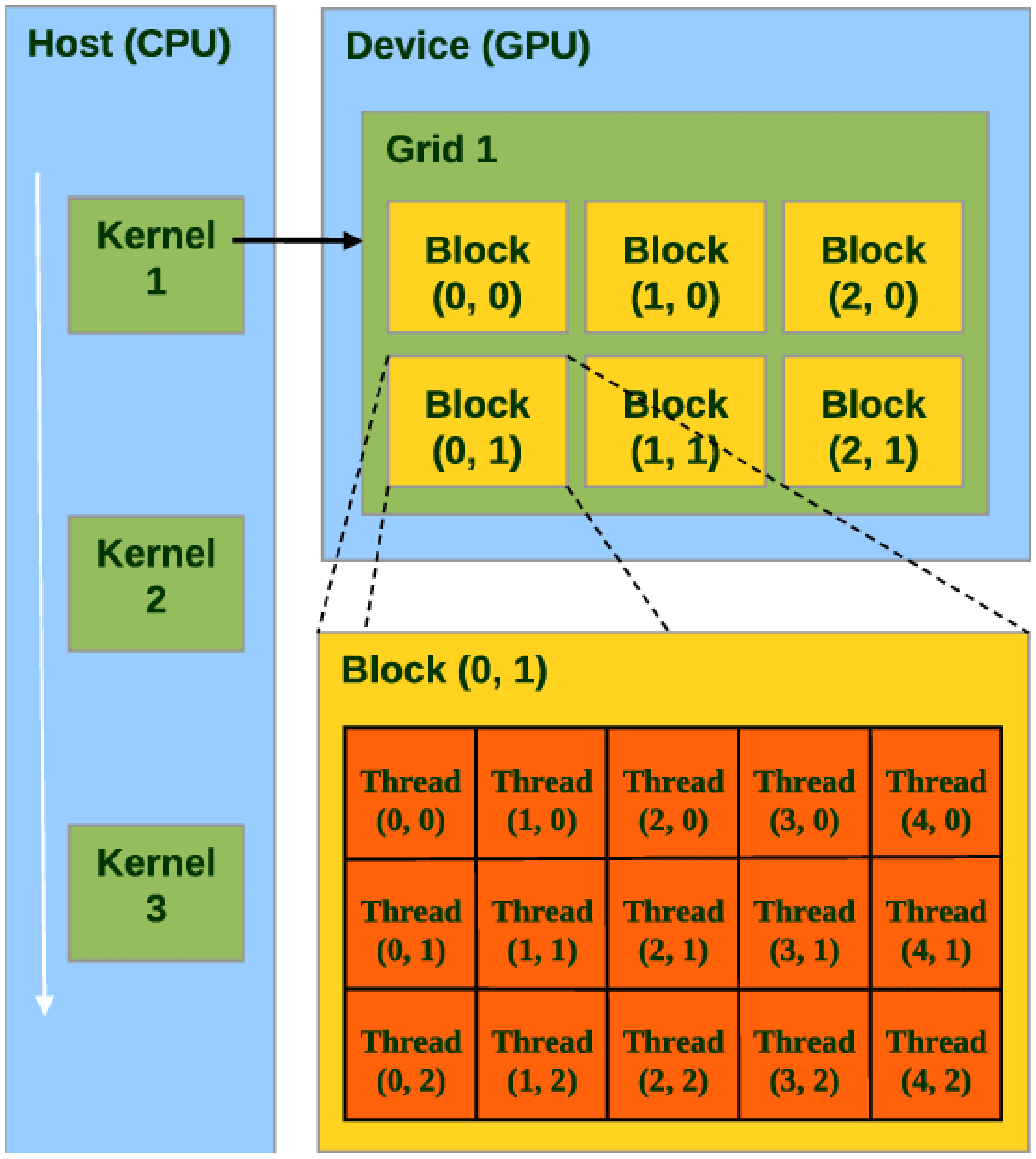


Figure 3.1: Thread, Block, Grid

Reference: Rafael S. Parpinelli, Marlon H. Scalabrin, *Int. J. Computational Science and Engineering*, Vol. 9, Nos. 1/2, 2014

- **Thread** - The thread is an abstract entity that represents the execution of the kernel.
- **Block** - A group of threads is called a CUDA block.
- **Grid -** CUDA blocks are grouped into a grid. A kernel is executed as a grid of blocks of threads.
- **CUDA kernel** - CUDA kernels are subdivided into blocks. A group of threads is called a CUDA block. CUDA blocks are grouped into a grid.The kernel is **a function executed on the GPU**. Every CUDA kernel starts with a __global__ declaration specifier. Programmers provide a unique global ID to each thread by using built-in variables
- **threadIdx** - This variable contains the thread index within the block.
- **blockIdx** - This variable contains the block index within the grid.
- **blockDim** - This variable and contains the dimensions of the block.
- **cudaMalloc** - function that can be called from the host or the device to allocate memory on the device.
- **cudaMemcpy** - Blocks the CPU until the copy is complete. Copy begins when all preceding CUDA calls have completed.
- **Dim 3** -is an integer vector type that can be used in CUDA code. Its most common application is to pass the grid and block dimensions in a kernel invocation. It can also be used in any user code for holding values of 3 dimensions.

Specifications of Google Collab's GPU

- name = 'Tesla K80'
- maxThreadsPerBlock = 1024
- maxBlockDimX = 1024
- maxBlockDimY = 1024
- maxBlockDimZ = 64
- maxGridDimX = 2147483647
- maxGridDimY = 65535
- maxGridDimZ = 65535

Chapter 4
Methodology

4.1 GPU based Jacobi iterative linear solver for DDC Matrix

We learned about Jacobi's iterative method, learned the basics of parallel computing, then we started Cuda programming by learning the basic syntax of Cuda programming. We first wrote code for host code using C language and for testing, we generated sample test cases that are diagonally dominant coefficient matrices up to size 2048 by writing code in C++. We accelerated host code using Cuda programming i.e writing Cuda C code on google colab followed by analysing time for different test cases generated.

4.2 GPU based Jacobi iterative solver for the solution of the heat equation in 2D

1) Analytical Method: The mathematical equation can be solved using techniques like the method of separation of variables.

$$\frac{T(x,y) - T_{low}}{T_{high} - T_{low}} = \sum_{n=1}^{\infty} \frac{2}{\pi} \frac{(-1)^{n+1} + 1}{n} \sin\left(\frac{n\pi}{L} x\right) \frac{sinh(\frac{n\pi}{L} y)}{sinh(\frac{n\pi}{L} W)} \qquad \text{(i)}$$

2) Numerical Method: Finite difference or finite volume schemes, usually will be solved using computers.

4.2.1 Procedure for Numerical analysis :

1) **Formulation of Governing Differential Equation for 2D steady state heat conduction.**
   a) Energy Balance(Law of conservation of energy)
   b) This results into laplace equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \qquad \text{(ii)}$$

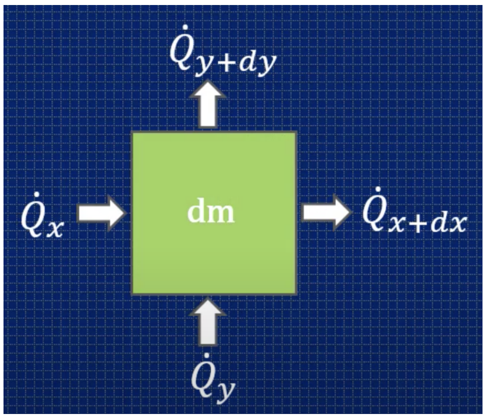


Figure 4.1: Energy Balance for small element dm

2) Discretization of domain into subdomain

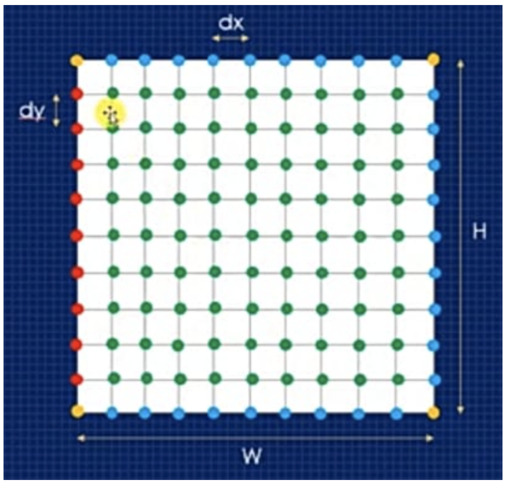


Figure 4.2: Discretization of domain into subdomain

$$dx = \frac{W}{Nx-1}$$

$$dy = \frac{H}{Ny-1}$$

3) Converting differential equations into algebraic equations.

Taylor Series Expansion

$$f(x + dx) = f(x) + f'(x)\frac{dx}{1!} + f''(x)\frac{dx^2}{2!} \qquad (1)$$

$$f(x - dx) = f(x) - f'(x)\frac{dx}{1!} + f''(x)\frac{dx^2}{2!} \qquad (2)$$

Adding Equation (1) and Equation (2)

$$f(x + dx) + f(x - dx) = 2f(x) + f''(x)dx \quad (3)$$

Rearranging the equation we get

$$f''(x) = \frac{(f(x+dx) + f(x-dx) - 2f(x))}{dx^2} \quad (4)$$

Now $f''(x)$ can be written as $\frac{d^2T}{dx^2}$

$$\frac{d^2T}{dx^2} = \frac{(T(x+dx,y) + T(x-dx,y) - 2T(x,y))}{dx^2} \quad (5)$$

Similarly for y:

$$\frac{d^2T}{dy^2} = \frac{(T(x,y+dy) + T(x,y-dy) - 2T(x,y))}{dy^2} \quad (6)$$

$$\frac{d^2T}{dx^2} + \frac{d^2T}{dy^2} = 0 \quad \text{(from above discussion at point (i))} \quad (7)$$

Substituting Equation (5) and (6) in Equation (7)

$$\frac{(T(x+dx,y) + T(x-dx,y) - 2T(x,y))}{dx^2} + \frac{(T(x,y+dy) + T(x,y-dy) - 2T(x,y))}{dy^2} = 0 \quad (8)$$

Rearranging the equation(8) we get:

$$T(x,y) = \frac{[(T(x+dx,y) + T(x-dx,y))/dx^2 + (T(x,y+dy) + T(x,y-dy))/dy^2]}{[2/dx^2 + 2/dy^2]} \quad (9)$$

Now Assume dx=dy, we get:

$$T(x,y) = \frac{(T(x+dx,y) + T(x-dx,y) + T(x,y+dy) + T(x,y-dy))}{4} \quad (10)$$

Final equation comes as

$$T(i,j) = \frac{(T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1))}{4} \quad (11)$$

4) **Solution of the algebraic equations (11) using code.**
5) **Plotting the result.**

4.2.2 Converting 2D array to 1D

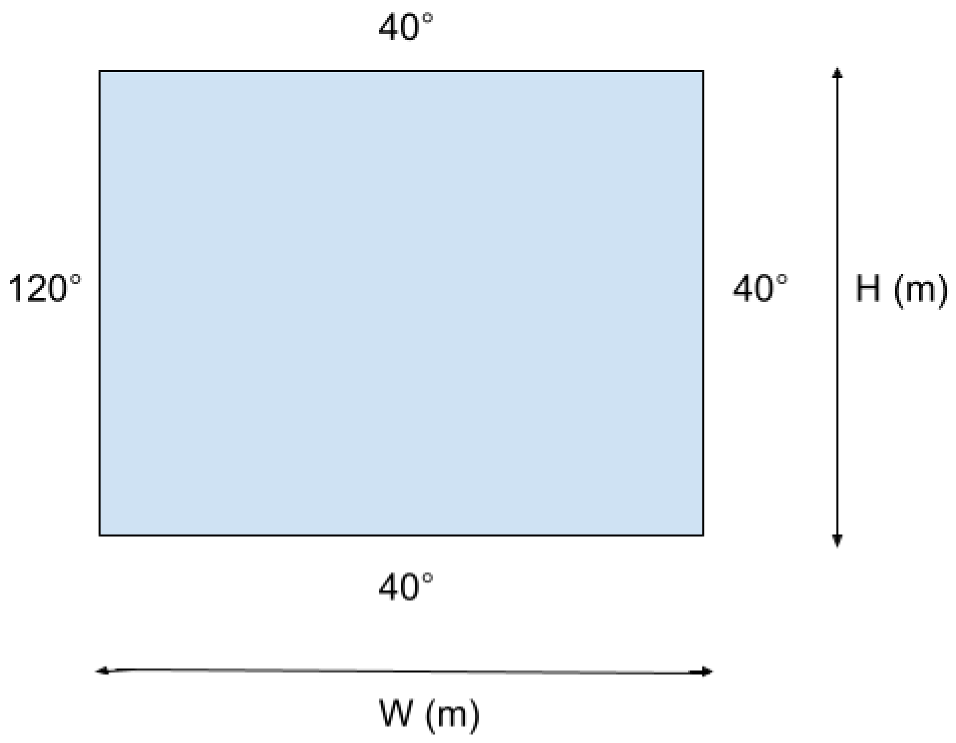


Figure 4.3: Rectangular Plate

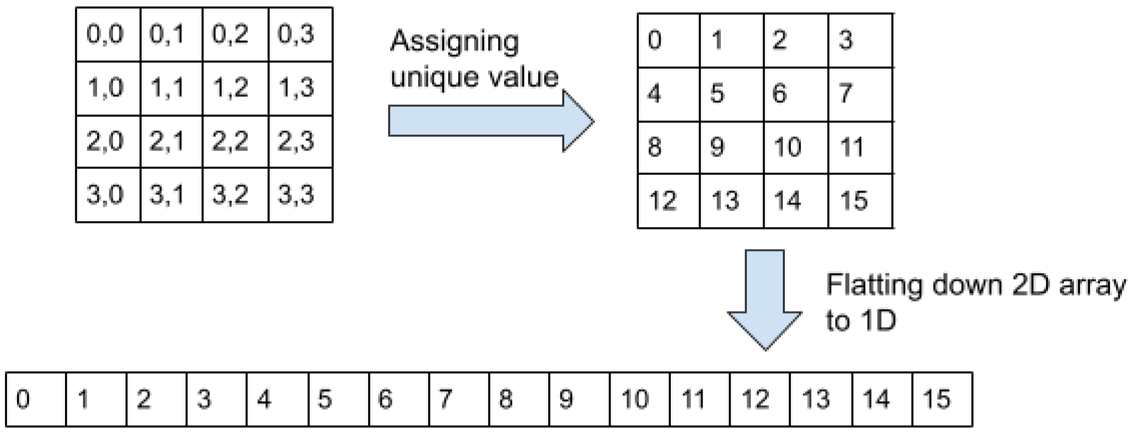
Height H, Width W and Temperatures at boundary

N = number of points	
Nx = N;	No. of grids in x direction
Ny = N;	No. of grids in y direction
dx = W/(Nx);	Width of element in (m)
dy = H/(Ny);	Height of element in (m)

Four corner points are initialised by taking average of two adjacent temperature. All points inside mesh are initialised with 25 degree celsius

Example 4 x 4

To solve 2D matrix in cuda function we need to convert 2D Matrix to 1D Array below figure shows how we first calculated unique index for each block and then arranged them to 1D array.



1 x 16

Figure 4.4: Converting 2D Matrix into 1D Array

As per result temperature at any point other than boundary points is average of four neighbouring points

$$T(i, j) = \frac{T(i+1,j) + T(i-1,j) + T(i,j+1) + T(i,j-1)}{4}$$

(where i, j are the index of the 2D grid)

Since we have 1D array:-

$$T(i) = \frac{T(i+1) + T(i-1) + T(i+n) + T(i-n)}{4}$$

(n - number of rows and i - index of the 1D array)

For boundary points Point at i will be multiple of n and i+1 will be multiple of n too.

4.3 Unique Index calculation:

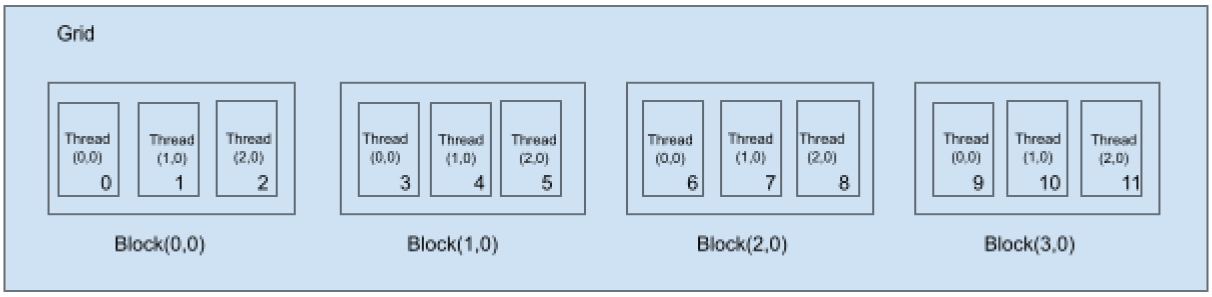


Figure 4.5: 1D grid of 1D blocks representation with thread Idx

threadId = (blockIdx.x * blockDim.x) + threadIdx.x

Let's check the equation for Thread (2,0) in Block (1,0).

Thread ID = (1 * 3) + 2 =3+2

Chapter 5
Jacobi's Method

Jacobi’s method in numerical linear algebra is an iterative method to compute the solution of a strictly diagonally dominant system of linear equations. The method is named after Carl Gustav Jacob Jacobi. The method is a shorter version of the Jacobi transformation method of matrix diagonalization.
Being diagonally dominant is a sufficient condition for Jacobi’s method to work. The method can work in other cases, but there is no guarantee that it will.

Diagonally Dominant Coefficient (DDC) Matrix

A matrix is said to be a diagonally dominant coefficient matrix if in each row the magnitude of diagonal coefficient is greater or equal to the sum of magnitude of all other entries in the row.
Matrix *A* is diagonally dominant if

$$\left|a_{ii}\right| \geq \sum_{j\neq i}\left|a_{ij}\right| \text{ for all i}$$

where a_{ij} denotes the entry in the *i*th row and *j*th column.

Example

$$A = \begin{bmatrix} 3 & -2 & 1 \\ 1 & -3 & 2 \\ -1 & 2 & 4 \end{bmatrix}$$

is diagonally dominant because

$$\left|a_{11}\right| \geq \left|a_{12}\right| + \left|a_{13}\right| \text{ since } |+3| \geq |-2| + |+1|$$

$$\left|a_{22}\right| \geq \left|a_{21}\right| + \left|a_{23}\right| \text{ since } |-3| \geq |+1| + |+2|$$

$$|a_{33}| \geq |a_{31}| + |a_{32}| \text{ since } |+4| \geq |-1| + |+2|$$

Two assumptions made on Jacobi Method:

1. The system given by-

$$a_{11}x_1 + a_{12}x_2 + \dots a_{1n}xn = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \dots a_{zn}x_n = b_2$$
$$a_{n1}x_1 + a_{nz}x_2 + \dots a_{nn}x_n = b_n$$

has a unique solution.

2. The coefficient matrix A has no zeros on its main diagonal, namely, a_{11}, a_{22},..., a_{nn} are nonzeros.

Main Idea of Jacobi

To begin, solve the 1st equation for x_1, the 2nd equation for x_2 and so on to obtain the rewritten equations:

$$x_1=1/a_{11}(b_1- a_{12}x_2- a_{13}x_3 - \dots a_{1n}x_n)$$
$$x_2 =1/a_{22}(b_2- a_{21}x1_1- a_{23}x_3 - \dots a_{2n}x_n)$$
$$x_n=1/a_{nn}(b_n- a_{n1}x_1 - a_{n2}x_2 - \dots a_{n,n-1}x_{n-1})$$

Then make an initial guess of the solution $x^{(0)} = (x_1^{(0)},x_2^{(0)}, x_3^{(0)},\dots X_n^{(0)})$. Substitute these values into the right hand side of the rewritten equations to obtain the first approximation, $(x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \dots x_n^{(1)})$.
This accomplishes one Iteration.

In the same way, the second approximation $(x_1^{(2)}, x_2^{(2)} ,x_3^{(2)},\dots x_n^{(2)})$ is computed by substituting the first approximation's x- values into the right hand side of the rewritten equations.
By repeated iterations, we form a sequence of approximations
$x^{(k)} =(x_1^{(k)}, x_2^{(k)} ,x_3^{(k)},\dots x_n^{(k)})^t$, $k = 1,2,3,\dots$

<u>**The Jacobi Method.**</u> For each $k \geq 1$, generate the components $x_i^{(k)}$ of $x^{(k)}$ from $x^{(k-1)}$ by

$$x_i^{(K)} =1/a_{ii} \left[\sum_{j=1,j\neq i}^{n} (-a_{ij}\, x_j^{(k-1)}) +b_i \right], \qquad \text{for } i=1,2,, \dots n.$$

Chapter 6
Jacobi Iterative Method Implementation and Results

6.1 GPU based Jacobi iterative linear solver for DDC Matrix

6.1.1 Algorithm for Host

The given algorithm is an implementation of the Jacobi iteration method for solving a system of linear equations on the host (CPU). The algorithm takes the following parameters:

- n: integer value representing the number of equations in the system
- a: pointer to the coefficient matrix of size n x n
- c: pointer to an array of size n to store intermediate values
- b: pointer to the right-hand side vector of size n
- x: pointer to the solution vector of size n
- l: integer value representing the number of iterations to perform

The algorithm works as follows:

- Initialize variables i and j.
- Run a loop for l iterations.
- For each iteration, run a loop for n equations.
- For each equation, set c[i] to the value of b[i].
- For the i-th equation, run a loop for all n coefficients.
- For each coefficient a[i*n + j], if i is not equal to j, subtract the product of a[i*n + j] and x[j] from c[i].
- After the inner loop, set x[i] to the value of c[i] divided by a[i*n + i].
- Return from the function.

For complete code refer to appendix 1.1

6.1.2 Algorithm for Device

The given algorithm is an implementation of the Jacobi iteration method for solving a system of linear equations on a CUDA device. The algorithm is implemented as a CUDA kernel function and takes the following parameters:

- n: integer value representing the number of equations in the system
- a_d: pointer to the coefficient matrix of size n x n stored in device memory
- c_d: pointer to an array of size n to store intermediate values in device memory
- b_d: pointer to the right-hand side vector of size n stored in device memory
- x_d: pointer to the solution vector of size n stored in device memory
- l: integer value representing the number of iterations to perform

The algorithm works as follows:

1. Calculate a unique index idx for each thread based on the block and thread indices.
2. Run a loop for l iterations.
3. For each iteration, check if idx is less than n to avoid errors.
4. If idx is less than n, initialise a variable sum to 0 and calculate the index idx_ai corresponding to the row of a_d that the thread is responsible for.
5. Run a loop for all n coefficients.
6. For each coefficient a_d[idx_ai + j], if idx is not equal to j, subtract the product of a_d[idx_ai + j] and x_d[j] from sum.
7. After the inner loop, use __syncthreads() to synchronise all threads in the block and then set x_d[idx] to the value of (b_d[idx] - sum) / a_d[idx_ai + idx].
8. Return from the kernel function.

For complete code refer to appendix 1.2

6.2 Result for GPU based Jacobi iterative linear solver for DDC Matrix

6.2.1 For 0.01 Million Iteration

Table 6.1 Data for 0.01 Million Iterations

Time taken by host code and device code when it was executed for 0.01 million iterations

Number of variables (n)	Time taken by Host function	Time taken by only Device function	Time taken by all processes in Device code
2	0.000236	0.000157	0.025611
3	0.000539	0.00013	0.030999
4	0.000771	0.000114	0.024962
8	0.00309	0.000165	0.036928
16	0.010307	0.00013	0.083766
32	0.047763	0.000111	0.257319
64	0.187935	0.000106	0.470635
128	0.71794	0.000143	0.813001
256	2.800634	0.000125	1.881661
512	10.999227	0.000124	7.060506
1024	42.455662	0.000115	14.156701
2048	164.389114	0.000107	28.675806

6.2.2 For 0.1 Million Iteration

Table 6.2 Data for 0.1 Million Iterations

Time taken by host code and device code when it was executed for 0.1 million iterations.

Number of variables (n)	**Time taken by host function**	**Time taken by only device function**	**Time taken by all processes in device code**
2	0.002227	0.000115	0.248982
3	0.004516	0.000101	0.297242
4	0.010064	0.00011	0.240644
8	0.028182	0.000098	0.339835
16	0.110166	0.000109	0.617595
32	0.46278	0.000168	1.784157
64	1.813967	0.000116	3.376125
128	6.855566	0.000109	6.380914
256	27.103855	0.000118	17.362448
512	107.230431	0.00013	69.333061
1024	419.23291	0.000139	140.342105
2048	1661.432373	0.000145	285.538780

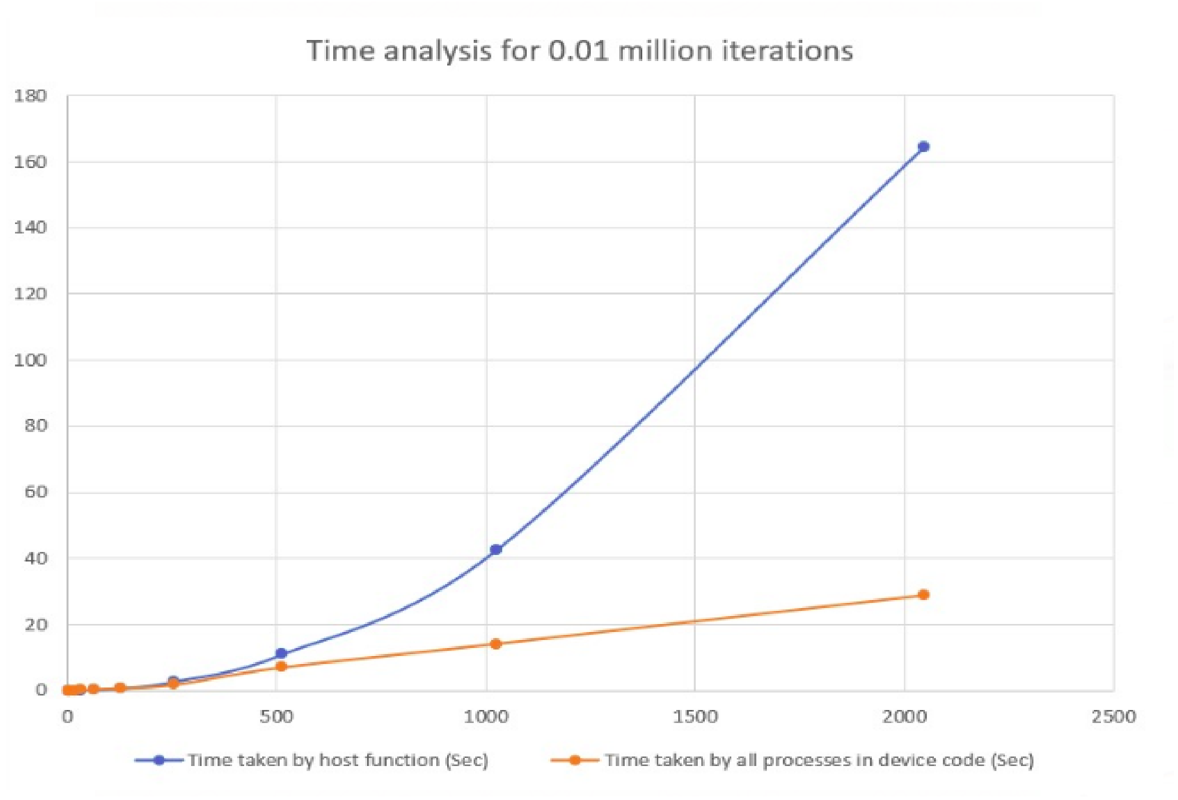


Figure 6.1:Graphical representation for table 6.1

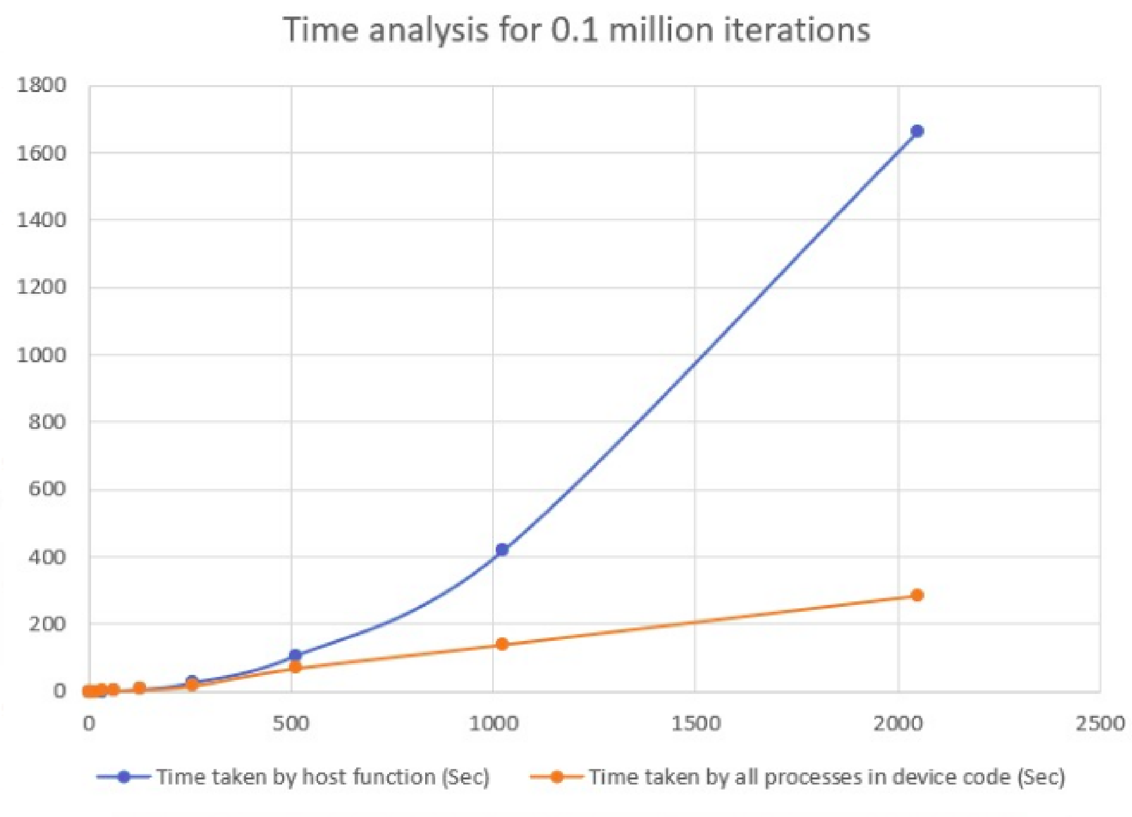


Figure 6.2 : Graphical representation for table 6.2

The comparison between CPU and GPU computation in this project revealed an interesting trend. At first, CPU-based computations seemed to outperform GPU-based computations due to their ability to use fewer resources. However, as the size of the problem increased, the GPU showed a significant increase in performance due to the larger number of resources available for computation.

This trend was evident from Figure 6.1 and Figure 6.2, which show the comparison between CPU and GPU computation times for different problem sizes. As the number of points in the problem increased, the computation time for the CPU increased rapidly, while the computation time for the GPU remained relatively constant.

This trend can be attributed to the parallel processing capabilities of the GPU. GPUs have a significantly larger number of processing cores than CPUs, which allows them to perform a large number of computations simultaneously. This makes GPUs particularly effective for handling large-scale, computationally intensive problems.

Overall, the findings of this project demonstrate the importance of selecting the appropriate computing platform based on the size and complexity of the problem at hand. While CPU-based computations may be more efficient for smaller problems, GPU-based computations are more effective for handling larger, more complex problems.

6.3 GPU based Jacobi iterative solver for the solution of the heat equation in 2D

6.3.1 Algorithm for Host

The given code implements the Jacobi iterative method for solving a two-dimensional heat conduction problem. The function jacobi_host takes the following parameters:

- T: pointer to the temperature array of size n x n
- Told: pointer to the old temperature array of size n x n
- l: integer value representing the number of iterations to perform

- n: integer value representing the number of grid points in each direction

The algorithm works as follows:

1. Run a loop for l iterations.
2. Inside the loop, copy the new temperature array T to the old temperature array Told by running a loop over all elements of T and copying their values to the corresponding elements of Told.
3. Run a loop over all elements of T, except for the boundary points.
4. For each element T[i], calculate the new temperature value as the average of the four neighbouring points (Told[i+1], Told[i-1], Told[i+n], Told[i-n]), and store the result in T[i].
5. Repeat the inner loop for all elements of T.
6. Return from the function.
7. Note that the Jacobi iterative method is an iterative algorithm, so running it for multiple iterations improves the accuracy of the solution. In each iteration, the new temperature values are calculated based on the old values, and the old values are updated with the new values at the end of the iteration. The process is repeated for multiple iterations until the solution converges to a stable value.

For complete code refer to appendix 1.3

6.3.2 Algorithm for device

The given code is a CUDA C implementation of the Jacobi iterative method for solving a heat diffusion problem. The algorithm can be summarised as follows:

1. Initialise the grid of temperatures T, and a temporary grid Told with the same values.
2. Copy the values of T to the device memory using cudaMemcpy.
3. Launch the device kernel jacobi_device with the specified grid and block sizes.
4. Synchronise the device using cudaDeviceSynchronize to ensure that all threads have completed their calculations before proceeding.

5. Copy the resulting values of T from the device memory to the host memory using cudaMemcpy.
6. Repeat steps 2 to 5 for the desired number of iterations.

The jacobi_device kernel takes as input the grid size n, the input temperature grid T_d, the temporary grid Told_d, and the number of iterations l. Each thread is assigned a unique index idx based on the block size and grid size. The kernel then loops through the specified number of iterations, updating the values of T_d based on the values of Told_d using the Jacobi iterative method.

The Jacobi iterative method involves computing the average of the temperatures at the four neighbouring grid points (up, down, left, and right) and using that average to update the temperature value at the current grid point. This computation is performed for all grid points except for the boundary points, which are not updated.

After the specified number of iterations, the final values of T_d are copied back to the host memory using cudaMemcpy.

For complete code refer to appendix 1.4

6.4 Result for GPU based Jacobi iterative solver for the solution of the heat equation in 2D

6.4.1 For 0.01 Million Iteration

Table 8.3 Data for 0.01 Million Iterations

Time taken by host code and device code when it was executed for 0.01 million iterations

Time for 0.01million Iterations		
n	Time for host	Time for device
100	1.329332	0.03925
200	5.367696	0.11465
300	12.29463	0.255854
400	22.257401	0.398463
500	33.194008	0.54755
1000	132.815933	1.714681
2000	557.023499	6.625637
4000	**	7.174869
5000	**	36.951527
7500	**	81.727203
10000	**	149.152756

6.4.2 For 0.1 Million Iteration

Table 6.4 Data for 0.1 Million Iterations

Time taken by host code and device code when it was executed for 0.1 million iterations.

Time for 0.1 million Iterations		
n	**Time for host**	**Time for device**
100	13.159575	0.376229
200	52.970924	0.937724
300	118.110786	1.691003
400	209.986694	2.546163
500	345.706604	4.062819
1000	1323.091431	14.959066
2000	5658.568848	60.203522
4000	**	242.647736
5000	**	375.67984
7500	**	854.272461
10000	**	1539.091064

**** Time taken by host was too long**

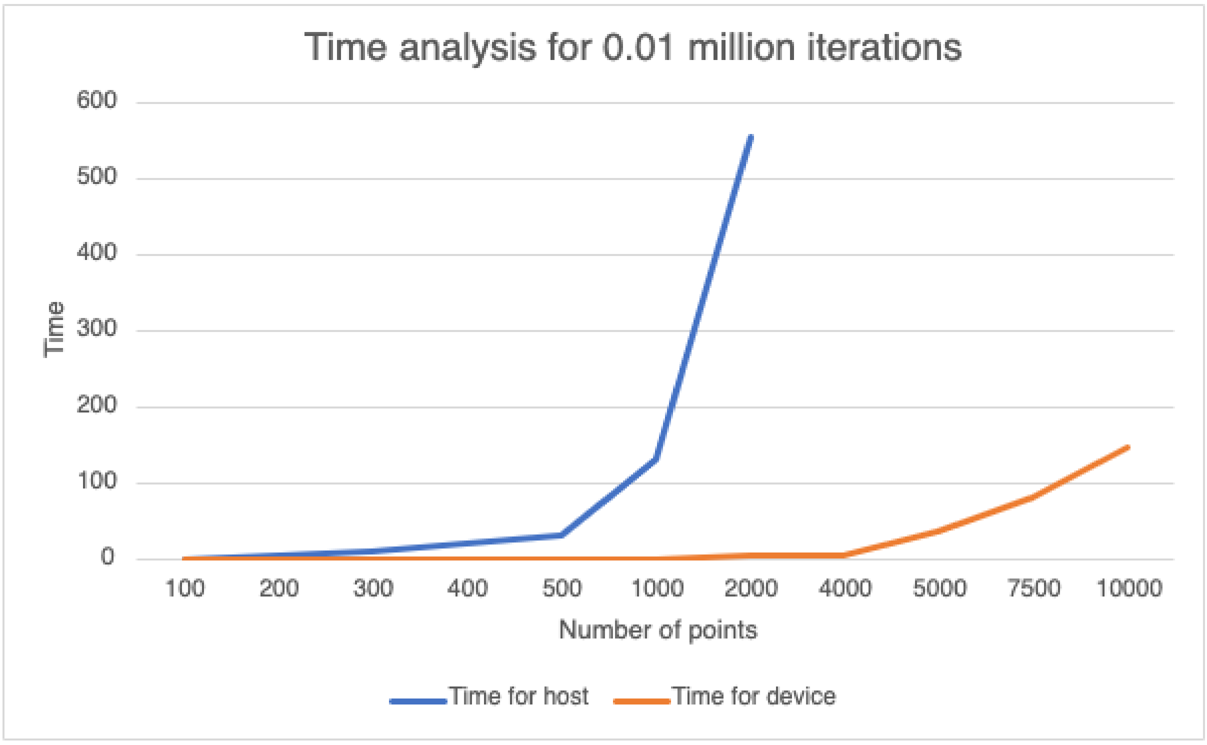


Figure 6.3: Graphical representation for table 6.3

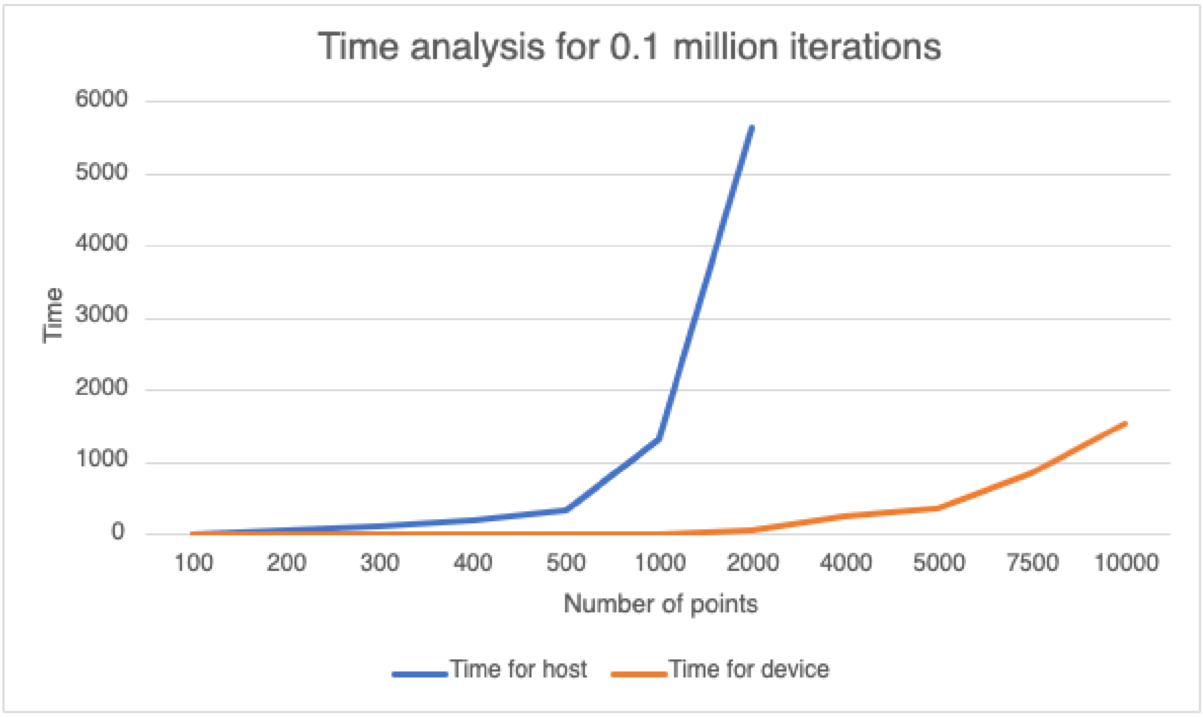


Figure 6.4:Graphical representation for table 6.4

As discussed above in section 6.2 we can observe a similar trend in figure 6.3 and 6.4 for heat equation in 2D.

6.5 Error analysis for the solution of the heat equation in 2D

Residual error is calculated between Told and T where Told is the value at the second last iteration and T is the value at the last iteration.

T is the array containing the current values of temperature whereas Told is an array of the same size but a copy of T. Mean square root is calculated using the above function where T and Told are the arguments passed.

```
double calculate_error(double* T,double *Told,int n) //function
for calculating error
{
    double error = 0;
    for(int i = 0;i<n*n;i++)
    {
            float x = abs(T[i] - Told[i]);
            error += x*x;
    }
    return sqrt(error)/(n*n);
}
```

Table 6.5 Error in host function for 0.01 million and 0.1 million iterations

Host Error		
n	**0.01 million iterations**	**0.1 million iterations**
100	0.009195	0
200	0.205495	0.000003
300	0.311205	0.001876
400	0.377429	0.015812
500	0.436416	0.038732
1000	0.657504	0.102178
2000	0.957099	0.15964

Table 6.6 Error in Device function for 0.01 million and 0.1 million iterations

Device Error		
n	**0.01 million iterations**	**0.1 million iterations**
100	0.006564	0
200	0.62735	0.11703
300	0.570809	0.261872
400	0.384035	0.310214
500	1.393137	0.368785
1000	1.507518	0.21539
2000	1.271833	0.319294
4000	10.29012	3.58269
5000	36.220596	10.016305
7500	25.86822	0.361833
10000	27.075108	1.093284

From table 6.5 and 6.6 we can observe that as the number of iterations and points increases error in both host as well as device increases.

6.6 Contour plot for temperature distribution

- Contour plot was made using matplotlib library in python using the points found in cuda code function where a meshgrid of size 100 x 100 was used for plotting temperature distribution for 10000 points.
- As we can see in fig 6.5 that temperature at the left wall is around 120 degrees and decreases as we move towards the right.
- The contour is symmetric about the horizontal axis from the centre which depicts temperature is symmetric about the horizontal axis.

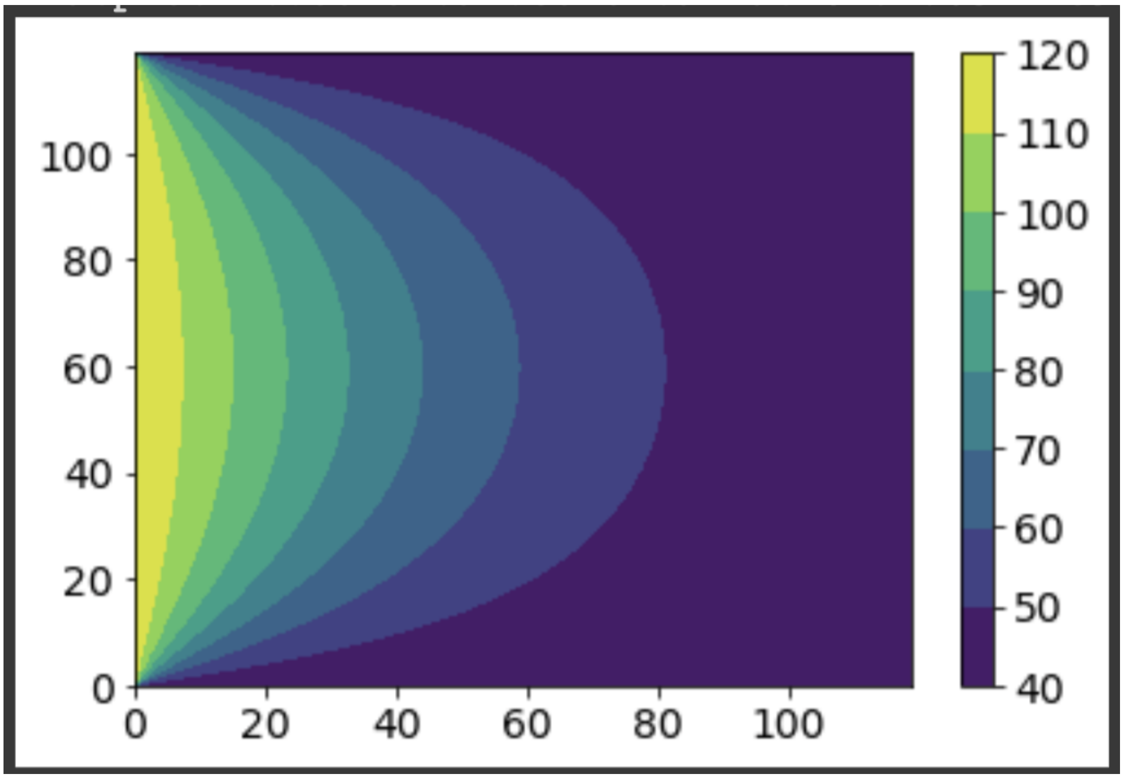


Figure 6.5 Contour plot for Temperature distribution for 100 points

Chapter 7
Red Black gauss seidel

The Red Black gauss seidel is applied optimizations technique and it exploits the computational capabilities of GPUs under the CUDA environment in solving the linear equations. Additionally, we also developed CPU code as a performance reference.
Significant performance improvements were achieved by using optimization methods which proved to have substantial speedup in performance.

The red/black method belongs to the iterative methods family like the Jacobi. The red black coloring allows easy parallelization. The calculation of all elements in the same colour can be handled independently of the others since there is no data dependence between them(figure 1). Therefore, this problem is an ideal one for parallelization.

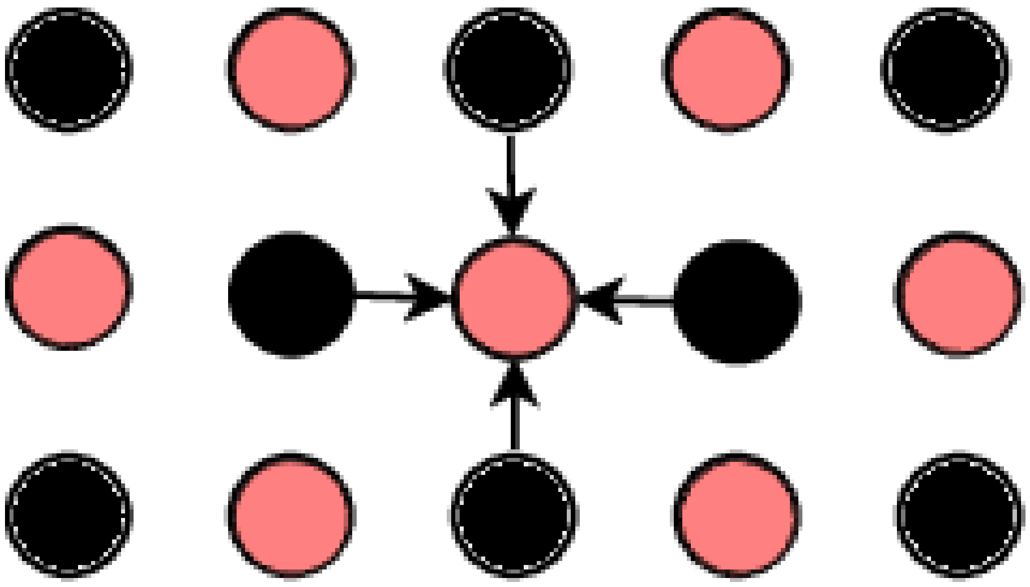


Fig. 7.1. *Red point values are affected only by the neighbouring black points.*

Reference:Accelerating the red/black SOR method using GPUs with CUDA .Elias Konstantinidis and Yiannis Cotronis

Moving on to the iterative process, values are calculated in two phases.

At first, red elements get updated and then black elements follow. Every point is updated according to the neighbour point values (fig 7.1), as the equations (1) and (2) indicate. The boundary values are assumed constant and predetermined.

This calculation is executed iteratively until the matrix values converge. Checking for convergence is performed by calculating the sum of squared differences between the current values and the previous iteration values.

7.1 Understanding the Algorithm

This method uses the same idea as discussed in the last section for jacobi iterative method the only difference one will notice is that Red/Black method is that instead of getting all the values for the previously calculated matrix we compute half the points (suppose marked as red) using the previously calculated matrix and for computing the other half (suppose marked as black) we take the newly calculated value of red points in this way it converge faster.

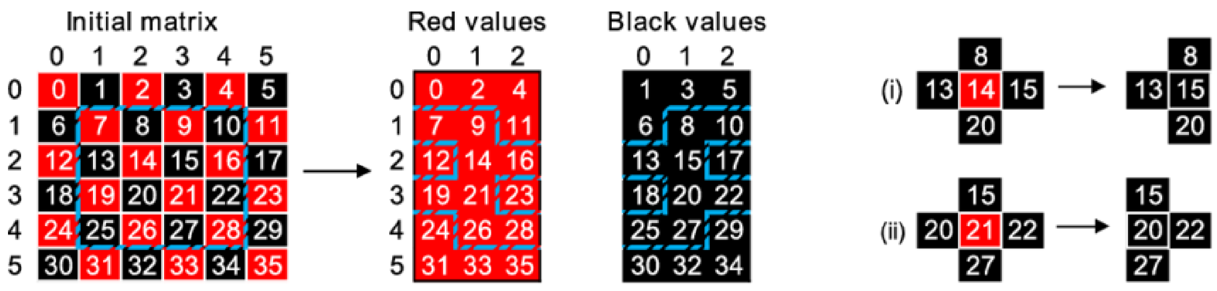


Fig.7.2. *Red and black points division of the matrix*

Reference: Accelerating the red/black SOR method usingGPUs with CUDA
By Elias Konstantinidis and Yiannis Cotronis

Function Discription:

1) Calculate Error
2) Red_black_host

7.1.1 Host Code:

Input:

- A matrix T representing the initial temperature distribution
- A matrix Told representing the temperature distribution of the previous iteration
- An integer l representing the number of levels of refinement

- An integer n representing the number of nodes along each axis

Output:

- The temperature distribution matrix T after convergence
- The number of iterations required for convergence

```
1. Initialize error to a large value, set No_of_iterations to 0.
2. While error is more than 1e-3:
    a. Copy the current temperature distribution T to Told.
    b. For each red point i in the domain (excluding boundaries):
        i. If i is not on the left or right boundary:
            1. Calculate the row and column of i using integer division and modulo.
            2. If the sum of the row and column of i is odd:
                a. Update T[i] using the adjacent and independent black points: T[i] = (Told[i+1] + Told[i-1] + Told[i+n] + Told[i-n])/4
    c. For each black point i in the domain (excluding boundaries):
        i. If i is not on the left or right boundary:
            1. Calculate the row and column of i using integer division and modulo.
            2. If the sum of the row and column of i is even:
                a. Update T[i] using the adjacent and independent red points: T[i] = (T[i+1] + T[i-1] + T[i+n] + T[i-n])/4
    d. Calculate the error between T and Told using the calculate_error() function: error = calculate_error(T, Told, n)
    e. Increment No_of_iterations by 1.
3. Return T and No_of_iterations.
```

For complete code refer to appendix 1.5

7.1.2 Device Code:

The following is an algorithm for the Jacobi method CUDA kernel for solving a heat transfer problem on a 2D grid:

Inputs:

- n: the size of the grid (assuming a square grid)
- T_d: a pointer to a device array storing the initial temperature values
- Told_d: a pointer to a device array storing the previous iteration's temperature values
- l: the number of iterations to perform

Outputs: None (the updated temperature array is stored in T_d)

1. Calculate the unique index for the current thread using blockIdx.x, blockDim.x, and threadIdx.x.
2. Set the error and number of iterations to 5 and 0, respectively.
3. Enter a loop that runs l iterations.
4. If the current thread's index is within the range of the grid (less than n*n), update Told_d to match the current T_d values.
5. Synchronise all threads using __syncthreads().
6. If the current thread's index is within the interior of the grid (away from the boundary), and the sum of the row and column indices is even, update T_d using the Jacobi formula with neighbouring grid points.
7. Synchronise all threads using __syncthreads().
8. If the current thread's index is within the interior of the grid (away from the boundary), and the sum of the row and column indices is odd, update T_d using the Jacobi formula with neighbouring grid points.
9. Synchronise all threads using __syncthreads().
10. Calculate the absolute difference between the current T_d and Told_d values for the current thread, and add it to the temp variable.
11. Update the error variable to be the value of temp.
12. Exit the loop.

For complete code refer to appendix 1.6

7.2 Results for Red Black iterative solver for the solution of the heat equation in 2D For 0.01 Million Iteration

Table 7.1 Data for 0.01 Million Iterations

Time taken by host code and device code when it was executed for 0.01 million iterations.

Time for 0.01 million Iterations		
n	**Time for host**	**Time for device**
100	2.755923	0.019625
200	11.814506	0.052739
300	25.302134	0.117692
400	45.283028	0.183291
500	70.362274	0.251873
1000	285.768768	0.788753
2000	1142.945068	3.047793
4000	**	3.300439
5000	**	18.51527
7500	**	40.827203
10000	**	75.635427

**** Time taken by host was too long**

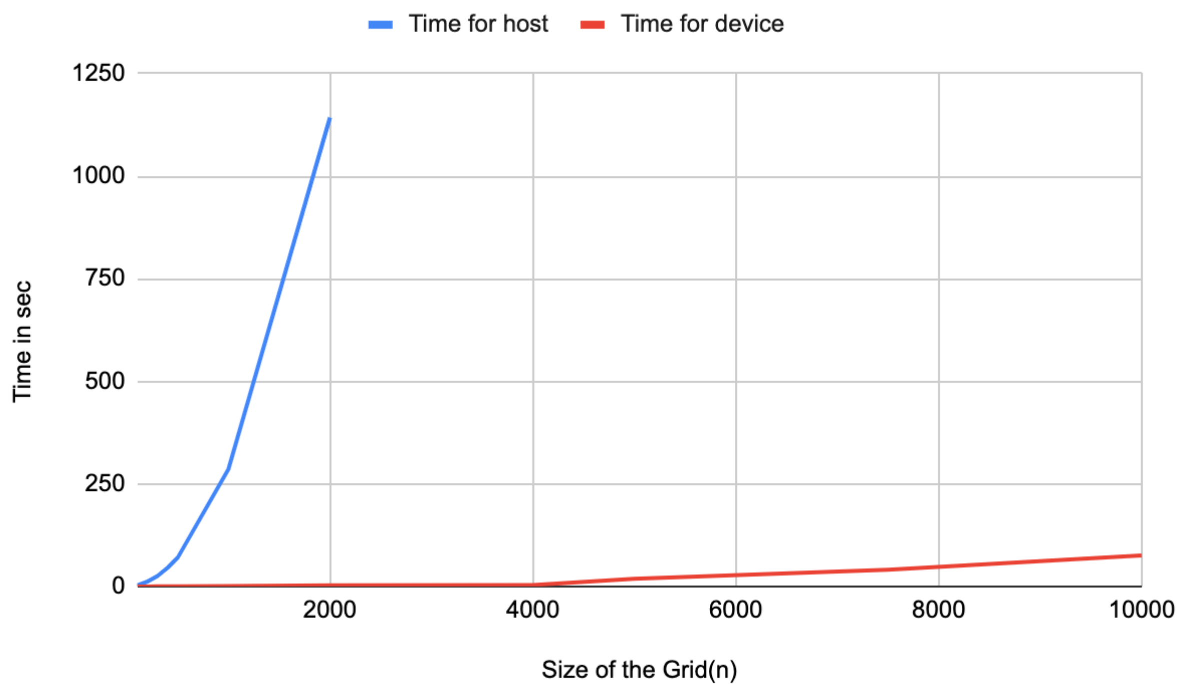


Figure 7.1: Graphical representation for table 7.1

As discussed above in section 6.2 we can observe a similar trend in figure 7.1 for red black iterative solver equation in 2D.The Red-Black Gauss-Seidel method is known to perform better than the Jacobi's iterative method for certain types of linear systems of equations. There are a few reasons why this is the case.Firstly, the Red-Black Gauss-Seidel method is a more advanced iterative method that takes into account the dependencies between different unknowns in the linear system. In contrast, Jacobi's iterative method only updates one unknown at a time and does not consider the dependencies between different unknowns. This can result in slower convergence and more iterations required to reach the desired solution.

Secondly, the Red-Black Gauss-Seidel method uses a more efficient updating scheme where unknowns are updated in a checkerboard pattern, alternating between red and black cells. This approach helps to reduce the number of iterations required to reach the desired solution, as each unknown is updated using more up-to-date information from the neighbouring unknowns.Overall, the Red-Black Gauss-Seidel method's superior performance over Jacobi's iterative method can be attributed to its more advanced updating scheme, consideration of the dependencies between different unknowns, and its ability to take advantage of parallel computing using GPUs.

Chapter 8
Conjugate gradient method

The conjugate gradient method is a procedure used in mathematics to solve certain systems of linear equations numerically, namely those whose matrix is symmetric and positive definite. Suppose we want to solve the following system of linear equations Ax = b where x is an unknown vector, b is a known vector, and A is a known, square, symmetric, positive-definite (or positive-indefinite) matrix.

Suppose we need to solve the below expression:

$$min\, f(x) := \frac{1}{2}x^T A.X \; - \; b^T X \qquad eq\ (1)$$

Lets define residual as $r_k = A.X_k - b$ at initial points X_k in next step we will find the search direction and determine the optimal step length to minimise f along $X_k + \alpha p\square$
By setting the first derivative of $f(x_k + \alpha p\square)$ w.r.t. α to be zero we get

$$\alpha_k = -\frac{\Delta f(X_k)^T p\square}{p\square^T A p\square} = \frac{r_k^T r_k}{p\square^T A p\square} \qquad eq\ (2)$$

We choose search direction p such that its element are conjugate w.r.t A that is

$$p_i^T A p_j = 0\; for\; all\; i \neq j \qquad eq\ (3)$$

Each direction $p\square$ is chosen to be a linear combination of the negative residual $-r\square$ and we write

$$p_k = -r_k + \beta_k p_{k-1} \qquad eq\ (4)$$

where the scalar $\beta\square$ is to be determined by the requirement that $p\square_{-1}$ and $p\square$ must be conjugate with respect to A.
We get

$$\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \qquad eq\ (5)$$

Knowing $\alpha\square$ and $p\square$, the next iterate could be calculated as usual using

$$x_{k+1} = x_k + \alpha_k p_k \qquad eq\ (6)$$

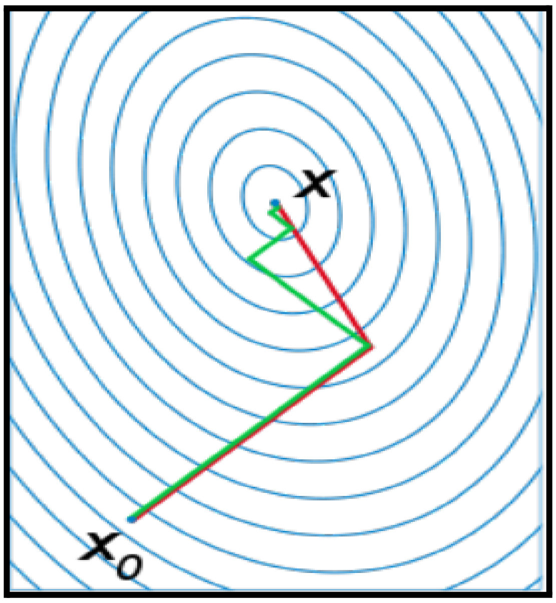


Figure 8.1 Comparison of convergence of gradient descent with optimal step size.

Reference:Conjugate gradient method parallel implementation by Hemza Keurti

A comparison of the convergence of gradient descent with optimal step size (in green) and conjugate vector (in red) for minimising a quadratic function associated with a given linear system. Conjugate gradient, assuming exact arithmetics, converges in at most n steps where n is the size of the matrix of the system

8.1 Algorithm for initialising the base values for the matrix

To solve the 2D heat equation problem, we first initialise three vectors: A, b, and x. The vector A is a 2D vector of size (NxN)x(NxN), and the other two vectors, b and x, are 1D arrays of double type and size (NxN).

We start by initialising the temperature values of the walls or boundaries of the 2D space. For instance, the left wall temperature is set to 120 degrees Celsius, and the other three walls are set to 40 degrees Celsius.

Next, we traverse all the rows and points in the vectors A, b, and x. If the point is on the boundary, we only have one coefficient for that equation, which is xii = 1. All other coefficients for this equation will be 0, and the value of b for this corresponding equation will be the temperature of that particular wall or boundary point.

If the point is not on the boundary, we have five coefficients, out of which four are the neighbouring elements and one is on that point. All the neighbouring coefficients will have a value of 0.25, and the value of xii will be -1. The corresponding value of b for this equation will be 0.

After initialising the coefficients and the values of the vector x, we initialise the vector x0 with a value of 25 degrees Celsius if it's not a boundary point. If it is a boundary point, we initialise it with the corresponding boundary temperature. For the four corner points, we initialise the value of x0 by taking the average of the two neighbouring boundary points. The corresponding value of b for these four corner points is set to the corresponding average temperature.

Once we have initialised all the required parameters, we can use the linear_cg function to solve the problem and obtain the final solution.

For complete code refer to appendix 1.8

8.2 Algorithm for conjugate gradient function

Input:

- A: a matrix represented as a vector of vectors
- b: a vector representing the right-hand side of the linear system Ax = b
- x_o: a vector representing the initial guess for x

Output:

- x_k: the approximate solution to the linear system Ax = b

1. Set the tolerance "tol" to 1e-5
2. Initialise x_k to be the initial guess x_o
3. Compute r_k = Ax_k - b
4. Set p_k = -r_k
5. Compute the norm of r_k and store it as r_k_norm
6. Initialise num_itr to be 0

```
7. While r_k_norm > tol:
    a. Compute Apk = A * p_k
    b. Compute rk_rk = dot_product(r_k, r_k)
    c. Compute alpha = rk_rk / dot_product(p_k, Apk)
    d. Update x_k = x_k + alpha * p_k
    e. Update r_k = r_k + alpha * Apk
    f. Compute beta = dot_product(r_k, r_k) / rk_rk
    g. Update p_k = -r_k + beta * p_k
    h. Increment num_itr
    i. Recompute r_k_norm = norm(r_k)
8. Return x_k as the approximate solution to Ax = b.
```

Function descriptions:

1. Subtract(a,b) :- subtract vector b from vector a and returns vector
2. Matrix_multi(A,b) :- matrix multiplication of two vectors if valid
3. negative(a) :- return a vector with negative of vector a
4. norm(a) :- returns a double value after calculating norm of a vector
5. dot_product(a,b) :- returns a double value after dot product of 2 vectors
6. addi(a,b) :- returns a vector after adding 2 given vectors
7. const_multi(a,b) :- returns a vector after multiplying a constant value a with vector b

For complete code refer to appendix 1.7

8.3 Limitations of the method

- For conjugate gradient method we are doing many redundant operations as our coefficient matrix is sparse and have many zeroes so we need to use compression vector method to reduce cost and we can increase time and can do for many points.
- Since in C++ standard library vectors we can have an array upto 1e7 and in the conjugate gradient method we require memory of size n^4 so we were able to run only upto n = 50.
- To run conjugate gradients on device we need to have compressed matrix perform operations.

Chapter 9
Conjugate gradient method using Sparse matrix

Sparse matrix: When a matrix has a lot of zero value in the data then it is called sparse matrix.
Sparse matrix or CSR(Compressed Sparse Row) consist of three main components i) Row number ii) Column Number iii) Value at that position
All these components are converted into a vector and then pushed into a vector of vectors for further calculation.

For conjugate gradient method we need following function for sparse matrix:
i) Addition of sparse matrix
ii) Multiplication of sparse matrix
iii) Transpose of sparse matrix
iv) Norm of sparse matrix
v) Dot product of sparse matrix
vi) Insert new value in sparse matrix
vii) Const Multiplication with sparse matrix

The sparse_matrix class has the following member variables:

- data: a vector of vectors to store the non-zero values of the sparse matrix, where each inner vector stores the row index, column index, and value of a non-zero element.
- row: the number of rows in the sparse matrix
- col: the number of columns in the sparse matrix
- len: the number of non-zero elements in the sparse matrix

The sparse matrix that is used throughout the programme is sorted by its row values. The column values of two elements that share the same row values are used to further sort them.

i) Addition

In order to combine the two matrices, we simply go over each one element at a time and add the smaller element—the one with the smaller row and col values—to the resulting matrix. If an element has the same row and column values as another element, we simply add their values and insert the new data into the final matrix.

ii) Transpose

Simply changing each column value to the row value and vice versa will transpose a matrix, but the resulting matrix won't be sorted as we need it to be. Consequently, we start by counting the number of elements to determine the precise index of the resulting matrix, where the current element should be placed, the current element's column must be smaller than the column being inserted. This is accomplished by keeping track of an array index[] whose ith value represents the number of matrix items beneath column i.

iii) Multiplication

To make our comparisons easier and keep the sorted order, we first calculate the transpose of the second matrix before multiplying the matrices. In order to obtain the resultant matrix, the length of both matrices must be traversed, and the relevant multiplied values must be added.

Each row in the first matrix with a value equal to x and any row in the second matrix (the transposed one) with a value equal to y will contribute to the result[x][y].
The result [x][y] is derived by multiplying all such elements with col values in both matrices and only adding those with row values of x in the original matrix and y in the second transposed matrix.

iv) Norm

In this we take the numeric values of the sparse matrix. Keep on adding the square of the value in a variable say x and return the square root of x.

v)Dot Product

Two sparse 1D matrices can only have dot products. So firstly we check if the two matrices undergoing Dot product must be of suitable size and then the first element of the first matrix is multiplied with the first element of the second matrix and added to a variable, say x. Later this x is returned as a scalar quantity.

vi) Insert new value in sparse matrix

To insert a new value in the sparse matrix we make a vector whose first index contains information about the row in the original matrix and the second index contains the information of the column of the element in the original matrix. The last information about the element that is its value is stored on the third index and this vector is then pushed to the sparse matrix.

vii) Constant Multiplication with sparse matrix

In this function we just updated the original values in the sparse matrix with constant*(original value).

For complete code refer to appendix 1.9

9.1 Result for Host Code

Table 9.1 *Time taken by host code*

n	Time	No of Iterations
10	0.003998	9
20	0.016006	25
50	0.245956	106
100	3.880889	409
200	69.090346	1647
500	2938.499016	6694

Table 9.1 in this project reveals that the computation time for the CPU increases exponentially as the number of points in the problem increases. This trend is particularly noticeable after 200 points. Additionally, the number of iterations required to reach the desired tolerance level is consistently less than the total number of variables, which is equal to n^2.

Chapter 10
GPU based Jacobi iterative solver for the solution of the heat equation in 3D

10.1 Algorithm for Host

Arguments for host function are as follows:

Table 10.1 *Arguments for host function*

n	Total number of grid points in mesh
l	Number of iterations to be performed
T	Double array to store temperature of size (N*N*N)
Told	Double array to store old temperature of size (N*N*N)

Algorithm:

```
1. While l is greater than 0, repeat steps 2 to 5 l times.
2. For i from 0 to n^3-1, copy the new temperature array to the old temperature array:
   a. Set Told[i] to T[i].
3. For i from n^2 to n^3-n^2-1, update the new temperature array:
   a. If i is not a boundary point (i.e. it is an interior point), do the following:
      i. Set T[i] to the average of the temperatures of its six neighbors: Told[i+1], Told[i-1],
Told[i+n], Told[i-n], Told[i+n^2], and Told[i-n^2], divided by 6.
4. Decrement l by 1.
5. Return from the function.
```

For complete code refer to appendix 1.10

10.2 Algorithm for Device

1. Define the size of the block and grid dimensions to launch the kernel on the GPU.
2. Allocate memory on the device using cudaMemcpy() to copy the host array T to the device array T_d.

3. Launch the kernel jacobi_device on the device using <<< nBlocks, blockSize >>>.
4. Synchronise the device with the host using cudaDeviceSynchronize().
5. Copy the device array T_d back to the host array T using cudaMemcpy().
6. The kernel function jacobi_device takes in the 3D grid dimensions n, the device array for the new temperature T_d, the device array for the old temperature Told_d, and the number of iterations l.
7. Calculate the unique index of each thread in the 3D grid using blockIdx.x, blockDim.x, and threadIdx.x.
8. Run a loop for l iterations.
9. Copy the new temperature array T_d to the old temperature array Told_d using the unique index.
10. Synchronise the threads using __syncthreads() to ensure all threads have copied the new temperature array before proceeding.
11. Check if the unique index corresponds to a non-boundary point in the 3D grid using the condition (idx<=((n*n*n)-n*n) and idx>=n*n) and !((idx%n==0) || ((idx)%n==n-1) || idx%n*n==0).
12. If the condition is true, calculate the average temperature of the surrounding points and update the new temperature array T_d at the unique index.
13. Repeat the loop until all l iterations are completed.
14. The kernel function jacobi_device returns void.

For complete code refer to appendix 1.11

10.3 Result for GPU based Jacobi iterative solver for the solution of the heat equation in 3D

10.3.1 For 0.01 Million Iteration

Table 10.2 Data for 0.01 Million Iterations

Time taken by host code and device code when it was executed for 0.01 million iterations

Time for 0.01million Iterations		
n	Time for host	Time for device
10	0.147128	0.0000234
20	1.382947	0.0000821
30	6.066671	0.0001973

40	13.568029	0.0003542
50	26.003992	0.0005718
100	225.158173	0.0136728
200	2365.34278	0.3018759
400	**	3.2845932
500	**	7.9723751
750	**	31.2790524
1000	**	87.4395732

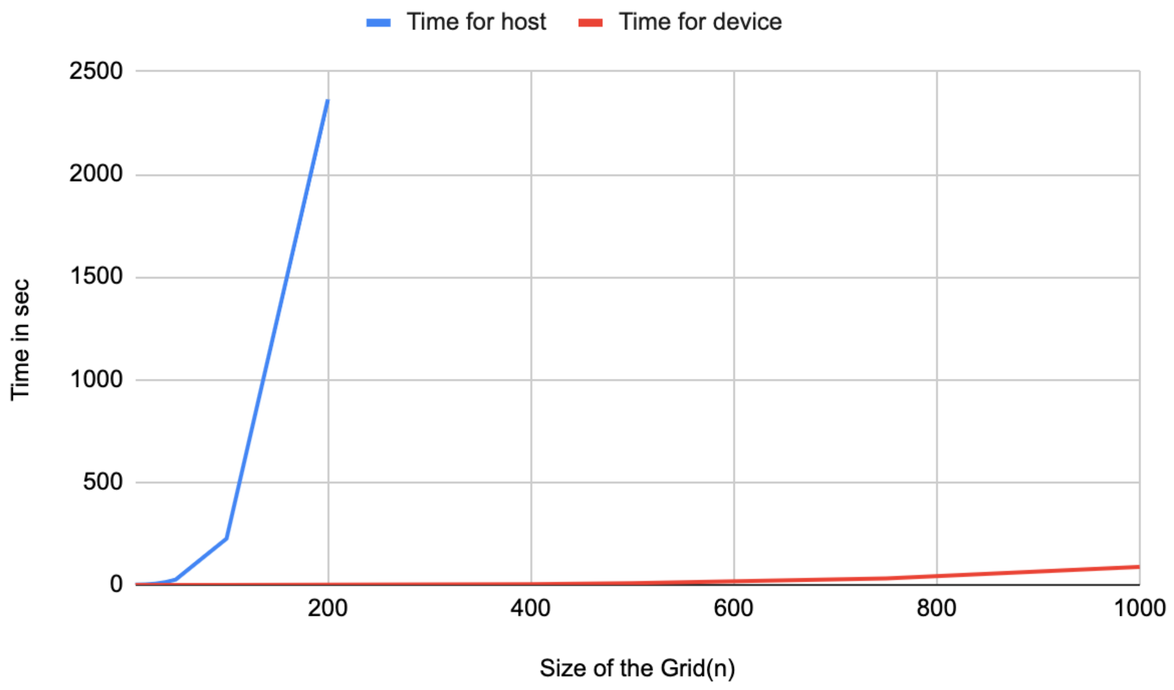


Figure 10.1: Graphical representation for table 10.2

10.3.2 GPU based Jacobi iterative solver for the solution of the heat equation in 3D For 0.1 Million Iteration

Table 10.3 Data for 0.1 Million Iterations

Time taken by host code and device code when it was executed for 0.1 million iterations.

Time for 0.1 million Iterations		
n	**Time for host**	**Time for device**
10	1.45691	0.0001784
20	15.450091	0.0008241
30	55.158867	0.0019128
40	134.298782	0.0040453
50	259.854492	0.0074869
100	2245.578125	0.2745212
200	**	3.8536517
400	**	34.8937262
500	**	83.9421191
750	**	330.9013255
1000	**	933.8136832

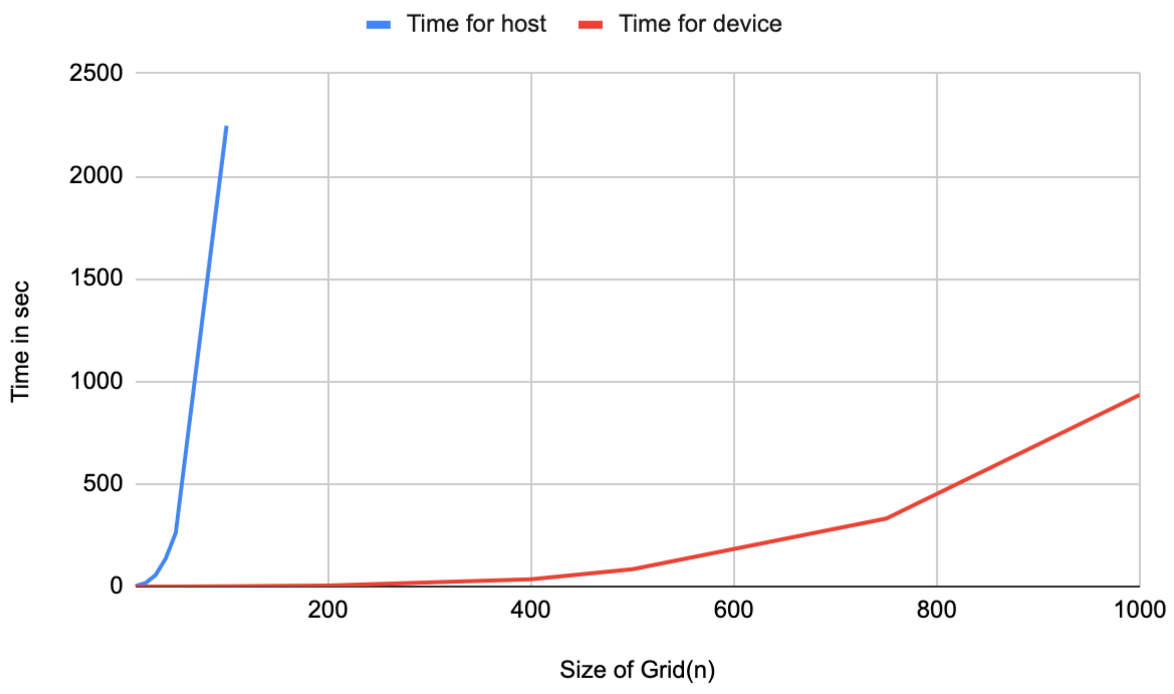


Figure 10.2: Graphical representation for table 10.3

Similar observation can be seen from figure 10.2 (same as in section 6.2) but here for very few points we can see the significant change in time for host as well as on device.

Chapter 11
Analysis

- The time complexity for host code (6.1.1) is O(1*N^2) whereas for the device (6.1.2) is O(l*N).where l is the number of iterations and N is the number of variables.
- The time complexity for host code is O(l*n2) (6.3.1) whereas for the device (6.3.2) is O(l) where l is the number of iterations and n is the number of points.
- Time complexity is of less order in Cuda code as compared to host code because of parallelization using multiple threads.
- From figure 6.1 and figure 6.2, we can see that time taken by the host increases rapidly as compared to Cuda code i.e. on GPU.
- Error decreases as we increase the number of iterations for both host as well as device.
- For a large number of points, the time difference in host code as well as device code is very large.
- It wasn't feasible for time calculation for host code for more than 2000 points whereas for cuda it was 10000 points.
- In red black gauss seidel method we can see that it is much faster than jacobi method
- Conjugate gradient method on host code converges in n steps as for 10 by 10 size matrix it converges 21 times for 1e-5 tolerance value.
- The conjugate gradient method involves performing many unnecessary operations because the coefficient matrix used in the method is sparse, meaning it has many zeroes. To reduce the computational cost and increase efficiency, a compressed vector method can be used. This would allow the method to be run for many points, as the memory required for the method scales with n^4. In C++, the standard library vector can only accommodate arrays up to 1e7, so the conjugate gradient method can only be run up to n = 50. To run the conjugate gradient method on a device, the matrix must first be compressed, allowing for more efficient operations.

Chapter 12
Conclusion

In conclusion, the project successfully implemented and compared different numerical solvers for linear systems of equations on a GPU using CUDA programming. The methods included Jacobi's iterative method, Red-Black Gauss-Seidel, and Conjugate Gradient method for solving 2D steady-state heat equations. The project also extended Jacobi's iterative method for 3D problems and developed a GPU code for it. The results showed that the GPU-accelerated numerical solvers were significantly faster than their CPU counterparts, particularly for larger problem sizes. The Red-Black Gauss-Seidel method was found to be faster than Jacobi's iterative method, and the Conjugate Gradient method was shown to be more efficient due to its ability to handle sparse matrices. Overall, the project demonstrated the potential of GPU acceleration in numerical computing and provided valuable insights into its practical applications Cuda programming provides a platform to perform parallel computing on devices i.e GPU. this is the reason parallel computing is used in many fields such as Data science and analytics.

References

[1] Roman Trobec, Boštjan Slivnik, Patricio Bulić, Borut Robicˇ. (2018). "*Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*"

[2] Duane Storti, Mete Yurtoglu. (2016). "*CUDA FOR ENGINEERS:An Introduction to High Performances Parallel Computing*"

[3] NVIDIA, NVIDIA corporation. (2008-2009). "*NVIDIA CUDA Software and GPU Parallel Computing Architecture"*

[4] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni and Amit Bawaskar. (2012). *"GPGPU PROCESSING IN CUDA ARCHITECTURE, Advanced Computing:An International Journal (ACIJ), Vol.3."*

[5] H.L.Royden ,P.M.Fitzpatrick. *"Real Analysis"* 4th edition.

[6] Michael L. Overton. (2001). "*Numerical Computing with IEEE Floating Point Arithmetic*"

[7] Elias Konstantinidis, Yiannis Cotronis.(2012). *"Accelerating the red/black SOR method usingGPUs with CUDA".*

[8] V.B.K. Vatti.(2016). "*Numerical Analysis: Iterative Methods*".

[9] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. (2008). *"A Survey of General-Purpose Computation on Graphics Hardware"*

[10] David B. Kirk and Wen-mei W. Hwu. (2010). *"Programming Massively Parallel Processors: A Hands-on Approach"*

[11] Anders Logg, Kent-Andre Mardal, and Garth Wells. (2012). *“Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book”*

[12] Nicholas J. Higham. (2002). *“Accuracy and Stability of Numerical Algorithms”*

[13] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. (2007). “*Numerical Recipes 3rd Edition: The Art of Scientific Computing*”

Appendix

1.1

```
// function for running code on host
void jacobi_host(int n,double* a,double* c,double* b,double* x,int
l)
{
   // initialising variables
   int i,j;

   // while loop runs for number of iterations
   while(l--)
   {
      // for loop runs for all n equations
           for(i=0;i<n;i++) {
               c[i]=b[i];

           // for loop runs for ith equation and perform
calculation for jacobi iteration
           for(j=0;j<n;j++) {
      // checking if it's not the diagonal element
               if(i!=j) {
                   c[i]=c[i]-a[i*n + j]*x[j];
               }
           }
       }

      // do calculations for all n equations
       for(i=0;i<n;i++) {
           x[i]=c[i]/a[i*n + i];
       }
   }
   return;
}
```

1.2

```
// jacobi function which will run on device
// __global__ is a CUDA C keyword which says function will
executes on device
__global__ void jacobi_device(int n,double* a_d,double*
c_d,double* b_d,double* x_d,int l)
{
   // calculation of unique index
   int idx = blockIdx.x*blockDim.x + threadIdx.x;

   // while loop runs for number of iterations
   while(l--)
   {
      // condition for checking if unique index value is less than
n to avoid error
if (idx < n)
       {
           // initialization of sum and index
           double sum = 0.0;
           int idx_ai = idx*n;

           // only a single for loop is there as compared to host
code as calculation for each equations are independent so we can
parallelize the calculations by launching multiple threads
           for (int j=0; j<n; j++){

              // checking if it's not diagonal element
               if (idx != j)
                   sum -= a_d[idx_ai + j] * x_d[j];

           // using the below keyword we assure that all the
threads have completed their calculations before performing
another calculation
           __syncthreads();
           x_d[idx] = (b_d[idx] - sum) / a_d[idx_ai + idx];
       }
   }
}
```

1.3

```
void jacobi_host(double* T,double *Told,int l,int n)
{
    while(l--)// running loop till l iterations
    {
      for(int i=0;i<n*n;i++)//copying new temperature array to 
Told
      {
           Told[i] = T[i];
      }
     for(int i=n;i<((n*n)-n);i++) // new temperature array to Told
     {
     if(!((i%n==0) || ((i+1)%n==0)))//not a boundary point check
     {
   T[i] = (Told[i+1] + Told[i-1] + Told[i+n] + Told[i-n])/4
           }
      }
    }
    return;
}
```

1.4

```
dim3 blockSize(1024);                    //1024 threads per block
dim3 nBlocks(16);                        //16 blocks per grid
```

```
cudaMemcpy(T_d, T, sizeof(double)*nn, cudaMemcpyHostToDevice);
//copying data from host array to device using cudaMemcpy function

jacobi_device<<< nBlocks, blockSize >>>(n,T_d,c_d,l); //calling 
device function

cudaDeviceSynchronize();
//to synchronise

cudaMemcpy(f_d, T_d, sizeof(double)*nn, 
cudaMemcpyDeviceToHost);//copying data from device array to host 
using cudaMemcpy function
```

```
__global__ void jacobi_device(int n,double* T_d,double* Told_d,int
l)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x; //unique index
calculation
    while(l--) //running loop till l iterations
    {
        if (idx < n*n)
//unique index should be less than n*n
        {
           Told_d[idx] = T_d[idx];//copying new Temperature array
to Told
            __syncthreads();//synchronising threads
            if (idx<=((n*n)-n) and idx>=n) //condition to avoid
boundary points
            {
                if(!((idx%n==0) || ((idx+1)%n==0)))
                 {
                        T_d[idx] = (Told_d[idx+1] + Told_d[idx-1]
+ Told_d[idx+n] + Told_d[idx-n])/4; //Temperature average
                 }
            }
        }
    }
}
```

1.5

```
//this function return the root mean square of the deflection in the
value.
double calculate_error(//data from red_black_host funtion)
{
      double error = 0;

for(int i= 0;i<n*n;i++)
      float x = abs(T[i] - Told[i]) , error += x*x;
return sqrt(error);
}
void red_black_host(double* T,double *Told,int l,int n)
```

```
{
      while(error is more than 1e-3)
      {
            //creating copy of old matrix
            for(int i=0;i<n*n;i++)
            {
                  Told[i] = T[i];
            }

            for(int i=n;i<((n*n)-n);i++){
            // this for loop is for the red point in which we  update
using the adjacent and independent black points
       if(!((i%n==0) || ((i+1)%n==0)))
      {
         int row = (i/n);
        int col = (i%n);
        if((row+col)%2)
        T[i] = (Told[i+1] + Told[i-1] + Told[i+n] + Told[i-n])/4;
            }
        }
for(int i=n;i<((n*n)-n);i++){
// this for loop is for the black point in which we update using the
adjacent and independent newly calculated red points for the
previous loop.
      if(!((i%n==0) || ((i+1)%n==0))){
            int row = (i/n);
            int col = (i%n);
            if((row+col)%2==0)
                 T[i] = (T[i+1] + T[i-1] + T[i+n] + T[i-n])/4;
      }
 }
    error = calculate_error(T,Told,n); //function calculate error is
called to update the value of error
    No_of_iterations++; // calculation no of iteration to converge
to the ans
}}
```

1.6

```
__global__ void jacobi_device(int n,double* T_d,double* Told_d,int
l)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;//unique index
calculation
    double epsilon = 1e-3;
    double error = 5;
    int no_of_iterations = 0;
    while(l-){//running loop till l iterations
         no_of_iterations++;//calculating the no of iterations
        //unique index should be less than n*n
        if (idx < n*n){
           Told_d[idx] = T_d[idx];
            //copying new Temperature array to Told
            __syncthreads(); //synchronising threads
        if (idx<=((n*n)-n) and idx>=n){//condition to avoid boundary
points
            int row = (idx/n);
            int col = (idx%n);
            if(!((idx%n==0) || ((idx+1)%n==0))&& ((row+col)%2))
            {
             //condition to avoid boundary points and black points
               T_d[idx] = (Told_d[idx+1] + Told_d[idx-1] +
Told_d[idx+n] + Told_d[idx-n])/4;
             }}
            __syncthreads();//synchronising threads
            if (idx<=((n*n)-n) and idx>=n){
                int row = (idx/n);
                int col = (idx%n);
            if(!((idx%n==0) ||((idx+1)%n==0))&((row+col)%2==0))
            {
             //condition to avoid boundary points and red points
                  T_d[idx] = (T_d[idx+1] + T_d[idx-1] + T_d[idx+n] +
T_d[idx-n])/4 }}
            __syncthreads();    //synchronising threads
           double temp = 0;
           temp += abs(T_d[idx] - Told_d[idx]);
          error = temp;
      }}}
```

1.7

```
vector<double> linear_cg(vector<vector<double>>& A,vector<double>&
b,vector<double>& x_o)
{
    double tol = 1e-5;
    vector<double> x_k = x_o;
    vector<double> r_k = matrix_multi(A,x_k);
    r_k = subtract(r_k,b);
    vector<double> p_k = negative(r_k);
    double r_k_norm = norm(r_k);
    int num_itr = 0;
    while(r_k_norm>tol)
    {
        vector<double> Apk = matrix_multi(A,p_k);
        double rk_rk = dot_product(r_k,r_k);
        double alpha = rk_rk/dot_product(p_k,Apk);
        x_k = addi(x_k,const_multi(alpha,p_k));

        r_k = addi(r_k,const_multi(alpha,Apk));
        double beta = dot_product(r_k,r_k)/rk_rk;

        p_k =addi(negative(r_k),const_multi(beta,p_k));
        Num_itr++;

        r_k_norm = norm(r_k);
    }
    return x_k;
}
```

1.8

```
int N = 10;

    vector<vector<double>> A(N*N,vector<double>(N*N,0));
    vector<double> X(N*N,0);
    vector<double> b(N*N,0);

    double TL = 120;      //Temperature of left wall in celsius
    double TR = 40;       //Temperature of Right wall in celsius
    double TT = 40;       //Temperature of Top wall in celsius
    double TB = 40;       //Temperature of Bottom wall in celsius
    for(int i = 0;i<N*N;i++)
    {
        //boundary points
        if(i<N or i>=(N*N - N) or i%N==0 or (i+1)%N==0)
        {
            //for coefficient matrix
            A[i][i] = 1;

            //for X and b
            if(i%N==0)
            {
                X[i] = TL;
                b[i] = TL;
            }
            else
            {
                X[i] = TT;
                b[i] = TT;
            }
        }
        else
        {
            A[i][i] = -1;
            A[i][i-1] = 0.25;
            A[i][i+1] = 0.25;
            A[i][i-N] = 0.25;
            A[i][i+N] = 0.25;

            X[i] = 25;
        }
    }

      X[0] = (TL + TT)/2;
      X[N-1] = (TR + TT)/2;
      X[(N*N)-N] = (TL + TB)/2;
      X[N*N - 1] = (TR + TB)/2;

      b[0] = (TL + TT)/2;
      b[N-1] = (TR + TT)/2;
      b[(N*N)-N] = (TL + TB)/2;
      b[N*N - 1] = (TR + TB)/2;
```

```
    vector<double> final = linear_cg(A,b,X);
```

1.9

```
class sparse_matrix
{
public:
    vector<vector<double>> data;
     int row, col;
     int len;
     sparse_matrix(int r, int c)
     {
           row = r;
           col = c;
           len = 0;
     }

    void sortAfter()
    {
        sort(data.begin(),data.end());
    }
    sparse_matrix operator=(const sparse_matrix other)
    {
        if (this == &other) {
            return *this;
        }
        row = other.row;
        col = other.col;
        len = other.len;
        vector<vector<double>> data = other.data;
        return *this;
    }

     void insert(double r, double c, double val)
     {
           if (r > row || c > col)
           {
                 cout << "wrong";
           }
```

```
          else
          {
           data.push_back({r,c,val});
          }
       len = data.size();
     }

    sparse_matrix add(sparse_matrix b)
    {
        sparse_matrix result(row, col);

        if (row != b.row || col != b.col)
        {
            cout << "Error: dimensions do not match." << endl;
            return result;
        }

        // sort data arrays
        sortAfter();
        b.sortAfter();

        int i = 0, j = 0;

        while (i < len && j < b.len)
        {
            int r1 = data[i][0], c1 = data[i][1], r2 =
b.data[j][0], c2 = b.data[j][1];
            double v1 = data[i][2], v2 = b.data[j][2];

            if (r1 < r2 || (r1 == r2 && c1 < c2))
            {
                result.insert(r1, c1, v1);
                i++;
            }
            else if (r1 > r2 || (r1 == r2 && c1 > c2))
            {
                result.insert(r2, c2, v2);
                j++;
            }
            else
            {
                double sum = v1 + v2;
                if (sum != 0)
                {
                    result.insert(r1, c1, sum);
                }
                i++;
                j++;
            }
        }

        // add remaining elements
        while (i < len)
        {
            result.insert(data[i][0], data[i][1], data[i][2]);
            i++;
        }
        while (j < b.len)
        {
            result.insert(b.data[j][0], b.data[j][1],
b.data[j][2]);
            j++;
        }

        return result;
    }

     sparse_matrix transpose()
     {

          sparse_matrix result(col, row);
          result.len = len;

        for(int i = 0;i<data.size();i++)
        {
            result.insert(data[i][1],data[i][0],data[i][2]);
        }


          return result;
     }
     sparse_matrix multiply(sparse_matrix b)
     {
          if (col != b.row)
          {
              return b;
          }
          b = b.transpose();
          int apos, bpos;

          sparse_matrix result(row, b.row);
          for (apos = 0; apos < len;)
          {
              int r = data[apos][0];

                for (bpos = 0; bpos < b.len;)
                {
                    int c = b.data[bpos][0];
                    int tempa = apos;
                    int tempb = bpos;

                    int sum = 0;
                    while (tempa < len && data[tempa][0] == r &&
                        tempb < b.len && b.data[tempb][0] == c)
                    {
                        if (data[tempa][1] < b.data[tempb][1])

                            tempa++;

                        else if (data[tempa][1] >
b.data[tempb][1])
                            tempb++;
                        else
                            sum += data[tempa++][2] *
                                b.data[tempb++][2];
                    }
                    if (sum != 0)
                        result.insert(r, c, sum);

                    while (bpos < b.len &&
                        b.data[bpos][0] == c)
```

```
                           bpos++;
                }
                while (apos < len && data[apos][0] == r)
                      apos++;
           }
        return result;
     }
     void print()
     {
           cout << "\nDimension: " << row << "x" << col;
           cout << "\nSparse Matrix: \nRow\tColumn\tValue\n";

           for (int i = 0; i < data.size(); i++)
           {
                cout << data[i][0] << "\t " << data[i][1]
                      << "\t " << data[i][2] << endl;
           }
     }

    void redeclare(sparse_matrix c)
    {
        row = c.row;
        col = c.col;
        len = c.len;
        data = c.data;
    }

};


sparse_matrix neg(sparse_matrix A){
    for(int i = 0;i<A.len;i++)
    {
        A.data[i][2] = -A.data[i][2];
    }
    return A;
}

double Norm(sparse_matrix &A)
{
    double res = 0;
    for(int i = 0;i<A.len;i++)
    {
        res += A.data[i][2]*A.data[i][2];
    }
    return sqrt(res);
}

int check_(sparse_matrix &A)
{
    int ln = A.len;
    int r = 0, c = 0;
    for(int i = 0;i<ln;i++)
    {
        if(A.data[i][0]==0)r++;
        if(A.data[i][1]==0)c++;
    }
    if(r==ln)return 1;
    return 0;
}

double solve_(sparse_matrix &A,sparse_matrix &B,int c)
{
    double res = 0;
    int l1 = A.len;
    int l2 = B.len;
    if(c==0)
    {
        // both row
        for(int i = 0;i<l1;i++)
        {
            for(int j = 0;j<l2;j++)
            {
                if(A.data[i][0]==B.data[j][0])
                {
                    res += A.data[i][2]*B.data[j][2];
                }
            }
        }
    }
    else if(c==1)
```

```
{
    // A-> r but B->c
    for(int i = 0;i<l1;i++)
    {
        for(int j = 0;j<l2;j++)
        {
            if(A.data[i][0]==B.data[j][1])
            {
                res += A.data[i][2]*B.data[j][2];
            }
        }
    }
}
else if(c==2)
{
    // A->c but B->r
    for(int i = 0;i<l1;i++)
    {
        for(int j = 0;j<l2;j++)
        {
            if(A.data[i][1]==B.data[j][0])
            {
                res += A.data[i][2]*B.data[j][2];
            }
        }
    }
}
else
{
    // A->c but B->c
    for(int i = 0;i<l1;i++)
    {
        for(int j = 0;j<l2;j++)
        {
            if(A.data[i][1]==B.data[j][1])
            {
                res += A.data[i][2]*B.data[j][2];
            }
        }
    }
}
```

```
    return res;
}

sparse_matrix Const_multi(double alpha,sparse_matrix A)
{
    for(int i = 0;i<A.len;i++)
    {
        A.data[i][2] = alpha*A.data[i][2];
    }
    return A;
}

double Dot_product(sparse_matrix& A, sparse_matrix& B)
{
    double result = 0.0;

    A.sortAfter();
    B.sortAfter();

    int i = 0, j = 0;
    while (i < A.len && j < B.len) {
        int c1 = A.data[i][1], c2 = B.data[j][1];

        if (c1 == c2) {
            result += A.data[i][2] * B.data[j][2];
            i++;
            j++;
        }
        else if (c1 < c2) {
            i++;
        }
        else {
            j++;
        }
    }

    return result;
}
vector<double> linear_cg(sparse_matrix& A,sparse_matrix&
b,sparse_matrix& x_o)
{
 double tol = 1e-5;

 sparse_matrix x_k = x_o;

 x_k.sortAfter();
 A.sortAfter();
 sparse_matrix tempr_k = A.multiply(x_k);
 cout<<"here"<<endl;
 A.print();
 x_k.print();
 tempr_k.print();

 sparse_matrix negb = neg(b);

 negb.print();

 sparse_matrix r_k = tempr_k.add(negb);

 sparse_matrix p_k = neg(r_k);


 double r_k_norm = Norm(r_k);
 int num_itr = 0;

 while(r_k_norm>tol)
 {
 sparse_matrix Apk = A.multiply(p_k);
 double rk_rk = Dot_product(r_k,r_k);
 double alpha = rk_rk/Dot_product(p_k,Apk);
 cout<<alpha<<endl;
 p_k.print();
 Apk.print();

 sparse_matrix n_x_k = x_k.add(Const_multi(alpha,p_k));
 sparse_matrix n_r_k = r_k.add(Const_multi(alpha,Apk));

double beta = Dot_product(n_r_k,n_r_k)/rk_rk;
sparse_matrix n_p_k = neg(n_r_k).add(Const_multi(beta,p_k));

 p_k.redeclare(n_p_k);
 r_k.redeclare(n_r_k);
```

```
     x_k.redeclare(n_x_k);

    num_itr++;
          r_k_norm = Norm(n_r_k);

     }

     x_k.print();
     return x_k;
 }
```

1.10

```
void jacobi_host(double* T,double *Told,int l,int n)
{
    while(l--)// running loop till l iterations
    {
      for(int i=0;i<n*n*n;i++)//copying new temperature array to
Told
      {
            Told[i] = T[i];
      }
     for(int i=n*n;i<=((n*n*n)-n*n);i++) // new temperature array
to Told
     {
     if(!((i%n==0) || ((i)%n==n-1) || i%(n*n)==0)//not a boundary
point check
     {
   T[i] = (Told[i+1] + Told[i-1] + Told[i+n] + Told[i-n] +
Told[i+n*n] + Told[i-n*n])/6;
            }
      }
    }
    return;
}
```

1.11

```
dim3 blockSize(1024);                    //1024 threads per block
dim3 nBlocks(16);                        //16 blocks per grid
```

```
cudaMemcpy(T_d, T, sizeof(double)*nn, cudaMemcpyHostToDevice);
//copying data from host array to device using cudaMemcpy function

jacobi_device<<< nBlocks, blockSize >>>(n,T_d,c_d,l); //calling
device function

cudaDeviceSynchronize();
//to synchronise

cudaMemcpy(f_d, T_d, sizeof(double)*nn,
cudaMemcpyDeviceToHost);//copying data from device array to host
using cudaMemcpy function

__global__ void jacobi_device(int n,double* T_d,double* Told_d,int
l)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x; //unique index
calculation
    while(l--) //running loop till l iterations
    {
        if (idx < n*n*n)
//unique index should be less than n*n
        {
           Told_d[idx] = T_d[idx];//copying new Temperature array
to Told
            __syncthreads();//synchronising threads
            if (idx<=((n*n*n)-n*n) and idx>=n*n) //condition to
avoid boundary points
            {
                if(!((idx%n==0) || ((idx)%n==n-1) ||idx%n*n==0 ))
                 {
                         T_d[idx] = (Told_d[idx+1] + Told_d[idx-1]
+ Told_d[idx+n] + Told_d[idx-n]  + Told_d[idx-n*n] +
Told_d[idx+n*n])/6; //Temperature average
                 }
            }
        }
    }
}
```