GPU computing of Heat Equations

by

Junchi Zhang

A Project Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Applied Mathematics

by

April 2015

APPROVED:

Professor Sarah D. Olson, Project Advisor

Abstract

There is an increasing amount of evidence in scientific research and industrial engineering indicating that the graphic processing unit (GPU) has a higher efficiency and a stronger ability over CPUs to process certain computations. The heat equation is one of the most well-known partial differential equations with well-developed theories, and application in engineering. Thus, we chose in this report to use the heat equation to numerically solve for the heat distributions at different time points using both GPU and CPU programs.

The heat equation with three different boundary conditions (Dirichlet, Neumann and Periodic) were calculated on the given domain and discretized by finite difference approximations. The programs solving the linear system from the heat equation with different boundary conditions were implemented on GPU and CPU. A convergence analysis and stability analysis for the finite difference method was performed to guarantee the success of the program. Iterative methods and direct methods to solve the linear system are also discussed for the GPU. The results show that the GPU has a huge advantage in terms of time spent compared with CPU in large size problems.

Key words: GPU, Heat equation, CPU, linear systems.

List of Figures

List of Tables

1 GPU computing

1.1 Introduction of GPU

Everything a computer does is controlled by its Central Processing Unit (CPU), which is the brain of a computer. All computations were done by CPU several years ago, but with new technology, computations are now faster on a GPU. A graphics processing unit (GPU), is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. The term GPU was popularized by Nvidia in 1999, who marketed the GeForce 256 as 'the world's first GPU', or Graphics Processing Unit, a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that are capable of processing a minimum of 10 million polygons per second [18].

GPUs were initially used for rendering graphics for video games only. Rendering a figure from stored data requires many calculations. The GPU takes much less time to render a vivid picture of high quality in comparison to CPU. Look at Figure 1 [11] shown below, the picture rendered by the GPU is also perfect in detail and saved a lot of time, which means GPUs are absolutely qualified for rendering and even a better choice. As technology has advanced, the large number of cores in GPUs relative to CPUs was exploited by developing computational capabilities for GPUs so that they can process many parallel streams of data simultaneously, no matter what that data may be. The use of the GPU to accelerate non-graphics computation has drawn much attention [3]. While GPUs can have hundreds or even thousands of stream processors, they each run slower than a CPU core.

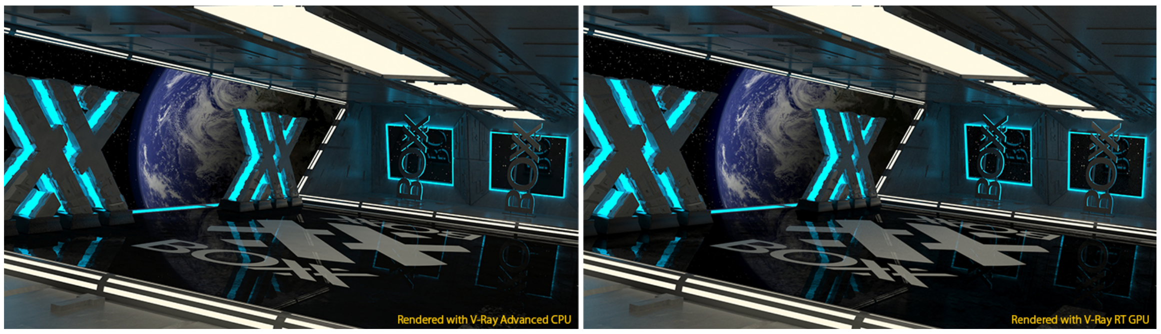


Figure 1: The left picture is rendered by CPU and the right one is by GPU. It's hard to tell the difference between them, but the GPU takes less time to render the graphic in comparison to the CPU. Figure taken from [11].

Modern GPUs are very efficient at manipulating computer graphics and image processing. The highly parallel structure of GPUs makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. Additionally, modern GPUs run 10,000s of threads concurrently [18].

1.2 GPU vs CPU

A simple way to understand the difference between a CPU and a GPU is to compare how they process tasks. Basically, CPUs and GPUs have significantly different architectures that make them better suited to different tasks. As shown in Figure 2 [14], a CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of hundreds of smaller cores. Although GPUs have a large amount of cores, all of these cores share the same device memory, which means input and output data from GPU takes extra time. Therefore, for problems with frequent data changing , the GPU is not a smart choice. The structure of the GPU enables them to handle large amounts of data in many streams and deal with multiple tasks simultaneously, performing relatively simple operations on them. However, it can be hard to deal with complex processing. A CPU is much faster on a single stream and can perform complex operations more easily, but cannot efficiently handle many streams simultaneously [15].

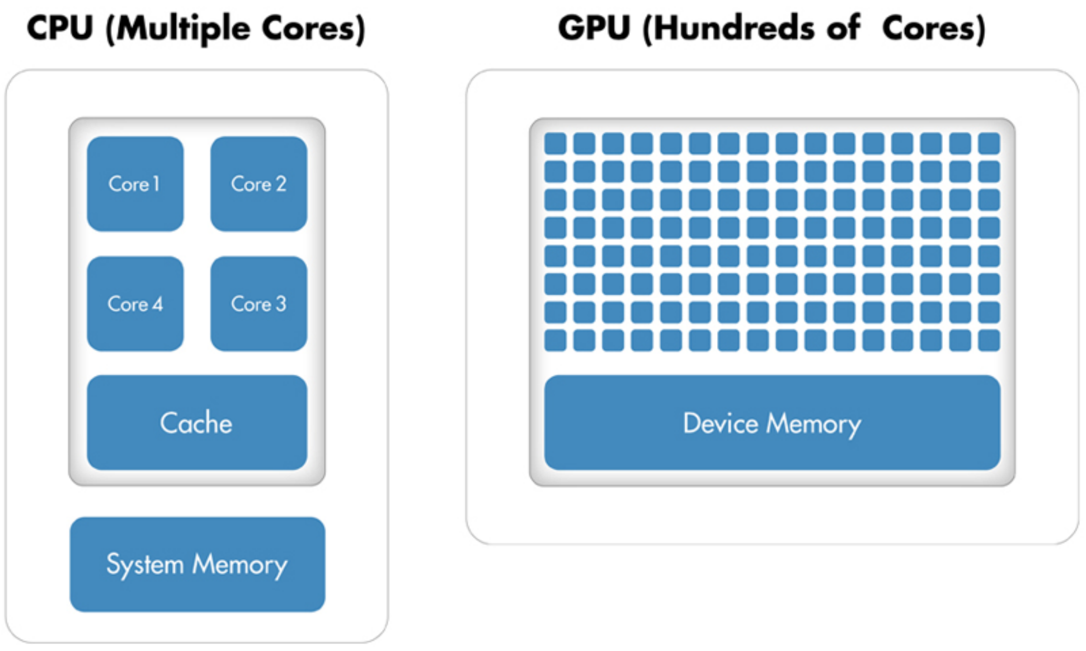


Figure 2: The CPU on the left has several cores that share the same memory in a cache and one machine may have several caches. The GPU on the right has hundreds of cores but all the cores share one device memory. Figure taken from [14].

1.3 GPU Programming

One would have to program an application specifically for a GPU for it to work, and significantly different techniques are required to program GPUs. These different techniques include new programming languages, modifications to existing languages, and new programming strategies that are better for expressing a computation as a parallel operation to be performed by many stream processors.

Frameworks such as CUDA and OpenCL enable programs to be written for GPUs, and the nature of GPUs make them most suited to highly parallelizable operations [9]. CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce.

MATLAB is a high-level language and interactive environment for numerical computation, visualization, and programming [9]. It can support CUDA kernel prototyping and development by providing an environment for quick evaluation and visualization using the CUDA Kernel object. *MATLAB* can be used to [2] :

1. Write prototype code to explore algorithms before implementing them in CUDA.

2. Quickly evaluate CUDA kernels for different input data.

3. Analyze and visualize kernel results.

4. Write test harnesses to validate that kernels are working correctly.

By calling $gpuArray$ in *MATLAB* we can operate arrays by passing it to the GPU, or using one of the methods defined for $gpuArray$ objects to establish an array directly on the GPU. $Parfor$ is a *MATLAB* loop command for the statements that could be executed in parallel. All work is completed by a cluster of workers where each worker has its own unique workspace which are identified and reserved with the $parpool$ command. $Parpool$ starts a pool using the default cluster profile, with the pool size specified by your parallel preferences and the default profile [17]. If we want to have a certain number of workers or run a certain profile (a profile is a local modification), it is easy to call it directly using the *MATLAB* commands. In the following example code, we start a parallel pool of 16 workers using a profile called $myProf$.

```
parpool('myProf',16)
```

We are going to show the different performances between GPU and CPU by the same test case using the fast Fourier transform. A fast Fourier transform (FFT) is an algorithm to compute the discrete Fourier transform (DFT) from consecutive data. FFT is used widely in many scientific, engineering and mathematical applications. The DFT is defined by the following formula [19]:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, 1, ..., N-1 \tag{1}$$

where x_n is the nth element of the original vector and X_k is the kth element of the vector after FFT, as usual $i = \sqrt{-1}$.

Note that the algorithms could be parallel since for each X_k, we could do the summation independently. That's why FFT is a good test case for GPU versus CPU. FFT is a method to compute the same results in $O(NlogN)$ operations while by using Equation (1) directly requires $O(N^2)$ operations [19]. There are N outputs X_i, and each output requires a sum of N terms. By recording the running time of the FFT code using CPU and GPU separately, we will see how they perform. The code is shown below, here fft is a *MATLAB* command to do a FFT for the random initial vector x. Tic, toc are the start and end of a *MATLAB* timer to record running time of the program. To make it a fair comparison, we run the code three times for each size of data, then calculate the average computing time, all the data are shown in Table 1. In Figure 3 we observe that, for small size problems, the time cost for the GPU and CPU calculation are almost the same. However, from Table 1, we know as the vector size goes to 10^7 or larger, the GPU takes approximately 5 percent of the time as the CPU did. We find that when computing the FFT, the GPU can have a dramatic advantage in terms of computation time compared with the CPU for big data.

```
%——Code for FFT
M=1000
tic
x=rand(M,1);
fft(x);
cpu_time=toc

tic
xx=gpuArray.rand(M,1);
fft(xx);
gpu_time=toc
```

vector size	10^3	10^4	10^5	10^6	10^7	10^8
GPU time	0.000397	0.000463	0.000740	0.000817	0.000802	0.12102
CPU time	0.000150	0.000618	0.005042	0.055253	0.562628	2.60096

Table 1: CPU and GPU average computing time of FFT.

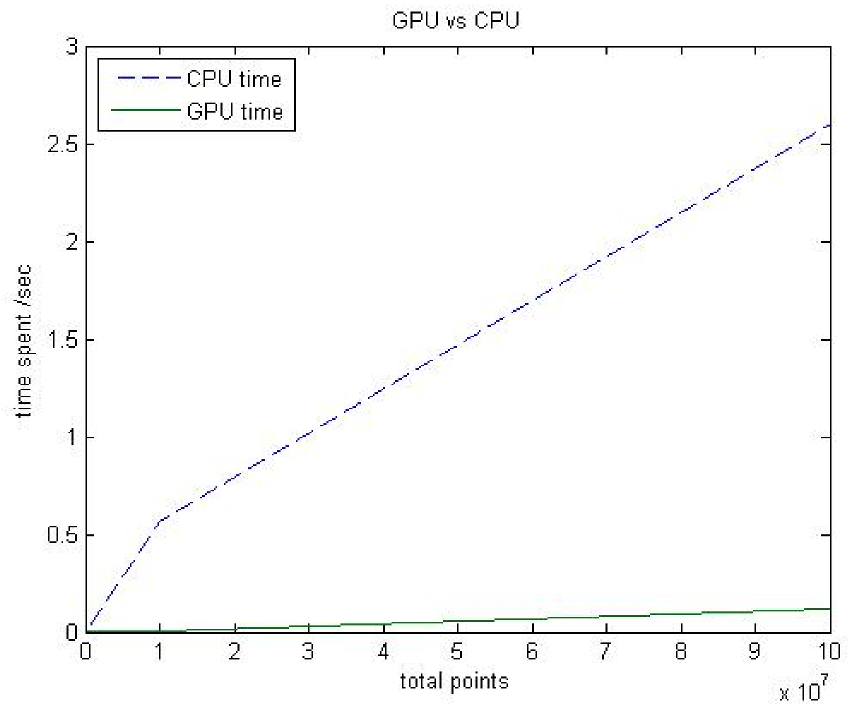


Figure 3: Time spent by the CPU and GPU in computing the fast Fourier transform. The slope of the GPU is flat. This shows that the computation time is much smaller for the GPU for a larger number of points. This will be a big advantage for problems of large size.

2 The Heat Equations

2.1 Introduction

The heat equation is a parabolic partial differential equation that describes the distribution of heat (or variation in temperature) in a given region over time. This equation is often used to construct models of the most basic theories underlying physics and engineering [10, 13] . The heat equation is given by:

$$u_t = \alpha u_{xx}, \qquad u(x,0) = sin\left(2k\pi x\right) \tag{2}$$

with $x \in [0,1], \;\; t \in [0,T]$, the boundary condition is given by

$$u(0,t) = u(1,t) = 0.$$

Here, α is a positive constant, k is an integer, and $u(x,0)$ is the initial condition of equation (2). The material property α is the thermal diffusivity. It is the thermal conductivity divided by the volumetric heat capacity. The formula of α is

$$\alpha = \frac{k}{\rho c_p},$$

where k is thermal conductivity, ρ is density and c_p is specific heat capacity [10]. This is a model of transient heat conduction in a slab of material. The domain of the solution is a strip of width 1 that continues indefinitely in time. In a practical computation, the solution is obtained only for a finite time, up to time T.
The exact solution is given by the following equation:

$$u(x,t) = e^{-\alpha 4k^2\pi^2 t} sin\left(2k\pi x\right).$$

If we set $\alpha = 1$ and $k = 1$, the graph of the solution to the heat equation is shown in Figure 4.

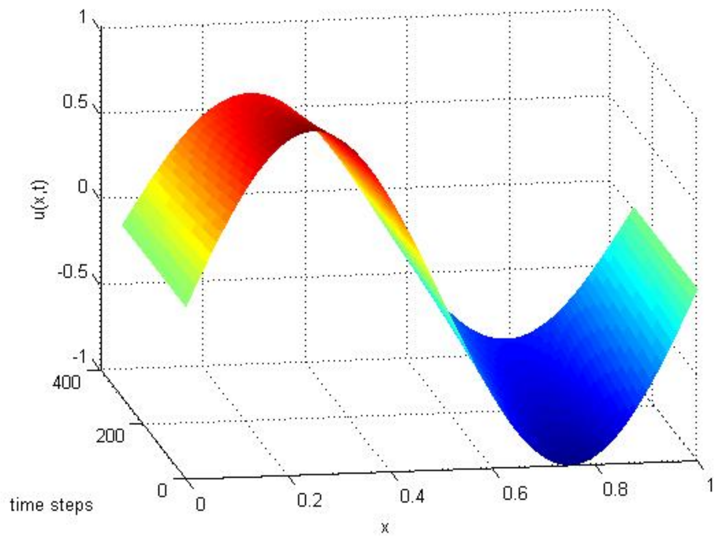


Figure 4: Exact solution to heat equation $e^{-4\pi^2 t}sin(2\pi x)$ with 100 points in x-coordinate and 300 time steps .

We can verify equation (2) by taking partial derivatives of the exact solution:

$$u_x = 2k\pi cos(2k\pi x)e^{-\alpha 4k^2\pi^2 t},$$

$$u_{xx} = -4k^2\pi^2 sin(2k\pi x)e^{-\alpha 4k^2\pi^2 t}.$$

Looking at the left hand side of equation (2),

$$u_t = -4k^2\pi^2\alpha sin(2k\pi x)e^{-\alpha 4k^2\pi^2 t}.$$

Thus, we have verified that $u_t = \alpha u_{xx}$ for the test case.

Actually, it's very hard to get the exact solution for different combinations of initial conditions and boundary conditions though the test case is simple. Therefore, we need to approximate the solution using numerical methods [10]. Of course there are a lot of different approaches to solving the heat equation numerically, including: finite differences, finite elements, spectral methods, and collocation methods. In this report, we will focus on finite differences methods.

2.2 Domain Discretization

We begin our discussion of numerical methods for partial differential equations by getting started solving a problem numerically. We consider the following one dimensional heat equation:

$$u_t = \alpha u_{xx}, \tag{3}$$

$$u_x(0,t) = \chi, \quad u_x(1,t) = \omega \quad \text{with} \qquad x \in [0,1], \quad t \in [0,T], \tag{4}$$

$$u(x,0) = f(x), \tag{5}$$

where α is a constant coefficient and $f(x)$ is an independent function corresponding to initial conditions.

The finite difference method obtains an approximate solution for (x,t) at a finite set of x and t [13]. We first partition the intervals $[0,1]$ and $[0,T]$ into respective finite grids as follows. We divide the interval $0 \leq x \leq 1$ such that

$$x_i = i\triangle x, \quad i = 0,1,2,...,M, \qquad \text{where} \quad \triangle x = \frac{1}{M},$$

and M is the total number of spatial nodes, including those on the boundary. Similarly we partition $[0,T]$ as

$$t_n = n\triangle t, \quad n = 1,2,...,N, \qquad \text{where} \quad \triangle t = \frac{T}{N}.$$

2.3 Finite Difference Approximations

We now have a grid that approximates our domain. The next step is to approximate the solution on this grid.

The finite difference method involves using discrete approximations such as a Taylor expansion of $f(x)$ at x_0:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n.$$

From the Taylor series, we can get the following equation by setting $x_0 + h = x$,

$$f(x_0 + h) = f(x_0) + f'(x_0)h + O(h^2).$$

$O(h^2)$ is the truncation error of the approximation, where the big O corresponds to the error term. The significant terms can be written explicitly, and the least-significant terms can be summarized in a single big O term. For example, we expand e^x at $x = 0$:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ... \quad \text{as } x \to 0, \tag{6}$$

$$e^x = 1 + x + \frac{x^2}{2!} + O(x^3) \quad \text{as } x \to 0, \tag{7}$$

$$e^x = 1 + x + O(x^2) \quad \text{as } x \to 0. \tag{8}$$

We can then approximate $f'(x)$ by following equation:

$$f'(x) = \frac{\partial f}{\partial x} \approx \frac{f(x+h) - f(x)}{h} + O(h). \tag{9}$$

In order to get a second order approximation, we expand $f(x+h)$ and $f(x-h)$

as follows:

$$f(x+h) = f(x) + f'(x)h + f''(x)h^2 + f'''(x)h^3 + O(h^4),$$

$$f(x-h) = f(x) - f'(x)h + f''(x)h^2 - f'''(x)h^3 + O(h^4).$$

To offset $f'(x)h$ and $f'''(x)h^3$, we add these two equations together to get:

$$f(x+h) + f(x-h) = 2f(x) + f''(x)h^2 + O(h^4), \tag{10}$$

$$f''(x)h^2 = f(x+h) + f(x-h) - 2f(x) + O(h^4). \tag{11}$$

Finally, we get the approximation,

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2). \tag{12}$$

This is a centred finite difference approximation with 2nd order accuracy. Thus, approximation of $u_t(x,t)$ could be $\frac{u_k^{n+1} - u_k^n}{\triangle t}$ with 1st order accuracy. Then the PDE in Equation (3) can be discrectized and approximated at the point $(k\triangle x, n\triangle t)$ by

$$\frac{u_k^{n+1} - u_k^n}{\triangle t} = \alpha \frac{u_{k+1}^n - 2u_k^n + u_{k-1}^n}{\triangle x^2}. \tag{13}$$

2.4 Boundary Conditions

We will study the three different boundary conditions. For each boundary condition, we have a slightly different numerical method to approximate the solution.

Dirichlet Boundary Conditions: This is the case when the boundary of the rod has a constant temperature. This can be considered as a model of an ideal cooler having infinitely large thermal conductivity,

$$u(0,t) = \chi, \quad u(1,t) = \omega.$$

Neumann Boundary Conditions: If we have a constant heat flux at the boundary then it will be Neumann Boundary Conditions. If the flux is equal to zero, the boundary conditions describe the ideal heat insulator with heat diffusion [13],

$$u_x(0,t) = \chi, \quad u_x(1,t) = \omega.$$

Periodic Boundary Conditions: We can use this boundary condition when we have an infinite length rod where we have periodic copies along the length,

$$u(0,t) = u(1,t).$$

2.4.1 Dirichlet Boundary Conditions

For the 1-d case. We consider the Forward in Time Central in Space Scheme (FTCS) where we replace the time derivative by the forward differencing scheme and the space derivative by the central differencing scheme. This yields,

$$\begin{aligned}
\frac{u_k^{n+1} - u_k^n}{\triangle t} &= \alpha \frac{u_{k+1}^n - 2u_k^n + u_{k-1}^n}{\triangle x^2}, \\
u_k^{n+1} - u_k^n &= \left(\frac{\alpha \triangle t}{\triangle x^2}\right)(u_{k+1}^n - 2u_k^n + u_{k-1}^n), \\
u_k^{n+1} &= \left(\frac{\alpha \triangle t}{\triangle x^2}\right) u_{k+1}^n + \left(1 - 2\left(\frac{\alpha \triangle t}{\triangle x^2}\right)\right) u_k^n + \left(\frac{\alpha \triangle t}{\triangle x^2}\right) u_{k-1}^n \Bigg).
\end{aligned}$$

Now we will denote $(\frac{\alpha\triangle t}{\triangle x^2})$ as θ. So the formula will be rewritten as :

$$u_k^{n+1} = \theta u_{k+1}^n + (1-2\theta)u_k^n + \theta u_{k-1}^n. \tag{14}$$

This equation only works on the interior nodes, namely for $i = 1, 2, ..., M-1$ and for $n = 1, 2, ..., N$. Let $i = 0$ correspond to $x = 0$ and $i = M$ correspond to $x = 1$. Since the initial condition is given by $u(x, 0) = f(x)$, numerically we have

$$u_i^1 = f(x_i), \quad i = 0, 1, 2, ..., M.$$

Moreover, the boundary conditions $u(0, t) = \chi,\ \ u(1, t) = \omega$ become

$$u_0^j = \chi \ \ u_M^j = \omega, \quad j = 1, 2, ..., N.$$

The explicit nature of the difference method can then be expressed in matrix form as:

$$\begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ \vdots \\ u_{M-1}^{n+1} \end{bmatrix} = \begin{bmatrix} 1-2\theta & \theta & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ \theta & 1-2\theta & \theta & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & \theta & 1-2\theta & \theta & 0 & \dots & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & \theta & 1-2\theta & \theta \\ \vdots & \dots & \dots & \dots & \dots & \dots & 0 & \theta & 1-2\theta \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ \vdots \\ \vdots \\ u_{M-1}^n \end{bmatrix}.$$

By doing the iterative multiplication we will have the numerical approximation of all the points on the grid.

Instead of the FTCS, one could have alternatively considered the following Backward in Time Central in Space (BTCS) differencing scheme, also called an implicit method. The finite difference approximation is:

$$\begin{aligned}
\frac{u_k^{n+1}-u_k^n}{\triangle t} &= \alpha\frac{u_{k+1}^{n+1}-2u_k^{n+1}+u_{k-1}^{n+1}}{\triangle x^2},\\
u_k^{n+1}-u_k^n &= \left(\frac{\alpha\triangle t}{\triangle x^2}\right)(u_{k+1}^{n+1}-2u_k^{n+1}+u_{k-1}^{n+1}),\\
u_k^n &= \left(1+2\left(\frac{\alpha\triangle t}{\triangle x^2}\right)\right)u_k^{n+1}-\left(\frac{\alpha\triangle t}{\triangle x^2}\right)u_{k+1}^{n+1}-\left(\frac{\alpha\triangle t}{\triangle x^2}\right)u_{k-1}^{n+1}.
\end{aligned}$$

Noting that the spatial derivative term is now differenced at the new time step. Again using $(\frac{\alpha\triangle t}{\triangle x^2})=\theta$, we can rewrite the equation as:

$$u_k^n = -\theta u_{k+1}^{n+1}+(1+2\theta)u_k^{n+1}-\theta u_{k-1}^{n+1}. \qquad (15)$$

The matrix form will be:

$$\begin{bmatrix} u_1^n \\ u_2^n \\ \vdots \\ \vdots \\ u_{M-1}^n \end{bmatrix} = \begin{bmatrix} 1+2\theta & -\theta & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ -\theta & 1+2\theta & -\theta & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & -\theta & 1+2\theta & -\theta & 0 & \dots & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & -\theta & 1+2\theta & -\theta \\ \vdots & \dots & \dots & \dots & \dots & \dots & 0 & -\theta & 1+2\theta \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ \vdots \\ u_{M-1}^{n+1} \end{bmatrix}.$$

Solving the linear system in iterative manner, we will have the approximations u^{n+1}. Noting that we already have u^1 as initial conditions and u_0, u_M as boundary conditions. Starting at the initial condition u^1, then we will have u^2 by solving the equations. For each time step, the linear system is solved in an iterative manner to obtain the approximations of u^2, u^3,...., u^{N-1}, u^N.

2.4.2 Neumann Boundary Conditions

We next consider what changes are necessary to consider Neumann boundary conditions, which means that a boundary condition on the derivative u_x is given rather than a condition on the value of u itself. We introduce a boundary condition treatment by considering the following problem. The equation (3) and the initial condition in (5) are as described earlier. Now the boundary condition is

$$u_x(0,t) = \chi, \quad u_x(1,t) = \omega.$$

To approximate the Neumann Boundary Condition, we derive a second order approximation. If we apply a centered difference at the boundary, the operator will be outside of the domain. Thus, placing a ghost point is needed and the approximation of the boundary condition will be :

$$\frac{u_1 - u_{-1}}{2\triangle x} = \chi \text{ and } \frac{u_{M+1} - u_{M-1}}{2\triangle x} = \omega, \tag{16}$$

where u_{-1} and u_{M+1} are ghost points. By the explicit approximation mentioned before, we know u_0^{n+1} is approximated by u_{-1}^n, u_0^n and u_1^n. Also, u_M^{n+1} approximated by u_{M-1}^n, u_M^n and u_{M+1}^n as follows:

$$u_0^{n+1} = \theta u_{-1}^n + (1-2\theta)u_0^n + \theta u_1^n, \tag{17}$$

$$u_M^{n+1} = \theta u_{M-1}^n + (1-2\theta)u_M^n + \theta u_{M+1}^n. \tag{18}$$

Now, use $u_{-1} = u_1 + 2\chi\triangle x$ and $u_{M+1} = u_{M-1} + 2\omega\triangle x$ from equation (16) into the above equation to offset the ghost points:

$$\begin{aligned} u_0^{n+1} &= \theta(u_1 - 2\chi\triangle x) + (1-2\theta)u_0^n + \theta u_1^n, \\ u_0^{n+1} &= 2\theta u_1^n + (1-2\theta)u_0^n - 2\theta\chi\triangle x, \\ u_0^{n+1} + 2\theta\chi\triangle x &= 2\theta u_1^n + (1-2\theta)u_0^n. \end{aligned}$$

Similarly, we have the approximation of u_M:

$$u_M^{n+1} + 2\theta\omega\triangle x = 2\theta u_M^n + (1-2\theta)u_{M-1}^n.$$

For the explicit method, we obtain the following system of equations:

$$\begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ \vdots \\ u_{M-1}^n \\ u_M^n \end{bmatrix} = \begin{bmatrix} 1-2\theta & 1+\theta & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ \theta & 1-2\theta & \theta & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & \theta & 1-2\theta & \theta & 0 & \dots & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & \theta & 1-2\theta & \theta \\ \vdots & \dots & \dots & \dots & \dots & \dots & 0 & 1-2\theta & 1+\theta \end{bmatrix} \begin{bmatrix} u_0^{n+1} + 2\theta\chi\triangle x \\ u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ u_{M-1}^{n+1} \\ u_M^{n+1} + 2\theta\omega\triangle x \end{bmatrix}.$$

This is because we have

$$u_k^{n+1} = \theta u_{k+1}^n + (1-2\theta)u_k^n + \theta u_{k-1}^n$$

for $k = 1, 2, ..., M-1$, we use the derived finite difference expression (17) and (18) for u_0 and u_M.

2.4.3 Periodic Boundary Conditions

Let's consider the linear system

$$A\overrightarrow{u} = \overrightarrow{b}.$$

First of all, we will introduce a homogeneous system. A system of linear equations is homogeneous if $\overrightarrow{b}$ is zero vector:

$$\begin{array}{lcr} a_{11}u_1{+}a_{12}u_2{+} & \ldots a_{1n}u_n & = 0 \\ a_{21}u_1{+}a_{22}u_2{+} & \ldots a_{2n}u_n & = 0 \\ \vdots & \vdots & \vdots \\ a_{m1}u_1{+}a_{m2}u_2{+} & \ldots a_{mn}u_n & = 0 \end{array}$$

where a_{ij} is the element in the i^{th} row and j^{th} column of A. Thus, the homogeneous system will be rewritten as:

$$A\overrightarrow{u} = \overrightarrow{0},$$

where A is an m by n matrix, $\vec{u}$ is a column vector with n entries, and $\vec{0}$ is the zero vector with m entries.

There is a close relationship between the solution to a linear system and the solutions to the corresponding homogeneous system. For the linear system $A\vec{u} = \vec{b}$ (for nonzero $\vec{b}$), the entire solution can be described as

$$\vec{u} = \vec{u}^H + \vec{u}^P,$$

where $\vec{u}^H$ is a solution to $A\vec{u} = \vec{0}$ and $\vec{u}^P$ is a particular solution to $A\vec{u} = \vec{b}$.

To solve the linear system with periodic boundary conditions, we seek solutions of the form:

$$\vec{u} = \beta\vec{u}^H + \vec{u}^P.$$

Let the linear system have periodic boundary conditions and work on a grid that is indexed $i = 0, ..., M$. Periodic boundary conditions implies that the solution is equal at the first and last grid point as $u_0 = u_M$. We will use an implicit method to show how to solve this problem.

First of all, we will solve the homogeneous system letting the right hand side be the zero vector and $u_0^H = u_M^H = 1$. Setting up an $(M-1)$ by $(M-1)$ linear system since $u_0^H = u_M^H$. For the heat equation $u_t = \alpha u_{tt}$, based on Equation (13), we have

$$u_i^{H,n} = -\theta u_{i-1}^{H,n+1} + (1+2\theta)u_i^{H,n+1} - \theta u_{i+1}^{H,n+1}, \qquad \text{where} \ \ \theta = \frac{\alpha\triangle t}{\triangle x^2}.$$

Noticing that $u_0^H = u_M^H = 1$ and $\vec{R} = 0$. Thus for $i = 1$,

$$\begin{aligned} u_1^{H,n} &= -\theta u_0^{H,n+1} + (1+2\theta)u_1^{H,n+1} - \theta u_2^{H,n+1}, \\ 0 &= -\theta + (1+2\theta)u_1^{H,n+1} - \theta u_2^{H,n+1}, \\ \theta &= (1+2\theta)u_1^{H,n+1} - \theta u_2^{H,n+1}. \end{aligned}$$

Also for $i = M-1$:

$$\begin{aligned} u_{M-1}^{H,n} &= -\theta u_{M-2}^{H,n+1} + (1+2\theta)u_{M-1}^{H,n+1} - \theta u_M^{H,n+1}, \\ 0 &= -\theta + (1+2\theta)u_{M-1}^{H,n+1} - \theta u_M^{H,n+1}, \\ \theta &= (1+2\theta)u_{M-1}^{H,n+1} - \theta u_M^{H,n+1}. \end{aligned}$$

While for $i = 2 : M-2$, the equation will be

$$0 = -\theta u_{i-1}^{H,n+1} + (1+2\theta)u_i^{H,n+1} - \theta u_{i+1}^{H,n+1}.$$

The linear system is

$$\begin{bmatrix} 1+2\theta & -\theta & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ -\theta & 1+2\theta & -\theta & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & -\theta & 1+2\theta & -\theta & 0 & \dots & \dots & \dots & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & -\theta & 1+2\theta & -\theta \\ \vdots & \dots & \dots & \dots & \dots & \dots & 0 & -\theta & 1+2\theta \end{bmatrix} \begin{bmatrix} u_1^H \\ u_2^H \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_{M-1}^H \end{bmatrix} = \begin{bmatrix} \theta \\ 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ \theta \end{bmatrix}$$

We denote the coefficient matrix as D. Then, $Du^{H,n} = \vec{R}$ and we can solve $u^{H,n}$ for $i = 1 : M-1$. Next, we wish to find a particular solution. Here we use the correct right hand side from the problem and assume $u_0^P = u_M^P = 0$.

For $i = 1$:

$$u_1^{P,n} = -\theta u_0^{P,n+1} + (1+2\theta)u_1^{P,n+1} - \theta u_2^{P,n+1},$$
$$u_1^{P,n} = (1+2\theta)u_1^{P,n+1} - \theta u_2^{P,n+1}.$$

Also for $i = M-1:$

$$u_{M-1}^{P,n} = (1+2\theta)u_{M-1}^{P,n+1} - \theta u_M^{P,n+1}.$$

For $i = 2 : M - 2 :$

$$u_i^{P,n} = -\theta u_{i-1}^{P,n+1} + (1+2\theta)u_i^{P,n+1} - \theta u_{i+1}^{P,n+1}.$$

The linear system will be

$$Du^{P,n+1} = u^{P,n},$$

which gives us the particular solution for $i = 1 : M-1$. Now, we are setting out to determine $\vec{u}$, where: $\vec{u} = \beta\vec{u}^H + \vec{u}^P$. We can use the periodic boundary condition information at u_M to solve for $\beta :$

$$\begin{aligned} -\theta u_{M-1}^{n+1} + (1+2\theta)u_M^{n+1} - \theta u_{M+1}^{n+1} &= u_M^n, \\ -\theta(u_{M-1}^{P,n+1} + \beta u_{M-1}^{H,n+1}) + (1+2\theta)(u_M^{P,n+1} + \beta u_M^{H,n+1}) - \theta(u_{M+1}^{P,n+1} + \beta u_{M+1}^{H,n+1}) &= u_M^n, \end{aligned}$$

$$\begin{aligned} \beta(-\theta u_{M-1}^{H,n+1} + (1+2\theta)u_M^{H,n+1} - \theta u_{M+1}^{H,n+1}) &= u_M^n + \theta u_{M-1}^{P,n+1} - (1+2\theta)u_M^{P,n+1} + \theta u_{M+1}^{P,n+1}, \\ \beta &= \frac{u_M^n + \theta u_{M-1}^{P,n+1} - (1+2\theta)u_M^{P,n+1} + \theta u_{M+1}^{P,n+1}}{-\theta u_{M-1}^{H,n+1} + (1+2\theta)u_M^{H,n+1} - \theta u_{M+1}^{H,n+1}}, \\ \beta &= \frac{u_M^n + \theta u_{M-1}^{P,n+1} - (1+2\theta)u_M^{P,n+1} + \theta u_1^{P,n+1}}{-\theta u_{M-1}^{H,n+1} + (1+2\theta)u_M^{H,n+1} - \theta u_1^{H,n+1}} \end{aligned}$$

since $u_{M+1}^{P,n+1} = u_1^{P,n+1}$ and $u_{M+1}^{H,n+1} = u_1^{H,n+1}$. Thus, we have β, then plug $\vec{u}^H$ and $\vec{u}^P$ in to get $\vec{u}$:

$$\vec{u} = \beta\vec{u}^H + \vec{u}^P.$$

Now we have solved the heat equation with periodic boundary conditions in 1-D.

2.5 Convergence, and Stability

2.5.1 Convergence

Definition: We can say a finite difference scheme is a convergent scheme if for any x and t, $(k\triangle x, (n+1)\triangle t)$ will converge to (x,t). u_k^n converges to the exact solution $u*$ as $\triangle x$ and $\triangle t$ converges to 0 [6, 13].

For an explicit method, first of all we denote v_k^n as

$$v_k^n = u_k^n - u *_k^n \quad \text{where} \quad u*_k^n = u^*(k\triangle x, n\triangle t).$$

For the heat equation, we can express u^{n+1} in terms of u^n using an explicit finite difference approximation as :

$$u*_k^{n+1} = (1-2\theta)u*_k^n + \theta(u*_{k+1}^n + u*_{k-1}^n) + O(\triangle t^2) + O(\triangle t \triangle x^2).$$

By subtracting the exact solution with the approximation, we have

$$v_k^{n+1} = (1-2\theta)v_k^n + \theta(v_{k+1}^n + v_{k-1}^n) + O(\triangle t^2) + O(\triangle t \triangle x^2).$$

Taking norms and using the triangle inequality, we can rewrite as:

$$\begin{aligned}
\|v_k^{n+1}\| &\leq (1-2\theta)\|v_k^n\| + \theta\|(v_{k+1}^n\| + \|v_{k-1}^n\|) + O(\triangle t^2 + \triangle t\triangle x^2),\\
&\leq (1-2\theta)V^n + 2\theta V^n + O(\triangle t^2 + \triangle t\triangle x^2),\\
&\leq V^n + O(\triangle t^2 + \triangle t\triangle x^2).
\end{aligned}$$

We denote V^n as $sup_k\{\|v_k^n\|\}$ and this gives us

$$\begin{aligned} V^{n+1} &\leq V^n + O(\triangle t^2 + \triangle t\triangle x^2), \\ &\leq V^{n-1} + 2O(\triangle t^2 + \triangle t\triangle x^2), \\ &\leq V^0 + (n+1)O(\triangle t^2 + \triangle t\triangle x^2), \\ &= (n+1)O(\triangle t^2 + \triangle t\triangle x^2) \quad (Since\ \ u^0 = u*^0). \end{aligned}$$

The error of this approximation will be

$$u_k^n - u*_k^n \leq (n+1)O(\triangle t^2 + \triangle t\triangle x^2) \to 0 \quad \text{as } \ \triangle t, \triangle x \to 0$$

which shows that the explicit method is a convergent method [7]. The conclusion still holds for an implicit method.

2.5.2 Stability

Definition: A finite difference scheme is said to be stable if for any positive time T, there is a constant K which is independent of n, k and for any initial data u^0 we have [12] :

$$\|u^k\| \leq K\|u^0\| \quad \forall \ 0 \leq k\triangle t \leq T.$$

The problem of stability is pervasive in the numerical solution of partial differential equations. Proving stability directly from the definition is quite difficult. Instead, it is easy to use tools from Fourier analysis to evaluate the stability of finite difference schemes [13].

As we know, the general solution to the heat equation can be decomposed into a sum over the various Fourier modes ,

$$u_k^n = Z^n e^{i\xi x_k}. \tag{19}$$

The Von Neumann stability criteria is for any norm of $\|Z\|$, we have [8]

$$\|Z\| \le 1 + \triangle\gamma$$

with γ an independent constant of k, $\triangle t$ and $\triangle x$. As a consistent scheme converges for $\triangle t, \triangle x \to 0$, and the boundary may be limited to

$$\|Z\| \le 1. \tag{20}$$

The equation of all u_k^{n+1} could be rewritten in Fourier form [8]

$$\begin{aligned} u_k^{n+1} &= (1-2\theta)u_k^n + \theta u_{k+1}^n + \theta u_{k-1}^n, \\ Z^{n+1}e^{i\xi x_k} &= (1-2\theta)Z^n e^{i\xi x_k} + \theta Z^n e^{i\xi x_{k+1}} + \theta Z^n e^{i\xi x_{k-1}}, \\ &= Z^n e^{i\xi x_k}(1 - 2\theta + \theta e^{i\triangle x} + \theta e^{-i\triangle x}). \end{aligned}$$

Based on $e^{i\theta} = cos\theta + isin\theta$, the formula will be

$$\begin{aligned} Z^{n+1}e^{i\xi x_k} &= Z^n e^{i\xi x_k}(1 - 2\theta + 2\theta cos\triangle x), \\ &= Z^n e^{i\xi x_k}(1 - 2\theta(1 - cos\triangle x)), \\ &= Z^n e^{i\xi x_k}(1 - 4\theta sin^2\frac{\triangle x}{2}). \end{aligned} \tag{21}$$

Since $\theta = \frac{\alpha\triangle t}{\triangle x^2}$ is positive, we have the conclusion that

$$1 - 4\theta sin^2\left(\frac{\triangle x}{2}\right) \le 1. \tag{22}$$

This satisfies the condition in (20), and according to (21) and (22) we have the following inequality:

$$Z^{n+1}e^{i\xi x_k} \leq Z^n e^{i\xi x_k},$$
$$Z \leq 1.$$

Thus,

$$\|u^k\| \leq \|u^{k-1}\|,$$
$$\|u^k\| \leq K\|u^0\|,$$

which shows the scheme is stable.

Richard Courant, Kurt Friedrichs, and Hans Lewy have pointed out that a great deal can be learned by considering the domains of dependence of a partial differential equation and of its discrete approximation. This requirement is known as the Courant-Friedrichs-Levy or CFL condition [5].

For the heat equations with the three boundary conditions we mentioned before, the CFL condition for one dimension is :

$$\theta = \frac{\alpha \triangle t}{(\triangle x)^2} \leq \frac{1}{2}.$$

This tells us a relationship between $\triangle t$ and $\triangle x$. In order to be stable, we need to set up $\triangle t$ and $\triangle x$ properly. For two dimensions, since the scheme is related to not only $\triangle x$ but also $\triangle y$, the CFL condition will be:

$$\theta = \frac{\alpha \triangle t}{(\triangle x)^2 + (\triangle y)^2} \leq \frac{1}{2}.$$

For the more common situation of i dimensions, we have

$$\alpha \sum_{i=1}^{n} \frac{\triangle t}{(\triangle x_i)^2} \leq \frac{1}{2}.$$

It should be noted that the CFL criteria is a necessary, but not a sufficient stability condition.

3 Solving Linear Equations

3.1 Iterative Methods

An iterative method is a mathematical procedure that generates a sequence of improving approximate solutions for problems. Generally, an iterative method includes the termination criteria and an initial guess. An iterative method is called convergent if the corresponding sequence converges for given initial approximations [6, 12].

An iterative method for solving a linear system $Ax = b$ (with A a square invertible matrix and b a vector), constructs an iteration series that under some conditions converges to the exact solution x for the system $Ax = b$. Thus, it is necessary to choose a starting point x_0 and iteratively apply a rule that computes x_{i+1} from an already known x_i. A starting vector x_0 is usually chosen as some approximation of the solution $x*$. The following is an iterative update:

$$x_{i+1} = Bx_i + Cb \quad i = 1, 2, ...,$$

where $B, C \in \mathbb{R}^{n\times n}, i \in \mathbb{N}$. Different choices of B and C define different iterative methods [1].

We can rewrite A in terms of three submatrices D, L, U, as

$$A = D + L + U,$$

where the only nonzero entries in D are the diagonal entries in A, the only nonzero entries in L are entries in A below the the diagonal, and the only nonzero entries in U are entries in A above the diagonal [16].

$$D = \begin{bmatrix} a_{1,1} & 0 & 0 & \dots & \dots & 0 \\ 0 & a_{2,2} & 0 & \dots & \dots & 0 \\ 0 & 0 & a_{3,3} & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & a_{n-1,n-1} & 0 \\ \vdots & \dots & \dots & \dots & 0 & a_{n,n} \end{bmatrix} L = \begin{bmatrix} 0 & 0 & 0 & \dots & \dots & 0 \\ a_{2,1} & 0 & 0 & \dots & \dots & 0 \\ a_{3,1} & a_{3,2} & 0 & \dots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ a_{n-1,1} & a_{n-1,2} & \ddots & \ddots & 0 & 0 \\ a_{n,1} & a_{n,2} & \dots & \dots & a_{n,n-1} & 0 \end{bmatrix}$$

$$U = \begin{bmatrix} 0 & a_{1,2} & a_{1,3} & \dots & \dots & a_{1,n} \\ 0 & 0 & a_{2,3} & \dots & \dots & a_{2,n} \\ 0 & 0 & 0 & \dots & \dots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & a_{n-2,n} \\ 0 & \ddots & \ddots & \ddots & 0 & a_{n-1,n} \\ 0 & 0 & \dots & \dots & 0 & 0 \end{bmatrix}$$

3.1.1 Convergence

Suppose the sequence $\{x_i\}_{i=0}^{\infty}$ converges to x, where

$$x_{i+1} = Bx_i + Cb. \tag{23}$$

If x_i converges to x as $i \to \infty$, then x satisfies the equation:

$$x = Bx + Cb. \tag{24}$$

Subtracting (24) from equation (23)

$$x_{i+1} - x = B(x_i - x),$$

and we will have following inequality by taking norms :

$$\|x_{i+1} - x\| \leq \|B\| \|x_i - x\|.$$

Suppose $\|B\| < 1$, which means

$$\|x_{i+1} - x\| < \|x_i - x\|.$$

Starting from an initial guess x_0, we get $x_1 - x = B(x_0 - x)$. We will have a monotonically decreasing sequence $\{\|x_i - x\|\}_{i=0}^{\infty}$ [1], and the error in the approximations is decreasing. Then:

$$\begin{aligned} x_{i+1} - x &= B(x_i - x), \\ &= B^2(x_{i-1} - x), \\ &= B^{i+1}(x_0 - x). \end{aligned}$$

Since $\|B\| < 1$,

$$\|x_i - x\| \leq \|B\| \|x_{i-1} - x\| \leq ..... \leq \|B\|^i \|x_0 - x\|.$$

Notice that $\|x_0 - x\|$ is a constant and $\|B\|^i \to 0$ as $i \to \infty$. So $\|B\| < 1$ is a necessary condition for convergence [1].

3.1.2 Jacobi Iterative Method

The Jacobi method is motivated by the following observation. Let A have nonzero diagonal elements (the rows of any nonsingular matrix can be reorganized to achieve this) [1] . Then $Ax = b$ can be rewritten as $Dx + (L + U)x = b$, thus ,

$$x = D^{-1}(-L - U)x + D^{-1}b.$$

Replacing x on the left-hand side by x_{i+1} and x on the right-hand side by x_i leads to the iteration formula of the Jacobi method:

$$x_{i+1} = -D^{-1}(L + U)x_i + D^{-1}b. \tag{25}$$

This formula also could be written as

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{i \neq j} a_{ij}x_j^{(k)}) \quad i, j = 1, 2...n. \tag{26}$$

The Jacobi method converges for any starting vector x_0 as long as $\|D^{-1}(L+U)\| < 1$ [5] . This condition is satisfied for a relatively large class of matrices including diagonally dominant matrices (matrices A such that $\sum_{j=1,j\neq i}^{n} |A_{ij}| \leq |A_{ii}|$ for $i = 1, \ldots, n$).

3.1.3 Gauss-Seidel Iterative Method

Analogous to the Jacobi method, we can rewrite $Ax = b$ as $(L + D)x + Ux = b$, which further implies $x = (L + D)^{-1}[-Ux + b]$. This leads to the iteration formula of the Gauss-Seidel method:

$$x_{i+1} = (L + D)^{-1}b - (L + D)^{-1}Ux_i, \tag{27}$$

or the recurrence would be:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{i<j} a_{ij}x_j^{(k+1)} - \sum_{i>j} a_{ij}x_j^{(k)}), i, j = 1, 2...n. \tag{28}$$

The main difference to the Jacobi method is that the Gauss-Seidel method is calculating a more efficient version of equation (26).

In Jacobi's method, the values of x_i^k obtained in the kth iteration remain unchanged until the entire $(k+1)$th iteration has been calculated. With the Gauss-Seidel method, we use the new values x_i^{k+1} as soon as they are known. For example, once we have computed x_1^{k+1} from the first equation, its value is then used in the second equation to obtain the new x_2^{k+1} [1].

3.2 Direct Methods

3.2.1 Gaussian Elimination

Gaussian elimination (also known as row reduction) is an algorithm for solving systems of linear equations. It is usually understood as a sequence of operations performed on the associated matrix of coefficients. This method is named after Carl Friedrich Gauss [4].

To illustrate with an example, we look at the following equation system for the system

$$2x + 5y = 12,$$

$$x - 3y = -5,$$

First, we can rewrite the matrix in augmented matrix form,

$$\begin{bmatrix} 2 & 5 & 12 \\ 1 & -3 & -5 \end{bmatrix}.$$

Then, we can use row operations to put the matrix in echelon form,

$$\begin{bmatrix} 2 & 5 & 12 \\ 1 & -3 & -5 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & -3 & -5 \\ 2 & 5 & 12 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & -3 & -5 \\ 0 & 11 & 22 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & -3 & -5 \\ 0 & 1 & 2 \end{bmatrix}$$

Here, row 1 has been exchanged with row 2 since the first element of row 2 is 1. Next, we calculate Row 2 -2 (Row 1) in order to let the first element of row 2 become 0. Notice the second element and third element of row 2 could be divided by 11, so we will rewrite them as 1 and 2. Now, our equations are:

$$x - 3y = -5,$$

$$y = 2.$$

It's easy to solve the equation since we know the value of y to get:

$$\begin{bmatrix} 1 & -3 & -5 \\ 0 & 1 & 2 \end{bmatrix} \implies x = 1, \;\; y = 2.$$

4 Comparison

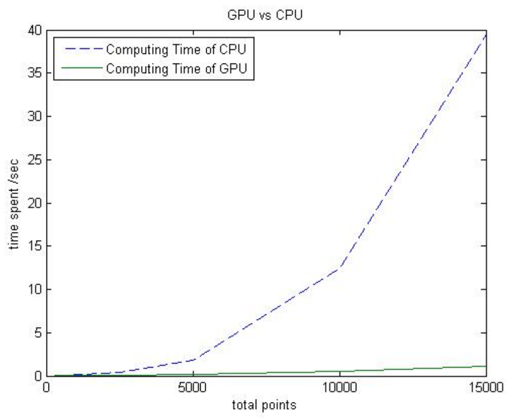
4.1 Multiplication

To optimize our algorithm, we need to make comparisons solving the heat equation with different boundary conditions on both the GPU and CPU. For all explicit methods, to get every u^n we need to do a multiplication between a large size matrix and a vector. To complete the multiplication directly on the CPU is not a smart choice since we can use row-based, column-based, or even a block-based strategy. A row-based multiplication with matrix of size $n \times n$ could be multiplication of a vector with n different vectors. Obviously, it's a parallel strategy. But as we mentioned before, copying data and transferring data to the GPU will take extra time. Thus to make a complete fairly comparison we need to record not only computing time but also the data transfer time. Code for this is in the Appendix 6.1 $matrix\ multiplication$.

In Table 2, we record the calculating time of Ax where A is a $n \times n$ matrix and x is a $n \times 1$ vector on the CPU and GPU. This is a common calculation in an explicit method. Figure 5 gives us a more direct observation.

size	300	625	1250	2500	5000	10000	15000
GPU computing time	0.0021	0.0025	0.00877	0.03292	0.1271	0.5046	1.1337
CPU computing time	0.0034	0.0510	0.1293	0.3972	1.8483	12.4439	39.4346
GPU total time	0.0027	0.0051	0.018571	0.08522	0.4380	2.5915	7.7294
CPU total time	0.0053	0.0595	0.165447	0.538	2.4119	14.6855	44.4883

Table 2: Time spent in seconds to complete multiplication by CPU and GPU.



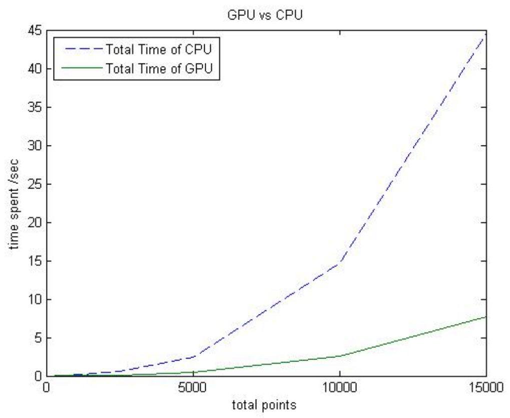


Figure 5: The plots show the calculation time spent by GPU is much less than CPU. The difference is significant when the size is larger than 10000 of data points. Although GPU takes extra time to transfer data, it's still a faster method compared with CPU. The graph on the left shows only the computing time and the graph on the right shows the total time, including data transfer.

Looking at Table 2 and Figure 5 , when the size of matrix is 300×300, computing time for both the CPU and GPU calculations are almost the same. But as the size grows to tens of thousands, the time spent for the GPU calculation is much less than the CPU. The GPU spent nearly 1 sec for size 15000 compared with 39 sec by CPU. Noticing that the total time of GPU calculation when the size is 15000, we find that the declare and gather time are the majority of the total time. Computing time is still small but the input and output time between device memory and GPU plays an important role.

4.2 Solving Linear Systems

Solving the heat equation with an explicit method, we will calculate many multiplications since we are solving u^n from $u^n = Au^{n-1}$ where A is a square coefficient matrix. The previous results showed the advantage of GPUs. But for the implicit method, we are solving u^n from $u^{n-1} = Au^n$, which means we are solving a system of linear equation in the form $Ax = b$ instead of a matrix multiplication.

Let's see the performance of the CPU and GPU for the Gauss-Seidel Method. Here we set up A as an $n \times n$ random matrix and b as a random vector with n dimensions. The code for this is in Appendix 6.2 $Gauss - Seidel\ \ Method$.

Size	50	125	250	500	1000	2000	4000
GPU time	0.2548	0.6221	1.2271	2.4433	4.8946	9.7800	20.3150
CPU time	0.0043	0.0089	0.0163	0.0497	0.1168	0.3200	1.3923

Table 3: CPU and GPU computing time of Gauss-Seidel Method in seconds.

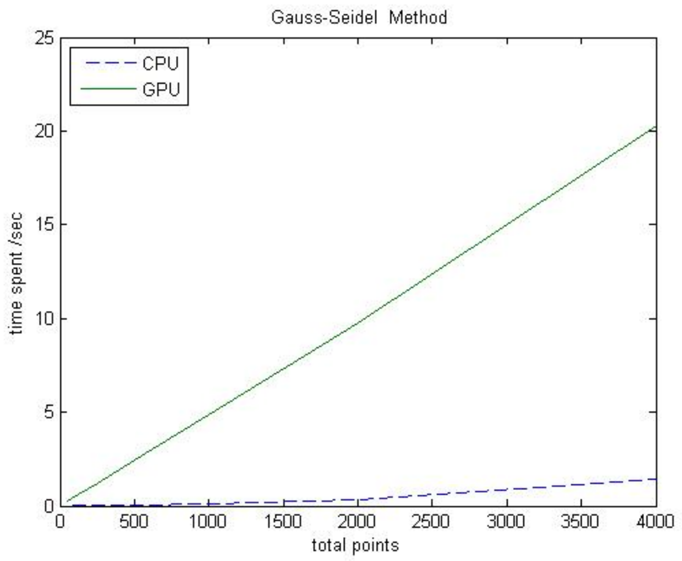


Figure 6: Solving linear system using Gauss-Seidel Method by CPU is always faster than GPU regardless of the size of the problem.

Looking at Table 3 and Figure 6, we find that solving the linear system $Ax = b$ by Gauss-Seidel Method on the CPU and GPU separately, no matter the size of the system, the CPU takes less time since the method is not parallel. As we said, the GPU performs better than CPU for parallel method, while the Gauss-Seidel Method is a one-thread method.

The GPU is a better choice for multiplication Ax while the CPU runs faster for solving linear equations $u^{n-1} = Au^n$ by the Gauss-Seidel Method. But we need to point out the fact that when solving $u^{n-1} = Au^n$, the square coefficient matrix A is fixed and nonsingular. According to the following equations, we know once we have the inverse of A, the problem will be almost the same one as multiplications.

$$\begin{aligned} u^{n-1} &= Au^n \\ A^{-1}u^{n-1} &= A^{-1}Au^n \\ u^n &= A^{-1}u^{n-1} \end{aligned}$$

According to Table 2, we know the time for GPU to calculate a multiplication like Ax of size 2500 using a row-based strategy is 0.08522 sec and 0.4380 sec for size 5000. In contrast, from Table 3,we know the CPU takes 0.32 sec to solve a linear system like $Ax = b$ of size 2000 with the Gauss-Seidel Method, 1.39 sec for a size of 4000. We can see the fact that the GPU takes less time to do the calculation. The main problem left is the time it takes to calculate A^{-1}.

Size	300	625	1250	2500	5000	10000
time spent	0.0097	0.0196	0.0412	0.1549	0.7199	4.2432

Table 4: GPU computing time in seconds for a matrix inverse by Gaussian Elimination method.

Table 4 shows the GPU computing time of a matrix inverse by Gaussian Elimination method. From the data, we can say if we are solving the heat equation with size no more than 10000 or only several time step like only u^1, u^2, ..., u^5, using CPU with the Gauss-Seidel Method is a wise choice that takes less time. On the contrast, when the problem size is very large and the number of time steps to be evaluated is large, the GPU with inverse A is a more practical choice.

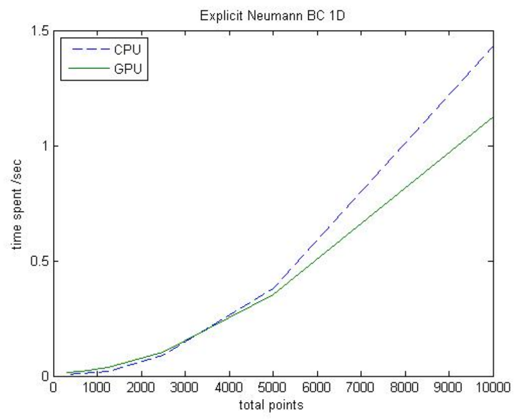
4.3 Heat Equation

Finally, let's see how the GPU perform in the 1-D heat equation with different boundary conditions compared with the CPU. We set up time steps N as 20 for all boundary conditions with both implicit and explicit methods. Codes can be found in Appendix 6.3 $1D\ Neumann\ BC$, 6.4 $1D\ Dirichlet\ BC$ and 6.5 $Periodic\ BC$.

Table 5 shows us the time for the CPU and GPU to complete an entire program of the 1-D heat equation approximation with different number of points M for the spatial discretization.. In the table, for short, we define Exp as the explicit method, Imp as the implicit method, Neu as the Neumann boundary condition, Dir as the Dirichlet boundary condition. As an example, Exp Neu CPU means the time for the CPU to calculate the heat equation with Neumann boundary condition by an explicit method. The reason why we don't have the data for implicit method for size 10000 is because it will run out of machine memory.

Size	300	625	1250	2500	5000	10000
Exp Neu CPU	0.0055	0.0110	0.0213	0.0898	0.3785	1.4320
Exp Neu GPU	0.0166	0.0205	0.0377	0.1009	0.3522	1.1239
Imp Neu CPU	0.0181	0.0821	0.2697	1.0198	3.8142	
Imp Neu GPU	0.0227	0.0456	0.0817	0.2512	1.0641	
Exp Dir CPU	0.0051	0.0101	0.0193	0.0838	0.3145	1.2320
Exp Dir GPU	0.0136	0.0198	0.0307	0.0902	0.3122	1.1129
Imp Dir CPU	0.0173	0.0791	0.2433	1.008	3.4112	
Imp Dir GPU	0.0207	0.0421	0.0794	0.2311	1.01	
Periodic CPU	0.0092	0.0259	0.0812	0.3828	1.8442	
Periodic GPU	0.0667	0.0891	0.1326	0.2993	1.1353	

Table 5: The heat equation with three different boundary conditions in one dimension on GPU and CPU. All time steps are 20 and calculation time is reported in seconds.



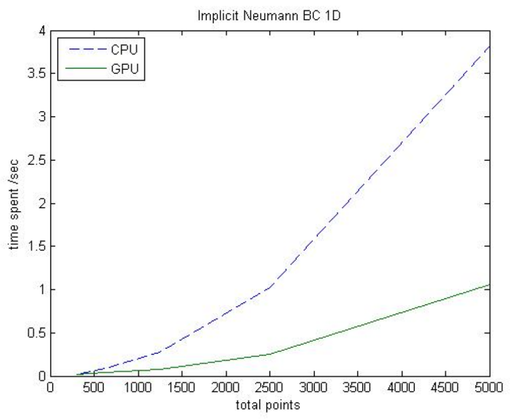


Figure 7: For the Neumann boundary condition in 1-D, time spent by the GPU and CPU are almost the same with the explicit method (left), but the GPU is significantly faster than CPU with implicit method (right).

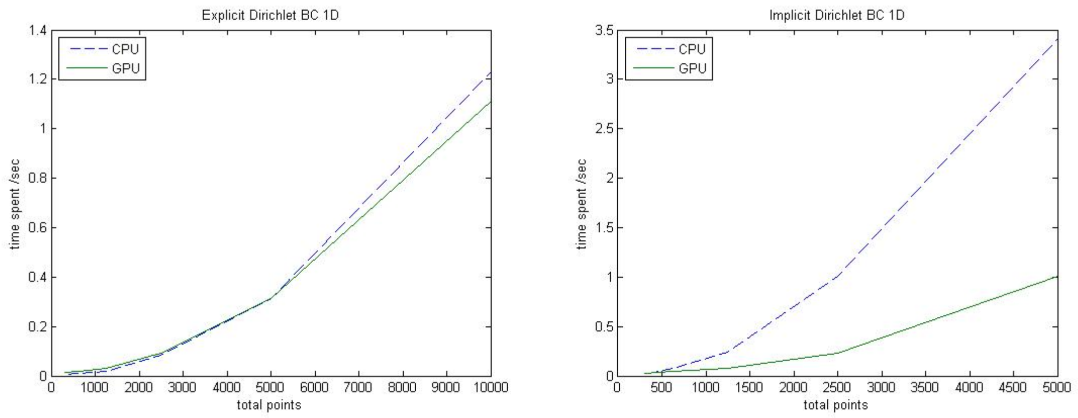


Figure 8: The left figure is using the explicit method while the right one is using the implicit method. The result of Dirichlet B.C in 1-D are similar to Neumann B.C. The GPU is faster when we look at the implicit method.

From Table 5, we know for both the Neumann and Dirichlet boundary conditions, using the explicit method takes shorter time for both the GPU and CPU than the implicit method. Time for the GPU to do the calculation is nearly the same with the CPU. Looking at the Figure 7 and Figure 8, the GPU line is not far away from the CPU line though the time for the GPU to complete the calculation is much less than for CPU when the problem size is large. Also, the table and figures tell us that with the implicit method the calculation time of the CPU is almost three times of the GPU for both Neumann and Dirichlet conditions.

When the problems turn to 2 Dimensions, let's see how the final results vary. Codes are much more complicated in 2-D, we have to reflect all points of 2-D to a matrix according to some strategies, thus the inner points will initialized by the boundary conditions compared with 1-D problems. Codes are in the Appendix 6.6 $2D\ Neumann\ BC$ and 6.7 $2D\ Dirichlet\ BC$. For the GPU part, since we are using the explicit method, all the multiplication are calculated by row- based strategy. Table 6 is the calculation time with the explicit method and Table 7 is with the implicit method. 40×40 means there is a 40 point partition in x and 40

point partition in y. Figure 9 is the graphic of Table 6 and Figure 10 is the graphic of Table 7.

Size	40×40	60×60	80×80	100×100	120×120	140×140	160×160
Exp Dir CPU	0.0288	0.1483	0.3867	0.8266	1.6590	2.9204	4.8497
Exp Dir GPU	0.0274	0.0850	0.1991	0.4384	0.8597	1.5299	2.5686
Exp Neu CPU	0.0308	0.1521	0.4043	0.8606	1.8659	3.1287	5.1815
Exp Neu GPU	0.0292	0.08901	0.2110	0.4884	0.9091	1.7298	2.7686

Table 6: The heat equation with the Neumann B.C and Dirichlet B.C in two dimensions with the explicit method. All time steps are 20 and calculation time is reported in seconds.

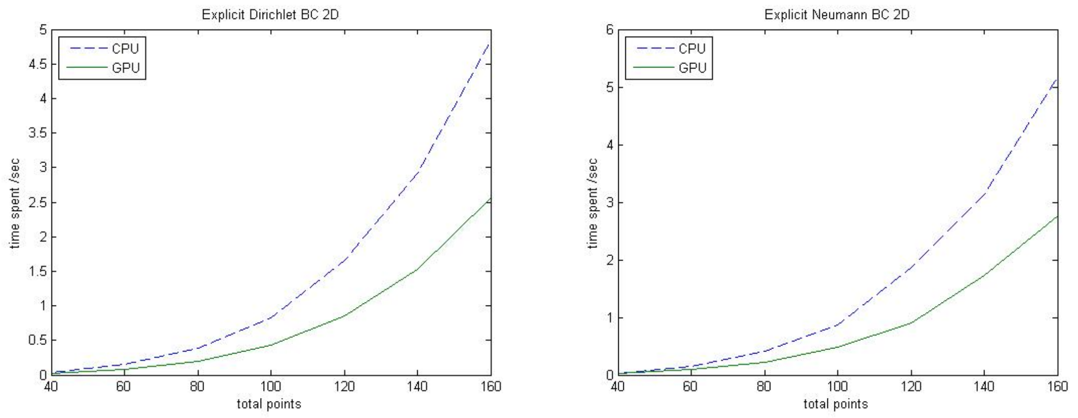


Figure 9: In 2D explicit case, no matter the size of the problem, the GPU took nearly half time as CPU did. The heat equation with Dirichlet B.C on the left and Neumann B.C on the right.

Size	40×40	60×60	80×80	100×100
Imp Dir CPU	0.1792	1.8424	3.7297	11.4264
Imp Dir GPU	0.1211	0.6151	2.0244	5.7823
Imp Neu CPU	0.1831	1.9441	3.8212	13.1431
Imp Neu GPU	0.149	0.67	2.1541	5.9801

Table 7: The heat equation with the Neumann B.C and Dirichlet B.C in two dimensions with the implicit method. All time steps are 20 and calculation time is reported in seconds.

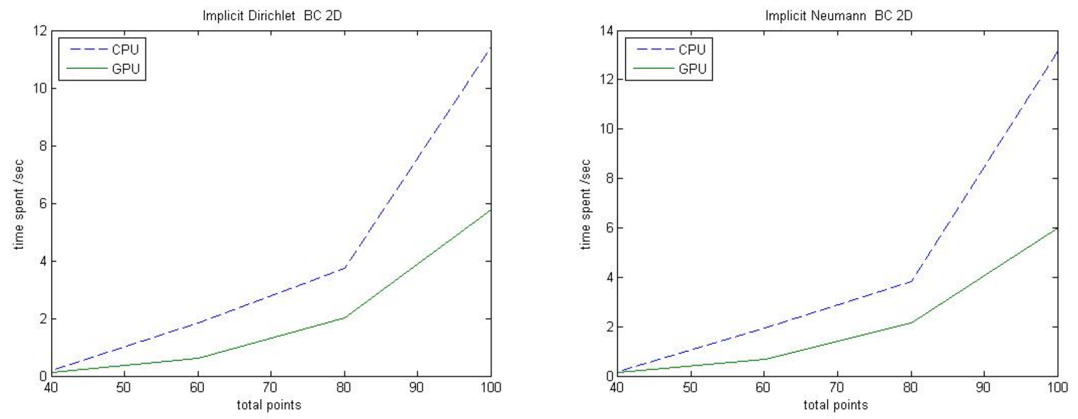


Figure 10: In 2-D implicit case, GPU is still faster than CPU, but the time to complete the calculations and the problem size are not linearly related. The left figure is the Dirichlet B.C and the right is the Neumann B.C.

From Table 6 and Table 7, the 2-D problem, we observe that the implicit method still takes a longer time to complete the calculation for both the GPU and CPU. In 2-D, the calculation time of the GPU is half of the calculation time of the CPU with explicit method. Both Figure 9 and Figure 10 show us that the GPU is much better than the CPU in terms of calculation times. For both the implicit and explicit methods, the GPU calculated the solution faster than the CPU for a large number of points. Hence, if we are dealing heat equations on a fine grid with thousands or millions of discretization points, of course GPU will cost less time.

For small size like hundreds or even less, time spent by GPU is almost the same with CPU.

5 Conclusion

The results in this report are from the codes in *MATLAB* that were written and summarized in the Appendices. For this code, we find that solving the heat equation with various boundary conditions, the implicit method and explicit method have differences on the GPU and CPU. Although in doing matrix multiplications, the GPU takes much less time than the CPU does, the advantage is not that remarkable when the data transferring time is taken into consideration. The GPU can solve the heat equation with an explicit method in a shorter time than the CPU, but the advantage is not as obvious as the implicit method.

For implicit method, if we use the Jacobi iterative method or the Gauss-Seidel iterative method, the CPU is always faster than the GPU regardless of problem size. Because they are methods run by a single core and can't be parallelized. But if we choose a row-based strategy, we find that the GPU showed a huge superiority compared with the CPU. Because for the CPU to solve a linear system as $Ax = b$, it takes much longer time even the data transferring time of the GPU is taken into considerations. As the the size of problem goes to larger or the dimensions goes to larger (both of these mean the size of the matrix goes to larger), the GPU will be a better choice compared with the CPU in terms of calculation time.

Using other methods to solve complete the multiplication and solve the linear systems with CUDA provided by Nvidia, or other parallel computing platform besides CUDA, may cause different results. For a matrix with certain properties, we may have a different result as solving the dense matrix with Gauss-Seidel method [5]. The results from this report might not be true for all cases and the codes could be written more optimally.

References

[1] L Argyros. *Computational Theory of Iterative Methods*. Elsevier B.V, Atlanta, GA, 2007.

[2] S Chapman. *MATLAB Programming for Engineers*. Cengage Learning, Boston MA, 2007.

[3] Z Fan, F Qiu, A Kaufman, and S Suzanne. Gpu cluster for high performance computing. *ACM / IEEE Supercomputing Conference*, 45, 2004.

[4] A Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[5] C Hadrien and J Allard. Parallel dense gauss-seidel algorithm on many-core processors. *High Performance Computation Conference*, 2009.

[6] JD Hoffman and S Frankel. *Numerical Methods for Engineers and Scientists*. McGraw-Hill Inc, Switzerland, 2001.

[7] MM Meerschaert and C Tadjeran. *Finite difference approximations for two-sided space-fractional partial differential equations*. Elsevier, Atlanta, GA, 2006.

[8] AR Mitchell and DF Griffiths. *The finite difference method in partial differential equations*. John Wiley, New York, 1980.

[9] Nvidia. *MATLAB for CUDA Development*, 2015. `https://developer.nvidia.com/matlab-cuda`.

[10] O Peter. *Introduction to Partial Differential Equations*. Springer, New York,USA, 2014.

[11] J Pizzini. *GPU Rendering vs CPU Rendering A method to compare render times with empirical benchmarks*, 2014. `http://blog.boxxtech.com/...gpu-rendering-vs-cpu-rendering-with-empirical-benchmarks`.

[12] A Quarteroni, R Sacco, and F Saleri. *Numerical Mathematics*. Springer Berlin Heidelberg, USA, 2007.

[13] J Randall and LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 2007.

[14] J Reesea. *GPU Programming in MATLAB*, 2015. `http://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html`.

[15] A Rege. *An Introduction to Modern GPU Architecture*, 2015. `ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf`.

[16] Y Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[17] L Shure. *Using parfor Loops: Getting Up and Running*, 2009. `http://blogs.mathworks.com/loren/2009/10/02/using-parfor-loops-getting-up-and-running/`.

[18] N Sonawane and B Nandwalkar. *Time Efficient Sentinel Data Mining using GPU*, 2015.

[19] DA Yuen, L Wang, X Chi, L Johnsson, W Ge, and Y Shi. *Algorithms and Architectures for Parallel Processing*. Springer, New York City, 2012.

6 Appendix

6.1 matrix multiplication

```
clear
clc

tic
L=10000

A=gpuArray.rand(L,L);
B=gpuArray.rand(L,1);
creat_time=toc

tic
C=A*B;
n=norm(C,'fro');
 compute_time=toc

tic
D=gather(n);
gather_time=toc
total_time=creat_time+compute_time+gather_time

fid = fopen('data.txt', 'a');
fprintf(fid, 'The size of matrices is \n%f\n',L);
fprintf(fid, 'Time costs by computing is %f\n', compute_time);
fprintf(fid, 'Time costs by gathering is %f\n', gather_time);
fprintf(fid, 'Total time costs is %f\n', total_time);
fclose(fid);
```

```
clear
clc

tic
L=5000*3
A=rand(L,L);
B=rand(L,1);
 C=A*B;
 cpu_time=toc

 fid = fopen('data.txt', 'a');
fprintf(fid, 'The size of matrices is \n%f\n',L);
fprintf(fid, 'Time costs by creating is %f\n', creat_time);
fprintf(fid, 'Total time costs is %f\n', total_time);
fprintf(fid, 'Time costs by computing is %f\n', cpu_time);
fclose(fid);
```

6.2 Gauss-Seidel

```

clear
clc


tic
n=50;  % set up the size of matrix
nOnes = ones(n, 1) ;
AA = diag(12 * nOnes, 0) - diag(nOnes(1:n-1), -1) - diag(nOnes(1:n-1),
    1);
```

```
% set up a diagonal matrix
A=gpuArray(AA);

  b=gpuArray.rand(n,1);     % the b of equation Ax=b
  x=gpuArray.zeros(n,1);     % the final result
  x1=gpuArray.zeros(n,1);    %  result of last step
  x2=gpuArray.zeros(n,1);    %  result of this step
  D=gpuArray.zeros(n,1);      % the diagonal elements


D=diag(A);
A=A-diag(D);
D=1./D;
tol=10;         % initial tolerance
x1=x;
count=0;       % count #
x2=x;
while tol>10^-10
  x2=x1;
for k=1:n       %main method
 x(k)=(b(k)-A(k,:)*x)*D(k);
end
x1=x;
tol=max(abs(x1-x2));
count=count+1;
end
count;
time=toc

max(abs(AA*x-b))  % check the answer
```

```

clear
clc


tic
n=50;  % set up the size of matrix
nOnes = ones(n, 1) ;
AA = diag(12 * nOnes, 0) - diag(nOnes(1:n-1), -1) - diag(nOnes(1:n-1),
    1);
% set up a diagonal matrix
A=AA;

   b=rand(n,1);     % the b of equation Ax=b
   x=zeros(n,1);     % the final result
   x1=zeros(n,1);    %  result of last step
   x2=zeros(n,1);    %  result of this step
   D=zeros(n,1);      % the diagonal elements

D=diag(A);
A=A-diag(D);
D=1./D;
tol=10;         % initial tolerance
x1=x;
count=0;       % count #
x2=x;
while tol>10^-10
   x2=x1;
for k=1:n       %main method
 x(k)=(b(k)-A(k,:)*x)*D(k);
end
```

```
x1=x;
tol=max(abs(x1-x2));
count=count+1;
end
count;
time=toc

max(abs(AA*x-b))  % check the answer
```

6.3 1D Neumann BC

```
function exp_neu_1d_cpu
%--------------Coded  junchi Zhang
%--------------Instructor: Prof Sarah Olson
clear
clc



tic
chi=0;
omega=2;

%------define the the length and steps
alpha=1;
X=1;
%# of grid points on the domain, 1-d line
M=10000
%spacing between the M points on the 1-d line
```

```
dx=X/M;

%----set up the proper dt
dt=((dx)^2)/(6*alpha);
N=20;
%-----set up the boundary conditions
A=zeros(M+1,N+1);
for i=1:M+1
    A(i,1) = cos(pi*(i-1)*dx)+1+((i-1)*dx)^2;
end

%-----put the difference method in a matrix
theta=(dt*alpha)/((dx)^2);
D=zeros(M+1,M+1);
D(1,1)=1-2*theta;
D(1,2)=2*theta;
D(M+1,M)=1-2*theta;
D(M+1,M+1)=2*theta;
for i=2:M
    D(i,i)=1-2*theta;
    D(i,i-1)=theta;
    D(i,i+1)=theta;
end


%-----calculate all points
 for i=2:N+1
     A(:,i)=D*A(:,i-1);
     A(1,i)=A(1,i)-2*chi*dx*theta;
     A(M+1,i)=A(M+1,i)-2*omega*dx*theta;
 end
 for  i=1:M+1
      for  j=1:N+1
 Exact_A(i,j)=exp((-pi)*pi*(j-1)*dt)*cos(pi*dx*(i-1))+1+(dx*(i-1))
     ^2+2*(j-1)*dt;
      end
 end
time=toc
```

```
function  exp_neu_1d_gpu
%—————Coded  junchi  Zhang
%—————Instructor:  Prof  Sarah  Olson
clear
clc



tic
chi=0;
omega=2;

%———define  the  the  length  and  steps
alpha=1;
X=1;
%#  of  grid  points  on  the  domain,  1-d  line
M=625*2*2*2
%spacing  between  the  M  points  on  the  1-d  line
dx=X/M;

%——set  up  the  proper  dt
```

```
dt=((dx)^2)/(6*alpha);
N=20;
%------set up the boundary conditions
A=zeros(M+1,N+1);
for i=1:M+1
    A(i,1) = cos(pi*(i-1)*dx)+1+((i-1)*dx)^2;
end

%------put the difference method in a matrix
theta=(dt*alpha)/((dx)^2);
D=zeros(M+1,M+1);
D(1,1)=1-2*theta;
D(1,2)=2*theta;
D(M+1,M)=1-2*theta;
D(M+1,M+1)=2*theta;
for i=2:M
    D(i,i)=1-2*theta;
    D(i,i-1)=theta;
    D(i,i+1)=theta;
end

AA=gpuArray(A);
DD=gpuArray(D);
%------calculate all points
 for i=2:N+1
     AA(:,i)=DD*A(:,i-1);
     AA(1,i)=AA(1,i)-2*chi*dx*theta;
     AA(M+1,i)=AA(M+1,i)-2*omega*dx*theta;
 end

 for i=1:M+1
```

```
      for j=1:N+1
  Exact_A(i,j)=exp((-pi)*pi*(j-1)*dt)*cos(pi*dx*(i-1))+1+(dx*(i-1))
      ^2+2*(j-1)*dt;
      end
  end

time=toc
```

```
function imp_neu_1d_cpu
%---------Coded  junchi Zhang
%---------Instructor: Prof Sarah Olson
clear
clc
tic
chi=0;
omega=2;

%-----define the the length and steps
alpha=1;
X=1;
%# of grid points on the domain, 1-d line
M=300
%spacing between the M points on the 1-d line
dx=X/M;

%---set up the proper dt
dt=((dx)^2)/(6*alpha);
N=20;
%----set up the boundary conditions
A=zeros(M+1,N+1);
```

```
for i=1:M+1
    A(i,1) = cos(pi*(i-1)*dx)+1+((i-1)*dx)^2;
end

%----put the difference method in a matrix
theta=(dt*alpha)/((dx)^2);
D=zeros(M+1,M+1);
D(1,1)=1+2*theta;
D(1,2)=-2*theta;
D(M+1,M)=-2*theta;
D(M+1,M+1)=1+2*theta;
for i=2:M
    D(i,i)=1+2*theta;
    D(i,i-1)=-theta;
    D(i,i+1)=-theta;
end


%----calculate all points
 for i=2:N+1

      veca=A(:,i-1);
      veca(1)=veca(1)-2*chi*dx*theta;
      veca(M+1)=veca(M+1)-2*omega*dx*theta;
   A(:,i)=D\veca;
 end

 for i=1:M+1
      for j=1:N+1
 Exact_A(i,j)=exp((-pi)*pi*(j-1)*dt)*cos(pi*dx*(i-1))+1+(dx*(i-1))
     ^2+2*(j-1)*dt;
```

```
        end
    end
 time=toc
```

```
function imp_neu_1d_gpu
%————Coded  junchi Zhang
%————Instructor: Prof Sarah Olson
clear
clc
tic
chi=0;
omega=2;

%———define the the length and steps
alpha=1;
X=1;
%# of grid points on the domain, 1-d line
M=300
%spacing between the M points on the 1-d line
dx=X/M;

%——set up the proper dt
dt=((dx)^2)/(6*alpha);
N=20;
%———set up the boundary conditions
A=zeros(M+1,N+1);
for i=1:M+1
    A(i,1) = cos(pi*(i-1)*dx)+1+((i-1)*dx)^2;
end

```

```
%——put the difference method in a matrix
theta=(dt*alpha)/((dx)^2);
D=zeros(M+1,M+1);
D(1,1)=1+2*theta;
D(1,2)=-2*theta;
D(M+1,M)=-2*theta;
D(M+1,M+1)=1+2*theta;
for i=2:M
    D(i,i)=1+2*theta;
    D(i,i-1)=-theta;
    D(i,i+1)=-theta;
end

AA=gpuArray(A);
DD=gpuArray.eye(M+1)/gpuArray(D);
veca=gpuArray.zeros(M+1,1);
%——calculate all points
 for i=2:N+1

      veca=AA(:,i-1);
      veca(1)=veca(1)-2*chi*dx*theta;
      veca(M+1)=veca(M+1)-2*omega*dx*theta;
   AA(:,i)=DD*veca;
 end

 for i=1:M+1
      for j=1:N+1
 Exact_A(i,j)=exp((-pi)*pi*(j-1)*dt)*cos(pi*dx*(i-1))+1+(dx*(i-1))
     ^2+2*(j-1)*dt;
      end
 end
```

```
time=toc
```

6.4 1D Dirichlet BC

```
function exp_dir_1d_cpu
%-----------Coded  by junchi
%-----------Instructor: Prof Sarah Olson
clear

clc
tic
%------define the the length and steps
alpha=1;
X=2;
%# of grid points on the domain, 1-d line
M=80;
%spacing between the M points on the 1-d line
dx=X/M;

%----set up the proper dt
dt=((dx)^2)/(6*alpha);
N=15;
%-----set up the boundary conditions
A=zeros(M+1,N+1);
for i=1:M+1
    A(i,1) = sin(pi*(i-1)*dx)+1;
end
for i=1:N+1
    A(1,i)=1;
```

```
        A(M+1,i)=1;
end
%-----put the difference method in a matrix
a=1-2*((dt*alpha)/(dx)^2);
b=(dt*alpha)/(dx)^2;
D=zeros(M+1,M+1);
D(1,1)=a;
D(1,2)=b;
D(M+1,M)=b;
D(M+1,M+1)=a;
for i=2:M
        D(i,i)=a;
        D(i,i-1)=b;
        D(i,i+1)=b;
end
%-----calculate all points
  for i=2:N+1
          A(:,i)=D*A(:,i-1);
          A(1,i)=1;
        A(M+1,i)=1;
  end

  for i=1:M+1
          for j=1:N+1
  Exact_A(i,j)=exp((-pi)*pi*(j-1)*dt)*sin(pi*dx*(i-1))+1;
          end
  end

  time=toc
```

```
function exp_dir_1d_gpu
%———————Coded  by junchi
%———————Instructor: Prof Sarah Olson
clear

clc
tic
%————define the the length and steps
alpha=1;
X=2;
%# of grid points on the domain, 1−d line
M=80;
%spacing between the M points on the 1−d line
dx=X/M;

%——set up the proper dt
dt=((dx)^2)/(6*alpha);
N=15;
%———set up the boundary conditions
A=zeros(M+1,N+1);
for i=1:M+1
    A(i,1) = sin(pi*(i−1)*dx)+1;
end
for i=1:N+1
    A(1,i)=1;
    A(M+1,i)=1;
end
%———put the difference method in a matrix
a=1−2*((dt*alpha)/(dx)^2);
b=(dt*alpha)/(dx)^2;
D=zeros(M+1,M+1);
```

```
D(1,1)=a;
D(1,2)=b;
D(M+1,M)=b;
D(M+1,M+1)=a;
 for  i=2:M
       D(i,i)=a;
       D(i,i-1)=b;
       D(i,i+1)=b;
end
AA=gpuArray(A);
DD=gpuArray(D);
%----calculate all points
  for  i=2:N+1
        AA(:,i)=DD*AA(:,i-1);
        AA(1,i)=1;
       AA(M+1,i)=1;
  end

  for  i=1:M+1
        for  j=1:N+1
  Exact_A(i,j)=exp((-pi)*pi*(j-1)*dt)*sin(pi*dx*(i-1))+1;
        end
  end

  time=toc
```

6.5 Periodic BC

```
function PeriodicBC_cpu
```

```
%-----------Coded  by junchi
%-----------Instructor: Prof Sarah Olson
clear
clc
tic
%------define the the length and steps
alpha=1;
X=1;
%# of grid points on the domain, 1-d line
M=625*2*2*2
%spacing between the M points on the 1-d line
dx=X/M;

%---set up the proper dt
dt=((dx)^2)/(6*alpha);
N=20;
%----set up the boundary conditions
A=zeros(M+1,N+1);

for i=1:M+1
    A(i,1) = sin(2*pi*(i-1)*dx);
    %above is the initial condition at t=0
    %solution: e^()*sin()
end

for i=1:N+1
    %BC at x=0, e^()*sin(0)=0
    A(1,i)=0;
    %BC at x=1, e^*sin(2*pi)=0
    A(M+1,i)=0;
end
```

```
%-----put the difference method in a matrix
 a=1+2*((dt*alpha)/(dx)^2);
b=-(dt*alpha)/((dx)^2);

D=zeros(M-1,M-1);
D(1,1)=a;
D(1,2)=b;
D(M-1,M-1)=a;
D(M-1,M-2)=b;

 for i=2:M-2
     D(i,i)=a;
     D(i,i-1)=b;
     D(i,i+1)=b;
end
%----set the RHS of the homogeneous equations
R=zeros(M-1,1);
R(1,1)=-b;
R(M-1,1)=-b;
%-----calculate all points
invD=eye(M-1)/D;
 for i=2:N+1
     %--the homogeneous one

     XH(1,1)=1;
     XH(M+1,1)=1;

     XH(2:M,1)=invD*R;
     %--the particular one
     XP(1,1)=1;
     XP(M+1,1)=1;
```

```
    %below is new line of code bringing in i-1 column, which should be
        data
    %values at previous time step

    RH=A(2:M,i-1);
      RH(1)=RH(1)-b;
    RH(M-1)=RH(M-1)-b;
    XP(2:M,1)=invD*RH;

    %below line was what was there - bringing a vectors of zeros

 %---the coefficient
beta=(A(M+1,i)+b*XP(M)-a*XP(M+1)+b*XP(2))/(-b*XH(M)+a*XH(M+1)-b*XH(2));
A(:,i)=XP+beta*XH;
end
for i=3:N+1
      A(2,i-1)=A(2,i-1)+b;
    A(M,i-1)=A(M,i-1)+b;
end

for i=1:M+1

    A(i,1) = sin(2*pi*(i-1)*dx);
    %above is the initial condition at t=0
    %solution: e^()*sin()
end
 for i=1:M+1
        for j=1:N+1
 Exact_A(i,j)=exp((-pi)*(pi)*2*2*(j-1)*dt)*sin(2*pi*dx*(i-1));
        end
 end
```

```
time=toc
```

```
function PeriodicBC_gpu
%————————Coded  by junchi
%————————Instructor: Prof Sarah Olson
clear
clc
tic
%————define the the length and steps
alpha=1;
X=1;
%# of grid points on the domain, 1−d line
M=300
%spacing between the M points on the 1−d line
dx=X/M;

%——set up the proper dt
dt=((dx)^2)/(6*alpha);
N=20;
%————set up the boundary conditions
A=zeros(M+1,N+1);

for i=1:M+1
    A(i,1) = sin(2*pi*(i−1)*dx);
    %above is the initial condition at t=0
    %solution: e^()*sin()
end

for i=1:N+1
    %BC at x=0, e^()*sin(0)=0
    A(1,i)=0;
    %BC at x=1, e^*sin(2*pi)=0
    A(M+1,i)=0;
end
%----put the difference method in a matrix
a=1+2*((dt*alpha)/(dx)^2);
b=-(dt*alpha)/((dx)^2);

D=zeros(M-1,M-1);
D(1,1)=a;
D(1,2)=b;
D(M-1,M-1)=a;
D(M-1,M-2)=b;

for i=2:M-2
    D(i,i)=a;
    D(i,i-1)=b;
    D(i,i+1)=b;
end
%---set the RHS of the homogeneous equations
R=zeros(M-1,1);
R(1,1)=-b;
R(M-1,1)=-b;
%----calculate all points
XH=gpuArray.zeros(M+1,1);
XP=gpuArray.zeros(M+1,1);
RH=gpuArray.zeros(M+1,1);
AA=gpuArray(A);
DD=gpuArray.eye(M-1)/D;
for i=2:N+1
    %--the homogeneous one
```

```
        XH(1,1)=1;
        XH(M+1,1)=1;

        XH(2:M,1)=DD*R;
        %--the particular one
        XP(1,1)=1;
        XP(M+1,1)=1;
        %below is new line of code bringing in i-1 column, which should be
            data
        %values at previous time step

        RH=AA(2:M,i-1);
          RH(1)=RH(1)-b;
        RH(M-1)=RH(M-1)-b;
        XP(2:M,1)=DD*RH;

        %below line was what was there - bringing a vectors of zeros

  %----the coefficient
 beta=(AA(M+1,i)+b*XP(M)-a*XP(M+1)+b*XP(2))/(-b*XH(M)+a*XH(M+1)-b*XH(2))
      ;
AA(:,i)=XP+beta*XH;
end
 for  i=3:N+1
          AA(2,i-1)=AA(2,i-1)+b;
        AA(M,i-1)=AA(M,i-1)+b;
end

 for  i=1:M+1
```

```
    A(i,1) = sin(2*pi*(i-1)*dx);
    %above is the initial condition at t=0
    %solution: e^()*sin()
end
 for i=1:M+1
      for j=1:N+1
 Exact_A(i,j)=exp((-pi)*(pi)*2*2*(j-1)*dt)*sin(2*pi*dx*(i-1));
      end
 end
time=toc
```

6.6 2D Neumann BC

```
function exp_neu_2d_cpu
%----------Coded by junchi Zhang
%----------Instructor: Prof Sarah Olson
clear
close all
clc


tic
%------define the the length and steps
alpha=1;
X=1;
Y=1;
%# of grid points on the domain x
M=40;
```

```
%# of grid points on the domain y
L=40;
% # of time step
N=20;

%spacing between the M points on x
dx=X/M;
%spacing between the M points on x
dy=Y/L;

%----set up the proper dt
dt=((dx)^2)/(6*alpha);

%------denote total points as tp
tp=(M+1)*(L+1);


%------put the difference method in a matrix
bx=(dt*alpha)/(dx)^2;
by=(dt*alpha)/(dy)^2;
a=1-2*((dt*alpha)/(dx)^2)-2*((dt*alpha)/(dy)^2);


D=zeros(tp,tp);
%------set up the boundary conditions
A=zeros(tp,N+1);
for j=1:L+1
    for i=1:M+1
        A((j-1)*(M+1)+i,1)= cos(pi*(i-1)*dx)*cos(pi*(j-1)*dy);
    end
end
```

```
for i=1:N+1
    for j=1:M+1
        A(j,i)=((j-1)*dy)^2; %the boundary conditions of bottom
        A(L*(M+1)+j,i)=2+((j-1)*dy)^2; %the boundary conditions of top
    end
    for k=1:L
        A(k*(M+1),i)=2+((k-1)*dx)^2; %the boundary conditions of right
        A(k*(M+1)+1,i)=((k-1)*dx)^2; %the boundary conditions of left
    end
end


% set up the coefficient matrix for the explicit parts
for j=1:L-1
    for i=2:M
   D(j*(M+1)+i,j*(M+1)+i)=a;
   D(j*(M+1)+i,j*(M+1)+i-1)=by;
   D(j*(M+1)+i,j*(M+1)+i+1)=by;
   D(j*(M+1)+i,j*(M+1)+i-M-1)=bx;
   D(j*(M+1)+i,j*(M+1)+i+M+1)=bx;
    end
end

% set up the coefficient matrix for the boundary parts
for i=1:M+2
    D(i,i)=a;
    D(tp+1-i,tp+1-i)=1-a;
end
for i=1:L
```

```
    D((M+1)*i,(M+1)*i)=1;
    D((M+1)*i+1,(M+1)*i+1)=1-a;
end


%----calculate all points by times the coefficient matrix
 %tic
 for p=2:N+1
      A(:,p)=D*A(:,p-1);

     for j=1:M+1
          A(j,p)=A(j,p)-2*dx*bx*((j-1)*dy)^2;
          A(L*(M+1)+j,p)=A(L*(M+1)+j,p)-2*dx*bx*(2+((j-1)*dy)^2);
     end
     for k=1:L
          A(k*(M+1),p)=A(k*(M+1),p)-2*dy*by*(2+((k-1)*dx)^2);
          A(k*(M+1)+1,p)=A(k*(M+1)+1,p)-2*dx*bx*(((k-1)*dx)^2);
     end

 end


  %cpu_main_computing_time=toc;

 %-----calculate the exact points

 %tic
 for k=1:N+1
 for j=1:L+1
     for i=1:M+1
```

```
          Exact_A((j-1)*(M+1)+i,k)=-exp(-(pi^2)*(k-1)*dt)*sin(pi*(i-1)*dx
              )*sin(pi*(j-1)*dy);
      end
  end
  end

time=toc
```

6.7 2D Dirichlet BC

```
function imp_dir_2d_cpu
%-----------Coded by Junchi Zhang
%-----------Instructor: Prof Sarah Olson
clear
close all
clc

tic
%-----define the the length and steps
alpha=1;
X=1;
Y=1;
N=20;  % # of time step

%# of grid points on the domain x
M=160;
%# of grid points on the domain y
L=160;
%spacing between the M points on x
```

```
dx=X/M;
%spacing between the M points on x
dy=Y/L;
%——set up the proper dt
dt=((dx)^2)/(6*alpha);

%———denote total points as tp
tp=(M+1)*(L+1);

%———put the difference method in a matrix
bx=-(dt*alpha)/((dx)^2);
by=-(dt*alpha)/((dy)^2);
a=1+2*((dt*alpha)/(dx)^2)+2*((dt*alpha)/(dy)^2);


cpu_A=zeros(tp,N+1);
cpu_D=zeros(tp,tp);
%———set up the boundary conditions

for j=1:L+1
    for i=1:M+1
        cpu_A((j-1)*(M+1)+i,1)=-sin(pi*(i-1)*dx)*sin(2*pi*(j-1)*dy);
    end
end

for i=1:N+1
    for j=1:M+1
        cpu_A(j,i)=0; %the boundary conditions of bottom
        cpu_A(L*(M+1)+j,i)=0; %the boundary conditions of top
    end
    for k=1:L
```

```
          cpu_A(k*(M+1),i)=0; %the boundary conditions of right
          cpu_A(k*(M+1)+1,i)=0; %the boundary conditions of left
     end
end

% set up the coefficient matrix for the explicit parts
for j=1:L-1
     for i=2:M
    cpu_D(j*(M+1)+i,j*(M+1)+i)=a;
    cpu_D(j*(M+1)+i,j*(M+1)+i-1)=by;
    cpu_D(j*(M+1)+i,j*(M+1)+i+1)=by;
    cpu_D(j*(M+1)+i,j*(M+1)+i-M-1)=bx;
    cpu_D(j*(M+1)+i,j*(M+1)+i+M+1)=bx;
     end
end

% set up the coefficient matrix for the boundary parts
for i=1:M+2
     cpu_D(i,i)=1;
     cpu_D(tp+1-i,tp+1-i)=1;
end
for i=1:L
     cpu_D((M+1)*i,(M+1)*i)=1;
     cpu_D((M+1)*i+1,(M+1)*i+1)=1;
end

invD=eye(tp)/cpu_D;
%----calculate all points by times the coefficient matrix
 for i=2:N+1
       cpu_A(:,i)=invD*cpu_A(:,i-1);
 end
```

```
%-----the exact points
 for k=1:N+1
 for j=1:L+1
     for i=1:M+1
          cpu_Exact_A((j-1)*(M+1)+i,k)=-exp(-5*(pi^2)*(k-1)*dt)*sin(pi*(i
              -1)*dx)*sin(2*pi*(j-1)*dy);
     end
 end
 end

   cpu_total_time=toc;
Error=abs(cpu_A-cpu_Exact_A);
 Max_Cpu_Error=max(max(Error));



 fid = fopen('data.txt', 'w');
fprintf( 'The arguments are M: %f  L: %f  N: %f\n', M,L,N);
fprintf( 'Max error of CPU result is %f\n', Max_Cpu_Error);
fprintf( 'Total time costs of CPU is %f sec\n', cpu_total_time);
fclose(fid);
```

```
function imp_dir_2d_gpu
%---------Coded  by Junchi Zhang
%---------Instructor: Prof Sarah Olson
clear
close all
clc
```

```
tic
%———define the the length and steps
alpha=1;
X=1;
Y=1;
N=20; % # of time steps
%# of grid points on the domain x
M=120;
%# of grid points on the domain y
L=120;
%spacing between the M points on x
dx=X/M;
%spacing between the M points on x
dy=Y/L;
%——set up the proper dt
dt=((dx)^2)/(6*alpha);

%———denote total points as tp
tp=(M+1)*(L+1);

%———put the difference method in a matrix
bx=-(dt*alpha)/((dx)^2);
by=-(dt*alpha)/((dy)^2);
a=1+2*((dt*alpha)/(dx)^2)+2*((dt*alpha)/(dy)^2);

gpu_A=zeros(tp,N+1);
gpu_D=zeros(tp,tp);

for j=1:L+1
   temp_b=zeros(M+1,1);
     for i=1:M+1
```

```
          temp_b(i)=-sin(pi*(i-1)*dx)*sin(2*pi*(j-1)*dy);
      end
    temp_B(j,:)=temp_b;
end
for k=1:L+1
gpu_A((k-1)*(M+1)+1:k*(M+1),1)=temp_B(k,:);
end



for i=1:N+1
      for j=1:M+1
          gpu_A(j,i)=0; %the boundary conditions of bottom
          gpu_A(L*(M+1)+j,i)=0; %the boundary conditions of top
      end
      for k=1:L
          gpu_A(k*(M+1),i)=0; %the boundary conditions of right
          gpu_A(k*(M+1)+1,i)=0; %the boundary conditions of left
      end
end




% set up the coefficient matrix for the explicit parts
for j=1:L-1
      for i=2:M
    gpu_D(j*(M+1)+i,j*(M+1)+i)=a;
    gpu_D(j*(M+1)+i,j*(M+1)+i-1)=by;
    gpu_D(j*(M+1)+i,j*(M+1)+i+1)=by;
    gpu_D(j*(M+1)+i,j*(M+1)+i-M-1)=bx;
    gpu_D(j*(M+1)+i,j*(M+1)+i+M+1)=bx;
      end
end

% set up the coefficient matrix for the boundary parts
for i=1:M+2
      gpu_D(i,i)=1;
      gpu_D(tp+1-i,tp+1-i)=1;
end
for i=1:L
      gpu_D((M+1)*i,(M+1)*i)=1;
      gpu_D((M+1)*i+1,(M+1)*i+1)=1;
end


%-----calculate all points by times the coefficient matrix

gpu_d=gpuArray(gpu_D);
gpu_inv_d=(eye(tp)/gpu_d);


  for i=2:N+1
         gpu_a(:,i)=gpu_inv_d*gpu_A(:,i-1);
  end
    Gpu_A=gather(gpu_a);


     for k=1:N+1
    temp_c=zeros(M+1,L+1);
  for j=1:L+1
       for i=1:M+1
```

```
            temp_c(i,j)=k+i-j;
        end
    end
    temp_C(:,:,k)=temp_c(:,:);
    end
    for k=1:N+1
    for j=1:L+1
        for i=1:M+1
            gpu_Exact_A((j-1)*(M+1)+i,k)=-exp(-5*(pi^2)*(k-1)*dt)*sin(pi*(i
                -1)*dx)*sin(2*pi*(j-1)*dy);
        end
    end
    end

    gpu_total_time=toc;


Error=abs(Gpu_A-gpu_Exact_A);
    Max_Gpu_Error=max(max(Error));

    fid = fopen('data.txt', 'w');
fprintf( 'The arguments are M: %f  L: %f  N: %f\n', M,L,N);
fprintf( 'Max error of GPU result is %f\n', Max_Gpu_Error);
fprintf( 'Total time costs of GPU is %f sec\n', gpu_total_time);
fclose(fid);
```

```
function imp_dir_2d_gpu
%----------Coded by Junchi Zhang
%----------Instructor: Prof Sarah Olson
clear
```

```
close all
clc

tic
%———define the the length and steps
alpha=1;
X=1;
Y=1;
N=20; % # of time steps
%# of grid points on the domain x
M=120;
%# of grid points on the domain y
L=120;
%spacing between the M points on x
dx=X/M;
%spacing between the M points on x
dy=Y/L;
%——set up the proper dt
dt=((dx)^2)/(6*alpha);

%———denote total points as tp
tp=(M+1)*(L+1);

%———put the difference method in a matrix
bx=-(dt*alpha)/((dx)^2);
by=-(dt*alpha)/((dy)^2);
a=1+2*((dt*alpha)/(dx)^2)+2*((dt*alpha)/(dy)^2);

gpu_A=zeros(tp,N+1);
gpu_D=zeros(tp,tp);
```

```
for  j =1:L+1
   temp_b=zeros (M+1 ,1) ;
     for  i =1:M+1
             temp_b ( i )=−sin ( pi *( i −1)*dx ) * sin (2* pi *( j −1)*dy ) ;
      end
     temp_B ( j , : ) =temp_b ;
end
for  k=1:L+1
gpu_A ( ( k−1) *(M+1) +1:k *(M+1) ,1)=temp_B ( k , : ) ;
end



for  i =1:N+1
      for  j =1:M+1
             gpu_A ( j , i ) =0; %the  boundary  conditions  of  bottom
             gpu_A (L *(M+1)+j , i ) =0; %the  boundary  conditions  of  top
      end
      for  k=1:L
             gpu_A ( k *(M+1) , i ) =0; %the  boundary  conditions  of  right
             gpu_A ( k *(M+1) +1, i ) =0; %the  boundary  conditions  of  left
      end
end




% set  up  the  coefficient  matrix  for  the  explicit  parts
for  j =1:L−1
      for  i =2:M
     gpu_D( j *(M+1)+i , j *(M+1)+i )=a ;
```

```
    gpu_D(j*(M+1)+i,j*(M+1)+i-1)=by;
    gpu_D(j*(M+1)+i,j*(M+1)+i+1)=by;
    gpu_D(j*(M+1)+i,j*(M+1)+i-M-1)=bx;
    gpu_D(j*(M+1)+i,j*(M+1)+i+M+1)=bx;
     end
end

% set up the coefficient matrix for the boundary parts
 for i=1:M+2
     gpu_D(i,i)=1;
     gpu_D(tp+1-i,tp+1-i)=1;
end
 for i=1:L
     gpu_D((M+1)*i,(M+1)*i)=1;
     gpu_D((M+1)*i+1,(M+1)*i+1)=1;
end


%-----calculate all points by times the coefficient matrix

gpu_d=gpuArray(gpu_D);
gpu_inv_d=(eye(tp)/gpu_d);


  for i=2:N+1
         gpu_a(:,i)=gpu_inv_d*gpu_A(:,i-1);
  end
   Gpu_A=gather(gpu_a);


    for k=1:N+1
  temp_c=zeros(M+1,L+1);
 for  j=1:L+1
     for  i=1:M+1
          temp_c(i,j)=k+i-j;
     end
 end
 temp_C(:,:,k)=temp_c(:,:);
 end
 for  k=1:N+1
 for  j=1:L+1
     for  i=1:M+1
          gpu_Exact_A((j-1)*(M+1)+i,k)=-exp(-5*(pi^2)*(k-1)*dt)*sin(pi*(i
              -1)*dx)*sin(2*pi*(j-1)*dy);
     end
 end
 end

 gpu_total_time=toc;

Error=abs(Gpu_A-gpu_Exact_A);
 Max_Gpu_Error=max(max(Error));

 fid = fopen('data.txt', 'w');
fprintf( 'The arguments are M: %f  L: %f  N: %f\n', M,L,N);
fprintf( 'Max error of GPU result is %f\n', Max_Gpu_Error);
fprintf( 'Total time costs of GPU is %f sec\n', gpu_total_time);
fclose(fid);
```